# Špecifikácia a verifikácia programov v Peanovej aritmetike

DIZERTAČNÁ PRÁCA

Ján Komara

**UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA APLIKOVANEJ INFORMATIKY**

Teoretická informatika 11-80-9

Vedúci dizertačnej práce
doc. PhDr. Ján Šefránek, CSc.

BRATISLAVA 2009

*Vyhlasujem, že predložená práca je mojím pôvodným dielom, ktoré som vypracoval samostatne s použitím zdrojov uvedených v zozname literatúry.*

# Abstrakt

V tejto práci sa zaoberáme matematickými základmi návrhu programovacieho jazyka a jeho formálneho systému. Pri našom prístupe programy sú vlastnosti totálnych funkcií nad oborom prirodzených čísel spĺňajúce isté vstupné podmienky. To nám umožňuje analýzu nekončiacich sa programov v rámci formalizmu totálnych funkcií. Dátové štruktúry aritmetizujeme do prirodzených čísel pomocou párovacej funkcie. Špecifikačno-verifikačný systém je klasická prvorádová formalizácia aritmetiky, ktorá sa volá Peanova aritmetika. Náš hlavný prínos spočíva v návrhu programovacieho jazyka s takmer neohraničenou rekurziou a s flexibilnou syntaxou programátorských konštrukcií, ktorým je expresívny jazyk podmienkových výrazov. Navyše tento jazyk je plne sformalizovaný v Peanovej aritmetike. Časť týchto idey už bola implementovaná v programovacom jazyku CL vrátane interaktívneho dokazovacieho systému.

**Kľúčové slová:** Deklaratívne programovanie, Rekurzívne funkcie, Špecifikácia a verifikácia programov, Peanova aritmetika.

# Specification and Verification of Programs in Peano Arithmetic

DISSERTATION

Ján Komara

**COMENIUS UNIVERSITY BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS
AND INFORMATICS
DEPARTMENT OF APPLIED
INFORMATICS**

Theoretical informatics 11-80-9

Supervisor
doc. PhDr. Ján Šefránek, CSc.

BRATISLAVA 2009

*Hereby I declare that I wrote this thesis myself with the help of no more than the referenced sources and the work represents the original contribution of the author unless stated otherwise.*

# Abstract

In this work we are concerned with the problem of mathematical explication of a programming language and its formal system. In our approach programs are properties of total functions over natural numbers satisfying certain preconditions. This allows analysis of non-terminating programs within the framework of total functions. Data structures are arithmetized into natural numbers with the help of a pairing function. The specification-verification system is based on the classical first order formalization of arithmetic known as Peano Arithmetic. Our main contribution lies in the design of the programming language with almost unrestricted recursion and with extensible syntax of programming construct (expressive language of conditionals). Moreover, the language is completely formalized within Peano Arithmetic. Some of these ideas have already been implemented in the declarative programming language and proof assistant CL.

**Keywords:** Declarative Programming, Recursive Functions, Specification and Verification of Programs, Peano Arithmetic.

# Preface

**Goals.** The main objective of this work is to give a mathematical explication of a programming language together with its formal system. We are looking for a solution in the paradigm of declarative programming which combines simple semantics with expressive power of programming constructs.

**Current state of the area.** Declarative programming is a modern trend of programming because programs, being definitions (properties) of mathematical objects can be rigorously analysed. This aspect allows formal specification of programs as well as formal verification of their properties. This is a very active field of research as demonstrated by at least a dozen projects listed below. Some of them come with an integrated proof assistant.

Current specification-verification languages widely differ in the strength of the theory they used. We mention here just some of them. The languages Vdm [18, 9] and Raise [14] are based on a first order logic with partial functions. The Z specification language [48] is based on a typed first order set theory. Algebraic specification languages are Obj [10, 11] and Larch [15]. Higher order logic is used by many systems such as Pvs [8, 35, 36], Ehdm [32], Isabelle [37, 33]. and Hol [12]. Some systems like Coq [17, 3] and Nuprl [7, 1] are based on intuitionistic higher order logics.

In our research we have been influenced also by the systems Nqthm [4] and Acl2 [20, 19, 13]. The system Nqthm is a theorem prover developed by R.S. Boyer and J.S. Moore in 1970's. Its logic is a first order *quantifier-free* theory of *S-expressions* as known from Lisp. The logic permits the user to axiomatize inductively constructed data types and recursively define (total) functions. The logic provides also well-founded induction on the ordinals up to $\epsilon_0$ and the witnessed constraint of new function symbols giving the logic some of the features of a high-order logic. A Computational Logic for Applicative Common Lisp (Acl2) is a direct descendant of Nqthmintended for large scale verification projects. While the logic of Nqthm is based on pure Lisp, the logic of Acl2 is based on the applicative subset of Common Lisp. The system is written in the logic it supports.

**Proposed solutions.** Our approach is based on the thesis of Church asserting that the class of effectively computable functions over natural numbers coincides with general recursive functions as defined by Herbrand-Gödel. We use Herbrand-Gödel-like recursive equations because they offer the programming comfort with almost unrestricted kinds of recursion and the computation of recursive equations by reductions permits a fine degree of control over the length of reduction sequences. We interpret the recursive equations into natural numbers because the concept of natural numbers is well understood even by beginners and the theory of recursive functions and arithmetic offers a firm natural semantic background.

A possible objection by computer scientists that the domain of natural numbers means unpleasant coding (arithmetization) of the rich set of data structures as used in computer programming is answered by coding into natural numbers in the style of Lisp with a pairing function (instead of *cons*). We obtain a degree of comfort as it is known from declarative programming languages. The examples given in the last two chapters, where we deal with programs operating over lists, binary trees and symbolic expressions, should convince the reader.

The computational model is based on reduction of terms. Programs are properties of (total) functions over natural numbers. Each program has assigned certain precondition describing which elements can be used as its inputs. Regularity conditions for a program guarantee that computation terminates correctly for every input satisfying its precondition, for other inputs computation might return wrong answer or even diverges. This conception of programs being properties with preconditions allows analysis of non-terminating programs within the framework of total functions.

For efficient computation one needs strong schemes of recursion and case analysis. Our strongest definitional schemes are regular recursive definitions into well-founded relations. The language of expressions is extended with a powerful generalization of (non-extensible) case constructs and pattern matching known from declarative programming languages. These new constructs, called case discrimination terms in this text, have flexible syntax which legality must be certified by a formal proof. Each case discrimination term has assigned certain precondition prescribing for which inputs the case analysis must be pairwise-disjoint and exhaustive.

This should be contrasted with the discipline of *Total Functional Programming* discussed in [51, 52]. In order to avoid the problems with partial functions, Turner proposed a language with limited expressivity of some programming constructs (notably recursion and case analysis) to exclude non-terminating programs. A similar approach to ours has been used in [56] to tackle the problem of non-termination but there are some significant differences between both methods. The idea of extensible syntax of case constructs is not new: in [56, 30] they consider approach which expressivity is similar to ours. We think however that our proposed solution has certain advantages because our case constructs are just ordinary expressions with simple semantics. This approach has been already studied by the author in [28].

We are interested not only in a programming language but also in verification of its programs. By restricting ourselves to the domain of natural numbers we can use the first order formalization of arithmetic called Peano Arithmetic as our specification-verification system. This is probably the most simple formal theory suitable for the theory of programming languages. But we still have to introduce the derived concepts as used in programming, data structures or recursive programs, for instance. This is done by showing that these new concepts can be introduced and then prove their properties in some of definitional extensions of Peano Arithmetic.

**Contributions.** In the part dealing with recursive bootstrapping of Peano Arithmetic we have contributed the following:

- A quick and easy introduction of coding into Peano Arithmetic (Chap. 2). We use a modified version of Cantor pairing function to develop the coding of tuples and finite sequences. The pairing function is then used in recursive bootstrapping of Peano Arithmetic in the next chapter.
- A novel proof that Peano Arithmetic admits definitions by nested simple recursion (Sect. 3.5). We have already investigated the scheme in [25] but we hope that this new presentation, which is supplied with a detailed formal proof for the first time, is much simpler and easier to follow.
- Showing that Peano Arithmetic admits definitions by powerful recursive schemes such as recursion with measure (Sect. 4.2), well-founded recursion (Sect. 4.3) and regular recursion (Sect. 4.4). The first two schemes are reduced to nested simple recursion by the technique already developed in [25, 54]. Dropping the syntactic restrictions imposed on well-founded recursion leads to regular recursive definitions. This is our most general scheme of recursion formalized within Peano Arithmetic.
- By allowing programs to be properties obeying certain input conditions we may have non-terminating algorithms but they can be analysed fully within the framework of Peano Arithmetic (Sect. 4.5).

In the part dealing with theory of programming our main contribution is in the choice of a simple semantics of a programming language and in the design of its formal system. This is discussed in Chap. 5. More specifically, we have contributed the following:

- Design of a programming language with extensible syntax of programming constructs (Sect. 5.2). We add to the language of Peano Arithmetic case discrimination terms, which are powerful generalizations of case constructs known from declarative programming languages. These new constructs have flexible syntax which legality must be certified by a formal proof in Peano Arithmetic.
- Design of a programming language with almost unrestricted recursion (Sect. 5.2). We will show that Peano Arithmetic admits a very flexible

kind of extensions by regular recursive definitions in which user-defined conditionals can be used. This is our most expressive scheme of recursive definitions formalized within PA. Clausal form of such definitions is discussed in Sect. 5.4.
- Arithmetization of common data structures needed in the computer programming with the level of comfort comparable to that in the declarative programming languages. Formalization of structural recursion/induction as shown in Chaps. 6 and 7, where we use these constructs to define functions operating over lists, binary trees and symbolic expressions.
- The use of Peano Arithmetic as a single framework for the design of a programming language and its specification-verification system.

Finally, we mention here also two of our contributions which did not come into this text. Nevertheless, they play important part either in the part where we borrow from mathematical logic or in the part where we are concerned with the design of the logical framework for verification of programs:

- In [28] we gave a simple *finitary* proof of conservativity of Skolem axioms. (see Thm. 1.3.11). The proof is based on the ideas developed in [22, 23].
- In [26] we have proved the admissibility of so-called *predicate induction rules*, which are very expressive schemes of induction derived from recursive definitions of predicates. Boyer and Moore [4] were the first to use such rules, and similar rules are used in HOL [31, 12]

Many of the forementioned contributions are joint results of the scientific collaboration of the author and his colleague Pavol Voda.

**Implementations.** We have already achieved some practical results in the application of our theoretical investigation in the area of computer programming. We have provided our own design and implementation of the declarative programming language CL (Clausal Language) [42, 24]. It comes with its own theorem prover which enables to formalize and prove properties of programs in Peano Arithmetic. The system was designed and implemented in 1997–2003. The authors are J. Kľuka (interface), J. Komara (proof system) and P.J. Voda (processor).

We use the programming and verification system CL in the following courses of undergraduate/graduate study at our university:

- *Declarative Programming*, where we teach the introduction into declarative programming within the formalism of primitive recursive functions.
- *Theory of Declarative Programming*, where we teach theory of declarative programming languages within the framework of the classical recursion theory (primitive, general and partial recursive functions).
- *Specification and Verification of Programs*, where we teach the introduction into first order arithmetic and where we use Peano Arithmetic as a specification, implementation and verification framework.
- *Computability Theory*, where we teach the classical theory of computability based on Turing machines.

Our experience shows that the undergraduate students have no problems defining functions in Peano arithmetic and have little problems doing formal proofs of their properties. This is primarily because they have a good intuition about the domain of natural numbers.

The interested reader will find in our homepage the executable file of an implementation of the system as well as the lecture notes for the courses we teach with the system (see [42, 28, 27]).

**Acknowledgements.** The author would like to thank his colleague Pavol Voda for his enormous contribution to this text. We have worked jointly over many years and the clausal language was designed and implemented as our common project in the form of the system CL. The author would also like to thank Ján Šefránek for his support during the last years.

# Contents

# Prerequisites and Notation

The only prerequisite is a knowledge of naive set theory and familiarity with basic logic notation. Important facts from the mathematical logic which are assumed to be known will be introduced in the next chapter.

**Logical notation.** We will use the symbol $\equiv$ as the syntactical identity over syntactical objects such as terms and formulas. Also we will use $\equiv$ as the syntactical identity over finite sequences of such objects.

Terms are formed from variables and constants by applications of function symbols in the usual way. Closed terms do not have free variables. We use lower Greek letters $\tau, \rho, \theta$ as syntactic variables ranging over terms.

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

When we write $\tau[f; \vec{x}]$ we indicate that the term $\tau$ 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 the term $\tau$ by the substitution of terms $\vec{\rho}$ for the corresponding variables of $\vec{x}$ as well as by the replacement of all applications $f(\vec{\theta})$ by applications $g(\vec{\theta})$.

An atomic formula is either a predicate application or an identity $\tau = \rho$. Formulas are formed from atomic formulas and propositional constants by applications of propositional connectives and quantifiers in the usual way:

| | | |
|---|---|---|
| $\top$ (true) | $\varphi \land \psi$ (conjunction) | $\varphi \leftrightarrow \psi$ (equivalence) |
| $\bot$ (falsehood) | $\varphi \lor \psi$ (disjunction) | $\forall x \varphi$ (universal quantifier) |
| $\neg \varphi$ (negation) | $\varphi \to \psi$ (implication) | $\exists x \varphi$ (existential quantifier). |

Closed formulas (i.e. sentences) do not have free variables. We will use lower Greek letters $\varphi, \psi$ as syntactic variables ranging over formulas.

In order to improve readability of formulas, we let all binary propositional connectives group to the right. We assign the highest precedence to the quantifiers and the negation. Next lower precedence has the conjunction and then the disjunction. The connectives of implication and equivalence have the lowest precedence. For instance, the formula $\varphi_1 \rightarrow \varphi_2 \leftrightarrow \neg\varphi_3 \vee \exists x \varphi_4 \vee \varphi_5$ should be read as $\varphi_1 \rightarrow (\varphi_2 \leftrightarrow \neg(\varphi_3 \vee ((\exists x \varphi_4) \vee \varphi_5)))$.

By $\tau \neq \rho$ we designate the formula $\neg \tau = \rho$. We generalize some of the propositional connectives to for finite sequences. The generalized conjunction $\bigwedge_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \wedge \cdots \wedge \varphi_n$ if $n \geq 1$ and for $\top$ if $n = 0$. We define the generalized disjunction $\bigvee_{i=1}^{n} \varphi_i$ similarly. By $\forall \vec{x} \varphi$ and $\exists \vec{x} \varphi$ we designate the formulas $\forall x_1 \ldots \forall x_n \varphi$ and $\exists x_1 \ldots \exists x_n \varphi$, respectively. By $\forall \varphi$ we denote the universal closure of the formula $\varphi$.

Similar conventions as those for terms will be adopted also for formulas. Only substitution requires a brief explanation. Whenever we write $\varphi[\vec{\tau}]$ it is assumed that the bound variables of the formula $\varphi[\vec{x}]$ are first renamed so that they do not appear in the terms $\vec{\tau}$. Recall that a formula does not change its meaning if one of its bound variables is changed to another.

**Natural numbers.** If we do not state explicitly $n$-ary functions and predicates are over the domain of natural numbers

$$\mathrm{N} = \{0, 1, 2, 3, 4, 5, \ldots\}.$$

We implicitly assume that we have $n \geq 1$; this means that our functions and predicates have always non-zero arity. Furthermore, $n$-ary functions are always total, i.e. with the domain being the whole Cartesian product $\mathrm{N}^n$.

Natural numbers are closed under the operations of addition $x + y$ and multiplication $x \times y$ (written $xy$ for short) but not under subtraction $x - y$ and division $\frac{x}{y}$. For instance, we have $3 - 5 = -2 < 0$ and $1 < \frac{5}{3} < 2$.

Instead of subtraction we will use modified subtraction $x \dot{-} y$ which is over natural numbers and it is defined by

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

The modified subtraction has the following basic properties:

$$y \leq x \rightarrow x = y + (x \dot{-} y) \qquad x \leq y \rightarrow x \dot{-} y = 0.$$

Note that we then have $5 \dot{-} 3 = 2$ and $3 \dot{-} 5 = 0$.

Instead of division we will use euclidean division. Recall that for every natural numbers $x$ and $y \neq 0$ there exist unique natural numbers $q$ and $r < y$ such that $x = qy + r$ holds. The numbers $q$ and $r$ are called respectively the quotient and the remainder of the euclidean division of $x$ by $y$. We denote by $x \div y$ the binary integer division function and by $x \bmod y$ the binary remainder function yielding respectively the quotient and remainder of the euclidean division of the number $x$ by $y$. The functions are defined to satisfy:

$$x \div y = \begin{cases} q & \text{if } y \neq 0 \text{ and } x = qy + r \text{ for some } r < y, \\ 0 & \text{otherwise.} \end{cases}$$

$$x \bmod y = \begin{cases} r & \text{if } y \neq 0 \text{ and } x = qy + r \text{ for some } q \text{ such that } r < y, \\ 0 & \text{otherwise.} \end{cases}$$

The functions have the following basic properties:

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

For instance, we have $5 \div 3 = 1$ and $5 \bmod 3 = 2$.

The binary exponentiation function $x^y$ has a following recursive definition:

$$x^0 = 1$$
$$x^{y+1} = xx^y.$$

Note that we have $x^y = 0 \leftrightarrow x = 0 \wedge y \neq 0$ and $x^y = 1 \leftrightarrow x = 1 \vee y = 0$.

For an $n$-ary predicate $R$, we denote by $R_*$ its characteristic function which is an $n$-ary function such that

$$R_*(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}), \\ 0 & \text{if not } R(\vec{x}). \end{cases}$$

Note that the value 1 means truth while the value 0 means falsehood. We designate by $x =_* y$, $x \leq_* y$ and $x <_* y$ the characteristic functions of the binary predicates $x = y$, $x \leq y$ and $x < y$, respectively. We adopt the same convention also for other binary predicates written in infix notation.

**References and meta-logical notation.** Chapters are divided into sections and these into consecutively numbered paragraphs such as definitions, theorems and remarks. Thus 5.3.4 is the 4th paragraph of the 3rd section of the 5th chapter. When a reference is made to a numbered equation within the same paragraph, both chapter and section numbers are omitted.

The word "iff" abbreviates "if and only if"; "s.t." abbreviates "such that"; "IH" abbreviates "induction hypothesis" and "IHs" is the plural form of "IH". The symbol $\Rightarrow$ denotes the word "implies", while the symbol $\Leftrightarrow$ means "implies and is implied by". Finally note that the conclusion of a proof is usually indicated by the symbol $\square$.

# Chapter 1
# Natural Numbers

We begin by introducing the paradigm of declarative programming and give some arguments over programming over natural numbers within the framework of (total) functions (Sect. 1.1). To make this text self-contained we have included three sections where we survey some basic facts from recursion theory and mathematical logic. In Sect. 1.2 we give a brief overview of primitive and μ-recursive functions. The subsequent two sections discuss first order theories in general (Sect. 1.3) and Peano Arithmetic in detail (Sect. 1.4).

## 1.1 Declarative Programming

**1.1.1 Introduction.** The style of programming where programs modify memory by obeying sequences of commands is called *imperative programming*. The term *declarative programming* is used for the style of programming where programs are definitions (properties) of mathematical objects such as functions or predicates. To illustrate this paradigm we give here a few typical algorithms written declaratively. At the end of this section we give some arguments in favour of declarative programming over natural numbers.

**1.1.2 Euclidean algorithm.** The *greatest common divisor* of two numbers $x$ and $y$, where at least one is non-zero, is the largest number that divides them both; we designate it by $\mathrm{gcd}(x, y)$. The function gcd is uniquely determined by the following *specification*:

$$\begin{gathered}\mathrm{gcd}(0,0) = 0\\ x \neq 0 \lor y \neq 0 \to \mathrm{gcd}(x,y) \mid x \land \mathrm{gcd}(x,y) \mid y\\ (x \neq 0 \lor y \neq 0) \land z \mid x \land z \mid y \to z \leq \mathrm{gcd}(x,y).\end{gathered}$$

Here, the $x \mid y$ is the binary *divisibility* predicate holding if the number $x$ divides the number $y$, i.e. if we have $y = xz$ for some number $z$.

An efficient algorithm for computing the greatest common divisor was described by the ancient mathematician Euclid which relies on the following property of divisibility:

$$x > y \land z \mid y \to z \mid x \leftrightarrow z \mid x \dot{-} y$$

This means that the greatest common divisor does not change if the smaller number is subtracted from the larger. Thus $\mathrm{gcd}(x, y)$ is $\mathrm{gcd}(x \dot{-} y, y)$ if $x > y$ and $\mathrm{gcd}(x, y \dot{-} x)$ if $x < y$.

The algorithm of Euclid can be expressed by the following *declarative* program which is also a recursive definition of the greatest divisor function:

```
$\mathrm{gcd}(x,y)$ = if $x \neq 0 \land y \neq 0$ then
                        case
                          $x > y \Rightarrow \mathrm{gcd}(x \dot{-} y, y)$
                          $x = y \Rightarrow x$
                          $x < y \Rightarrow \mathrm{gcd}(x, y \dot{-} x)$
                        end
                      else
                        $\max(x,y)$.
```

Here, the $\max(x, y)$ is the *maximum* of the numbers $x$ and $y$. The definition is legal because the arguments of both recursive applications go down in the *measure* $\max(x, y)$ as we have

$$x \neq 0 \land y \neq 0 \land x > y \to \max(x \dot{-} y, y) < \max(x, y)$$
$$x \neq 0 \land y \neq 0 \land x < y \to \max(x, y \dot{-} x) < \max(x, y).$$

The properties are called the *conditions of regularity* of the definition. They guarantee that the functional equation has a unique solution.

The expression on the right side of the definition applies two *conditionals*. The first one is an ordinary test on whether or not $x \neq 0 \land y \neq 0$. The second conditional is *trichotomy discrimination* on the numbers $x, y$ by examining which one of the following properties $x < y$, $x = y$ and $x > y$ holds.

We can use the defining equation as a computation rule from left to right to evaluate applications of the greatest divisor function. For instance, the following is the reduction sequence for evaluation of $\mathrm{gcd}(21, 12)$:

$$\mathrm{gcd}(21,12) = \mathrm{gcd}(9,12) = \mathrm{gcd}(9,3) = \mathrm{gcd}(6,3) = \mathrm{gcd}(3,3) = 3.$$

Regularity conditions guarantee that computation of gcd using the defining identity always terminates.

**1.1.3 Euclidean algorithm revisited.** The program for the greatest divisor function described in the previous paragraph is less optimal than it should be due to repeated test $x \neq 0 \land y \neq 0$ in each recursive call. We obtain a better one by computing the greatest divisor using the following identity as reduction rule:

$$\begin{aligned} \mathrm{gcd}(x,y) = \mathbf{case}\; & \\ & x > y \Rightarrow \mathrm{gcd}(x \dot{-} y, y) \\ & x = y \Rightarrow x \\ & x < y \Rightarrow \mathrm{gcd}(x, y \dot{-} x) \\ \mathbf{end}. & \end{aligned}$$

The program works correctly for those inputs that satisfy the following property: $x \neq 0 \land y \neq 0$. This is called the *precondition* of the program. Note that if we write the above program together with its precondition

$$\begin{aligned} x \neq 0 \land y \neq 0 \rightarrow \mathrm{gcd}(x,y) = \mathbf{case}\; & \\ & x > y \Rightarrow \mathrm{gcd}(x \dot{-} y, y) \\ & x = y \Rightarrow x \\ & x < y \Rightarrow \mathrm{gcd}(x, y \dot{-} x) \\ \mathbf{end}, & \end{aligned}$$

we obtain an assertion in the form of conditional identity which is also the property of the greatest divisor function.

The following properties are called the *(extended) conditions of regularity* of the program:

$$\begin{aligned} &x \neq 0 \land y \neq 0 \land x > y \rightarrow \max(x \dot{-} y, y) < \max(x,y) \land x \dot{-} y \neq 0 \land y \neq 0 \\ &x \neq 0 \land y \neq 0 \land x < y \rightarrow \max(x, y \dot{-} x) < \max(x,y) \land x \neq 0 \land y \dot{-} x \neq 0. \end{aligned}$$

These conditions guarantee that computation of the greatest divisor function for inputs $x, y$ satisfying the input condition always terminates yielding the correct result $\mathrm{gcd}(x,y)$. Note that we require not only that recursion goes down in the measure $\max(x,y)$ but also that the arguments of recursive applications satisfy the precondition of the program.

For inputs violating the precondition of the program computation might not terminate as it shown in the following reduction sequence:

$$\mathrm{gcd}(1,0) = \mathrm{gcd}(1 \dot{-} 0, 0) = \mathrm{gcd}(1,0) = \cdots.$$

This means that if we allow unrestricted recursion in programs we would have to deal with partial functions. By insisting that inputs should satisfy preconditions of programs we remain within the framework of (total) functions.

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

$$\begin{aligned} \mathrm{fib}(0) &= 0 \\ \mathrm{fib}(1) &= 1 \\ \mathrm{fib}(n+2) &= \mathrm{fib}(n+1) + \mathrm{fib}(n). \end{aligned}$$

This is an example of *course of values recursive* definition, where the arguments of recursive applications decrease in the relation $<$: we have $n+1 < n+2$ for the first recursive application and $n < n+2$ for the second.

We can use the recurrences directly for computation. For instance:

$$\begin{aligned}
&\mathrm{fib}(4) = \mathrm{fib}(3) + \mathrm{fib}(2) = \big(\mathrm{fib}(2) + \mathrm{fib}(1)\big) + \mathrm{fib}(2) = \\
&\quad = \Big(\big(\mathrm{fib}(1) + \mathrm{fib}(0)\big) + \mathrm{fib}(1)\Big) + \mathrm{fib}(2) = \Big(\big(1 + \mathrm{fib}(0)\big) + \mathrm{fib}(1)\Big) + \mathrm{fib}(2) = \\
&\quad = \big((1+0) + \mathrm{fib}(1)\big) + \mathrm{fib}(2) = \big(1 + \mathrm{fib}(1)\big) + \mathrm{fib}(2) = (1+1) + \mathrm{fib}(2) = \\
&\quad = 2 + \mathrm{fib}(2) = 2 + \big(\mathrm{fib}(1) + \mathrm{fib}(0)\big) = 2 + \big(1 + \mathrm{fib}(0)\big) = \\
&\quad = 2 + (1+0) = 2 + 1 = 3.
\end{aligned}$$

The only problem is that the computation sequence is too long. In order to compute the number $\mathrm{fib}(n)$ one needs to use the defining recurrences approximately $\mathrm{fib}(n)$ times. The Fibonacci function grows as fast as the exponential function and to compute the function in this way is simply too wasteful.

We give here more satisfactory implementation of the Fibonacci function by an imperative program. The following PASCAL-like program computes $\mathrm{fib}(n+1)$ into the variable $a$:

```
$a := 1;\ b := 0;$
$\mathit{while}\ n \neq 0\ \mathit{do}$
    $n := n - 1;\ c := a;\ a := a + b;\ b := c;$
```

The reader will note that the while-loop is executed only $n$ times. This example is usually given as the 'standard argument' against declarative programming where the recursive version is clearly inferior to the imperative.

The argument is fallacious as one should define an auxiliary ternary function $g(n, a, b)$ with two *accumulators* $a$ and $b$ by *primitive recursion*:

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

and then we take the following identity as an alternate program for $\mathrm{fib}(n)$:

$$\begin{aligned}
\mathrm{fib}(0) &= 0 \\
\mathrm{fib}(n+1) &= g(n, 1, 0). \qquad (1)
\end{aligned}$$

The number of recursions of $g(n, a, b)$ is exactly the same as the number of iterations of the loop of the imperative program. Moreover, a good compiler can remove the so-called *tail recursion* in the definition of $f$ and compile it similarly as the while-loop in the above PASCAL-like program.

It remains to show that the identity (1) is true. For that we need the following property of the auxiliary function $g$:

$$\forall k\, g\big(n, \mathrm{fib}(k+1), \mathrm{fib}(k)\big) = \mathrm{fib}(n+1+k) \qquad (2)$$

which is proved by induction on $n$. In the base case take any $k$ and we have

$$g\big(0, \mathrm{fib}(k+1), \mathrm{fib}(k)\big) = \mathrm{fib}(k+1) = \mathrm{fib}(0+1+k).$$

In the induction step take any $k$ and we have

$$\begin{aligned} g\big(n+1, \mathrm{fib}(k+1), \mathrm{fib}(k)\big) &= g\big(n, \mathrm{fib}(k+1) + \mathrm{fib}(k), \mathrm{fib}(k+1)\big) = \\ &= g\big(n, \mathrm{fib}(k+2), \mathrm{fib}(k+1)\big) = g\big(n, \mathrm{fib}(k+1+1), \mathrm{fib}(k+1)\big) \overset{\mathrm{IH}}{=} \\ &= \mathrm{fib}(n+1+k+1) = \mathrm{fib}(n+1+1+k). \end{aligned}$$

We are now ready to prove (1):

$$\begin{aligned} \mathrm{fib}(n+1) = \mathrm{fib}(n+1+0) &\overset{(2)}{=} g\big(n, \mathrm{fib}(0+1), \mathrm{fib}(0)\big) = \\ &= g\big(n, \mathrm{fib}(1), \mathrm{fib}(0)\big) = g(n, 1, 0). \end{aligned}$$

**1.1.5 Arithmetization of word domains.** In the examples discussed so far the domain of values over which the functions were operated was the domain of natural numbers. But what about complex data structures used in computer programming? Is not the restriction to natural numbers unrealistic one? The answer lies in the *coding* of data structures into the domain of natural numbers. The process of going from operations over certain domain to the operations over the codes of elements of the domain in N is called the *arithmetization* of the domain. In this paragraph we illustrate the problem of arithmetization for word domains.

Consider the two-elements alphabet $\Sigma = \{1, 2\}$. We can code words over $\Sigma$ with the help of *dyadic successors* functions explicitly defined by:

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

It is not difficult to see that every natural number has a unique representation as a *dyadic numeral* which are terms built up from the constant 0 by applications of dyadic successors. This is called *dyadic representation* of natural numbers. Consider, for instance, the first eight words from the sequence of words over the alphabet $\Sigma$ which is ordered first on the length and then within the same length lexicographically:

$$\varnothing, 1, 2, 11, 12, 21, 22, 111.$$

The corresponding dyadic numerals are shown in Fig. 1.1. Arithmetization is so straightforward that, from now on, we will usually identify dyadic words with their code numbers.

The *dyadic size* function $|x|_{\mathrm{d}}$ 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 $\Sigma$ and yielding its length. The

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

**Fig. 1.1** Dyadic representation of natural numbers

function is defined by

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

This is a correct definition because recursion decreases the argument since we clearly have $x < x\mathbf{1}$ and $x < x\mathbf{2}$.

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 $x \star y$ is the arithmetization of the word function concatenating words over the alphabet $\Sigma$. The function is defined by

$$\begin{aligned}
x \star 0 &= x && (1)\\
x \star y\mathbf{1} &= (x \star y)\mathbf{1} && (2)\\
x \star y\mathbf{2} &= (x \star y)\mathbf{2}. && (3)
\end{aligned}$$

We can use the identities for computations. For instance, we have

$$0\mathbf{21} \star 0\mathbf{121} \overset{(2)}{=} (0\mathbf{21} \star 0\mathbf{12})\mathbf{1} \overset{(3)}{=} (0\mathbf{21} \star 0\mathbf{1})\mathbf{21} \overset{(2)}{=} (0\mathbf{21} \star 0)\mathbf{121} \overset{(1)}{=} 0\mathbf{21121}.$$

Note that during the computation there is no need to convert the values into other, say decimal, notation.

The connection between dyadic concatenation and dyadic size is captured by the following property:

$$x \star y = x2^{|y|_{\mathrm{d}}} + y.$$

This is proved by complete induction on $y$. So take any $y$ and consider three cases. If $y = 0$ then we have

$$x \star 0 = x = x2^0 + 0 = x2^{|0|_{\mathrm{d}}} + 0.$$

If $y = z\mathbf{1}$ for some $z$ then $z < y$ and thus

$$x \star z\mathbf{1} = (x \star z)\mathbf{1} \overset{\mathrm{IH}}{=} (x2^{|z|_{\mathrm{d}}} + z)\mathbf{1} = x2^{|z|_{\mathrm{d}}+1} + z\mathbf{1} = x2^{|z\mathbf{1}|_{\mathrm{d}}} + z\mathbf{1}.$$

The case when $y$ is $z\mathbf{2}$ for $z$ is proved similarly.

**1.1.6 Arguments in favour of declarative programming over natural numbers.** To understand the meaning of an imperative program requires to understand its effect on an entire memory which can be changed during the execution of the program. This impose a great difficulty in reasoning about the correctness of such programs. The main advantage of the imperative programming languages over the declarative ones is efficiency – imperative programs can usually run some times faster as their declarative counterparts.

On the other hand, if we care about the correctness of our programs then the proofs are much easier for declarative programs. For instance, proving that the declarative program from Par. 1.1.2 implements the greatest common divisor can be done within the elementary arithmetic.

In the above examples the domain of values over which the functions operated was the domain natural numbers. 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. How do we propose to deal with such a bewildering variety in the seemingly restricted setting of functions over natural numbers? As we have shown already in Par. 1.1.5, the answer lies in the *coding* of data structures into the domain of natural numbers.

A possible objection that the domain N means unpleasant coding of the rich set of data structures used in computer programming is answered by coding into N in the style of LISP with a pairing function (instead of *cons*). We obtain a degree of comfort as it is known from declarative programming languages. The examples in Chaps. 6–7 should convince the reader.

Many modern functional programming languages allowed arbitrary forms of recursive programs. With unrestricted recursion the best we can do is to compute partial functions. To overcome this obstacle, we propose that programs are properties of (total) functions satisfying certain input conditions. We do not insist that programs have to be definitions.

Each program $P$ is a property of some function $f$ which can be used as a computational rule to calculate this function. The program $P$ has assigned a precondition describing which elements can be used as inputs. Regularity conditions for the program $P$ guarantee that computation terminates for every input $\vec{x}$ which satisfies its precondition yielding the correct value $f(\vec{x})$. for other inputs computation might return wrong answer or even diverge. This will be proved in Sect. 4.5.

## 1.2 Recursive Functions

**1.2.1 Introduction.** In this section we review some basic facts about two (sub-)classes of effectively computable functions: primitive recursive (p.r.) functions and μ-recursive functions (see also [21, 41, 43, 54]). The section contains only basic development. We will show, for instance, that many well-known arithmetic functions and predicates are primitive recursive. We will also show some simple closure properties of both classes (closure under explicit definitions or bounded minimalization). Further investigation is the object of study of the next chapters.

**1.2.2 Basic primitive recursive functions.** The *zero* function $Z$ is such that $Z(x) = 0$; the *successor* function $S$ satisfies the equation $S(x) = x + 1$. For every $n \geq 1$ and $1 \leq i \leq n$, the $n$-ary *identity* function $I^n_i$ yields its $i$-th argument, i.e. we have

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

We usually write $I$ instead of $I^1_1$ and we have $I(x) = x$.

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

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

**1.2.4 Primitive recursion.** For every $n \geq 1$, the operator of *primitive recursion* takes an $n$-ary function $g$ and an $(n{+}2)$-ary function $h$ and yields an $(n{+}1)$-ary function $f$ such that

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

The first argument is the *recursive argument* whereas the remaining arguments are *parameters.* Note that the definition has at least one parameter.

**1.2.5 Primitive recursive functions.** A sequence of functions $f_1, \ldots, f_k$ is called a *primitive recursive derivation* of a function $f$ if

(i) $f = f_k$,
(ii) for every $i$ such that $1 \leq i \leq k$, the function $f_i$ is either one of the basic primitive recursive functions or is obtained from some of the previous functions $f_1, \ldots, f_{i-1}$ by composition or primitive recursion.

A function is primitive recursive if it has a primitive recursive derivation. A predicate is primitive recursive if its characteristic function is.

A class of functions is *primitively recursively closed* if it contains all basic primitive recursive functions and it is closed under composition and primitive recursion. It is easy to see that the class of primitive recursive functions is the smallest primitively recursively closed class of functions. Note that the properties of the class of primitive recursive functions discussed in the next paragraphs depend only on the fact that the class is primitively recursively closed. We will use this observation in Thm. 1.2.33.

**1.2.6 Constant functions are primitive recursive.** We first show, by induction on $m$, that every unary constant function $C_m(x) = m$ is primitive recursive. In the base case we have $C_0 = Z$ is one of the basic p.r. functions. In the induction step we assume that $C_m$ is primitive recursive by IH and define $C_{m+1}$ as primitive recursive by unary composition:

$$C_{m+1}(x) = S\,C_m(x).$$

The $n$-ary constant function $C^n_m(\vec{x}) = m$ is obtained as primitive recursive by the following composition:

$$C^n_m(x_1,\dots,x_n) = C_m\,I^n_1(x_1,\dots,x_n).$$

**1.2.7 Explicit definitions of functions.** Every explicit definition

$$f(x_1,\dots,x_n) = \tau[x_1,\dots,x_n]$$

can be viewed as a function operator which takes all functions applied in the term $\tau$ and returns as a result the function $f$ satisfying the identity. We suppose here that the term $\tau$ does not apply the symbol $f$ and that all its free variables are among the indicated ones.

**1.2.8 Theorem** *Primitive recursive functions are closed under explicit definitions of functions.*

*Proof.* By induction on the structure of terms $\tau$ we prove that primitive recursive functions are closed under explicit definitions of $n$-ary functions:

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

If $\tau \equiv x_i$ then the function $f$ is the $n$-ary identity function $I^n_i$ which is one of the basic primitive recursive functions.

If $\tau \equiv m$ then the function $f$ is the $n$-ary constant function $C^n_m$ which is primitive recursive by Par. 1.2.6.

If $\tau \equiv h(\rho_1,\dots,\rho_m)$, where $h$ is an $m$-ary primitive recursive function, then the $n$-ary functions $g_1,\dots,g_m$ defined explicitly by

$$g_1(\vec{x}) = \rho_1[\vec{x}] \qquad \dots \qquad g_m(\vec{x}) = \rho_m[\vec{x}]$$

are primitive recursive by IH. The function $f$ is obtained as primitive recursive by the following composition

$$f(\vec{x}) = h(g_1(\vec{x}), \dots, g_m(\vec{x})). \qquad \square$$

**1.2.9 Primitive recursive definitions.** Let $\tau_1[\vec{y}, \vec{z}]$ and $\tau_2[\vec{y}, x, a, \vec{z}]$ be terms containing at most the indicated variables free and neither of them applies the function symbol $f$. Then the functional equations

$$\begin{aligned} f(\vec{y}, 0, \vec{z}) &= \tau_1[\vec{y}, \vec{z}] \\ f(\vec{y}, x+1, \vec{z}) &= \tau_2[\vec{y}, x, f(\vec{y}, x, \vec{z}), \vec{z}] \end{aligned}$$

has a unique solution $f$. The definition is called *primitive recursive definition* of $f$. The definition can be viewed as a function operator which takes all functions applied in the terms $\tau_1$ and $\tau_2$ and yields the function $f$ as a result. Note that we do not exclude the case when the parameters $\vec{y}$ or $\vec{z}$ or both are empty. Also the variable $a$ does not have to occur freely in the term $\tau_2$.

*Example.* Note that the operator of *iteration of unary function* is a special case of primitive recursive definitions. The operator takes a unary function $f$ and yields a binary function $f^n(x)$ satisfying:

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

The function $f^n(x)$ is called the *iteration of $f$*. As a simple corollary of the next theorem we obtain that primitive recursive functions are closed also under iteration of unary functions.

**1.2.10 Theorem** *Primitive recursive functions are closed under primitive recursive definitions.*

*Proof.* Let $f$ be defined by the primitive recursive definition as in Par. 1.2.9 from p.r. functions. First we define explicitly two auxiliary functions

$$\begin{aligned} g(w, \vec{y}, \vec{z}) &= \tau_1[\vec{y}, \vec{z}] \\ h(x, a, w, \vec{y}, \vec{z}) &= \tau_2[\vec{y}, x, a, \vec{z}], \end{aligned}$$

which are primitive recursive by Thm. 1.2.8. Next we define a p.r. function $f_1$ by primitive recursion (note that we have at least one parameter!):

$$\begin{aligned} f_1(0, w, \vec{y}, \vec{z}) &= g(w, \vec{y}, \vec{z}) \\ f_1(x+1, w, \vec{y}, \vec{z}) &= h(x, f_1(x, w, \vec{y}, \vec{z}), w, \vec{y}, \vec{z}). \end{aligned}$$

We derive $f$ as primitive recursive by the following explicit definition

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

**1.2.11 Addition is primitive recursive.** The addition function $x + y$ is a p.r. function by the following primitive recursive definition:

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

Note that we have $x + y = S^x(y) = S^y(x)$.

**1.2.12 Multiplication is primitive recursive.** The multiplication function $x \times y$ is a p.r. function by the following primitive recursive definition:

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

**1.2.13 Exponentiation is primitive recursive.** The exponentiation function $x^y$ is a p.r. function by the following primitive recursive definition:

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= x x^y. \end{aligned}$$

**1.2.14 Summation function.** The summation function $\sum_{i=0}^{n} i$ is a p.r. function by the following primitive recursive definition:

$$\begin{aligned} \sum_{i=0}^{0} i &= 0 \\ \sum_{i=0}^{n+1} i &= \sum_{i=0}^{n} i + n + 1. \end{aligned}$$

This is an example of *parameterless* primitive recursive definition.

**1.2.15 Predecessor function is primitive recursive.** The unary predecessor function $x \dot{-} 1$ is defined by the following *explicit definition with monadic discrimination on* $x$:

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

The definition has a form of *parameterless* primitive recursive definition, where the term on the right hand side of the second identity is without any recursive application. Hence the predecessor function is primitive recursive.

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

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

Note that the last occurrence of the symbol $\dot{-}$ in the second equation belongs to the application of the predecessor function. Note also that we have $x \dot{-} y = P^y(x)$, where $P(y) = y \dot{-} 1$.

**1.2.17 Case discrimination function is primitive recursive.** The case discrimination function $D$ is defined by

$$D(x,y,z) = v \leftrightarrow x \neq 0 \land v = y \lor x = 0 \land v = z.$$

The function is primitive recursive by the following explicit definition which uses monadic discrimination on the first argument:

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

**1.2.18 Equality predicate is primitive recursive.** The characteristic function $x =_* y$ of the equality predicate $x = y$ is primitive recursive by the following explicit definition:

$$(x =_* y) = D(x \dot{-} y + (y \dot{-} x), 0, 1).$$

This is because we have $x = y \leftrightarrow x \dot{-} y + (y \dot{-} x) = 0$.

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

$$f(x,\vec{y}) = \begin{cases} \text{the least } z \leq x \text{ s.t. } g(z,\vec{y}) = 1 \text{ holds} & \text{if } \exists z \leq x\, g(z,\vec{y}) = 1; \\ 0 & \text{if there is no such number.} \end{cases}$$

This is usually abbreviated to

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

**1.2.20 Theorem** *Primitive recursive functions are closed under the operator of bounded minimalization.*

*Proof.* Suppose that $f$ is obtained by the bounded minimalization

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

of a primitive recursive function $g$. Clearly we have

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

We derive $f$ as a p.r. function by the following primitive recursive definition:

$$\begin{aligned}
f(0,\vec{y}) &= 0\\
f(x+1,\vec{y}) &= D\big(g(f(x,\vec{y}),\vec{y}) =_* 1, f(x,\vec{y}), D(g(x+1,\vec{y}) =_* 1, x+1, 0)\big). \qquad \square
\end{aligned}$$

**1.2.21 Boolean functions are primitive recursive.** The *boolean* functions are defined by

$$\begin{aligned}
(\neg_* x) = y &\leftrightarrow x \neq 0 \land y = 0 \lor x = 0 \land y = 1\\
(x \land_* y) = z &\leftrightarrow x \neq 0 \land y \neq 0 \land z = 1 \lor (x = 0 \lor y = 0) \land z = 0\\
(x \lor_* y) = z &\leftrightarrow (x \neq 0 \lor y \neq 0) \land z = 1 \lor x = 0 \land y = 0 \land z = 0\\
(x \to_* y) = z &\leftrightarrow (x = 0 \lor y \neq 0) \land z = 1 \lor x \neq 0 \land y = 0 \land z = 0\\
(x \leftrightarrow_* y) = z &\leftrightarrow x \neq 0 \land y \neq 0 \land z = 1 \lor x = 0 \land y = 0 \land z = 1 \lor\\
&\qquad x \neq 0 \land y = 0 \land z = 0 \lor x = 0 \land y \neq 0 \land z = 0.
\end{aligned}$$

Note that we identify non-zero values with truth and 0 with falsehood.

The functions are primitive recursive by the following explicit definitions:

$$\begin{aligned}
(\neg_* x) &= D(x, 0, 1)\\
(x \land_* y) &= D(x, D(y, 1, 0), 0)\\
(x \lor_* y) &= \big(\neg_*(\neg_* x \land_* \neg_* y)\big)\\
(x \to_* y) &= (\neg_* x \lor_* y)\\
(x \leftrightarrow_* y) &= \big((x \to_* y) \land_* (y \to_* x)\big).
\end{aligned}$$

**1.2.22 Formulas with bounded quantifiers.** *Bounded quantifiers* are formulas of the form $\forall x \leq \tau\, \varphi$ and $\exists x \leq \tau\, \varphi$, where the variable $x$ is not free in $\tau$. The bounded quantifiers abbreviate the formulas $\forall x(x \leq \tau \to \varphi)$ and $\exists x(x \leq \tau \land \varphi)$, respectively. *Strict* bounded quantifiers $\forall x < \tau\, \varphi$ and $\exists x < \tau\, \varphi$ are defined similarly.

*Bounded formulas* are formulas which are built from atomic formulas by propositional connectives and bounded quantifiers.

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

$$P(x_1, \ldots, x_n) \leftrightarrow \varphi[x_1, \ldots, x_n],$$

where $\varphi$ is a bounded formula with at most the indicated $n$-tuple of variables free and without any application of the predicate symbol $P$.

Every such definition can be viewed as a function operator which takes all functions occurring in the formula $\varphi$ (this also includes the characteristic functions of every predicate occurring in $\varphi$) and which yields as a result the characteristic function $P_*$ of the predicate $P$.

**1.2.24 Theorem** *Primitive recursive predicates are closed under explicit definitions of predicates with bounded formulas.*

*Proof.* We show that the class of primitive recursive predicates is closed under explicit definitions $P(\vec{x}) \leftrightarrow \varphi[\vec{x}]$ of $n$-ary predicates by induction on the structure of bounded formulas $\varphi$.

If $\varphi \equiv \tau = \rho$ then the characteristic function $P_*$ of $P$ is primitive recursive by the following explicit definition: $P_*(\vec{x}) = (\tau[\vec{x}] =_* \rho[\vec{x}])$.

If $\varphi \equiv R(\vec{\tau})$ then, since $R_*$ is primitive recursive, we define $P_*$ as primitive recursive by explicit definition: $P_*(\vec{x}) = R_*(\vec{\tau}[\vec{x}])$.

If $\varphi \equiv \neg\psi$ then we use IH and define an $n$-ary p.r. predicate $R$ by explicit definition: $R(\vec{x}) \leftrightarrow \psi[\vec{x}]$. Now we define $P_*$ as primitive recursive by the following explicit definition: $P_*(\vec{x}) = (\neg_* R_*(\vec{x}))$.

If $\varphi \equiv \psi \land \chi$ then we obtain as primitive recursive two auxiliary $n$-ary predicates $R(\vec{x}) \leftrightarrow \psi[\vec{x}]$ and $Q(\vec{x}) \leftrightarrow \chi[\vec{x}]$ by IH. We define $P_*$ as primitive recursive by explicit definition: $P_*(\vec{x}) = (R_*(\vec{x}) \land_* Q_*(\vec{x}))$.

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

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

The characteristic function $P_*$ of the predicate $P$ has the following explicit definition: $P_*(\vec{x}) = R_*\big(f(\tau[\vec{x}], \vec{x}), \vec{x}\big)$ as a p.r. function.

The remaining cases are treated similarly. □

**1.2.25 Comparison predicates are primitive recursive.** The standard comparison predicates are primitive recursive by explicit definitions:

$$x \leq y \leftrightarrow \exists z \leq y\, x = z \qquad\qquad x \geq y \leftrightarrow y \leq x$$
$$x < y \leftrightarrow y \nleq x \qquad\qquad x > y \leftrightarrow y < x.$$

**1.2.26 Divisibility is primitive recursive.** The binary divisibility predicate $x \mid y$ is a p.r. predicate by the following explicit definition:

$$x \mid y \leftrightarrow \exists z \leq y\, y = xz.$$

**1.2.27 Definitions by bounded minimalization.** Definitions of functions by *bounded minimalization* are of the form

$$f(\vec{x}) = \begin{cases} \text{the least } y \le \tau[\vec{x}] \text{ s.t. } \varphi[\vec{x}, y] \text{ holds} & \text{if } \exists y \le \tau[\vec{x}]\, \varphi[\vec{x}, y]; \\ 0 & \text{if there is no such number.} \end{cases}$$

Here $\tau[\vec{x}]$ is a term and $\varphi[\vec{x}, y]$ a bounded formula with at most the indicated variables free, both without any application of the symbol $f$. Every such definition can be viewed as a function operator taking all functions and the characteristic functions of all predicates occurring in either the term $\tau$ or formula $\varphi$ and yielding the function $f$.

In the sequel we abbreviate the definition to

$$f(\vec{x}) = \mu y \le \tau[\vec{x}]\big[\varphi[\vec{x}, y]\big].$$

We permit also strict bounds in definitions by bounded minimalization; i.e. we allow definitions of the form

$$f(\vec{x}) = \mu y < \tau[\vec{x}]\big[\varphi[\vec{x}, y]\big]$$

as abbreviation for $f(\vec{x}) = \mu y \le \tau[\vec{x}]\big[y < \tau[\vec{x}] \land \varphi[\vec{x}, y]\big]$.

**1.2.28 Theorem** *Primitive recursive functions are closed under definitions of functions with bounded minimalization.*

*Proof.* Consider an $n$-ary function $f$ defined by the bounded minimalization

$$f(\vec{x}) = \mu y \le \tau[\vec{x}]\big[\varphi[\vec{x}, y]\big]$$

from primitive recursive functions and predicates. We can define $f$ by the following series of definitions:

$$\begin{aligned} P(y, \vec{x}) &\leftrightarrow \varphi[\vec{x}, y] \\ g(z, \vec{x}) &= \mu y \le z[P_*(y, \vec{x}) = 1] \\ f(\vec{x}) &= g(\tau[\vec{x}], \vec{x}). \end{aligned}$$

By Thm. 1.2.24 and Thm. 1.2.20, the characteristic function $P_*$ of $P$ and the auxiliary function $g$ are primitive recursive, and so is the function $f$. □

**1.2.29 Integer division is primitive recursive.** The integer division function $x \div y$ is a p.r. function by the following bounded minimalization:

$$x \div y = \mu q \le x[x < (q+1)y].$$

**1.2.30 Remainder is primitive recursive.** The binary remainder function $x \bmod y$ is a p.r. function by the following explicit definition:

$$x \bmod y = D(y, x \dot{-} (x \div y)y, 0).$$

**1.2.31 Regular minimalization.** For every $n \geq 1$, the operator of *regular minimalization* takes an $(n{+}1)$-ary function $g$ satisfying the following *condition of regularity*:

$$\forall \vec{x} \exists y\, g(y, \vec{x}) = 1$$

and yields an $n$-ary function $f$ such that

$$f(\vec{x}) = \text{the least } y \text{ such that } g(y, \vec{x}) = 1 \text{ holds},$$

This is usually abbreviated to

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

**1.2.32 μ-Recursive functions.** The class of μ-recursive functions is the smallest class of functions containing the zero, successor and identity functions and which is closed under composition, primitive recursion and regular minimalization. A predicate is μ-recursive if its characteristic function is.

**1.2.33 Theorem** *μ-Recursive functions are closed under explicit definitions of functions, primitive recursive definitions, and definitions of functions with bounded minimalization. μ-Recursive predicates are closed under explicit definitions of predicates with bounded formulas.*

*Proof.* It follows from the fact that the class of μ-recursive functions is primitively recursively closed and from the proofs of the corresponding theorems for primitive recursive functions and predicates. □

**1.2.34 Definitions by regular minimalization.** Definitions of functions by *regular minimalization* are of the form

$$f(\vec{x}) = \text{the least } y \text{ such that } \varphi[\vec{x}, y] \text{ holds},$$

where $\varphi[\vec{x}, y]$ is a bounded formula with at most the indicated variables free and without any application of the symbol $f$. Moreover we require that the formula $\varphi$ satisfies the following *condition of regularity*:

$$\forall \vec{x} \exists y \varphi[\vec{x}, y].$$

Every such definition can be viewed as a function operator taking all functions and the characteristic functions of all predicates occurring in the formula $\varphi$ and yielding the function $f$.

In the sequel we will abbreviate the definition to

$$f(\vec{x}) = \mu y\big[\varphi[\vec{x}, y]\big].$$

**1.2.35 Theorem** *μ-Recursive functions are closed under definitions of functions with regular minimalization.*

*Proof.* Consider an $n$-ary function $f$ defined by the bounded minimalization

$$f(\vec{x}) = \mu y\big[\varphi[\vec{x}, y]\big]$$

from μ-recursive functions and predicates. We can define $f$ by the following series of definitions:

$$\begin{aligned} P(y, \vec{x}) &\leftrightarrow \varphi[\vec{x}, y] \\ f(\vec{x}) &= \mu y[P_*(y, \vec{x}) = 1]. \end{aligned}$$

By Par. 1.2.33 the char. function $P_*$ of $P$ is μ-recursive and so is $f$. □

## 1.3 First Order Logic

**1.3.1 Introduction.** In order to establish the terminology we start with a quick review of the standard syntactic and semantic notions for the first order logic and theories. The reader may refer for details to [44, 2].

**1.3.2 First order languages.** A first order language $\mathcal{L}$ is given by denumerable (finite or infinite) set of *non-logical* symbols, which are function and predicate symbols. This includes constants and *propositional variables* as they have the zero arity. Terms and formulas of $\mathcal{L}$ are formed in usual way.

The first order language $\mathcal{L}_2$ is an *extension* of the first language $\mathcal{L}_1$ if every nonlogical symbol of $\mathcal{L}_1$ is a nonlogical symbol of $\mathcal{L}_2$.

**1.3.3 Semantics.** A *first order structure* $\mathcal{M}$ for a first order language $\mathcal{L}$ consists of a non-empty *domain* together with an *interpretation* of function and predicate symbols of $\mathcal{L}$. The *denotation* $\tau^{\mathcal{M}}$ of closed terms and the notion *a formula* $\varphi$ *of* $\mathcal{L}$ *is true in the structure* $\mathcal{M}$, written as $\mathcal{M} \models \varphi$, is defined in the usual way.

A structure $\mathcal{M}$ is a *model* of a set of formulas $T$ if every formula of $T$ is true in $\mathcal{M}$. A formula $\varphi$ is a *logical consequence* of a set of formulas $T$, written as $T \models \varphi$, if it is true in every model of $T$. We abbreviate $\varnothing \models \varphi$ to $\models \varphi$. A formula is *logically valid* if $\models \varphi$, i.e. if it is true in every structure.

Let $\mathcal{M}_1$ be a structure for a first order language $\mathcal{L}_1$ and $\mathcal{L}_2$ an extension of $\mathcal{L}_1$. By adding interpretation of nonlogical symbols of $\mathcal{L}_2 \setminus \mathcal{L}_1$ we obtain a structure $\mathcal{M}_2$ for $\mathcal{L}_2$. We call $\mathcal{M}_2$ an *expansion* of $\mathcal{M}_1$ to $\mathcal{L}_2$.

**1.3.4 Example of a formal system.** We consider here a Hilbert-style *axiom system* H. Such systems are completely specified by their *logical axioms* and *inference rules.* The logical axioms of H are all *(propositional) tautologies*, all *equality axioms*, which are formulas of the form

$$\tau = \tau \qquad \tau = \rho \to \rho = \tau \qquad \tau = \rho \land \rho = \theta \to \tau = \theta$$
$$\tau_1 = \rho_1 \land \cdots \land \tau_n = \rho_n \to f(\tau_1, \dots, \tau_n) = f(\rho_1, \dots, \rho_n)$$
$$\tau_1 = \rho_1 \land \cdots \land \tau_n = \rho_n \to P(\tau_1, \dots, \tau_n) \to P(\rho_1, \dots, \rho_n),$$

and all *quantifier axioms*, which are formulas of the form

$$\forall x \varphi[x] \to \varphi[\tau] \qquad \varphi[\tau] \to \exists x \varphi[x].$$

Its inference rules are *modus ponens*, *generalization rules*, and *axiom rules* introducing formulas from a fixed set of formulas $T$ (listed in that order):

$$\frac{\varphi \to \psi \quad \varphi}{\psi} \qquad \frac{\varphi \to \psi[x]}{\varphi \to \forall x \psi[x]} \qquad \frac{\psi[x] \to \varphi}{\exists x \psi[x] \to \varphi} \qquad \frac{}{\varphi}\ \varphi \in T.$$

We suppose here that the variable $x$ is not free in $\varphi$. Formulas of the set $T$ are usually referred as *non-logical axioms* of the system H.

By a *proof* of a formula $\psi$ from a set of formulas $T$ in the system H we mean a sequence of formulas $\psi_1, \dots, \psi_n$ such that

(i) $\psi \equiv \psi_n$,
(ii) for every $i$ such that $1 \le i \le n$, the formula $\psi_i$ is either one of its logical axioms or it is a conclusion of one of its inference rules obtained from some of the previous formulas $\psi_1, \dots, \psi_{i-1}$ or it is a formula from $T$.

We say $\psi$ is *provable* in $T$ if there is a proof of $\psi$ from the set $T$.

**1.3.5 Formal systems and provability.** We write $T \vdash \varphi$ if the formula $\varphi$ is provable from the set of formulas $T$; we write $\vdash \varphi$ if the set $T$ is empty. To demonstrate provability, we will usually use natural language as a formal system. However, the reader is invite to use his own favourite formal system satisfying the following theorem; for instance, the Hilbert system H from the previous paragraph is one of such formal systems.

**1.3.6 Theorem** *$T \vdash \varphi$ if and only if $T \vDash \varphi$.*

**1.3.7 First order theories.** A first order theory $T$ of a first order language $\mathcal{L}$ is any set of formulas of $\mathcal{L}$. Formulas from the set $T$ are called the *axioms* of the theory $T$. We shall designate the language of the theory $T$ by $\mathcal{L}_T$. A formula $\varphi$ of $\mathcal{L}_T$ is called a *theorem* of $T$ if $T \vdash \varphi$. The theory $T$ is *consistent* if it does not prove contradiction, i.e. if $\bot$ is not a theorem of $T$. By Thm. 1.3.6, a theory is consistent iff it has a model.

We will write $T \vdash T'$ if every axiom of $T'$ is a theorem of $T$. A theory $T'$ is an *extension* of a theory $T$ if $\mathcal{L}_{T'}$ is an extension of $\mathcal{L}_T$ and every theorem of $T$ is also theorem of $T'$. A *conservative extension* of $T$ is an extension $T'$ of $T$ such that every formula of $\mathcal{L}_T$ which is a theorem of $T'$ is also a theorem of $T$. The theories $T$ and $T'$ are *equivalent* if they have the same language and the same theorems, i.e. if we have $T \vdash T'$ and $T' \vdash T$.

**1.3.8 Explicit definitions of predicates.** Let $T$ be a theory and $\varphi[\vec{x}]$ a formula of $\mathcal{L}_T$ with all its free variables among the $n$ indicated ones. Consider the theory $T'$ obtained from $T$ by adding a new $n$-ary predicate symbol $P$ and the defining axiom:

$$P(\vec{x}) \leftrightarrow \varphi[\vec{x}].$$

We say that $T'$ is an *extension of $T$ by explicit definition of a predicate.*

Given a formula $\psi$ of $\mathcal{L}_{T'}$, we designate by $\psi^\star$ a formula of $\mathcal{L}_T$ obtained from $\psi$ by replacing in it every application $P(\vec{\tau})$ by $\varphi[\vec{\tau}]$. The formula $\psi^\star$ is called the *translation of $\psi$ into $T$*. We clearly have

$$T' \vdash \psi \leftrightarrow \psi^\star. \tag{1}$$

**1.3.9 Theorem** *We have*

*(i)* $T' \vdash \psi$ *if and only if* $T \vdash \psi^\star$,
*(ii)* $T'$ *is a conservative extension of* $T$,
*(iii)* *every model of* $T$ *has a unique expansion to a model of* $T'$.

*Proof.* (i): We prove the direction ($\Rightarrow$) first; this will be demonstrated for the Hilbert formal system H (see Par. 1.3.4). Suppose that $\psi_1, \dots, \psi_m$ is a proof of the formula $\psi$ in the theory $T'$. By complete induction on $i$ we prove that every formula $\psi_i^\star$ is a theorem of $T$; consequently $\psi^\star \equiv \psi_n^\star$ is a theorem of $T$.

If $\psi_i$ is a logical axiom then there are two subcases to consider. If $\psi_i$ is the equality axiom of a form

$$\tau_1 = \rho_1 \land \dots \land \tau_n = \rho_n \to P(\tau_1, \dots, \tau_n) \to P(\rho_1, \dots, \rho_n)$$

then its translation $\psi_i^\star$ is a logically valid formula

$$\tau_1 = \rho_1 \land \dots \land \tau_n = \rho_n \to \varphi[\tau_1, \dots, \tau_n] \to \varphi[\rho_1, \dots, \rho_n].$$

If this is not the case then the formula $\psi_i^\star$ is a logical axiom of the same kind and hence trivially provable in $T$.

If $\psi_i$ is a conclusion of an axiom rule introducing the formula

$$\forall \vec{x}(P(\vec{x}) \leftrightarrow \varphi[\vec{x}])$$

then its translation $\psi_i^\star$ is a logically valid formula

$$\forall \vec{x}(\varphi[\vec{x}] \leftrightarrow \varphi[\vec{x}]).$$

If the formula $\psi_i$ is inferred from $\psi_j$ and $\psi_k \equiv \psi_j \to \psi_i$ by modus ponens then its translation $\psi_i^\star$ is inferred from $\psi_j^\star$ and $\psi_k^\star \equiv \psi_j^\star \to \psi_i^\star$ by the same rule. By

IH, the formulas $\psi_j^\star$ and $\psi_k^\star$ are theorems of $T$ and so is $\psi_i^\star$. The remaining inferences rules are treated similarly.

In the proof of the reverse direction ($\Leftarrow$)-direction assume that the formula $\psi^\star$ is a theorem of $T$. The theory $T'$ is an extension of $T$ and thus $\psi^\star$ is a theorem of $T'$ as well. We apply 1.3.8(1) to obtain that $\psi$ is a theorem of $T'$.

(ii): Let $\psi$ be a formula of $\mathcal{L}_T$ which is a theorem of the theory $T'$. By (i), its translation $\psi^\star$ is a theorem of $T$. But $\psi^\star \equiv \psi$ and we are done.

(iii): Obvious. □

**1.3.10 Skolem axioms.** Let $T$ be a theory whose language does not contain the $n$-ary function symbol $f$. Let further $\varphi[\vec{x}, y]$ be a formula of $\mathcal{L}_T$ with all its free variables among the $n+1$ indicated ones. The formula

$$\exists y\, \varphi[\vec{x}, y] \to \varphi[\vec{x}, f(\vec{x})] \tag{1}$$

is called the *Skolem axiom for $\varphi$ and $f$.*

Extension of $T$ to $T'$ by the addition of the symbol $f$ to its language and of the axiom (1) to its axioms is called a *Skolem extension of $T$*. The next theorem shows that $T'$ is a conservative extension of $T$.

**1.3.11 Theorem** *Skolem extensions are conservative extensions.*

*Proof.* A model-theoretic proof is easy. A syntactic proof is much harder; the reader is invited to consult Shoenfield's book [44]. For our contribution in this topic, see the proof in [28], which is based on the ideas developed in [22, 23]. □

**1.3.12 Contextual definitions of functions.** Let $T$ be a theory whose language does not contain the $n$-ary function symbol $f$. Let further $\varphi[\vec{x}, y]$ be a formula of $\mathcal{L}_T$ in which no other variable than the $n+1$ indicated ones is free. Suppose that $T$ proves the *existence and uniqueness conditions for $\varphi$*:

$$T \vdash \forall \vec{x} \exists y\, \varphi[\vec{x}, y] \qquad T \vdash \varphi[\vec{x}, y_1] \land \varphi[\vec{x}, y_2] \to y_1 = y_2.$$

Let $T'$ be the theory obtained from $T$ by adding $f$ and the defining axiom

$$f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y].$$

We say that $T'$ is an *extension of $T$ by contextual definition of $f$.*

Given a formula $\psi$ of $\mathcal{L}_{T'}$, we designate by $\psi^\star$ a formula of $\mathcal{L}_T$ obtained from $\psi$ by replacing in it each its subformula $\psi_1[f(\vec{\tau})]$ by the formula $\exists z(\varphi[\vec{\tau}, z] \land \psi_1[z])$ (or, alternatively, by the formula $\forall z(\varphi[\vec{\tau}, z] \to \psi_1[z])$). Here, the $z$ is a new variable. The formula $\psi^\star$ is called the *translation of $\psi$ into $T$*. We clearly have

$$T' \vdash \psi \leftrightarrow \psi^\star. \tag{1}$$

Note also that the following formula is always provable

$$\vdash \forall y(f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y]) \leftrightarrow \varphi[\vec{x}, f(\vec{x})] \land \forall y(\varphi[\vec{x}, y] \rightarrow f(\vec{x}) = y).$$

In the sequel we will use this observation without referring to it.

**1.3.13 Theorem** *We have*

*(i)* $T' \vdash \psi$ *if and only if* $T \vdash \psi^\star$,
*(ii)* $T'$ *is a conservative extension of* $T$,
*(iii)* *every model of* $T$ *has a unique expansion to a model of* $T'$.

*Proof.* (ii): Consider the extension $T_f$ of $T$ by adding the function symbol $f$ and nothing else. From the uniqueness condition for $\varphi$ we get

$$T_f \vdash \forall y(f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y]) \leftrightarrow \varphi[\vec{x}, f(\vec{x})].$$

On the other hand, the existence condition for $\varphi$ yields

$$T_f \vdash (\exists y\, \varphi[\vec{x}, y] \rightarrow \varphi[\vec{x}, f(\vec{x})]) \leftrightarrow \varphi[\vec{x}, f(\vec{x})].$$

By combining these facts together we can see that

$$T_f \vdash \forall y(f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y]) \leftrightarrow (\exists y\, \varphi[\vec{x}, y] \rightarrow \varphi[\vec{x}, f(\vec{x})]).$$

Thus $T'$ is equivalent to the Skolem extension $T''$ of $T$ with the axiom:

$$\exists y\, \varphi[\vec{x}, y] \rightarrow \varphi[\vec{x}, f(\vec{x})].$$

By Thm. 1.3.11, the theory $T''$ is conservative over $T$ and so is $T'$.

(i): For every formula $\psi$ of $\mathcal{L}_{T'}$, we have $T' \vdash \psi$ iff, by 1.3.12(1), $T' \vdash \psi^\star$ iff $T \vdash \psi^\star$ since $T'$ is conservative over $T$.

(iii): Obvious. □

**1.3.14 Extensions by definitions.** We say that a theory $T'$ is an *extension by definition* of a theory $T$ if the theory $T'$ is equivalent to a theory obtained from $T$ either by an explicit definition of a predicate or by a contextual definition of a function.

A theory $T'$ is an *extension by definitions* of a theory $T$ if $T'$ is obtained from $T$ by a finite number of extensions by definition.

**1.3.15 Theorem** *If* $T'$ *is an extension by definitions of* $T$ *then*

*(i)* *there is an effective translation of the formulas* $\psi$ *of* $\mathcal{L}_{T'}$ *to the formulas* $\psi^\star$ *of* $\mathcal{L}_T$ *such that for every formula* $\psi$ *of* $\mathcal{L}_{T'}$ *we have*

$$T' \vdash \psi \text{ if and only if } T \vdash \psi^\star,$$

*(ii)* $T'$ *is a conservative extension of* $T$,

*(iii)* *every model of $T$ has a unique expansion to a model of $T'$.*

*Proof.* It follows directly from Thm. 1.3.9 and Thm. 1.3.13. □

**1.3.16 Implicit definitions of functions.** Let $T$ be a theory and $\varphi[\vec{x}, y]$ a formula of $\mathcal{L}_T$ as in Par. 1.3.12. Let $T'$ be the theory obtained from $T$ by adding a new $n$-ary function symbol $f$ and the defining axiom

$$\varphi[\vec{x}, f(\vec{x})].$$

We say that $T'$ is an *extension of $T$ by implicit definition of a function.*

**1.3.17 Theorem** *Implicit definitions of functions are extensions by definitions.*

*Proof.* Let $T'$ be the extension of the theory $T$ by the implicit definition of $f$ as in the previous paragraph. It suffices to show that the theory $T'$ is equivalent to the extension $T''$ of $T$ by the contextual definition:

$$f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y].$$

Let us denote by $T_f$ the extension of $T$ by adding the function symbol $f$. From the uniqueness condition for $\varphi$ we get

$$T_f \vdash \forall y(f(\vec{x}) = y \leftrightarrow \varphi[\vec{x}, y]) \leftrightarrow \varphi[\vec{x}, f(\vec{x})]$$

from which the equivalence of $T'$ and $T''$ follows immediately. □

**1.3.18 Explicit definitions of functions.** Let $\tau[\vec{x}]$ be a term of a theory $T$ in which no other variables than the $n$ indicated ones are free. Let $T'$ be the theory obtained from the theory $T$ by adding a new $n$-ary function symbol $f$ and the defining axiom

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

We say that $T'$ is an *extension of $T$ by explicit definition of a function.*

**1.3.19 Theorem** *Explicit definitions of functions are extensions by definitions.*

*Proof.* Let $T'$ be the extension of $T$ by the explicit definition of $f$ as in the previous paragraph and $T''$ an extension of $T$ by the contextual definition

$$f(\vec{x}) = y \leftrightarrow y = \tau[\vec{x}],$$

whose existence and uniqueness conditions

$$\vdash \forall \vec{x} \exists y\, y = \tau[\vec{x}] \qquad \vdash y_1 = \tau[\vec{x}] \land y_2 = \tau[\vec{x}] \rightarrow y_1 = y_2$$

are always provable. The equivalence of $T'$ and $T''$ follows from

$$\vdash f(\vec{x}) = \tau[\vec{x}] \leftrightarrow \forall y(f(\vec{x}) = y \leftrightarrow y = \tau[\vec{x}]).$$

But $T''$ is an extension of $T$ by definition and so is $T'$. □

## 1.4 Peano Arithmetic

**1.4.1 Introduction.** In this section we introduce the classical first order axiomatic system for natural numbers called *Peano arithmetic* (PA). In the following discussion we state many properties of PA without proofs. The reader is advised to consult [16] for details.

**1.4.2 Peano arithmetic.** The language $\mathcal{L}_{\mathrm{PA}}$ consists of the constant 0, the unary function symbol $S$, two binary function symbols $+$ and $\times$, and two binary predicate symbols $\leq$ and $<$. We will use standard conventions in writing terms and formulas of PA. We use infix notation for binary symbols; e.g. we write $x + y$ instead of $+(x, y)$, similarly for $\times$, $\leq$ and $<$.

We denote by $\mathcal{N}$ the intended interpretation of the language $\mathcal{L}_{\mathrm{PA}}$ with the domain of natural numbers N and with the interpretation of its symbols in the above order as the zero number, the successor function (adding one), the addition and multiplication functions, and the non-strict and strict less-than linear order relations.

The axioms of PA consist of the following eight formulas:

$$\begin{aligned} 0 &\neq S(x) \\ S(x) = S(y) &\rightarrow x = y \\ 0 + y &= y \\ S(x) + y &= S(x + y) \\ 0 \times y &= 0 \\ S(x) \times y &= x \times y + y \\ x \leq y &\leftrightarrow \exists z x + z = y \\ x < y &\leftrightarrow \exists z x + S(z) = y \end{aligned}$$

together with all *mathematical induction axioms*

$$\varphi[0] \wedge \forall x(\varphi[x] \rightarrow \varphi[S(x)]) \rightarrow \forall x \varphi[x].$$

The *induction formula* $\varphi[x]$ may contain, in addition to the *induction variable* $x$, zero or more free variables as *parameters*. Clearly, every axiom of PA is true in the intepretation $\mathcal{N}$. The first order structure $\mathcal{N}$ is called the *standard model* of PA. We use the symbol $\vdash_{\mathrm{PA}} \varphi$ of provability in PA.

**1.4.3 Remark.** Many basic facts about addition, multiplication and order relations can be formalize in PA together with their proofs; for example, one can easily prove that addition is commutative, associative and distributive over multiplication. In the sequel, we will not explicitly refer to the properties of the basic arithmetic functions and relations of PA. We mention here only one property, which is called called *monadic case analysis*:

$$\vdash_{\mathrm{PA}} x = 0 \lor \exists y\, x = S(y).$$

The property can be proved by a straightforward induction on $x$.

**1.4.4 The principle of complete induction.** For every formula $\varphi[x]$, the formula of *complete induction on $x$ for $\varphi$* is the following one:

$$\forall x\big(\forall y(y < x \to \varphi[y]) \to \varphi[x]\big) \to \forall x \varphi[x]. \tag{1}$$

It is assumed here that the variable $y$ is different from the induction variable $x$ and it does not occur freely in $\varphi$. The induction formula $\varphi$ may contain additional variables as parameters.

**1.4.5 Theorem** PA *proves the principle of complete induction for each formula of $\mathcal{L}_{\mathrm{PA}}$.*

*Proof.* The principle of complete induction 1.4.4(1) is reduced to mathematical induction as follows. Under the assumption that $\varphi$ is *progressive*:

$$\forall x\big(\forall y(y < x \to \varphi[y]) \to \varphi[x]\big), \tag{$\dagger_1$}$$

we prove first, by induction on $n$, the following auxiliary property

$$\forall z(z < n \to \varphi[z]). \tag{$\dagger_2$}$$

In the base case there is nothing to prove. In the induction step take any $z < S(n)$ and consider two cases. If $z < n$ then we obtain $\varphi[z]$ by IH. If $z = n$ then by instantiating of $(\dagger_1)$ with $x := z$ we obtain

$$\forall y(y < n \to \varphi[y]) \to \varphi[z].$$

Now we apply IH to get $\varphi[z]$.

With the auxiliary property proved we obtain that $\varphi[x]$ holds for every $x$ by instantiating of $\forall n(\dagger_2)$ with $n := S(x)$ and $z := x$. □

**1.4.6 The least number principle.** For every formula $\varphi[x]$, the formula of *the least number principle for $\varphi$* is the following one:

$$\exists x \varphi[x] \to \exists x\big(\varphi[x] \land \forall y(y < x \to \neg\varphi[y])\big). \tag{1}$$

We assume here that $y$ is different from $x$ and that it does not occur in $\varphi$. The formula $\varphi$ may contain additional variables as parameters.

**1.4.7 Theorem** PA *proves the least number principle for each formula of* $\mathcal{L}_{\mathrm{PA}}$.

*Proof.* The claim follows from Thm. 1.4.5 by noting that the least number principle 1.4.6(1) is logically equivalent to the principle of complete induction for the formula $\neg\varphi[x]$:

$$\forall x\big(\forall y(y < x \to \neg\varphi[y]) \to \neg\varphi[x]\big) \to \forall x\neg\varphi[x]. \qquad \square$$

**1.4.8 Extensions of PA.** Now we consider the problem of introducing new functions and predicates into PA. Since we do not wish the extended theory to be inconsistent we are interested in extensions by definitions of PA. Recall that such extensions are conservative and thus they do not prove any new theorems in the language before the extension. Since the original theory PA is consistent, the same holds for any extension by definitions of PA. Moreover, any argument formally expressed in the extended language can be effectively translated back into the language of PA. New function and predicate symbols introduced by this kind of extensions are thus only a notational convenience which give us expressivity but not power.

For these reasons we keep the notation PA for the current extension by definitions of PA. We will be using the symbol of provability $\vdash_{\mathrm{PA}} \varphi$ in this relativized sense. We will also use the expression "standard model of PA" in the relativized sense to designate the unique expansion of the standard model $\mathcal{N}$ of PA to the model of the current extension of PA. The uniqueness of expansion is guaranteed by Thm. 1.3.15. Only in situations where we will be introducing new schemes of extension of PA we will temporary revert to designating the extensions of PA by symbols $T$, $T'$, etc.

**1.4.9 Theorem** *If $T$ is an extension by definitions of* PA *then it proves the principle of mathematical induction, the principle of complete induction, and the least number principle for each formulas of* $\mathcal{L}_T$.

*Proof.* Consider the following induction axiom for a formula $\varphi$ of $\mathcal{L}_T$:

$$\varphi[0] \land \forall x(\varphi[x] \to \varphi[S(x)]) \to \forall x\varphi[x]. \qquad (\dagger_1)$$

Its translation into PA is the induction axiom for the formula $\varphi^\star$ of $\mathcal{L}_{\mathrm{PA}}$:

$$\varphi^\star[0] \land \forall x(\varphi^\star[x] \to \varphi^\star[S(x)]) \to \forall x\varphi^\star[x] \qquad (\dagger_2)$$

The formula $(\dagger_2)$ is an axiom of PA and hence trivially provable in PA. Therefore, by Thm. 1.3.15, the induction axiom $(\dagger_1)$ is a theorem of $T$.

The other two principles are derived in $T$ similarly. $\square$

**1.4.10 Decimal constants.** In the sequel, we will often use decimal constants in our discussion. For instance, we can introduce the decimal constants 1, 2 and 3 into PA by the explicit definitions $1 = S(0)$, $2 = S\,S(0)$ and $3 = S\,S\,S(0)$. Other decimal constants are defined similarly. With the help of the constant 1 we can rewrite the principle of mathematical induction in a more convenient form:

$$\vdash_{\mathrm{PA}} \varphi[0] \land \forall x(\varphi[x] \to \varphi[x+1]) \to \forall x \varphi[x].$$

**1.4.11 Comparison predicates.** The inverse of the comparison predicates $\leq$ and $<$ are introduced into PA by the following explicit definitions:

$$\begin{aligned} x \geq y &\leftrightarrow y \leq x \\ x > y &\leftrightarrow y < x. \end{aligned}$$

**1.4.12 Modified subtraction.** The modified subtraction function $x \dot{-} y$ is introduced into PA by the following contextual definition:

$$x \dot{-} y = z \leftrightarrow x \leq y \land z = 0 \lor y \leq x \land x = y + z.$$

**1.4.13 Integer division and remainder.** The existence and uniqueness conditions of euclidean division can be easily expressed and proved in PA:

$$\begin{aligned} &\vdash_{\mathrm{PA}} y \neq 0 \to \exists q \exists r(x = qy + r \land r < y) \\ &\vdash_{\mathrm{PA}} r_1 < y \land r_2 < y \land q_1 y + r_1 = q_2 y + r_2 \to q_1 = q_2 \land r_1 = r_2. \end{aligned}$$

The integer division $x \div y$ and remainder function $x \bmod y$ are introduced into PA by the following contextual definitions:

$$\begin{aligned} x \div y = q &\leftrightarrow y = 0 \land q = 0 \lor y \neq 0 \land \exists r(x = qy + r \land r < y) \\ x \bmod y = r &\leftrightarrow y = 0 \land r = 0 \lor y \neq 0 \land \exists q(x = qy + r \land r < y). \end{aligned}$$

**1.4.14 Divisibility predicate.** The binary divisibility predicate $x \mid y$ is introduced into PA explicitly by

$$x \mid y \leftrightarrow \exists z\, y = xz.$$

**1.4.15 Extensions by regular minimalization.** Definitions of functions by regular minimalization (see Par. 1.2.34) are formalized as follows. Let $T$ be an extension by definitions of PA. Let further $\varphi[\vec{x}, y]$ be a formula of $\mathcal{L}_T$ in which no other variables than the $n+1$ indicated ones are free. Suppose that the theory $T$ proves the condition of regularity for $\varphi$:

$$T \vdash \forall \vec{x} \exists y\, \varphi[\vec{x}, y].$$

Let $T'$ be the theory obtained from the theory $T$ by adding a new $n$-ary function symbol $f$ and the defining axiom:

$$\varphi[\vec{x}, f(\vec{x})] \wedge \forall y < f(\vec{x})\, \neg\varphi[\vec{x}, y].$$

We say that $T'$ is an *extension of $T$ by regular minimalization.* As before we abbreviate the above definition by $f(\vec{x}) = \mu y[\varphi[\vec{x}, y]]$.

**1.4.16 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by regular minimalization is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by regular minimalization as in Par. 1.4.15 and $T''$ an extension of $T$ by the implicit definition of $f$

$$\varphi[\vec{x}, f(\vec{x})] \wedge \forall z < f(\vec{x})\, \neg\varphi[\vec{x}, z]$$

for the formula $\varphi[\vec{x}, y] \wedge \forall z < y\, \neg\varphi[\vec{x}, z]$. Since the defining axioms of both extensions are variant of each other, the theories $T'$ and $T''$ are equivalent. By Thm. 1.3.17, $T''$ is conservative over $T$ and so is $T'$.

It remains to check the legality of the implicit definition, i.e. that the theory $T$ proves its existence and uniqueness conditions

$$T \vdash \exists y\big(\varphi[\vec{x}, y] \wedge \forall z < y\, \neg\varphi[\vec{x}, z]\big)$$
$$T \vdash \varphi[\vec{x}, y_1] \wedge \forall z < y_1\, \neg\varphi[\vec{x}, z] \wedge \varphi[\vec{x}, y_2] \wedge \forall z < y_2\, \neg\varphi[\vec{x}, z] \rightarrow y_1 = y_2.$$

The existence condition is a straightforward consequence of the condition of regularity for $\varphi$ and the least number principle

$$T \vdash \exists y\, \varphi[\vec{x}, y] \rightarrow \exists y\big(\varphi[\vec{x}, y] \wedge \forall z(z < y \rightarrow \neg\varphi[\vec{x}, z])\big).$$

In the proof of the uniqueness property, assume its antecedent and consider three cases. If $y_1 < y_2$ then we obtain the contradiction $\neg\varphi[\vec{x}, y_1]$ from the assumption $\forall z < y_2\, \neg\varphi[\vec{x}, z]$. If $y_1 > y_2$ then we derive a similar contradiction. So it must be $y_1 = y_2$. □

**1.4.17 Extensions by bounded minimalization.** Definitions of functions by bounded minimalization (see Par. 1.2.27) are formalized as follows. Let $T$ be an extension by definitions of PA. Let further $\varphi[\vec{x}, y]$ be a formula of $\mathcal{L}_T$ in which no other variables than the $n+1$ indicated ones are free and $\tau[\vec{x}]$ a term of $\mathcal{L}_T$ with all its free variables among the indicated ones.

Suppose that $T'$ is the theory obtained from the theory $T$ by adding a new $n$-ary function symbol $f$ and the defining axiom:

$$f(\vec{x}) \leq \tau[\vec{x}] \wedge \varphi[\vec{x}, f(\vec{x})] \wedge \forall y < f(\vec{x})\, \neg\varphi[\vec{x}, y] \vee \forall y \leq \tau[\vec{x}]\, \neg\varphi[\vec{x}, y] \wedge f(\vec{x}) = 0.$$

We say that $T'$ is an *extension of $T$ by bounded minimalization* As before we abbreviate the above definition by $f(\vec{x}) = \mu_{y \leq \tau}[\varphi]$.

**1.4.18 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by bounded minimalization is an extension by definition.*

*Proof.* Note that the bounded minimalization

$$f(\vec{x}) = \mu y \leq \tau[\vec{x}]\big[\varphi[\vec{x}, y]\big]$$

is equivalent to the regular minimalization

$$f(\vec{x}) = \mu y\big[y \leq \tau[\vec{x}] \land \varphi[\vec{x}, y] \lor \forall z \leq \tau[\vec{x}]\, \neg\varphi[\vec{x}, z] \land y = 0\big],$$

which condition of regularity is always provable. The claim is now a direct consequence of Thm. 1.4.16. □

# Chapter 2
# Beginning of Arithmetization

The most difficult part of the recursive development of PA is the formalization of some kind of *coding of finite sequences.* The encoding is needed to show that various recursion schemes can be formalized within PA. The most famous encoding is via Gödel's *$\beta$-function* and relies on a rather non-trivial Chinese Remainder Theorem. For that reason many authors use a slightly simpler method of coding which is based on the binary representation of N. This approach requires a derivation of exponentiation, which is done achieved the help of a temporary, usually highly non-trivial, coding of finite sets.

In our approach we look for a solution to the programming language Lisp which offers excellent coding of programming data structures into the domain of *S-expressions.* The domain is freely generated from denumerable many *atoms* by a binary operation *cons*. We obtain the coding convenience of Lisp with help of a suitable *pairing function.* Numbers which are not in the range of the pairing function play the role of atoms while the pairing function itself plays the role of the operation *cons*. As we will see later there is no advantage in having infinitely many atoms; just one, say 0, suffices.

The *Cantor pairing function* $J(x,y) = \sum_{i=0}^{x+y} i + x$ is a classical example of pairing function. The function is a bijection and hence there are no atoms. By increasing $J$ by one we obtain the so-called *modified Cantor pairing function* $\langle x,y\rangle = J(x,y)+1$, which has the number 0 as the only atom. This simple modification enables us to use the coding techniques of Lisp with a degree of comfort comparable to that in other functional programming languages. Examples from the second part of this text should convince the reader.

Basic properties of the pairing function $\langle x,y\rangle$ are studied in Sect. 2.1. The next section contains probably the most difficult part of this text: we prove that PA is closed under a restricted form of iteration of unary functions. The formalization of the scheme is based on elementary facts about $p$-ary representation of numbers ($p$ is a prime); essentially no coding is needed for this. The scheme is then used in the last two sections where we investigate the coding of tuples and finite sequences based on the pairing function.

## 2.1 Pairing Function

**2.1.1 Introduction.** In this section we will be studying the properties of the binary function $\langle x, y\rangle$, which is the standard Cantor pairing function when offset by one:

$$\langle x, y\rangle = \sum_{i=0}^{x+y} i + x + 1,$$

Figure 2.1 shows the initial segment of values of this modified pairing function of Cantor in tabular form.

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

**Fig. 2.1** The modified Cantor pairing function

**2.1.2 Summation function.** The summation function

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + \cdots + n$$

can be introduced into PA by the following explicit definition as p.r. function

$$\sum_{i=0}^{n} i = n(n+1) \div 2.$$

**2.1.3 Recurrent properties of the summation function.** We have

$$\vdash_{\mathrm{PA}} \sum_{i=0}^{0} i = 0 \tag{1}$$

$$\vdash_{\mathrm{PA}} \sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + n + 1. \tag{2}$$

In the sequel we will use these properties without explicitly referring to them.

*Proof.* (1): Obvious. (2): It follows from

$$\sum_{i=0}^{n+1} i = (n+1)(n+1+1) \div 2 = \big(n(n+1) + 2(n+1)\big) \div 2 =$$
$$= n(n+1) \div 2 + n + 1 = \sum_{i=0}^{n} i + n + 1. \qquad \square$$

**2.1.4 Ordering properties of the summation function.** We have

$$\vdash_{\mathrm{PA}} k_1 \le n_1 \land k_2 \le n_2 \to \sum_{i=0}^{n_1} i + k_1 \le \sum_{i=0}^{n_2} i + k_2 \leftrightarrow n_1 < n_2 \lor n_1 = n_2 \land k_1 \le k_2 \tag{1}$$

$$\vdash_{\mathrm{PA}} k_1 \le n_1 \land k_2 \le n_2 \to \sum_{i=0}^{n_1} i + k_1 = \sum_{i=0}^{n_2} i + k_2 \leftrightarrow n_1 = n_2 \land k_1 = k_2. \tag{2}$$

*Proof.* First we prove

$$\vdash_{\mathrm{PA}} \forall m(n \le m \to \sum_{i=0}^{n} i \le \sum_{i=0}^{m} i) \tag{$\dagger_1$}$$

by induction on $n$. The base case is obvious. In the induction step take any $m$ such that $n+1 \le m$. Then $m = m_1 + 1$ for some $m_1$ s.t. $n \le m_1$. We obtain

$$\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + n + 1 \overset{\mathrm{IH}}{\le} \sum_{i=0}^{m_1} i + n + 1 \le \sum_{i=0}^{m_1} i + m_1 + 1 = \sum_{i=0}^{m_1+1} i.$$

Note that the induction hypothesis is applied with $m_1$ in place of $m$.

(1): Under the assumptions $k_1 \le n_1$ and $k_2 \le n_2$ we consider three cases. The case when $n_1 = n_2$ is obvious; the case $n_1 < n_2$ follows from ($\dagger_1$). So suppose that $n_1 > n_2$. The desired claim follows from

$$\sum_{i=0}^{n_2} i + k_2 < \sum_{i=0}^{n_2} i + n_2 + 1 = \sum_{i=0}^{n_2+1} i \overset{(\dagger_1)}{\le} \sum_{i=0}^{n_1} i \le \sum_{i=0}^{n_1} i + k_1.$$

(2): It follows from (1). $\square$

**2.1.5 Additional properties of the summation function.** We have

$$\vdash_{\mathrm{PA}} n \le \sum_{i=0}^{n} i \tag{1}$$

$$\vdash_{\mathrm{PA}} \exists n\, x < \sum_{i=0}^{n} i \tag{2}$$

$$\vdash_{\mathrm{PA}} \exists n(\sum_{i=0}^{n} i \le x < \sum_{i=0}^{n+1} i). \tag{3}$$

*Proof.* (1): By a straightforward induction on $n$. (2): By induction on $x$. The base case follows from $0 < 1 = \sum_{i=0}^{1} i$. In the induction step, we obtain from IH that $x < \sum_{i=0}^{n} i$ for some $n$. Now we consider two cases. If $x + 1 < \sum_{i=0}^{n} i$ then we are done. Otherwise we have $x+1 = \sum_{i=0}^{n} i$, but now the claim follows from the inequality $\sum_{i=0}^{n} i < \sum_{i=0}^{n+1} i$.

(3): We need the following auxiliary property

$$\vdash_{\mathrm{PA}} x < \sum_{i=0}^{k} i \rightarrow \exists n(\sum_{i=0}^{n} i \leq x < \sum_{i=0}^{n+1} i), \tag{$\dagger_1$}$$

which is proved by induction on $k$. In the base case there is nothing to prove. In the induction step assume $x < \sum_{i=0}^{k+1} i$ and consider two cases. If $x \geq \sum_{i=0}^{k} i$ then it suffices to take $k$ for $n$. Otherwise we have $x < \sum_{i=0}^{k} i$ and now the claim follows directly from IH.

With the auxiliary property ($\dagger_1$) proved we obtain (3) from (2). □

**2.1.6 Modified Cantor pairing function.** The *modified Cantor pairing function* is introduced into PA by the next explicit definition as a p.r. function

$$\langle x, y\rangle = \sum_{i=0}^{x+y} i + x + 1.$$

From now on by a pairing function we always mean the modified Cantor pairing function $\langle x, y\rangle$.

**2.1.7 Basic properties of the pairing function.** We have

$$\vdash_{\mathrm{PA}} \langle x_1, x_2\rangle = \langle y_1, y_2\rangle \rightarrow x_1 = y_1 \wedge x_2 = y_2 \tag{1}$$
$$\vdash_{\mathrm{PA}} x < \langle x, y\rangle \wedge y < \langle x, y\rangle \tag{2}$$
$$\vdash_{\mathrm{PA}} x = 0 \vee \exists y \exists z\, x = \langle y, z\rangle. \tag{3}$$

Property (1) is called the *pairing property* and it says that the function is an injection. Property (2) is needed for induction and/or recursion. From (2) we get that $0 \neq \langle x, y\rangle$ for every $x$ and $y$. This means that 0 is not in the range of the pairing function and plays the role of the atom *nil* of Lisp. From this and (3) we can see that the pairing function is onto the set $\mathrm{N} \setminus \{0\}$, i.e. that 0 is the only atom.

*Proof.* (1): It follows from

$$\langle x_1, x_2\rangle = \langle y_1, y_2\rangle \Leftrightarrow \sum_{i=0}^{x_1+x_2} i + x_1 + 1 = \sum_{i=0}^{y_1+y_2} i + y_1 + 1 \overset{2.1.4(2)}{\Leftrightarrow}$$
$$\Leftrightarrow x_1 + x_2 = y_1 + y_2 \wedge x_1 = y_1 \Leftrightarrow x_1 = y_1 \wedge x_2 = y_2.$$

(2): The first conjunct is obvious and the second one follows from

$$y < x + y + x + 1 \overset{2.1.5(1)}{\leq} \sum_{i=0}^{x+y} i + x + 1 = \langle x, y \rangle.$$

(3): Assume $x \neq 0$. By 2.1.5(3) there is a number $n$ such that

$$\sum_{i=0}^{n} i + 1 \leq x < \sum_{i=0}^{n+1} i + 1.$$

Clearly $x = \sum_{i=0}^{n} i + y + 1$ for some $y$. From this get

$$\sum_{i=0}^{n} i + y + 1 = x < \sum_{i=0}^{n+1} i + 1 = \sum_{i=0}^{n} i + n + 1 + 1$$

and therefore $y \leq n$. Let $z$ be the number such that $n = y + z$. We obtain

$$x = \sum_{i=0}^{y+z} i + y + 1 = \langle y, z \rangle.$$ □

**2.1.8 Ordering properties of the pairing function.** We have

$$\vdash_{\mathrm{PA}} \langle x_1, x_2 \rangle \leq \langle y_1, y_2 \rangle \leftrightarrow x_1 + x_2 < y_1 + y_2 \lor x_1 + x_2 = y_1 + y_2 \land x_1 \leq y_1 \quad (1)$$
$$\vdash_{\mathrm{PA}} \langle x_1, x_2 \rangle < \langle y_1, y_2 \rangle \leftrightarrow x_1 + x_2 < y_1 + y_2 \lor x_1 + x_2 = y_1 + y_2 \land x_1 < y_1. \quad (2)$$

*Proof.* (1): It follows from

$$\langle x_1, x_2 \rangle \leq \langle y_1, y_2 \rangle \Leftrightarrow \sum_{i=0}^{x_1+x_2} i + x_1 + 1 \leq \sum_{i=0}^{y_1+y_2} i + y_1 + 1 \overset{2.1.4(1)}{\Leftrightarrow}$$
$$\Leftrightarrow x_1 + x_2 < y_1 + y_2 \lor x_1 + x_2 = y_1 + y_2 \land x_1 \leq y_1.$$

(2): It follows from 2.1.7(1) and (1). □

**2.1.9 Pair representation of natural numbers.** The class of *pair numerals* consists of terms obtained from 0 by finitely many pairing operations. It can be easily proved by complete induction that every natural number $x$ can be uniquely presented as a pair numeral. We call this the *pair representation* of natural numbers. Pair numerals can be visualized as finite binary trees.

Zeroes are leaves and a pair numeral $\langle \tau_1, \tau_2 \rangle$ is a tree with two sons $\tau_1$ and $\tau_2$. Figure 2.2 enumerates the finite binary trees corresponding to the pair numerals. We will denote by $|x|_{\mathrm{p}}$ the number of nodes of the tree corresponding to the pair numeral $\tau = x$. In other words, $|x|_{\mathrm{p}}$ is the number of pairing operations needed to construct the pair numeral $\tau = x$.

In order to obtain a simple recursive characterization of subelementary complexity classes (such as PTIME) one should use a pairing function such that $|x|_{\mathrm{p}} = \Omega(\lg(x))$ (see [53]). The system CLuses such a function but for

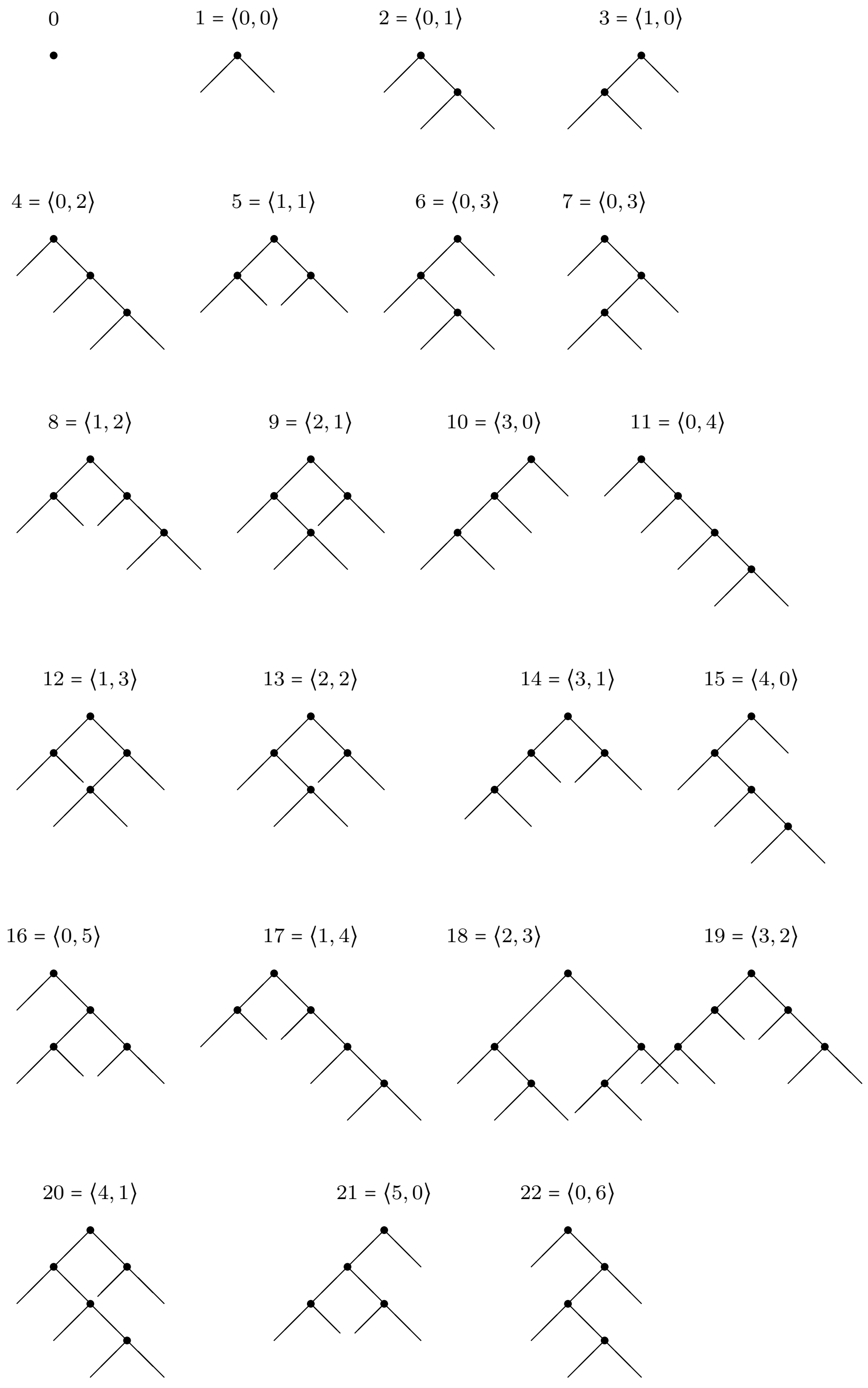

**Fig. 2.2** Pair representation of natural numbers

the purposes of this paper this requirement is not important and we use a much simpler pairing function which does not satisfy the requirement.

**2.1.10 Projection functions.** From the basic properties of the pairing function we can see that every non-zero number $x$ can be uniquely written in the form $x = \langle y, z \rangle$ for some $y, z$. The numbers $y$ and $z$ are called the *first* and the *second projection* of $x$, respectively.

The *first projection* function $\pi_1$ and *second projection* $\pi_2$ are unary functions satisfying

$$\begin{array}{ll} \vdash_{\mathrm{PA}} \pi_1(0) = 0 & \vdash_{\mathrm{PA}} \pi_2(0) = 0 \\ \vdash_{\mathrm{PA}} \pi_1\langle x, y\rangle = x & \vdash_{\mathrm{PA}} \pi_2\langle x, y\rangle = y. \end{array}$$

The projection functions are introduced into PA by contextual definitions:

$$\pi_1(x) = y \leftrightarrow x = 0 \wedge y = 0 \vee \exists z\, x = \langle y, z\rangle$$
$$\pi_2(x) = z \leftrightarrow x = 0 \wedge z = 0 \vee \exists y\, x = \langle y, z\rangle.$$

Both projections are primitive recursive as we might equally well have introduced them by bounded minimalization as shown here

$$\pi_1(x) = \mu y < x[\exists z < x\, x = \langle y, z\rangle] \qquad \pi_2(x) = \mu z < x[\exists y < x\, x = \langle y, z\rangle].$$

## 2.2 Contracted Iteration

**2.2.1 Introduction.** We now turn to the problem of formalization of the iteration of the second projection within PA. That is, we wish to introduce into PA the binary function $\pi_2^n(x)$ such that

$$\vdash_{\mathrm{PA}} \pi_2^0(x) = x$$
$$\vdash_{\mathrm{PA}} \pi_2^{n+1}(x) = \pi_2\, \pi_2^n(x).$$

The demonstration of this fact is far from obvious. We will not able to derive the iteration $\pi_2^n(x)$ directly, but we will do this by encoding its computation from these recurrences.

The derivation of $\pi_2^n(x)$ depends only on the following two simple properties of the second projection:

$$\vdash_{\mathrm{PA}} \pi_2(0) = 0$$
$$\vdash_{\mathrm{PA}} x \neq 0 \rightarrow \pi_2(x) < x.$$

This allows us to formulate our problem in a slightly general way.

**2.2.2 Extensions by contracted iteration.** Let $T$ be an extension by definitions of PA. Let further $f$ be a unary function of $T$ such that

$$\vdash_{\mathrm{PA}} f(0) = 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} x \neq 0 \to f(x) < x. \tag{2}$$

Consider the theory $T'$ obtained from $T$ by adding a new binary function symbol $f^n(x)$, the defining axioms

$$\vdash_{\mathrm{PA}} f^0(x) = x \tag{3}$$
$$\vdash_{\mathrm{PA}} f^{n+1}(x) = f\, f^n(x), \tag{4}$$

and the scheme of mathematical induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f^n(x)$. We say that $T'$ is an *extension of $T$ by contracted iteration.*

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 2.2.17 that the theory $T'$ is an extension by definition of $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this (inessential) extension of $T$.

**2.2.3 The outline of the proof.** For the introduction of contracted iteration into PA we will develop a temporary coding of finite sequences based on the standard notation of natural numbers. Recall that, given a base $p > 1$, every number $x > 0$ can be uniquely written in the form

$$x = x_0p^0 + x_1p^1 + x_2p^2 + \cdots + x_{n-1}x^{n-1} + x_np^n,$$

where $x_1, \ldots, x_n$ are single digits of $x$ in the base $p$, i.e.

$$x_0 < p \quad x_1 < p \quad x_2 < p \quad \ldots \quad x_{n-1} < p \quad 0 < x_n < p.$$

We can think of the number $x$ as the code of the finite sequence $x_0, \ldots, x_n$. We will write $y \in_p x$ if $y = x_i$ for some $i$.

We plan to introduce the contracted iteration by encoding the computation of $f^n(x)$ from its natural recurrences 2.2.2(3)(4). For that we define a ternary *course of values predicate $Cvs_p(s, x)$* which holds if the number $s$ is the code of the *course of values sequence*

$$\langle f^0(x), 0\rangle, \langle f^1(x), 1\rangle, \langle f^2(x), 2\rangle, \ldots, \langle f^{n-1}(x), n-1\rangle, \langle f^n(x), n\rangle$$

in the base $p$, i.e.

$$s = \sum_{i=0}^{n} \langle f^i(x), i\rangle \times p^i \text{ and } \langle f^i(x), i\rangle < p \text{ for every } i = 0, \ldots, n.$$

Note that we do not need to consider all combinations of arguments in the definition of the course values predicate. Namely from 2.2.2(1)(2) we can easily see that the following holds for some $n \leq x$:

$$x = f^0(x) > f^1(x) > \cdots > f^{n-1}(x) > 0 = f^n(x) = f^{n+1}(x) = \cdots .$$

The contracted iteration $f^n(x)$ can be introduced by

$$f^n(x) = y \leftrightarrow n \leq x \wedge \exists p \exists s\big(Cvs_p(s,x) \wedge \langle y,n\rangle \in_p s\big) \vee n > x \wedge y = 0.$$

Note also that for our purposes it suffices to consider encoding of sequences containing only nonzero elements.

The rest of this section is organized as follows. First we develop the coding of finite sequences, where we consider only notations with a prime base. Then we investigate the properties of course of values sequences. Finally we prove in Thm. 2.2.17 that the contracted iteration can be formalized within PA.

**2.2.4 Prime numbers.** Recall that a number $\geq 2$ is said to be prime if its only divisors are one and itself. This is captured in PA by the following explicit definition

$$Prime(p) \leftrightarrow p > 1 \wedge \forall d(d \mid p \to d = 1 \vee d = p).$$

Many properties of prime numbers can be easily formalized in PA together with their proofs; for example the second theorem of Euclid:

$$\vdash_{\mathrm{PA}} \exists p(x < p \wedge Prime(p)), \tag{1}$$

and the Fundamental theorem of arithmetic:

$$\vdash_{\mathrm{PA}} Prime(p) \leftrightarrow p > 1 \wedge \forall x \forall y(p \mid xy \to p \mid x \vee p \mid y). \tag{2}$$

Both properties will be needed below.

**2.2.5 Powers of primes.** A number $q$ is a power of the number $p$ if $q = p^i$ for some $i$. If $p$ is prime we can define the predicate of being a power of $p$ without recourse to exponential as we have the following for every $q$:

$q$ is a power of $p$ iff every divisor of $q$ is either 1 or is divisible by $p$.

This is formally captured by the following definition

$$Pow_p(q) \leftrightarrow \forall d(d \mid q \to d = 1 \vee p \mid d)$$

of the *binary* predicate $Pow_p(q)$.

**2.2.6 Discrimination property of powers of primes.** We have

$$\vdash_{\mathrm{PA}} Prime(p) \to Pow_p(q) \leftrightarrow q = 1 \vee \exists q_1(q = pq_1 \wedge Pow_p(q_1)). \tag{1}$$

*Proof.* We need the following properties:

$$\vdash_{\mathrm{PA}} Pow_p(0) \to p = 1 \qquad (\dagger_1)$$
$$\vdash_{\mathrm{PA}} Pow_p(1) \qquad (\dagger_2)$$
$$\vdash_{\mathrm{PA}} Pow_p(pq) \to Pow_p(q) \qquad (\dagger_3)$$
$$\vdash_{\mathrm{PA}} Prime(p) \land Pow_p(q) \to Pow_p(pq) \qquad (\dagger_4)$$

($\dagger_1$): Suppose, by contradiction, that we have $Pow_p(0)$ for some $p \neq 1$. Then $\forall d(d = 1 \lor p \mid d)$ by definition. Now either $p = 0$ and then $0 \mid 2$ or $p > 1$ and then $p \mid (p+1)$. In either case we have a contradiction.

($\dagger_2$): This is trivial. ($\dagger_3$): Straightforward.

($\dagger_4$): Under the assumption of the claim take any $d$ such that $d \mid pq$. We then have $pq = dx$ for some $x$. We have also $d = yp + r$ for some $y$ and $r < p$. By combining both equations we obtain that $pq = ypx + rx$. From this $p \mid rx$ and thus, by the Fundamental theorem of arithmetic (see 2.2.4(2)), the number $p$ divides either $r$ or $x$. The first case leads to contradiction with $r < p$. So it must be $p \mid x$, i.e. $x = pz$ for some $z$. We have $pq = dpz$ and thus $q = dz$. As a consequence $d \mid q$ and thus, since $q$ is a power of $p$, we have either $q = 1$ or $p \mid d$.

We are in position to prove the equivalence (1). The ($\leftarrow$)-direction is a straightforward consequence of ($\dagger_2$) and ($\dagger_4$). In the proof of the reverse direction ($\to$) under the assumptions of the claim we consider two cases according to ($\dagger_1$). If $q = 1$ then the claim holds trivially. If $q > 1$ then from the assumption $Pow_p(q)$ we obtain $p \mid q$ by noting that $q \mid q$. This means that we have $q = pq_1$ for some $q_1$. Now it suffices to apply ($\dagger_3$) to get $Pow_p(q_1)$. □

**2.2.7 Finite sequences in a prime base.** We now return to the problem of arithmetization of finite sequences as described in Par. 2.2.3. Let $p$ be a prime number. Let further $s = \sum_j x_j p^j$ be the code of the finite sequence of nonzero numbers $x_0, x_1, \ldots$ in the base $p$. We wish to introduce into PA the ternary predicate $x \in_p s$ which holds if $x = x_i$ for some $x_i$. First note that

$$\sum_j x_j p^j = \sum_{j>i} x_j p^j + x_i p^i + \sum_{j<i} x_j p^j = (\sum_{j>i} x_j p^{j-i-1}) p p^i + x_i p^i + \sum_{j<i} x_j p^j .$$

From this we obtain

the predicate $x \in_p s$ holds iff there is a power $q$ of $p$ and numbers $a$ and $b < q$ such that $s = apq + xq + b$ holds.

The last condition can be easily formalized within PA.

In order to shorten our presentation first we introduce into PA the following 4-ary auxiliary predicate:

$$x \in_p^q s \leftrightarrow x \neq 0 \land x = s \div q \bmod p.$$

The predicate has a simple meaning:

$x \in_p^{p^i} s$ iff $x$ is the $i$-th non-zero digit in the $p$-ary representation of $s$.

The ternary membership predicate $x \in_p s$ is defined explicitly by

$$x \in_p s \leftrightarrow \exists q(Pow_p(q) \land x \in_p^q s).$$

In the sequel we will write $x \notin_p^q s$ and $x \notin_p s$ as an abbreviation for $\neg(x \in_p^q s)$ and $\neg(x \in_p s)$, respectively.

In addition to the membership predicate we will also need two operations for creating finite sequences. The empty sequence is coded by the number 0. The ternary *insertion* operation $s \cup_p \{x\}$ adds the number $x$ to the sequence $s$ in the base $p$. The function has the following simple explicit definition:

$$s \cup_p \{x\} = sp + x.$$

From now on we will identify (finite) sequences with their codes and we say the *sequence* $s$ instead of the *code the sequence* $s$.

**2.2.8 Basic properties of the auxiliary predicate $x \in_p^q s$.** We have

$$\vdash_{\mathrm{PA}} x \notin_p^q 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} 0 < x < p \to y \in_p^1 s \cup_p \{x\} \leftrightarrow y = x \tag{2}$$
$$\vdash_{\mathrm{PA}} 0 < x < p \to y \in_p^{pq} s \cup_p \{x\} \leftrightarrow y \in_p^q s. \tag{3}$$

*Proof.* (1): It follows from

$$\vdash_{\mathrm{PA}} s = 0 \lor q = 0 \lor p = 0 \to s \div q \bmod p = 0.$$

(2): First note that for $x < p$ we have the following

$$(sp + x) \div 1 \bmod p = (sp + x) \bmod p = x \bmod p = x \tag{$\dagger_1$}$$

If $0 < x < p$ then we obtain

$$y \in_p^1 s \cup_p \{x\} \Leftrightarrow y \neq 0 \land y = (sp + x) \div 1 \bmod p \overset{(\dagger_1)}{\Leftrightarrow} y \neq 0 \land y = x \Leftrightarrow y = x.$$

(3): First note that for $x < p$ we have the following

$$(sp + x) \div (pq) \bmod p = (sp + x) \div p \div q \bmod p = s \div q \bmod p. \tag{$\dagger_2$}$$

If $0 < x < p$ then we obtain

$$y \in_p^{pq} s \cup_p \{x\} \Leftrightarrow y \neq 0 \land y = (sp + x) \div (pq) \bmod p \overset{(\dagger_2)}{\Leftrightarrow}$$
$$\Leftrightarrow y \neq 0 \land y = s \div q \bmod p \Leftrightarrow y \in_p^q s. \qquad \square$$

**2.2.9 Basic properties of the membership predicate.** We have

$$\vdash_{\mathrm{PA}} \; x \notin_p 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} \; Prime(p) \land 0 < x < p \to y \in_p s \cup_p \{x\} \leftrightarrow y \in_p s \lor y = x. \tag{2}$$

*Proof.* (1) follows from 2.2.8(1). (2): If $Prime(p)$ and $0 < x < p$ then we have

$$y \in_p s \cup_p \{x\} \Leftrightarrow \exists q(Pow_p(q) \land y \in_p^q s \cup_p \{x\}) \overset{2.2.6(1)}{\Leftrightarrow}$$
$$\Leftrightarrow \exists q(Pow_p(q) \land y \in_p^{pq} s \cup_p \{x\}) \lor y \in_p^1 s \cup_p \{x\} \overset{2.2.8(2)(3)}{\Leftrightarrow}$$
$$\Leftrightarrow \exists q(Pow_p(q) \land y \in_p^q s) \lor y = x \Leftrightarrow y \in_p s \lor y = x. \qquad \square$$

**2.2.10 Course of values sequences.** Now we are ready to define the ternary course of values predicate $Cvs_p(s,x)$ discussed in Par. 2.2.3. The predicate is defined explicitly by

$$\begin{aligned} Cvs_p(s,x) \leftrightarrow & \forall y(\langle y,0\rangle \in_p s \to y = x) \land \\ & \forall y \forall n\big(\langle y,n+1\rangle \in_p s \to \exists z(y = f(z) \land \langle z,n\rangle \in_p s)\big). \end{aligned}$$

**2.2.11 Uniqueness of course of values sequences.** We have

$$T \vdash \bigwedge_{i=1}^{2}(Cvs_{p_i}(s_i,x) \land \langle y_i,n\rangle \in_{p_i} s_i) \to y_1 = y_2. \tag{1}$$

*Proof.* By a straightforward induction on $n$ as $\forall y_1 \forall y_2(1)$. □

**2.2.12 Bounded property of course of values sequences.** We have

$$T \vdash n \le x \land Cvs_p(s,x) \land \langle y,n\rangle \in_p s \to \langle y,n\rangle \le \langle x,0\rangle. \tag{1}$$
$$T \vdash n \le x \land Cvs_p(s,x) \land \langle y,n\rangle \in_p s \to \langle f(y),n+1\rangle < p. \tag{2}$$

*Proof.* First we prove the following two auxiliary properties

$$T \vdash n < x \land \langle y,n\rangle \le \langle x,0\rangle \to \langle f(y),n+1\rangle \le \langle x,0\rangle \tag{$\dagger_1$}$$
$$T \vdash Cvs_p(s,x) \land \langle y,n\rangle \in_p s \to \langle x,0\rangle \in_p s. \tag{$\dagger_2$}$$

($\dagger_1$): Assume $n < x$ and $\langle y,n\rangle \le \langle x,0\rangle$. We consider two cases. If $y = 0$ then $f(0) = 0$ by 2.2.2(1) and therefore

$$\langle f(0),n+1\rangle = \langle 0,n+1\rangle \le \langle 0,x\rangle \le \langle x,0\rangle.$$

If $y \ne 0$ then $f(y) < y$ by 2.2.2(2) and thus

$$\langle f(y),n+1\rangle < \langle f(y)+1,n\rangle \le \langle y,n\rangle \le \langle x,0\rangle.$$

($\dagger_2$): By a straightforward induction on $n$ as $\forall y(\dagger_2)$.

(1): By induction on $n$ as $\forall y(1)$. The base case follows directly from the definition. In the induction step take any $y$ such that $n+1 \le x$, $\mathit{Cvs}_p(s,x)$, and $\langle y, n+1\rangle \in_p s$. By definition there is $z$ such that $y = f(z)$ and $\langle z,n\rangle \in_p s$. We then have $\langle z,n\rangle \le \langle x,0\rangle$ by IH and thus $\langle f(z), n+1\rangle \le \langle x,0\rangle$ by $(\dagger_1)$.

(2): Assume $n \le x$, $\mathit{Cvs}_p(s,x)$ and $\langle y,n\rangle \in_p s$. We have $\langle x,0\rangle \in_p s$ by $(\dagger_2)$ and hence $\langle x,0\rangle < p$. From (1) we get $\langle y,n\rangle \le \langle x,0\rangle < p$. □

**2.2.13 Extension properties of course of values sequences.** The following two properties

$$T \vdash \mathit{Prime}(p) \land \langle x,0\rangle < p \to \mathit{Cvs}_p(0 \cup_p \{\langle x,0\rangle\}, x) \tag{1}$$
$$T \vdash \mathit{Prime}(p) \land n < x \land \mathit{Cvs}_p(s,x) \land \langle y,n\rangle \in_p s \to \mathit{Cvs}_p(s \cup_p \{\langle f(y), n+1\rangle\}, x) \tag{2}$$

show the way how to construct course of values sequences.

*Proof.* (1): Let $\mathit{Prime}(p)$ and $\langle x,0\rangle$. From the results of Par. 2.2.9 we get

$$\begin{aligned}\langle y,n\rangle \in_p 0 \cup_p \{\langle x,0\rangle\} \Leftrightarrow \langle y,n\rangle \in_p 0 \lor \langle y,n\rangle = \langle x,0\rangle \Leftrightarrow \\ \Leftrightarrow \langle y,n\rangle = \langle x,0\rangle \Leftrightarrow y = x \land n = 0.\end{aligned}$$

Consequently, the number $0\cup_p\{\langle x,0\rangle\}$ is a course of values sequence for $f^n(x)$ in the base $p$.

(2): Suppose that $n < x$, $\mathit{Cvs}_p(s,x)$ and $\langle y,n\rangle \in_p s$. Let us denote by $s'$ the number $s \cup_p \{\langle f(y), n+1\rangle\}$. We wish to show that $s'$ is a course of values sequence for $f^n(x)$ in the base $p$, i.e.

$$\forall u(\langle u,0\rangle \in_p s' \to u = x) \tag{$\dagger_1$}$$
$$\forall u \forall k\big(\langle u,k+1\rangle \in_p s' \to \exists z(u = f(z) \land \langle z,k\rangle \in_p s')\big). \tag{$\dagger_2$}$$

Property $(\dagger_1)$ follows from

$$\langle u,0\rangle \in_p s' \overset{2.2.9(2)}{\Rightarrow} \langle u,0\rangle \in_p s \lor \langle u,0\rangle = \langle f(y), n+1\rangle \Rightarrow \langle u,0\rangle \in_p s \Rightarrow u = x$$

since $s$ is a course of values sequence for $f^n(x)$ in the base $p$.

In the proof of $(\dagger_2)$ suppose that $\langle u,k+1\rangle \in_p s'$. We consider two cases according to 2.2.9(2). If $\langle u,k+1\rangle \in_p s$ then from the assumption $\mathit{Cvs}_p(s,x)$ we obtain that there is a number $z$ such that $u = f(z)$ and $\langle z,k\rangle \in_p s$. By 2.2.9(2) again, we have $\langle z,k\rangle \in_p s'$ and thus we are done. Otherwise we have $\langle u,k+1\rangle = \langle f(y), n+1\rangle$, i.e. $u = f(y)$ and $k = n$. By 2.2.9(2) again, $\langle y,n\rangle \in_p s'$ and we are done. □

**2.2.14 The graph of the contracted iteration.** By $G(n,x,y)$ we denote the graph of the iteration of $f$ which is a ternary predicate defined by

$$G(n,x,y) \leftrightarrow n \le x \land \exists p \exists s(\mathit{Prime}(p) \land \mathit{Cvs}_p(s,x) \land \langle y,n\rangle \in_p s) \lor n > x \land y = 0.$$

**2.2.15 Extension properties the graph.** We have:

$$T \vdash G(0,x,x) \tag{1}$$
$$T \vdash G(n,x,y) \rightarrow G(n+1,x,f(y)). \tag{2}$$

*Proof.* (1): By the second theorem of Euclid (see 2.2.4(1)) there is a prime number $p > \langle x,0\rangle$. By 2.2.13(1) the number $0 \cup_p \{\langle x,0\rangle\}$ is a course of values sequence for $f^n(x)$ in the base $p$. We have also $\langle x,0\rangle \in_p 0 \cup_p \{\langle x,0\rangle\}$ and thus $G(0,x,x)$ by definition.

(2): First note that we have

$$T \vdash n \geq x \wedge G(n,x,y) \rightarrow y = 0. \tag{$\dagger_1$}$$

The claim holds trivially for $n > x$. If $n = x$ then we have $\langle y,x\rangle \leq \langle x,0\rangle$ by 2.2.12(1) and thus $y + x \leq x + 0$ from which we get $y = 0$.

Under the assumption $G(n,x,y)$ we consider two cases. If $n+1 \leq x$ then $n \leq x$ and thus there is a course of values sequence $s$ for $f^n(x)$ containing the pair $\langle y,n\rangle$. Then, by 2.2.13(2), the number $s \cup_p \{\langle f(y),n+1\rangle\}$ is a course of values sequence for $f^n(x)$ containing the pair $\langle f(y),n+1\rangle$, i.e. we have $G(n+1,x,f(y))$. If $n+1 > x$, i.e. $n \geq x$, we get $y = 0$ from above and since $f(0) = 0$ we clearly have $G(n+1,x,f(y))$ from the assumption $G(n,x,y)$. □

**2.2.16 Existence and uniqueness conditions for the graph.** We have

$$T \vdash \exists y G(n,x,y) \tag{1}$$
$$T \vdash G(n,x,y_1) \wedge G(n,x,y_2) \rightarrow y_1 = y_2. \tag{2}$$

*Proof.* (1): It follows from 2.2.15(1)(2) by a straightforward induction on $x$. The uniqueness property (2) is proved by considering two cases. If $n > x$ then $y_1 = 0 = y_2$ from the definition of the graph. The case when $n \leq x$ follows from the uniqueness property 2.2.11(1) of course of values sequences. □

**2.2.17 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by contracted iteration is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by contracted iteration as in Par. 2.2.2 and $T''$ an extension of $T$ by the contextual definition

$$f^n(x) = y \leftrightarrow G(n,x,y),$$

where $G$ is as in Par. 2.2.14. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$. In order to prove the theorem it suffices to show that the theories $T'$ and $T''$ have the same theorems.

We first show $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA, and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f^n(x)$. It remains to show that $T''$ proves both defining axioms of the unary iteration of $f$.

We have $G(0, x, f^0(x))$ from the definition of the iteration and $G(0, x, x)$ by 2.2.15(1). Consequently $f^0(x) = x$ by the uniqueness of the graph.

The second defining axiom is proved as follows. By definition $G(n, x, f^n(x))$ and thus $G(n+1, x, f\, f^n(x))$ by the second extension property of the graph. On the other hand, we have also $G(n+1, x, f^{n+1}(x))$ by definition. Now the uniqueness of the graph applies and we get $f^{n+1}(x) = f\, f^n(x)$.

Vice versa, in order to show $T' \vdash T''$ it suffices to show

$$T' \vdash G(n, x, f^n(x))$$

by Thm. 1.3.17. The property is proved by induction on $n$.[1] The base case follows immediately from the first defining axiom and 2.2.15(1). In the induction step we have $G(n, x, f^n(x))$ by IH and thus $G(n+1, x, f\, f^n(x))$ by 2.2.15(2). Consequently $G(n+1, x, f^{n+1}(x))$ by definition. □

**2.2.18 Basic properties of the contracted iteration.** We have

$$T \vdash f^{n+1}(x) = f^n\, f(x) \tag{1}$$
$$T \vdash \exists n \le x\, f^n(x) = 0 \tag{2}$$
$$T \vdash f^n(0) = 0. \tag{3}$$

*Proof.* (1): By induction on $n$. The base case follows from

$$f^{0+1}(x) = f\, f^0(x) = f(x) = f^0\, f(x).$$

In the induction step we have

$$f^{n+1+1}(x) = f\, f^{n+1}(x) \overset{\text{IH}}{=} f\, f^n\, f(x) = f^{n+1}\, f(x).$$

(2): By complete induction on $x$. Take any $x$ and consider two cases. If $x = 0$ then $f^0(0) = 0$ and thus it suffices to set $n := 0$. Otherwise we have $x = \langle v, w \rangle$ for some $v, w$. Since $w < x$ then by IH there is a number $m \le w$ such that $f^m(w) = 0$. We then obtain

$$f^{m+1}\langle v, w \rangle \overset{(1)}{=} f^m\, f\langle v, w \rangle = f^m(w) = 0.$$

We have $m + 1 \le w + 1 \le \langle v, w \rangle$ and thus it suffices to set $n := m + 1$.

(3): By a straightforward induction on $n$. □

**2.2.19 Iteration of the second projection.** As a straightforward corollary of Thm. 2.2.17 and the previous paragraph we obtain that there is a

[1] This is the only place that we use the principle of mathematical induction for a formula containing the symbol $f^n(x)$ inside the theory $T'$.

binary function $\pi_2^n(x)$ of PA such that

$$\vdash_{\mathrm{PA}} \pi_2^0(x) = x \tag{1}$$
$$\vdash_{\mathrm{PA}} \pi_2^{n+1}(x) = \pi_2\,\pi_2^n(x) \tag{2}$$

and also

$$\vdash_{\mathrm{PA}} \pi_2^{n+1}(x) = \pi_2^n\,\pi_2(x) \tag{3}$$
$$\vdash_{\mathrm{PA}} \exists n \leq x\,\pi_2^n(x) = 0 \tag{4}$$
$$\vdash_{\mathrm{PA}} \pi_2^n(0) = 0. \tag{5}$$

## 2.3 Tuples

**2.3.1 Introduction.** In this section we introduce a particular encoding of ordered $n$-tuples of natural numbers based on our pairing function $\langle x, y\rangle$. Our aim is to assign to each element $(x_1, \dots, x_n)$ of the Cartesian product $N^n$ a number $\ulcorner(x_1, \dots, x_n)\urcorner^n$, called the *code* of $(x_1, \dots, x_n)$, so that different codes are assigned to different $n$-tuples. Moreover we would like to have decoding effective. This means we can effectively decide whether a number is the code of an $n$-tuple and if it is, find that tuple. We will use this encoding throughout the rest of this text.

**2.3.2 Arithmetization of tuples.** Encoding of Cartesian products $N^n$, where $n \geq 0$, is defined inductively on $n$ as follows:

$$\begin{aligned} \ulcorner\varnothing\urcorner^0 &= 0 \\ \ulcorner x\urcorner^1 &= x \\ \ulcorner(x_1, x_2, \dots, x_n)\urcorner^n &= \left\langle x_1, \ulcorner(x_2, \dots, x_n)\urcorner^{n-1}\right\rangle \qquad \text{if } n \geq 2. \end{aligned}$$

The reader will note that the code of an 1-tuple $x$ is the number itself and the code of the empty tuple $\varnothing$ is the number 0. Note also that $\ulcorner(x, y)\urcorner^2 = \langle x, y\rangle$.

The reader will also note that these encodings may overlap. Consider, for instance, the number 2. We have $2 = \langle 0, 1\rangle$ and $1 = \langle 0, 0\rangle$. Therefore

$$\begin{aligned} 2 &= \langle 0, 1\rangle = \ulcorner(0, 1)\urcorner^2 \\ 2 &= \langle 0, 1\rangle = \big\langle 0, \langle 0, 0\rangle\big\rangle = \big\langle 0, \ulcorner(0, 0)\urcorner^2\big\rangle = \ulcorner(0, 0, 0)\urcorner^3. \end{aligned}$$

Hence, the number 2 is the code both of the ordered pair $(0, 1) \in \mathrm{N}^2$ and the ordered triple $(0, 0, 0) \in \mathrm{N}^3$.

**2.3.3 Notational conventions.** We will adopt the following conventions for the pairing function $\langle x, y\rangle$. We postulate that the pairing operator groups to the right, i.e. $\langle x, y, z\rangle$ abbreviates $\big\langle x, \langle y, z\rangle\big\rangle$. If $\vec{\tau} \equiv (\tau_1, \dots, \tau_n)$ is an $n$-tuple of terms then the term $\langle\vec{\tau}\rangle$ stands for $\langle\tau_1, \dots, \tau_n\rangle$ when $n \geq 2$, for $\tau_1$ when $n = 1$, and for 0 when $n = 0$. Note that we then have

$$\ulcorner(x_1, \dots, x_n)\urcorner^n = \langle x_1, \dots, x_n\rangle$$

for every $n$ and every element $(x_1, \dots, x_n)$ of $N^n$.

**2.3.4 Predicate holding of the codes of tuples.** For $n \geq 2$, we have

$$\vdash_{\mathrm{PA}} \exists x_1 \dots \exists x_n\, x = \langle x_1, \dots, x_n\rangle \leftrightarrow \pi_2^{n-2}(x) \neq 0.$$

Consequently, the binary predicate $\mathit{Tuple}(n, x)$, which holds when $x$ is the code of an $n$-tuple, is primitive recursive by the following explicit definition

$$\mathit{Tuple}(n, x) \leftrightarrow n = 0 \land x = 0 \lor n = 1 \lor n \geq 2 \land \pi_2^{n-2}(x) \neq 0.$$

**2.3.5 Projection function for tuples.** The ternary *projection* function $[x]_i^n$ selects the $i$-th element of the $n$-tuple coded by $x$, i.e.

$$\vdash_{\mathrm{PA}} [\langle x_1, \dots, x_n\rangle]_i^n = x_i \tag{1}$$

for every $i = 1, \dots, n$. We clearly have

$$\vdash_{\mathrm{PA}} x = \langle x_1, \dots, x_n\rangle \to \bigwedge_{i=1}^{n-1} x_i = \pi_1\, \pi_2^{i-1}(x) \land x_n = \pi_2^{n-1}(x)$$

for $n \geq 2$. Thus we can define $[x]_i^n$ explicitly as a p.r. function by

$$[x]_i^n = D\big(i <_* n, \pi_1\pi_2^{i \dot- 1}(x), \pi_2^{n \dot- 1}(x)\big).$$

The projection function satisfies

$$\begin{aligned}
&\vdash_{\mathrm{PA}} [x_1]_1^1 = x_1\\
&\vdash_{\mathrm{PA}} [\langle x_1, x\rangle]_1^{n+2} = x_1\\
&\vdash_{\mathrm{PA}} [\langle x_1, x\rangle]_{i+2}^{n+2} = [x]_{i+1}^{n+1}\,.
\end{aligned}$$

**2.3.6 Contraction to unary functions.** As a simple application of the arithmetization of $n$-tuples we obtain the following natural correspondence between $n$-ary and unary functions. If $f$ is an $n$-ary function then its *contraction* is the unary function $\langle f\rangle$ such that

$$\langle f\rangle(x) = \begin{cases} f(x_1, \dots, x_n) & \text{if } x = \langle x_1, \dots, x_n\rangle \text{ for some numbers } x_1, \dots, x_n,\\ 0 & \text{if there are no such numbers.}\end{cases}$$

Note that the contraction of an unary function is the function itself.

We can define the contraction of $f$ explicitly by

$$\langle f\rangle(x) = D\Big(\mathit{Tuple}_*(n,x), f\big([x]_1^n,\dots,[x]_n^n\big), 0\Big).$$

Vice versa, we can recover $f$ from its contraction by

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

Thus a function is primitive recursive if and only if its contraction is.

## 2.4 Finite Sequences

**2.4.1 Introduction.** Now we consider the problem of the arithmetization of finite sequences of natural numbers. Mathematically speaking, finite sequences are just tuples of variable length and so the set of all such sequences is the infinite union $\bigcup_{n\in\mathrm{N}} \mathrm{N}^n$. We cannot use the method of codings of tuples of fixed length since such encodings overlap. Our uniform encoding of finite sequences is based on the fact that the number 0 is the atom, i.e. it is not in the range of the pairing function $\langle x, y\rangle$.

**2.4.2 Arithmetization of finite sequences.** A uniform method for coding of finite sequences of numbers into N is obtained as follows. We assign the code 0 to the empty sequence $\varnothing$. A non-empty sequence $x_1,\dots,x_n$ is coded by the number $\langle x_1, x_2,\dots,x_n, 0\rangle$ as shown in Fig. 2.3. The number $\langle x_1, x_2,\dots,x_n, 0\rangle$ is often called the sequence number of the sequence $x_1,\dots,x_n$.

The reader will note that the assignment of codes is one to one: i.e. 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.

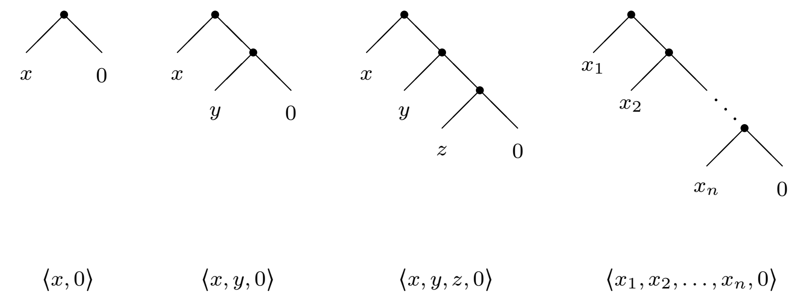

**Fig. 2.3** Arithmetization of finite sequences

**2.4.3 Length of sequences.** The code $x = \langle x_1, x_2, \ldots, x_n, 0\rangle$ of the sequence $x_1, \ldots, x_n$ has the *length* $n$. The function $L(x)$ yielding the length of $x$ is introduced into PA by the bounded minimalization as a p.r. function:

$$L(x) = \mu n \le x[\pi_2^n(x) = 0].$$

The function satisfies

$$\vdash_{\text{PA}} L(0) = 0 \tag{1}$$
$$\vdash_{\text{PA}} L\langle v, w\rangle = L(w) + 1. \tag{2}$$

*Proof.* First note that from the definition of $L$ and 2.2.19(4) we obtain that

$$\vdash_{\text{PA}} \pi_2^{L(x)}(x) = 0 \tag{$\dagger_1$}$$
$$\vdash_{\text{PA}} \pi_2^n(x) = 0 \to L(x) \le n. \tag{$\dagger_2$}$$

Note also that we have

$$\vdash_{\text{PA}} \pi_2^{n+1}\langle v, w\rangle = 0 \leftrightarrow \pi_2^n(w) = 0 \;. \tag{$\dagger_3$}$$

(1): We have $\pi_2^0(0) = 0$ and thus $L(0) \le 0$ by ($\dagger_2$), i.e. $L(0) = 0$. (2): We have $\pi_2^{L(w)}(w) = 0$ by ($\dagger_1$) and hence $\pi_2^{L(w)+1}\langle v, w\rangle = 0$ by ($\dagger_3$). From this and ($\dagger_2$) we obtain $L\langle v, w\rangle \le L(w) + 1$. The reverse inequality is proved as follows. We have $\pi_2^{L\langle v.w\rangle}\langle v.w\rangle = 0$ by ($\dagger_1$). Now since $\pi_2^0\langle v, w\rangle = \langle v, w\rangle \ne 0$ it must be $L\langle v, w\rangle \ne 0$ and thus $\pi_2^{L\langle v.w\rangle \dot{-} 1}(w) = 0$ by ($\dagger_3$). From this and ($\dagger_2$) we conclude that $L(w) \le L\langle v, w\rangle \dot{-} 1$, i.e. that we have $L(w) + 1 \le L\langle v, w\rangle$. □

**2.4.4 Indexing function.** The *indexing* function $(x)_i$ yields the $(i+1)$-st element of the sequence $x$, i.e.

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

The function is defined explicitly by

$$(x)_i = \pi_1\, \pi_2^i(x)$$

as a primitive recursive function.

The recurrent properties of the indexing function are:

$$\vdash_{\text{PA}} (\langle v, w\rangle)_0 = v \tag{1}$$
$$\vdash_{\text{PA}} (\langle v, w\rangle)_{i+1} = (w)_i\,. \tag{2}$$

*Proof.* (1): It follows from $(\langle v, w\rangle)_0 = \pi_1\, \pi_2^0\langle v, w\rangle = \pi_1\langle v, w\rangle = v$. The second property (2) follows from

$$(\langle v, w\rangle)_{i+1} = \pi_1\,\pi_2^{i+1}\langle v, w\rangle = \pi_1\,\pi_2^{i}\,\pi_2\langle v, w\rangle = \pi_1\,\pi_2^{i}(w) = (w)_i\,. \qquad \square$$

**2.4.5 Uniqueness of arithmetization.** We say that a number $y$ is an element of $x$ if there is $i < L(x)$ such that $y = (x)_i$. The following property

$$\vdash_{\mathrm{PA}} \; x = y \leftrightarrow L(x) = L(y) \land \forall i\big(i < L(x) \to (x)_i = (y)_i\big) \tag{1}$$

asserts that the codes of sequences are uniquely determined by their length and their elements.

*Proof.* Property $\forall y(1)$ is proved by complete induction on $x$. Take any $y$ and consider four cases according to whether $x$ or $y$ is 0 or not. We will show here only the case when $x = \langle v_1, w_1\rangle$ and $y = \langle v_2, w_2\rangle$ for some $v_1, w_1, v_2, w_2$; the remaining cases are straightforward. We have $w_1 < x$ and thus

$$\begin{aligned}
&\langle v_1, w_1\rangle = \langle v_2, w_2\rangle \Leftrightarrow v_1 = v_2 \land w_1 = w_2 \overset{\mathrm{IH}}{\Leftrightarrow}\\
&v_1 = v_2 \land L(w_1) = L(w_2) \land \forall j\big(j < L(w_1) \to (w_1)_j = (w_2)_j\big) \overset{(*_1)}{\Leftrightarrow}\\
&\;(\langle v_1, w_1\rangle)_0 = (\langle v_2, w_2\rangle)_0 \land L\langle v_1, w_1\rangle = L\langle v_2, w_2\rangle \land\\
&\qquad \land \forall j\big(j + 1 < L\langle v_1, w_1\rangle \to (\langle v_1, w_1\rangle)_{j+1} = (\langle v_2, w_2\rangle)_{j+1}\big) \overset{(*_2)}{\Leftrightarrow}\\
&L\langle v_1, w_1\rangle = L\langle v_2, w_2\rangle \land \forall i\big(i < L\langle v_1, w_1\rangle \to (\langle v_1, w_1\rangle)_i = (\langle v_2, w_2\rangle)_i\big).
\end{aligned}$$

The step marked by $(*_1)$ follows from the basic properties of the length and indexing functions; the second one marked by $(*_2)$ is just monadic case analysis on the bound variable $i$. Note also that the induction hypothesis is applied with $w_2$ in place of $y$. $\square$

# Chapter 3
# Primitive Recursive Schemes

We are now going to investigate various schemes of extensions of PA which support recursive definitions of primitive recursive functions. We start by showing PA admits definitions by *primitive recursion.* More precisely, we will prove in Sect. 3.1 that extensions of PA by primitive recursion are extensions by definitions. In the proof we use coding of finite sequences based on the modified Cantor pairing function introduced in the previous chapter.

The method of course of values sequences can be easily adapted to show that PA admits definitions by *course of values recursion* as shown in Sect. 3.2. By a reverting the flow of computation we obtain the scheme *backward recursion* which admissibility is proved in the next section. In Sect. 3.4 we will investigate recursive definitions for which parameters change in recursive applications. This is called *recursion with parameter substitution.* The admissibility of the scheme is proved by arithmetization of computation trees.

The hardest part of our recursive bootstrapping is the demonstration that PA admits definitions of functions by *nested simple recursion.* The scheme has a form of primitive recursion where parameters may be arbitrarily substituted for even with nested recursive applications. R. Péter has proved in [39] that primitive recursive functions are closed under such recursion. We have already investigated the scheme in [25], but we hope that this new presentation, which is supplied with a detailed formal proof for the first time, is much simpler and easier to follow (see Sect. 3.5).

## 3.1 Primitive Recursion

**3.1.1 Introduction.** We already know that some primitive recursive functions (e.g. $x \dot{-} y$ or $\sum_{i=0}^{n} i$) can be easily introduced into PA. For other primitive recursive functions, such as the exponentiation $x^y$ for instance, finding their definition in PA is a quite challenging task.

In this section we show that PA admits definitions by primitive recursion. Namely, we prove that if functions $g(\vec{y})$ and $h(x,z,\vec{y})$ are already introduced into PA then we can also introduce into PA a function $f(x,\vec{y})$ satisfying

$$\vdash_{\mathrm{PA}} \quad f(0,\vec{y}) = g(\vec{y})$$
$$\vdash_{\mathrm{PA}} f(x+1,\vec{y}) = h\bigl(x, f(x,\vec{y}), \vec{y}\bigr).$$

Note that this is more than just merely stating that PA contains all primitive recursive functions: we do not require that $g$ and $h$ are primitive recursive.

**3.1.2 Extensions by primitive recursion.** Let $T$ be an extension by definitions of PA. Let $\rho[\vec{y}]$ and $\tau[x,z,\vec{y}]$ be terms of $\mathcal{L}_T$ in which no other variables than the indicated ones are free. Consider a theory $T'$ obtained from the theory $T$ by adding a new $(n{+}1)$-ary function symbol $f$, the defining axioms

$$f(0,\vec{y}) = \rho[\vec{y}] \tag{1}$$
$$f(x+1,\vec{y}) = \tau[x, f(x,\vec{y}), \vec{y}], \tag{2}$$

and the scheme of mathematical induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of* $T$ *by primitive recursion.*

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 3.1.10 that the theory $T'$ is an extension by definition of the theory $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.1.3 The outline of the proof.** We plan to extend the theory $T$ by explicitly defining an $(n{+}2)$-ary predicate *Cvs* such that

$$Cvs(s,x,\vec{y}) \leftrightarrow s = \bigl\langle f(x,\vec{y}), f(x-1,\vec{y}), \ldots, f(2,\vec{y}), f(1,\vec{y}), f(0,\vec{y}), 0\bigr\rangle.$$

In other words, we would like to have

$$Cvs(s,x,\vec{y}) \leftrightarrow L(s) = x+1 \land \forall u < x\ (s)_{x \dot{-} u} = f(u,\vec{y}).$$

The number $s$ such that $Cvs(s,x,\vec{y})$ holds is called a *course of values (c.v.) sequence for* $f(x,\vec{y})$. With the course of values predicate introduced into PA we can define the function $f$ in PA by the following contextual definition:

$$f(x,\vec{y}) = z \leftrightarrow \exists s\bigl(Cvs(s,x,\vec{y}) \land (s)_0 = z\bigr).$$

**3.1.4 Course of values sequences.** The $(n{+}2)$-ary course of values predicate *Cvs* has the following explicit definition:

$$\begin{aligned} Cvs(s,x,\vec{y}) \leftrightarrow\ & L(s) = x{+}1 \land (s)_{x\dot{-}0} = \rho[\vec{y}] \land \\ & \forall u < x\ (s)_{x\dot{-}(u+1)} = \tau[u,(s)_{x\dot{-}u},\vec{y}]. \end{aligned}$$

**3.1.5 Extension properties of course of values sequences.** We have

$$T \vdash Cvs\big(\langle \rho[\vec{y}], 0\rangle, 0, \vec{y}\big) \tag{1}$$

$$T \vdash Cvs(s,x,\vec{y}) \to Cvs\left(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle, x+1, \vec{y}\right). \tag{2}$$

The first property says that the number $\langle \rho[\vec{y}], 0\rangle$ is a course of values sequence for $f(0,\vec{y})$. The second property shows how to built course of values sequences by stages: if the number $s$ is a course of sequence for $f(x,\vec{y})$ then the number $\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle$ is a course of values sequence for the next stage $f(x+1,\vec{y})$.

*Proof.* (1): Directly from definition. (2): Take any number $s$ which is a course of values sequence for $f(x,\vec{y})$, i.e.

$$L(s) = x+1 \tag{$\dagger_1$}$$

$$(s)_{x\dot{-}0} = \rho[\vec{y}] \tag{$\dagger_2$}$$

$$\forall u\big(u < x \to (s)_{x\dot{-}(u+1)} = \tau[u,(s)_{x\dot{-}u},\vec{y}]\big). \tag{$\dagger_3$}$$

We show that $\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle$ is a course of values sequence for $f(x+1,\vec{y})$:

$$L\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle = (x+1)+1 \tag{$\dagger_4$}$$

$$\Big(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle\Big)_{x+1\dot{-}0} = \rho[\vec{y}] \tag{$\dagger_5$}$$

$$\begin{aligned} \forall u\Bigg(u < x+1 \to & \Big(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle\Big)_{x+1\dot{-}(u+1)} = \\ & \tau\left[u, \Big(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle\Big)_{x+1\dot{-}u}, \vec{y}\right]\Bigg). \end{aligned} \tag{$\dagger_6$}$$

($\dagger_4$): It follows from

$$L\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle = L(s) + 1 \overset{(\dagger_1)}{=} (x+1)+1.$$

($\dagger_5$): It follows from

$$\Big(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle\Big)_{x+1\dot{-}0} = \Big(\left\langle \tau[x,(s)_0,\vec{y}], s\right\rangle\Big)_{x+1} = (s)_x \overset{(\dagger_2)}{=} \rho[\vec{y}].$$

($\dagger_6$): Take any $u < x+1$ and consider two cases. If $u < x$ then $u+1 \le x$ and we have

$$\begin{aligned}
&\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x+1\dot{-}(u+1)} = \Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x\dot{-}(u+1)+1} = \\
&\quad = (s)_{x\dot{-}(u+1)} \overset{(\dagger_3)}{=} \tau[u,(s)_{x\dot{-}u}\,,\vec{y}] = \\
&\quad = \tau\left[u,\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x\dot{-}u+1},\vec{y}\right] = \tau\left[u,\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x+1\dot{-}u},\vec{y}\right].
\end{aligned}$$

Otherwise we have $u = x$ and thus

$$\begin{aligned}
&\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x+1\dot{-}(x+1)} = \Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_0 = \tau[x,(s)_0\,,\vec{y}] = \\
&\quad = \tau\Bigl[x,\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_1,\vec{y}\Bigr] = \tau\Bigl[x,\Bigl(\bigl\langle\tau[x,(s)_0\,,\vec{y}],s\bigr\rangle\Bigr)_{x+1\dot{-}x},\vec{y}\Bigr]. \qquad \square
\end{aligned}$$

**3.1.6 The existential and uniqueness properties of c.v. sequences.** The following properties express the fact that course of values sequences can be uniquely constructed for all arguments:

$$T \vdash \exists s\, Cvs(s,x,\vec{y}) \tag{1}$$
$$T \vdash Cvs(s_1,x,\vec{y}) \land Cvs(s_2,x,\vec{y}) \to s_1 = s_2. \tag{2}$$

*Proof.* The existence property (1) is proved by a straightforward induction on $x$, where both the base case and the induction step follow directly from the extension properties of course of values sequences (see Par. 3.1.5).

The uniqueness property (2) is proved as follows. Assume $Cvs(s_1,x,\vec{y})$ and $Cvs(s_2,x,\vec{y})$. Then the course of values sequences $s_1$ and $s_2$ have the same length $x+1$. According to 2.4.5(1) it suffices to show that both sequences have the same elements up to the last index $x$, i.e. that we have

$$\forall i\bigl(i \le x \to (s_1)_i = (s_2)_i\bigr). \tag{$\dagger_1$}$$

For that we need the following auxiliary property

$$u \le x \to (s_1)_{x\dot{-}u} = (s_2)_{x\dot{-}u}\,, \tag{$\dagger_2$}$$

which is proved by induction on $u$. The base case is straightforward:

$$(s_1)_{x\dot{-}0} = \rho[\vec{y}] = (s_2)_{x\dot{-}0}\,.$$

In the induction step assume $u+1 \le x$. Then $u \le x$ and we have

$$(s_1)_{x\dot{-}(u+1)} = \tau[u,(s_1)_{x\dot{-}u}\,,\vec{y}] \overset{\text{IH}}{=} \tau[u,(s_2)_{x\dot{-}u}\,,\vec{y}] = (s_2)_{x\dot{-}(u+1)}\,.$$

We are now in position to prove $(\dagger_1)$. Take any $i \le x$. Then $i+u = x$ for some $u \le x$. We then have

$$(s_1)_i = (s_1)_{x\dot{-}u} \overset{(\dagger_2)}{=} (s_2)_{x\dot{-}u} = (s_2)_i\,. \qquad \square$$

**3.1.7 The graph predicate.** By $G(x,\vec{y},z)$ we denote the graph of the function $f$ which is a $(n{+}2)$-ary predicate explicitly defined by

$$G(x,\vec{y},z) \leftrightarrow \exists s\big(Cvs(s,x,\vec{y}) \wedge (s)_0 = z\big).$$

**3.1.8 Recurrent properties of the graph.** We have

$$T \vdash G\big(0,\vec{y},\rho[\vec{y}]\big) \tag{1}$$
$$T \vdash G(x,\vec{y},z) \to G\big(x+1,\vec{y},\tau[x,z,\vec{y}]\big). \tag{2}$$

*Proof.* It follows directly from the extension properties of c.v. sequences. □

**3.1.9 The existential and uniqueness properties of the graph.**

$$T \vdash \exists z\, G(x,\vec{y},z) \tag{1}$$
$$T \vdash G(x,\vec{y},z_1) \wedge G(x,\vec{y},z_2) \to z_1 = z_2. \tag{2}$$

*Proof.* This is a straightforward consequence of the existential and uniqueness properties of course of values sequences. □

**3.1.10 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by primitive recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by primitive recursion as in Par. 3.1.2 and $T''$ an extension of $T$ by the contextual definition

$$f(x,\vec{y}) = z \leftrightarrow G(x,\vec{y},z),$$

where $G$ is the graph predicate from Par. 3.1.7. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$. In order to prove the claim it suffices to show that both theories $T'$ and $T''$ have the same theorems.

First we show that $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA, and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.1.2(1)(2) of $f$.

We have $G\big(0,\vec{y},f(0,\vec{y})\big)$ by definition of $f$ and $G\big(0,\vec{y},\rho[\vec{y}]\big)$ by first recurrent property of the graph. Now the uniqueness property of the graph applies and we obtain $f(0,\vec{y}) = \rho[\vec{y}]$. This proves in $T''$ the first defining axiom of $f$.

We have $G\big(x,\vec{y},f(x,\vec{y})\big)$ by definition of $f$. Second recurrent property of the graph yields $G\big(x,\vec{y},\tau[x,f(x,\vec{y}),\vec{y}]\big)$. By definition of $f$ again we have also $G(x+1,\vec{y},f(x+1,\vec{y}))$ and thus $f(x+1,\vec{y}) = \tau[x,f(x,\vec{y}),\vec{y}]$ by the uniqueness of the graph. This proves in $T''$ the second defining axiom of $f$.

Vice versa, in order to prove $T' \vdash T''$, it suffices to show

$$T' \vdash G(x,\vec{y},f(x,\vec{y}))$$

in view of Thm. 1.3.17. This is proved by induction on $x$.[1]

The base case $G(0,\vec{y},f(0,\vec{y}))$ is a direct consequence of the defining axiom $f(0,\vec{y}) = \rho[\vec{y}]$ and first recurrent property $G(0,\vec{y},\rho[\vec{y}])$ of the graph.

In the induction step we obtain $G(x,\vec{y},f(x,\vec{y}))$ from IH and thus, by second recurrent property of the graph, we have $G\big(x+1,\vec{y},\tau[x,f(x,\vec{y}),\vec{y}]\big)$. From this and the second definition axiom $f(x+1,\vec{y}) = \tau[x,f(x,\vec{y}),\vec{y}]$ we can conclude that $G(x+1,\vec{y},f(x+1,\vec{y}))$ holds. □

**3.1.11 Remark.** A careful reader has noted that instead of adding infinitely many induction axioms to the non-logical axioms of $T'$ it was sufficient to add only one induction axiom. Its sole purpose is to show that the definition by primitive recursion has a unique solution. The induction formula can be easily read off from the proof of Thm. 3.1.10. We will encounter a similar situation with each of the remaining extension principles discussed in this chapter.

## 3.2 Course of Values Recursion

**3.2.1 Introduction.** In this section we will study a simple generalization of primitive recursion called *course of values recursion.* In this new scheme the value at the $(n{+}1)$-stage depends not only on the value from the previous $n$-th stage but on the values from $<n$-th stages as well. We show that course of values recursion is admissible in PA by reducing it to primitive recursion.

**3.2.2 Example.** The scheme of course of values recursion is best explained with an example. Consider again the function $\mathrm{fib}(n)$ from Par. 1.1.4 which yields the $n$-th element of the sequence of Fibonacci:

$$\begin{aligned}\mathrm{fib}(0) &= 0\\ \mathrm{fib}(1) &= 1\\ \mathrm{fib}(n+2) &= \mathrm{fib}(n+1) + \mathrm{fib}(n).\end{aligned}$$

These three identities can be re-written in a more compact form with the help of the case discrimination function $D$ (see Par. 1.2.17) as follows:

$$\begin{aligned}\mathrm{fib}(0) &= 0\\ \mathrm{fib}(n+1) &= D\big(n, \mathrm{fib}(n) + \mathrm{fib}(n \dot{-} 1), 1\big).\end{aligned}$$

This is not a definition by primitive recursion since the value $\mathrm{fib}(n+1)$ depends not only on the value $\mathrm{fib}(n)$ but on the value $\mathrm{fib}(n \dot{-} 1))$ as well.

[1] This is the only place that we use the principle of mathematical induction for a formula containing the symbol $f$ inside the theory $T'$.

**3.2.3 Extensions by course of values recursion.** Let $T$ be an extension by definitions of PA. Let further

$$\rho[\vec{y}], \tau[x, \vec{z}, \vec{y}], \xi_1[x, \vec{y}], \dots, \xi_k[x, \vec{y}]$$

be terms of $\mathcal{L}_T$ with all their free variables indicated such that

$$T \vdash \xi_1[x, \vec{y}] \leq x \quad \dots \quad T \vdash \xi_k[x, \vec{y}] \leq x. \tag{1}$$

Consider a theory $T'$ obtained from the theory $T$ by adding a new $(n{+}1)$-ary function symbol $f$, the defining axioms

$$f(0, \vec{y}) = \rho[\vec{y}] \tag{2}$$

$$f(x+1, \vec{y}) = \tau\left[x, f\big(\xi_1[x, \vec{y}], \vec{y}\big), \dots, f\big(\xi_k[x, \vec{y}], \vec{y}\big), \vec{y}\right], \tag{3}$$

and the scheme of complete induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of* $T$ *by course of values recursion.*

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 3.2.6 that the theory $T'$ is an extension by definition of the theory $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

*Remark.* The definition can be viewed as a function operator which takes all functions applied in the terms $\rho, \tau, \xi_1, \dots, \xi_k$ and yields the function $f$ as a result. We will prove in Thm. 3.2.7 that the class of primitive recursive functions is closed under the operator of course of values recursion.

**3.2.4 The outline of the proof.** We wish to introduce into $T$ the course of values function $\overline{f}(x, \vec{y})$ yielding the course of values sequence for $f(x, \vec{y})$, i.e. we would like to have

$$\overline{f}(x, \vec{y}) = \big\langle f(x, \vec{y}), f(x-1, \vec{y}), \dots, f(2, \vec{y}), f(1, \vec{y}), f(0, \vec{y}), 0\big\rangle.$$

Note that then the following holds for every $u \leq x$:

$$f(u, \vec{y}) = \big(\overline{f}(x, \vec{y})\big)_{x \dot{-} u}.$$

The function $f$ can be thus defined explicitly by

$$f(x, \vec{y}) = \big(\overline{f}(x, \vec{y})\big)_0.$$

**3.2.5 Course of values function.** We define the $(n{+}1)$-ary course of value function $\overline{f}(x, \vec{y})$ as primitive recursive by the primitive recursive definition:

$$\overline{f}(0,\vec{y}) = \langle \rho[\vec{y}], 0 \rangle$$
$$\overline{f}(x+1,\vec{y}) = \left\langle \tau\left[x, \left(\overline{f}(x,\vec{y})\right)_{x \dot{-} \xi_1[x,\vec{y}]}, \dots, \left(\overline{f}(x,\vec{y})\right)_{x \dot{-} \xi_k[x,\vec{y}]}, \vec{y}\right], \overline{f}(x,\vec{y}) \right\rangle.$$

The following holds for $i = 1, \dots, k$:

$$T \vdash \left(\overline{f}\left(\xi_i[x,\vec{y}],\vec{y}\right)\right)_0 = \left(\overline{f}(x,\vec{y})\right)_{x \dot{-} \xi_i[x,\vec{y}]}. \tag{1}$$

*Proof.* First note that we have

$$T \vdash \left(\overline{f}(x_1 + x_2, \vec{y})\right)_{x_2} = \left(\overline{f}(x_1,\vec{y})\right)_0. \tag{$\dagger_1$}$$

This is proved by induction on $x_2$. The base case is obvious and the induction step follows from

$$\begin{aligned}\left(\overline{f}(x_1 + x_2 + 1, \vec{y})\right)_{x_2+1} &= \left(\left\langle \tau[\dots], \overline{f}(x_1 + x_2, \vec{y}) \right\rangle\right)_{x_2+1} = \\ &= \left(\overline{f}(x_1 + x_2, \vec{y})\right)_{x_2} \overset{\mathrm{IH}}{=} \left(\overline{f}(x_1,\vec{y})\right)_0.\end{aligned}$$

We are now ready to prove (1). We have $\xi_i[x,\vec{y}] \leq x$ by 3.2.3(1) and thus

$$\xi_i[x,\vec{y}] + (x \dot{-} \xi_i[x,\vec{y}]) = x \tag{$\dagger_2$}$$

for every $i = 1, \dots, k$. We obtain

$$\begin{aligned}\left(\overline{f}\left(\xi_i[x,\vec{y}],\vec{y}\right)\right)_0 &\overset{(\dagger_1)}{=} \left(\overline{f}\left(\xi_i[x,\vec{y}] + (x \dot{-} \xi_i[x,\vec{y}]),\vec{y}\right)\right)_{x \dot{-} \xi_i[x,\vec{y}]} \overset{(\dagger_2)}{=} \\ &= \left(\overline{f}(x,\vec{y})\right)_{x \dot{-} \xi_i[x,\vec{y}]}.\end{aligned}$$

□

**3.2.6 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by course of values recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by course of values recursion as in Par. 3.2.3 and $T''$ an extension of $T$ by the explicit definition

$$f(x,\vec{y}) = \left(\overline{f}(x,\vec{y})\right)_0,$$

where $\overline{f}$ is the course of values function from Par. 3.2.5. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$ by Thm. 1.3.19. It suffices to show that both theories $T'$ and $T''$ have the same theorems.

First we show $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA, and therefore, by Thm. 1.4.9, it proves the principle of complete induction for each formula of $\mathcal{L}_{T'}$ containing $f$. It remains to show that $T''$ proves both defining axioms 3.2.3(2)(3) of $f$. The first defining axiom follows from

$$f(0,\vec{y}) = \left(\overline{f}(0,\vec{y})\right)_0 = \left(\langle \rho[\vec{y}], 0 \rangle\right)_0 = \rho[\vec{y}].$$

The second defining axiom follows from

$$\begin{aligned} f(x+1,\vec{y}) &= \big(\overline{f}(x+1,\vec{y})\big)_0 = \\ &= \tau\left[x, \big(\overline{f}(x,\vec{y})\big)_{x \dot{-} \xi_1[x,\vec{y}]}, \dots \big(\overline{f}(x,\vec{y})\big)_{x \dot{-} \xi_k[x,\vec{y}]}, \vec{y}\right] \overset{3.2.5(1)}{=} \\ &= \tau\left[x, \Big(\overline{f}\big(\xi_1[x,\vec{y}],\vec{y}\big)\Big)_0, \dots, \Big(\overline{f}\big(\xi_k[x,\vec{y}],\vec{y}\big)\Big)_0, \vec{y}\right] = \\ &= \tau\left[x, f\big(\xi_1[x,\vec{y}],\vec{y}\big), \dots, f\big(\xi_k[x,\vec{y}],\vec{y}\big), \vec{y}\right]. \end{aligned}$$

Vice versa, in order to show $T' \vdash T''$ it suffices to prove

$$T' \vdash f(x,\vec{y}) = \big(\overline{f}(x,\vec{y})\big)_0.$$

This is proved by complete induction on $x$. There are two cases to consider. If $x = 0$ then we have

$$f(0,\vec{y}) = \rho[\vec{y}] = \big(\langle \rho[\vec{y}], 0\rangle\big)_0 = \big(\overline{f}(0,\vec{y})\big)_0.$$

If $x = x' + 1$ for some $x'$ then $\xi_i[x',\vec{y}] \le x' < x' + 1$ by 3.2.3(1) and therefore

$$\begin{aligned} f(x'+1,\vec{y}) &= \tau\Big[x', f\left(\xi_1[x',\vec{y}],\vec{y}\right), \dots, f\big(\xi_k[x',\vec{y}],\vec{y}\big), \vec{y}\Big] \overset{k\times\mathrm{IH}}{=} \\ &= \tau\left[x', \Big(\overline{f}\big(\xi_1[x',\vec{y}],\vec{y}\big)\Big)_0, \dots, \Big(\overline{f}\big(\xi_k[x',\vec{y}],\vec{y}\big)\Big)_0, \vec{y}\right] \overset{3.2.5(1)}{=} \\ &= \tau\left[x', \big(\overline{f}(x',\vec{y})\big)_{x' \dot{-} \xi_1[x',\vec{y}]}, \dots, \big(\overline{f}(x',\vec{y})\big)_{x' \dot{-} \xi_k[x',\vec{y}]}, \vec{y}\right] = \big(\overline{f}(x'+1,\vec{y})\big)_0. \end{aligned}$$ □

**3.2.7 Theorem** *Primitive recursive functions are closed under course of values recursion.*

*Proof.* By inspection of the proof of Thm. 3.2.6. □

## 3.3 Backward Recursion

**3.3.1 Introduction.** In this section we will study a simple modification of course of values recursion which is called *backward recursion.* In this new scheme the computation goes from 0 to an arbitrary but fixed upper bound. We show that backward recursion is admissible in PA by reducing it to course of values recursion.

**3.3.2 Example.** Suppose that $f$ is defined by

$$f(x,y) = \begin{cases} g(y) & \text{if } x \ge b(y), \\ h\big(x, f(x+1,y), y\big) & \text{if } x < b(y). \end{cases}$$

We have $x < x+1 \leq b(y)$ for $x < b(y)$ and therefore the definition is legal because the value $b(y) \dot{-} x$ decreases for the arguments of the (only) recursive application $f(x+1,y)$:

$$x < b(y) \to b(y) \dot{-} (x+1) < b(y) < x.$$

We say that $f$ is defined by backward recursion on the difference $b(y) \dot{-} x$.

We want to show that if $g$, $h$ and $b$ are all p.r. functions then so is $f$. For that we shall define a new binary function $\hat{f}$ such that

$$v + x = b(y) \to \hat{f}(v,y) = f(x,y). \tag{1}$$

Under the assumption $v + x = b(y)$, if the number $x$ grows from 0 to the upper bound $b(y)$, the number $v$ decreases from $b(y)$ to 0. This suggests the following p.r. derivation of $\hat{f}$. If $0 + x = b(y)$ then $x = b(y)$ and so it must be

$$\hat{f}(0,y) \overset{(1)}{=} f\big(b(y),y\big) = g(y).$$

If $v+1+x = b(y)$ then $v+x+1 = b(y)$ and $x = b(y) \dot{-} (v+1)$, and so it must be

$$\begin{aligned}\hat{f}(v+1,y) \overset{(1)}{=} f(x,y) = h\big(x, f(x+1,y), y\big) &\overset{(1)}{=} \\ = h\big(x, \hat{f}(v,y), y\big) = h\big(b(y) \dot{-} (v+1), \hat{f}(v,y), y\big).\end{aligned}$$

It suffices to define $\hat{f}$ as a p.r. function by

$$\begin{aligned}\hat{f}(0,y) &= g(y) \\ \hat{f}(v+1,y) &= h\big(b(y) \dot{-} (v+1), \hat{f}(v,y), y\big).\end{aligned}$$

Now from (1) we get

$$x \leq b(y) \to f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

If $x \geq b(y)$ then $b(y) \dot{-} x = 0$ and thus $f(x,y) = g(y) = \hat{f}(0,y) = \hat{f}(b(y) \dot{-} x, y)$. This means that we have also

$$x \geq b(y) \to f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

By combining these last two properties together we obtain

$$f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

We can take this identity as an explicit definition of $f$ as a p.r. function.

**3.3.3 Example.** Our second example of backward recursion is the following definition of a binary function $f$:

$$f(x,y) = \begin{cases} g(y) & \text{if } x \geq b(y), \\ h\Big(x, f\big(\xi[x,y],y\big),y\Big) & \text{if } x < b(y), \end{cases}$$

where $x < \xi[x,y]$ for every $x, y$. We clearly have

$$x < b(y) \to b(y) \dot{-} \big(\xi[x,y]+1\big) < b(y) < x$$

and therefore the recursion is legal because the difference $b(y) \dot{-} x$ decreases for the arguments of the recursive application $f(\xi[x,y]+1,y)$.

We want to show that if $g$, $h$, $b$ and $\xi$ are are all primitive recursive then so is $f$. For that we shall define a new binary function $\hat{f}$ such that

$$v + x = b(y) \to \hat{f}(v,y) = f(x,y). \tag{1}$$

If $0 + x = b(y)$ then $x = b(y)$ and so it must be

$$\hat{f}(0,y) \overset{(1)}{=} f\big(b(y),y\big) = g(y).$$

Suppose now $v+1+x = b(y)$ and for simplicity assume that $\xi[x,y] \leq b(y)$. We have $x = b(y) \dot{-} (v+1)$. Let further $\hat{\xi}[v,y]$ be a term defined by

$$\hat{\xi}[v,y] \equiv b(y) \dot{-} \xi\big[b(y) \dot{-} (v+1), y\big].$$

Then $\hat{\xi}[v,y] + \xi[x,y] = b(y)$ and so it must be

$$\begin{aligned} \hat{f}(v+1,y) &\overset{(1)}{=} f(x,y) = h\Big(x, f\big(\xi[x,y],y\big),y\Big) \overset{(1)}{=} \\ &= h\Big(x, \hat{f}\big(\hat{\xi}[v,y],y\big),y\Big) = h\Big(b(y) \dot{-} (v+1), \hat{f}\big(\hat{\xi}[v,y],y\big),y\Big). \end{aligned}$$

Note also that the inequality $\hat{\xi}[v,y] \leq v$ holds as we have

$$\begin{aligned} \hat{\xi}[v,y] &= b(y) \dot{-} \xi\big[b(y) \dot{-} (v+1),y\big] \leq b(y) \dot{-} \Big(\big(b(y) \dot{-} (v+1)\big)+1\Big) \leq \\ &\leq b(y) \dot{-} \big(b(y)+1 \dot{-} (v+1)\big) = b(y) \dot{-} \big(b(y) \dot{-} v\big) \leq v. \end{aligned}$$

The following is a course of values recursive definition of $\hat{f}$ as a p.r. function:

$$\begin{aligned} \hat{f}(0,y) &= g(y) \\ \hat{f}(v+1,y) &= h\Big(b(y) \dot{-} (v+1), \hat{f}\big(\hat{\xi}[v,y],y\big),y\Big) \end{aligned}$$

From (1) we get

$$x \leq b(y) \to f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

If $x \geq b(y)$ then $b(y) \dot{-} x = 0$ and thus $f(x,y) = g(y) = \hat{f}(0,y) = \hat{f}(b(y) \dot{-} x, y)$. Consequently

$$x \geq b(y) \rightarrow f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

By combining these last two properties together we obtain

$$f(x,y) = \hat{f}\big(b(y) \dot{-} x, y\big).$$

We can take this identity as an explicit definition of $f$ as a p.r. function.

**3.3.4 The principle of backward induction.** Properties of functions defined by backward recursion are usually verified by backward induction. The principle of backward induction is formalized within PA as follows.

For every formula $\varphi[x,\vec{y}]$ and term $\theta[\vec{y}]$, the formula of *backward induction on the difference* $\theta[\vec{y}] \dot{-} x$ *for* $\varphi$ is the following one:

$$\forall x \forall \vec{y}\Big(\forall x_1\big(\theta[\vec{y}] \dot{-} x_1 < \theta[\vec{y}] \dot{-} x \rightarrow \varphi[x_1,\vec{y}]\big) \rightarrow \varphi[x,\vec{y}]\Big) \rightarrow \forall x \forall \vec{y} \varphi[x,\vec{y}]. \quad (1)$$

We assume here that the variable $x_1$ is different from $x, \vec{y}$ and does not occur freely in $\varphi$. The formula $\varphi$ and the term $\theta$ may contain additional variables as parameters.

**3.3.5 Theorem** *If* $T$ *is an extension by definitions of* PA *containing* $<$ *then it proves the principle of backward induction for each formula of* $\mathcal{L}_T$.

*Proof.* The principle of backward induction 3.3.4(1) of $\mathcal{L}_T$ is reduced to mathematical induction as follows. Under the assumption

$$\forall x \forall \vec{y}\Big(\forall x_1\big(\theta[\vec{y}] \dot{-} x_1 < \theta[\vec{y}] \dot{-} x \rightarrow \varphi[x_1,\vec{y}]\big) \rightarrow \varphi[x,\vec{y}]\Big) \quad (\dagger_1)$$

we first prove, by induction on $n$, the following auxiliary property

$$\forall v\big(\theta[\vec{w}] \dot{-} v < n \rightarrow \varphi[v,\vec{w}]\big). \quad (\dagger_2)$$

In the base case there is nothing to prove. In the induction step take any $v$ such that $\theta[\vec{w}] \dot{-} v < n+1$ and consider two cases. If $\theta[\vec{w}] \dot{-} v < n$ then we obtain $\varphi[v,\vec{w}]$ by IH. If $\theta[\vec{w}] \dot{-} v = n$ then by instantiating of $(\dagger_1)$ with $x, \vec{y} := v, \vec{w}$ we obtain

$$\forall x_1\big(\theta[\vec{w}] \dot{-} x_1 < n \rightarrow \varphi[x_1,\vec{w}]\big) \rightarrow \varphi[v,\vec{w}].$$

Now we apply IH to get $\varphi[x,\vec{y}]$.

With the auxiliary property proved we obtain that $\varphi[x,\vec{y}]$ holds for every $x, \vec{y}$ by instantiating of $\forall n \forall \vec{w}(\dagger_2)$ with $n, \vec{w}, v := \theta[\vec{y}] \dot{-} x + 1, \vec{y}, x$. □

**3.3.6 Extensions by backward recursion.** Let $T$ be an extension by definitions of PA. Let further

$$\rho[x,\vec{y}],\tau[x,\vec{z},\vec{y}],\theta[\vec{y}],\xi_1[x,\vec{y}],\dots,\xi_k[x,\vec{y}]$$

be terms of $\mathcal{L}_T$ with all their free variables indicated. Suppose that

$$T \vdash x < \xi_1[x,\vec{y}] \quad \dots \quad T \vdash x < \xi_k[x,\vec{y}]. \tag{1}$$

Consider a theory $T'$ obtained from the theory $T$ by adding a new $(n{+}1)$-ary function symbol $f$, the defining axioms

$$x \geq \theta[\vec{y}] \rightarrow f(x,\vec{y}) = \rho[x,\vec{y}] \tag{2}$$

$$x < \theta[\vec{y}] \rightarrow f(x,\vec{y}) = \tau\left[x, f\big(\xi_1[x,\vec{y}],\vec{y}\big),\dots,f\big(\xi_k[x,\vec{y}],\vec{y}\big),\vec{y}\right]. \tag{3}$$

and the scheme of backward induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of T by backward recursion on the difference* $\theta[\vec{y}] \dot{-} x$.

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 3.3.8 that the theory $T'$ is an extension by definition of the theory $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

*Remark.* The definition can be viewed as a function operator which takes all functions applied in the terms $\rho,\tau,\theta,\xi_1,\dots,\xi_k$ and yields the function $f$ as a result. We will prove in Thm. 3.3.9 that the class of primitive recursive functions is closed under the operator of backward recursion.

**3.3.7 Auxiliary function.** We will introduce the function $f$ with the help of an auxiliary $(n{+}1)$-ary function $\hat{f}$ such that

$$v + x = \theta[\vec{y}] \rightarrow \hat{f}(v,\vec{y}) = f(x,\vec{y}).$$

Let $\hat{\tau}[v,\vec{z},\vec{y}],\hat{\xi}_1[v,\vec{y}],\dots,\hat{\xi}_k[v,\vec{y}]$ be terms of $\mathcal{L}_T$ defined by

$$\begin{aligned}
&\hat{\tau}[v, z_1, \dots, z_k, \vec{y}] \equiv \\
&\quad \tau\Big[\theta[\vec{y}] \dot{-} (v+1), \\
&\qquad D\Big(\xi_1\big[\theta[\vec{y}] \dot{-} (v+1), \vec{y}\big] <_* \theta[\vec{y}], z_1, \rho\big[\xi_1[\theta[\vec{y}] \dot{-} (v+1), \vec{y}], \vec{y}\big]\Big), \dots, \\
&\qquad D\Big(\xi_k\big[\theta[\vec{y}] \dot{-} (v+1), \vec{y}\big] <_* \theta[\vec{y}], z_k, \rho\big[\xi_k[\theta[\vec{y}] \dot{-} (v+1), \vec{y}], \vec{y}\big]\Big), \vec{y}\Big] \\
&\hat{\xi}_i[v, \vec{y}] \equiv \theta[\vec{y}] \dot{-} \xi_i\big[\theta[\vec{y}] \dot{-} (v+1), \vec{y}\big].
\end{aligned}$$

For every $i = 1, \dots, k$ we have

$$T \vdash \hat{\xi}_i[v, \vec{y}] \le v. \tag{1}$$

The function $\hat{f}$ is defined by the following course of values recursion

$$\begin{aligned}
\hat{f}(0, \vec{y}) &= \rho\big[\theta[\vec{y}], \vec{y}\big] \\
\hat{f}(v+1, \vec{y}) &= \hat{\tau}\Big[v, \hat{f}\big(\hat{\xi}_1[v, \vec{y}], \vec{y}\big), \dots, \hat{f}\big(\hat{\xi}_k[v, \vec{y}], \vec{y}\big), \vec{y}\Big].
\end{aligned}$$

In the sequel we will need the following property of $\hat{f}$:

$$\begin{aligned}
&T \vdash x < \theta[\vec{y}] \to \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big) = \qquad (2)\\
&\quad = \tau\Big[x, D\Big(\xi_1[x, \vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_1[x, \vec{y}], \vec{y}\big), \rho\big[\xi_1[x, \vec{y}], \vec{y}\big]\Big), \dots, \\
&\qquad D\Big(\xi_k[x, \vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_k[x, \vec{y}], \vec{y}\big), \rho\big[\xi_k[x, \vec{y}], \vec{y}\big]\Big), \vec{y}\Big].
\end{aligned}$$

*Proof.* (1): It follows from

$$\begin{aligned}
\hat{\xi}_i[v, y] &= \theta[y] \dot{-} \xi_i\big[\theta[y] \dot{-} (v+1), y\big] \overset{3.3.6(1)}{\le} \theta[y] \dot{-} \Big(\big(\theta[y] \dot{-} (v+1)\big) + 1\Big) \le \\
&\le \theta[y] \dot{-} \big(\theta[y] + 1 \dot{-} (v+1)\big) = \theta[y] \dot{-} \big(\theta[y] \dot{-} v\big) \le v.
\end{aligned}$$

(2): If $x < \theta[\vec{y}]$ then $x + v + 1 = \theta[\vec{y}]$ for some $v$. We then have

$$\theta[\vec{y}] \dot{-} x = v + 1 \tag{$\dagger_1$}$$
$$\theta[\vec{y}] \dot{-} (v+1) = x \tag{$\dagger_2$}$$

and also

$$\Big(\xi_i\big[\theta[\vec{y}] \dot{-} (v+1), \vec{y}\big] <_* \theta[\vec{y}]\Big) = \big(\xi_i[x, \vec{y}] <_* \theta[\vec{y}]\big) \tag{$\dagger_3$}$$
$$\hat{f}\big(\hat{\xi}_i[v, \vec{y}], \vec{y}\big) = \hat{f}\Big(\theta[\vec{y}] \dot{-} \xi_i\big[\theta[\vec{y}] \dot{-} (v+1), \vec{y}\big], \vec{y}\Big) = \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_i[x, \vec{y}], \vec{y}\big). \tag{$\dagger_4$}$$

We now obtain

$$\begin{aligned}
&\hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big) \overset{(\dagger_1)}{=} \hat{f}\big(\theta[v+1, \vec{y}\big) = \\
&\quad = \hat{\tau}\Big[v, \hat{f}\big(\hat{\xi}_1[v,\vec{y}],\vec{y}\big), \dots, \hat{f}\big(\hat{\xi}_k[v,\vec{y}],\vec{y}\big), \vec{y}\Big] \overset{(\dagger_2),(\dagger_3),(\dagger_4)}{=} \\
&\quad = \tau\bigg[x, D\Big(\xi_1[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_1[x,\vec{y}],\vec{y}\big), \rho\big[\xi_1[x,\vec{y}],\vec{y}\big]\Big), \dots, \\
&\qquad\quad D\Big(\xi_k[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_k[x,\vec{y}],\vec{y}\big), \rho\big[\xi_k[x,\vec{y}],\vec{y}\big]\Big), \vec{y}\bigg].
\end{aligned}$$

This proves (2). □

**3.3.8 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by backward recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by backward recursion as in Par. 3.3.6 and $T''$ an extension of $T$ by the explicit definition

$$f(x,\vec{y}) = D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big),$$

where $\hat{f}$ is the function from Par. 3.3.7. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$ by Thm. 1.3.19. It suffices to show that both theories $T'$ and $T''$ have the same theorems.

We show $T'' \vdash T'$ first. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 3.3.5, it proves the principle of backward induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to derive both defining axioms 3.3.6(2)(3) of $f$ in $T''$. If $x \geq \theta[\vec{y}]$ then we have

$$\begin{aligned}
f(x,\vec{y}) &= D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \\
&= D\Big(0, \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \rho[x,\vec{y}].
\end{aligned}$$

This proves the first defining axiom of $f$ in $T''$. If $x < \theta[\vec{y}]$ then we have

$$\begin{aligned}
&f(x,\vec{y}) = D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \\
&\quad = D\Big(1, \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big) \overset{3.3.7(2)}{=} \\
&\quad = \tau\bigg[x, D\Big(\xi_1[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_1[x,\vec{y}],\vec{y}\big), \rho\big[\xi_1[x,\vec{y}],\vec{y}\big]\Big), \dots, \\
&\qquad\quad D\Big(\xi_k[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_k[x,\vec{y}],\vec{y}\big), \rho\big[\xi_k[x,\vec{y}],\vec{y}\big]\Big), \vec{y}\bigg] = \\
&\quad = \tau\Big[x, f\big(\xi_1[x,\vec{y}],\vec{y}\big), \dots, f\big(\xi_k[x,\vec{y}],\vec{y}\big), \vec{y}\Big].
\end{aligned}$$

This proves the second defining axiom of $f$ in $T''$.

Vice versa, in order to show $T' \vdash T''$ it suffices to prove

$$T' \vdash f(x,\vec{y}) = D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big).$$

The property is proved by backward induction on the difference $\theta[\vec{y}] \dot{-} x$. So take any $x, \vec{y}$ and consider two cases by dichotomy. If $x \geq \theta[\vec{y}]$ then we have

$$\begin{aligned} f(x,\vec{y}) = \rho[x,\vec{y}] = D\Big(0, \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \\ = D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big). \end{aligned}$$

If $x < \theta[\vec{y}]$ then $\theta[\vec{y}] \dot{-} \xi_i[x,\vec{y}] < \theta[\vec{y}] \dot{-} x$ by 3.3.6(1) for all $i = 1, \dots, k$. Thus

$$\begin{aligned} f(x,\vec{y}) &= \tau\Big[x, f\big(\xi_1[x,\vec{y}],\vec{y}\big), \dots, f\big(\xi_k[x,\vec{y}],\vec{y}\big), \vec{y}\Big] \overset{k\times\mathrm{IH}}{=} \\ &= \tau\Bigg[x, D\Big(\xi_1[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_1[x,\vec{y}],\vec{y}\big), \rho\big[\xi_1[x,\vec{y}],\vec{y}\big]\Big), \dots, \\ &\qquad D\Big(\xi_k[x,\vec{y}] <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} \xi_k[x,\vec{y}],\vec{y}\big), \rho\big[\xi_k[x,\vec{y}],\vec{y}\big]\Big), \vec{y}\Bigg] \overset{3.3.7(2)}{=} \\ &= \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big) = D\Big(1, \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big) = \\ &= D\Big(x <_* \theta[\vec{y}], \hat{f}\big(\theta[\vec{y}] \dot{-} x, \vec{y}\big), \rho[x,\vec{y}]\Big). \end{aligned}$$
□

**3.3.9 Theorem** *Primitive recursive functions are closed under backward recursion.*

*Proof.* By inspection of the proof of Thm. 3.3.8. □

## 3.4 Recursion with Parameter Substitution

**3.4.1 Introduction.** In this section we will investigate in PA recursive definitions for which parameters change in recursive applications. This new scheme is called recursion with parameter substitution.

**3.4.2 Example of primitive recursion with parameter substitution.** Our first example of recursion with parameter substitution is the efficient implementation of the sequence of Fibonacci (see Par. 1.1.4). Recall that the fast program for the Fibonacci function

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

is obtained with the help of the auxiliary ternary function $g$ defined by

$$\begin{aligned} g(0,a,b) &= a \\ g(n+1,a,b) &= g(n,a+b,a). \end{aligned}$$

Note that the value $g(n+1,a,b)$ depends on the value $g(n,a+b,a)$ from the previous stage, where the terms $a+b$ and $a$ has been substituted for the parameters $a$ and $b$, respectively. We say that the function $g(n,a,b)$ is defined by primitive recursion on $n$ with substitution in the parameters $a$ and $b$.

**3.4.3 Example of course of values recursion with parameter substitution.** The second example is the algorithm of Euclid for computing of the greatest common divisor of two numbers:

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

which relies on the following property of divisibility:

$$\vdash_{\mathrm{PA}} y \neq 0 \to z \mid x \land z \mid y \leftrightarrow z \mid y \land z \mid x \bmod y.$$

These two equations have a form of a course of values recursive definition because the first argument decreases in the recursive application:

$$y \bmod (x+1) \leq x < x+1.$$

The only difference between the above definition and the one discussed in Sect. 3.2 is that in this case the parameter changes in recursion. Namely, the term $x+1$ is substituted for the parameter $y$ in the second equation. We say that the function $\gcd(x,y)$ is defined by course of values recursion on $x$ with substitution in the parameter $y$.

**3.4.4 Extensions by recursion with substitution in parameters.** Let $T$ be an extension by definitions of PA and $f$ a new $(n{+}1)$-ary function symbol. Let further

$$\rho[\vec{y}], \tau[x,\vec{z},\vec{y}], \xi_1[x,\vec{y}], \vec{\sigma}_1[x,\vec{y}], \ldots, \xi_k[x,\vec{y}], \vec{\sigma}_k[x,\vec{y}]$$

be terms of $\mathcal{L}_T$ in which no other variables than the indicated ones are free such that

$$T \vdash \xi_1[x,\vec{y}] \leq x \quad \ldots \quad T \vdash \xi_k[x,\vec{y}] \leq x.$$

Consider the theory $T'$ obtained from the theory $T$ by adding the function symbol $f$, the defining axioms of $f$

$$f(0,\vec{y}) = \rho[\vec{y}]$$
$$f(x+1,\vec{y}) = \tau\left[x, f\big(\xi_1[x,\vec{y}],\vec{\sigma}_1[x,\vec{y}]\big),\dots,f\big(\xi_k[x,\vec{y}],\vec{\sigma}_k[x,\vec{y}]\big),\vec{y}\right].$$

and the scheme of mathematical induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of $T$ by (course of values) recursion with parameter substitution.*

The assertion that $T'$ is an extension by definition of $T$ will be proved at the end of this section. The proof proceeds in stages. First, we show that the scheme of primitive recursion with parameter substitution with two recursive applications ($k = 2$) and one parameter ($n = 1$) is admissible in PA. This is proved in Thm. 3.4.14 by reducing the scheme to backward recursion. Next, we extend the result to primitive recursion with arbitrary number of recursive applications (Thm. 3.4.18) and parameters (Thm. 3.4.22). Finally, we show how to reduce course of values recursion with parameter substitution to primitive recursion (Thm. 3.4.27).

*Remark.* The definition can be viewed as a function operator which takes all functions applied in the terms $\rho,\tau,\xi_1,\vec{\sigma}_1,\dots,\xi_k,\vec{\sigma}_k$ and yields the function $f$ as a result. We will prove in Thm. 3.4.28 that the class of primitive recursive functions is closed under the operator of course of values recursion with parameter substitution.

## *Primitive Recursion with Parameter Substitution: Case $k = 2$ and $n = 1$*

**3.4.5 Introduction.** In this subsection we will investigate the scheme of primitive recursion with substitution in one parameter ($n = 1$), where only two different recursive applications are allowed ($k = 2$). The admissibility of the scheme in PA will be shown in Thm. 3.4.14.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by primitive recursion with parameter substitution with the defining axioms

$$f(0,y) = \rho[y] \tag{1}$$
$$f(x+1,y) = \tau\big[x, f\big(x,\sigma_1[x,y]\big), f\big(x,\sigma_2[x,y]\big), y\big]. \tag{2}$$

Here $f$ is a new binary function symbol. We claim that $T'$ is an extension of $T$ by definitions. We will prove this fact by reducing the scheme to backward recursion.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.4.6 The outline of of the proof.** We will introduce the function $f$ into the theory $T$ by the arithmetization of computation trees for $f$ in which we use as computational rules its defining axioms 3.4.5(1)(2). The computation of the application $f(x,y)$ can be visualized as a binary tree with labels consisting of all applications $f(x_i,y_i)$ which are needed to compute the value $f(x,y)$.

Binary trees are coded as follows. The empty tree is coded by the number 0. A non-empty tree is coded by the number $\langle z,l,r\rangle$, where $z$ is the label of its root node, and $l$ and $r$ are the codes of its left and right subtrees, respectively. Note that if $t$ is the code of a non-empty tree then the label of its root node is the first projection of $t$, i.e. the number $\pi_1(t)$.

We intend to introduce $f$ into $T$ with the help of its course of values function $\overline{f}$. The function $\overline{f}(x,y)$ yields the computation tree for the application $f(x,y)$, i.e. we would like to have

$$\begin{aligned}\overline{f}(0,y) &= \big\langle f(0,y),0,0\big\rangle\\ \overline{f}(x+1,y) &= \Big\langle f(x+1,y),\overline{f}\big(x,\sigma_1[x,y]\big),\overline{f}\big(x,\sigma_2[x,y]\big)\Big\rangle\end{aligned}$$

in the standard model. The function $f$ can then be defined explicitly by

$$f(x,y) = \pi_1\overline{f}(x,y)$$

as primitive recursive.

Figure 3.1 shows an example of computation tree $\overline{f}(x,y)$ for the application $f(x,y)$. Note that each node of the tree has the form $f(x_i,y_i)$ for some word $i$ over two-symbol alphabet $\Sigma = \{1,2\}$. The dyadic word $i$ represents the path from the root to that node in obvious manner. Note that we can easily recover the arguments of the application $f(x_i,y_i)$ from the arguments of the application $f(x,y)$ and the path $i$. Clearly, the recursive argument $x_i$ is the difference between $x$ and the and the length of the dyadic word $i$. Moreover

$$y_\varnothing = y \qquad y_{i1} = \sigma_1[x_{i1},y_i] \qquad y_{i2} = \sigma_2[x_{i2},y_i].$$

This gives us simple recurrences for computing the parameter $y_i$.

By the results of Par. 1.1.5, there is simple correspondence between dyadic words and dyadic representation of natural numbers. Recall that every natural number has a unique representation as a dyadic numeral which are terms built up from the constant 0 by applications of the dyadic successors

$$x\mathbf{1} = 2x+1 \qquad x\mathbf{2} = 2x+2.$$

Dyadic representation allows simple coding of dyadic words into natural numbers. For instance, the code number of the dyadic word 221 is the number

$$0\mathbf{221} = 2\times(2\times(2\times 0+2)+2)+1 = 2\times 2^2+2\times 2^1+1\times 2^0 = 13.$$

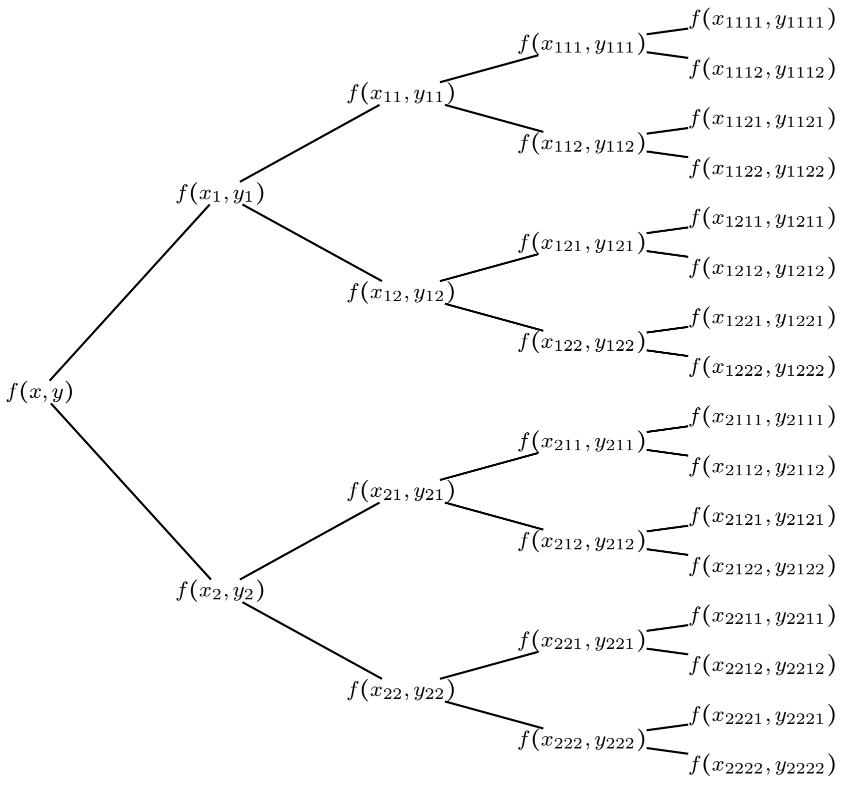

**Fig. 3.1** Computation tree of depth 5

Arithmetization is so straightforward that, from now on, we will usually identify dyadic words with their code numbers.

We intend to compute $\overline{f}(x,y)$ from bottom-up using backward recursion. This is done with the help of the course of values subtree function $\overline{f}(x,y).i$ which returns the subtree of the computation tree for $f(x,y)$ at position indexed by the dyadic path $i$. That is, we would like to have

$$\pi_1 \overline{f}(x,y).i = \overline{f}(x_i,y_i)$$

for every dyadic path $i$ in the computation tree $\overline{f}(x,y)$. Hence

$$\overline{f}(x,y) = \pi_1 \overline{f}(x,y).0$$

and we can take the identity as an explicit definition of the course of values function. Note that 0 is the code number of the empty dyadic word $\varnothing$.

**3.4.7 Dyadic case analysis.** Note that we have

$$\vdash_{\mathrm{PA}} x = 0 \lor \exists y\, x = x\mathbf{1} \lor \exists y\, x = x\mathbf{2}. \tag{1}$$

This is called the principle of *dyadic case analysis on the number $x$.*

**3.4.8 Dyadic size.** The unary function $|x|_{\mathrm{d}}$ yields the number of dyadic successors in the dyadic representation of $x$. The dyadic size function satisfies

$$\vdash_{\mathrm{PA}} \quad |0|_{\mathrm{d}} = 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} |x\mathbf{1}|_{\mathrm{d}} = |x|_{\mathrm{d}} + 1 \tag{2}$$
$$\vdash_{\mathrm{PA}} |x\mathbf{2}|_{\mathrm{d}} = |x|_{\mathrm{d}} + 1 \tag{3}$$

and it is defined by course of values recursion as a p.r. function by

$$|0|_{\mathrm{d}} = 0$$
$$|x+1|_{\mathrm{d}} = |x \div 2|_{\mathrm{d}} + 1.$$

The following property will be needed later:

$$|x|_{\mathrm{d}} < n \leftrightarrow x + 1 < 2^n. \tag{4}$$

In the sequel we will tacitly use the properties (1)-(3) of dyadic size.

*Proof.* (1): Directly from definition. (2): It follows from

$$|x\mathbf{1}|_{\mathrm{d}} = |2x+1|_{\mathrm{d}} = |2x \div 2|_{\mathrm{d}} + 1 = |x|_{\mathrm{d}} + 1.$$

(3): This is proved similarly.

(4): By complete induction on $x$ as $\forall n(4)$. In the induction step take any $n$ and consider three cases according to 3.4.7(1). If $x = 0$ then we have

$$|0|_{\mathrm{d}} < n \Leftrightarrow 0 < n \Leftrightarrow 1 < 2^n \Leftrightarrow 0 + 1 < 2^n.$$

If $x = y\mathbf{1}$ for some $y$ then $y < x$ and we obtain

$$|y\mathbf{1}|_{\mathrm{d}} < n \overset{(2)}{\Leftrightarrow} |y|_{\mathrm{d}} + 1 < n \Leftrightarrow \exists m\,(n = m+1 \land |y|_{\mathrm{d}} + 1 < m+1) \Leftrightarrow$$
$$\Leftrightarrow \exists m\,(n = m+1 \land |y|_{\mathrm{d}} < m) \overset{\mathrm{IH}}{\Leftrightarrow} \exists m\,(n = m+1 \land y + 1 < 2^m) \Leftrightarrow$$
$$\Leftrightarrow \exists m\left(n = m+1 \land 2y + 2 < 2^{m+1}\right) \Leftrightarrow 2y + 2 < 2^n \Leftrightarrow y\mathbf{1} + 1 < 2^n.$$

The case when $x = y\mathbf{2}$ for some $y$ has a similar proof. □

**3.4.9 Dyadic concatenation.** The binary function $x \star y$ yields a number which dyadic representation is obtained from the dyadic representations of $x$ and $y$ by appending the digits of $y$ after the digits of $x$. The dyadic concatenation function $x \star y$ satisfies the identities

$$\vdash_{\mathrm{PA}} \quad x \star 0 = x \tag{1}$$
$$\vdash_{\mathrm{PA}} x \star y\mathbf{1} = (x \star y)\mathbf{1} \tag{2}$$
$$\vdash_{\mathrm{PA}} x \star y\mathbf{2} = (x \star y)\mathbf{2} \tag{3}$$

and it is defined explicitly as a p.r. function by

$$x \star y = x2^{|y|_{\mathrm{d}}} + y.$$

We will need the following distributive property:

$$|x \star y|_{\mathrm{d}} = |x|_{\mathrm{d}} + |y|_{\mathrm{d}}\,. \tag{4}$$

In the sequel we will tacitly use the properties (1)-(3) of dyadic concatenation.

*Proof.* (1): We have $x \star 0 = x2^{|0|_{\mathrm{d}}} + 0 = x2^0 = x$. (2): It follows from

$$x \star y\mathbf{1} = x2^{|y\mathbf{1}|_{\mathrm{d}}} + y\mathbf{1} = x2^{|y|_{\mathrm{d}}+1} + 2y + 1 = 2\left(x2^{|y|_{\mathrm{d}}} + y\right) + 1 = (x \star y)\mathbf{1}.$$

(3): This is proved similarly.

(4): By complete induction on $y$. In the induction step we consider three cases according to 3.4.7(1). If $y = 0$ then

$$|x \star 0|_{\mathrm{d}} \overset{(1)}{=} |x|_{\mathrm{d}} = |x|_{\mathrm{d}} + 0 = |x|_{\mathrm{d}} + |0|_{\mathrm{d}}\,.$$

If $y = z\mathbf{1}$ for some $z$ then $z < y$ and we obtain

$$|x \star z\mathbf{1}|_{\mathrm{d}} \overset{(2)}{=} |(x \star z)\mathbf{1}|_{\mathrm{d}} = |x \star z|_{\mathrm{d}} + 1 \overset{\mathrm{IH}}{=} |x|_{\mathrm{d}} + |z|_{\mathrm{d}} + 1 = |x|_{\mathrm{d}} + |z\mathbf{1}|_{\mathrm{d}}\,.$$

The case when $y = z\mathbf{2}$ for some $z$ has a similar proof. □

**3.4.10 Selector function for recursive arguments.** By $\mathbf{x}_i(x)$ we denote the binary function which computes the recursive argument of the recursive application of $f$ at position indexed by the dyadic path $i$ in the computation tree for $f(x, y)$. The function satisfies

$$\vdash_{\mathrm{PA}} \mathbf{x}_0(x) = x \tag{1}$$
$$\vdash_{\mathrm{PA}} \mathbf{x}_{i\mathbf{1}}(x) = \mathbf{x}_i(x) \dot{-} 1 \tag{2}$$
$$\vdash_{\mathrm{PA}} \mathbf{x}_{i\mathbf{2}}(x) = \mathbf{x}_i(x) \dot{-} 1 \tag{3}$$

and it is defined explicitly as a p.r. function by

$$\mathbf{x}_i(x) = x \dot{-} |i|_{\mathrm{d}}\,.$$

We will need the following *composition* property of the selector function:

$$\vdash_{\mathrm{PA}} \mathbf{x}_{i\star j}(x) = \mathbf{x}_j\mathbf{x}_i(x). \tag{4}$$

In the sequel we will tacitly use the properties (1)-(3).

*Proof.* (1): We have $\mathbf{x}_0(x) = x \dot{-} |0|_\mathrm{d} = x \dot{-} 0 = x$. (2): It follows from

$$\mathbf{x}_{i\mathbf{1}}(x) = x \dot{-} |i\mathbf{1}|_\mathrm{d} = x \dot{-} (|i|_\mathrm{d} + 1) = x \dot{-} |i|_\mathrm{d} \dot{-} 1 = \mathbf{x}_i(x) \dot{-} 1.$$

(3): This is proved similarly. (4): It follows from

$$\begin{aligned}\mathbf{x}_{i\star j}(x) &= x \dot{-} |i \star j|_\mathrm{d} \overset{3.4.9(4)}{=} x \dot{-} (|i|_\mathrm{d} + |j|_\mathrm{d}) = x \dot{-} |i|_\mathrm{d} \dot{-} |j|_\mathrm{d} = \\ &= \mathbf{x}_i(x) \dot{-} |j|_\mathrm{d} = \mathbf{x}_j\mathbf{x}_i(x).\end{aligned}$$

□

**3.4.11 Selector function for parameters.** The ternary function $\mathbf{y}_i(x, y)$ computes the parameter of the recursive application of $f$ at position indexed by the dyadic path $i$ in the computation tree for $f(x, y)$. The function satisfies

$$T \vdash \mathbf{y}_0(x, y) = y \quad (1)$$
$$T \vdash \mathbf{y}_{i\mathbf{1}}(x, y) = \sigma_1[\mathbf{x}_{i\mathbf{1}}(x), \mathbf{y}_i(x, y)] \quad (2)$$
$$T \vdash \mathbf{y}_{i\mathbf{2}}(x, y) = \sigma_2[\mathbf{x}_{i\mathbf{2}}(x), \mathbf{y}_i(x, y)] \quad (3)$$

and it is defined by course of values recursion on $i$ as a p.r. function:

$$\begin{aligned}&\mathbf{y}_0(x, y) = y \\ &\mathbf{y}_{i+1}(x, y) = D\big((i+1) \bmod 2, \sigma_1[\mathbf{x}_{i+1}(x), \mathbf{y}_{i\div 2}(x, y)], \sigma_2[\mathbf{x}_{i+1}(x), \mathbf{y}_{i\div 2}(x, y)]\big).\end{aligned}$$

The following is the *composition* property of the parameter selector function:

$$T \vdash \mathbf{y}_{i\star j}(x, y) = \mathbf{y}_j\big(\mathbf{x}_i(x), \mathbf{y}_i(x, y)\big). \quad (4)$$

In the sequel we will use the properties (1)-(3) without explicit reference.

*Proof.* (1): From definition. (2): We have $(2i + 1) \bmod 2 = 1 \neq 0$ and therefore

$$\mathbf{y}_{i\mathbf{1}}(x, y) = \mathbf{y}_{2i+1}(x, y) = \sigma_1[\mathbf{x}_{2i+1}(x), \mathbf{y}_{2i\div 2}(x, y)] = \sigma_1[\mathbf{x}_{i\mathbf{1}}(x), \mathbf{y}_i(x, y)].$$

(3): This is proved similarly. (4): By complete induction on $j$. In the induction step we consider three cases according to 3.4.7(1). If $j = 0$ then

$$\mathbf{y}_{i\star 0}(x, y) = \mathbf{y}_i(x, y) = \mathbf{y}_0\big(\mathbf{x}_i(x), \mathbf{y}_i(x, y)\big).$$

If $j = k\mathbf{1}$ for some $k$ then $k < j$ and we obtain

$$\begin{aligned}\mathbf{y}_{i\star k\mathbf{1}}(x, y) &= \mathbf{y}_{(i\star k)\mathbf{1}}(x, y) \overset{(2)}{=} \sigma_1[\mathbf{x}_{(i\star k)\mathbf{1}}(x), \mathbf{y}_{i\star k}(x, y)] = \\ &= \sigma_1[\mathbf{x}_{i\star k\mathbf{1}}(x), \mathbf{y}_{i\star k}(x, y)] \overset{3.4.10(4)}{=} \sigma_1[\mathbf{x}_{k\mathbf{1}}\mathbf{x}_i(x), \mathbf{y}_{i\star k}(x, y)] \overset{\mathrm{IH}}{=} \\ &= \sigma_1\left[\mathbf{x}_{k\mathbf{1}}\mathbf{x}_i(x), \mathbf{y}_k\big(\mathbf{x}_i(x), \mathbf{y}_i(x, y)\big)\right] \overset{(2)}{=} \mathbf{y}_{k\mathbf{1}}\big(\mathbf{x}_i(x), \mathbf{y}_i(x, y)\big).\end{aligned}$$

The case when $j = k\mathbf{2}$ for some $k$ is similar. □

**3.4.12 Course of values subtree function.** The ternary function $\overline{f}(x,y).i$ returns the subtree of the computation tree for $f(x,y)$ at position indexed by the dyadic path $i$. It has the following basic properties:

$$T \vdash |i|_{\mathrm{d}} = x \to \overline{f}(x,y).i = \big\langle \rho[\mathbf{y}_i(x,y)],0,0\big\rangle \tag{1}$$

$$\begin{aligned} T \vdash |i|_{\mathrm{d}} < x \to \overline{f}(x,y).i = & \\ & \Big\langle \tau\big[\mathbf{x}_i(x) \mathbin{\dot{-}} 1, \pi_1\big(\overline{f}(x,y).i\mathbf{1}\big), \pi_1\big(\overline{f}(x,y).i\mathbf{2}\big), \mathbf{y}_i(x,y)\big], \\ & \overline{f}(x,y).i\mathbf{1}, \overline{f}(x,y).i\mathbf{2}\Big\rangle. \end{aligned} \tag{2}$$

The course of values subtree function $\overline{f}(x,y).i$ for $f$ is defined by backward recursion on the difference $2^x \mathbin{\dot{-}} 1 \mathbin{\dot{-}} i$ as a p.r. function by

$$\begin{aligned} i \geq 2^x \mathbin{\dot{-}} 1 \to \overline{f}(x,y).i &= \big\langle \rho[\mathbf{y}_i(x,y)],0,0\big\rangle \\ i < 2^x \mathbin{\dot{-}} 1 \to \overline{f}(x,y).i &= \Big\langle \tau\big[\mathbf{x}_i(x) \mathbin{\dot{-}} 1, \pi_1\big(\overline{f}(x,y).i\mathbf{1}\big), \pi_1\big(\overline{f}(x,y).i\mathbf{2}\big), \mathbf{y}_i(x,y)\big], \\ & \qquad \overline{f}(x,y).i\mathbf{1}, \overline{f}(x,y).i\mathbf{2}\Big\rangle. \end{aligned}$$

The *composition* property of the course of values subtree function is

$$T \vdash |i \star j|_{\mathrm{d}} \leq x \to \overline{f}(x,y).(i \star j) = \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big).j. \tag{3}$$

*Proof.* (1),(2): Directly from definition by noting that

$$i < 2^x \mathbin{\dot{-}} 1 \Leftrightarrow i+1 < 2^x \overset{3.4.8(4)}{\Leftrightarrow} |i|_{\mathrm{d}} < x.$$

(3): By backward induction on the difference $x \mathbin{\dot{-}} |j|_{\mathrm{d}}$. So take any $i,j,x$ such that $|i \star j|_{\mathrm{d}} \leq x$ and consider two cases. If $|i \star j|_{\mathrm{d}} = x$ then $|i|_{\mathrm{d}} + |j|_{\mathrm{d}} = x$ by 3.4.9(4) and therefore $|j|_{\mathrm{d}} = x \mathbin{\dot{-}} |i|_{\mathrm{d}} = \mathbf{x}_i(x)$. We obtain

$$\begin{aligned} \overline{f}(x,y).(i \star j) &\overset{(1)}{=} \big\langle \rho[\mathbf{y}_{i\star j}(x,y)],0,0\big\rangle \overset{3.4.11(4)}{=} \Big\langle \rho\big[\mathbf{y}_j\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big)\big],0,0\Big\rangle \overset{(1)}{=} \\ &= \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big).j \end{aligned}$$

So suppose that $|i \star j|_{\mathrm{d}} < x$. We then have $|i|_{\mathrm{d}} + |j|_{\mathrm{d}} < x$ by 3.4.9(4) and therefore $|j|_{\mathrm{d}} < x \mathbin{\dot{-}} |i|_{\mathrm{d}} = \mathbf{x}_i(x)$ and

$$|i \star j\mathbf{1}|_{\mathrm{d}} \overset{3.4.9(4)}{=} |i|_{\mathrm{d}} + |j\mathbf{1}|_{\mathrm{d}} = |i|_{\mathrm{d}} + |j|_{\mathrm{d}} + 1 \leq x.$$

From $|j|_{\mathrm{d}} < x$ we obtain $|j|_{\mathrm{d}} < |j|_{\mathrm{d}} + 1 = |j\mathbf{1}|_{\mathrm{d}} \leq x$ and thus $x \mathbin{\dot{-}} |j\mathbf{1}|_{\mathrm{d}} < x \mathbin{\dot{-}} |j|_{\mathrm{d}}$. Therefore

$$\begin{aligned}
&\overline{f}(x,y).(i\star j) \overset{(2)}{=}\\
&= \Big\langle \tau\big[\mathbf{x}_{i\star j}(x) \dot- 1, \pi_1\big(\overline{f}(x,y).(i\star j)\mathbf{1}\big), \pi_1\big(\overline{f}(x,y).(i\star j)\mathbf{2}\big), \mathbf{y}_{i\star j}(x,y)\big],\\
&\qquad \overline{f}(x,y).(i\star j)\mathbf{1}, \overline{f}(x,y).(i\star j)\mathbf{2}\Big\rangle \overset{3.4.10(4),3.4.11(4)}{=}\\
&= \Big\langle \tau\big[\mathbf{x}_j\mathbf{x}_i(x) \dot- 1, \pi_1\big(\overline{f}(x,y).(i\star j\mathbf{1})\big), \pi_1\big(\overline{f}(x,y).(i\star j\mathbf{2})\big),\\
&\qquad \mathbf{y}_j\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big)\big], \overline{f}(x,y).(i\star j\mathbf{1}), \overline{f}(x,y).(i\star j\mathbf{2})\Big\rangle \overset{2\times\mathrm{IH}}{=}\\
&= \Big\langle \tau\big[\mathbf{x}_j\mathbf{x}_i(x) \dot- 1, \pi_1\big(\overline{f}(\mathbf{x}_i(x), \mathbf{y}_i(x,y)).j\mathbf{1}\big), \pi_1\big(\overline{f}(\mathbf{x}_i(x), \mathbf{y}_i(x,y)).j\mathbf{2}\big),\\
&\qquad \mathbf{y}_j\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big)\big], \overline{f}(\mathbf{x}_i(x), \mathbf{y}_i(x,y)).j\mathbf{1}, \overline{f}(\mathbf{x}_i(x), \mathbf{y}_i(x,y)).j\mathbf{2}\Big\rangle \overset{(2)}{=}\\
&= \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big).j. \qquad \square
\end{aligned}$$

**3.4.13 Course of values function.** The binary function $\overline{f}(x,y)$ returns the computation tree for $f(x,y)$. The function satisfies

$$T \vdash \quad \overline{f}(0,y) = \big\langle \rho[y], 0, 0\big\rangle \tag{1}$$

$$\begin{aligned}T \vdash \overline{f}(x+1,y) = \Big\langle &\tau\big[x, \pi_1\overline{f}\big(x,\sigma_1[x,y]\big), \pi_1\overline{f}\big(x,\sigma_2[x,y]\big), y\big],\\ &\overline{f}\big(x,\sigma_1[x,y]\big), \overline{f}\big(x,\sigma_2[x,y]\big)\Big\rangle\end{aligned} \tag{2}$$

and it is defined explicitly with the help of the course of values subtree function $\overline{f}(x,y).i$ for $f$ as follows

$$\overline{f}(x,y) = \overline{f}(x,y).0.$$

*Proof.* (1): We have $|0|_{\mathrm{d}} = 0$ and thus

$$\overline{f}(0,y) = \overline{f}(0,y).0 \overset{3.4.12(1)}{=} \big\langle \rho[\mathbf{y}_0(0,y)], 0, 0\big\rangle = \big\langle \rho[y], 0, 0\big\rangle.$$

(2): First note that the following holds

$$T \vdash |i|_{\mathrm{d}} \le x \to \overline{f}(x,y).i = \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big). \tag{$\dagger_1$}$$

Indeed, if $|i|_{\mathrm{d}} \le x$ then $|i\star 0|_{\mathrm{d}} = |i|_{\mathrm{d}} \le x$ and therefore

$$\overline{f}(x,y).i = \overline{f}(x,y).(i\star 0) \overset{3.4.12(3)}{=} \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big).0 = \overline{f}\big(\mathbf{x}_i(x), \mathbf{y}_i(x,y)\big).$$

Further note that we have also

$$\mathbf{x}_{0\mathbf{1}}(x+1) = \mathbf{x}_0(x+1) \dot- 1 = x+1 \dot- 1 = x \tag{$\dagger_2$}$$

$$\mathbf{y}_{0\mathbf{1}}(x+1,y) = \sigma_1\big[\mathbf{x}_{0\mathbf{1}}(x+1), \mathbf{y}_0(x+1,y)\big] \overset{(\dagger_2)}{=} \sigma_1[x,y]\ . \tag{$\dagger_3$}$$

We have $|0|_{\mathrm{d}} = 0 < x + 1$ and therefore

$$\begin{aligned}
&\overline{f}(x+1,y) = \overline{f}(x+1,y).0 \overset{3.4.12(2)}{=} \\
&= \Big\langle \tau\big[\mathbf{x}_0(x+1) \dot{-} 1, \pi_1\big(\overline{f}(x+1,y).0\mathbf{1}\big), \pi_1\big(\overline{f}(x+1,y).0\mathbf{2}\big), \mathbf{y}_0(x+1,y)\big], \\
&\qquad \overline{f}(x+1,y).0\mathbf{1}, \overline{f}(x+1,y).0\mathbf{2}\Big\rangle = \\
&= \Big\langle \tau\big[x, \pi_1\big(\overline{f}(x+1,y).0\mathbf{1}\big), \pi_1\big(\overline{f}(x+1,y).0\mathbf{2}\big), y\big], \\
&\qquad \overline{f}(x+1,y).0\mathbf{1}, \overline{f}(x+1,y).0\mathbf{2}\Big\rangle \overset{(\dagger_1)}{=} \\
&= \Big\langle \tau\big[x, \pi_1\overline{f}\big(\mathbf{x}_{0\mathbf{1}}(x+1), \mathbf{y}_{0\mathbf{1}}(x+1,y)\big), \pi_1\overline{f}\big(\mathbf{x}_{0\mathbf{2}}(x+1), \mathbf{y}_{0\mathbf{2}}(x+1,y)\big), y\big], \\
&\qquad \overline{f}\big(\mathbf{x}_{0\mathbf{1}}(x+1), \mathbf{y}_{0\mathbf{1}}(x+1,y)\big), \overline{f}\big(\mathbf{x}_{0\mathbf{2}}(x+1), \mathbf{y}_{0\mathbf{2}}(x+1,y)\big)\Big\rangle \overset{(\dagger_2),(\dagger_3)}{=} \\
&= \Big\langle \tau\big[x, \pi_1\overline{f}\big(x, \sigma_1[x,y]\big), \pi_1\overline{f}\big(x, \sigma_2[x,y]\big), y\big], \overline{f}\big(x, \sigma_1[x,y]\big), \overline{f}\big(x, \sigma_2[x,y]\big)\Big\rangle.
\end{aligned}$$

This proves the property (2). □

**3.4.14 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by primitive recursion with parameter substitution for the case $k = 2$ and $n = 1$ is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by recursion with parameter substitution as in Par. 3.4.5 and $T''$ an extension of $T$ by the explicit definition

$$f(x,y) = \pi_1\overline{f}(x,y),$$

where $\overline{f}$ is the course of values function for $f$ from Par. 3.4.13. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$ by Thm. 1.3.19. It suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA and thus, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing $f$. It remains to show that $T''$ proves both defining axioms 3.4.5(1)(2) of $f$. The first defining axiom follows from

$$f(0,y) = \pi_1\overline{f}(0,y) \overset{3.4.13(1)}{=} \pi_1\langle\rho[y],0,0\rangle = \rho[y].$$

The second defining axiom follows from

$$\begin{aligned}
f(x+1,y) = \pi_1\overline{f}(x+1,y) &\overset{3.4.13(2)}{=} \tau\big[x, \pi_1\overline{f}\big(x,\sigma_1[x,y]\big), \pi_1\overline{f}\big(x,\sigma_2[x,y]\big), y\big] = \\
&= \tau\big[x, f\big(x,\sigma_1[x,y]\big), f\big(x,\sigma_2[x,y]\big), y\big].
\end{aligned}$$

Vice versa, in order to show $T' \vdash T''$ it suffices to prove

$$T' \vdash \forall y\, f(x,y) = \pi_1 \overline{f}(x,y).$$

This is proved by induction on $x$. In the base case take any $y$ and we have

$$f(0,y) = \rho[y] = \pi_1\langle g(y),0,0\rangle \overset{3.4.13(1)}{=} \pi_1\overline{f}(0,y).$$

In the induction step take any $y$ and we obtain

$$\begin{aligned} f(x+1,y) &= \tau\big[x, f\big(x,\sigma_1[x,y]\big), f\big(x,\sigma_2[x,y]\big), y\big] \overset{2\times\mathrm{IH}}{=} \\ &= \tau\big[x, \pi_1\overline{f}\big(x,\sigma_1[x,y]\big), \pi_1\overline{f}\big(x,\sigma_2[x,y]\big), y\big] \overset{3.4.13(2)}{=} \pi_1\overline{f}(x+1,y). \end{aligned}$$ □

**3.4.15 Theorem** *Primitive recursive functions are closed under primitive recursion with parameter substitution for the case $k = 2$ and $n = 1$.*

*Proof.* By inspection of the proof of Thm. 3.4.14. □

## *Primitive Recursion with Parameter Substitution: Case $n = 1$*

**3.4.16 Introduction.** In this subsection we will show that the scheme of primitive recursion with substitution in one parameter is admissible in PA. This will be proved in Thm. 3.4.18 by reducing the number of recursive applications. This leads eventually to primitive recursion with substitution in one parameter and two recursive applications.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by primitive recursion with parameter substitution with the defining axioms

$$f(0,y) = \rho[y] \tag{1}$$

$$f(x+1,y) = \tau\left[x, f\big(x,\sigma_1[x,y]\big),\dots, f\big(x,\sigma_{k+1}[x,y]\big), y\right]. \tag{2}$$

Here $f$ is a new binary function symbol. We claim that $T'$ is an extension of $T$ by definitions.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.4.17 Reduction of the number of recursive applications.** We will reduce the above definition for $k \geq 2$ to a new one, where only $k$ recursive applications are allowed. This new definition is for a binary function $\hat{f}(u,v)$ such that $\hat{f}(2x,y) = f(x,y)$ and it is of the form

$$\begin{aligned}\hat{f}(0,v) &= \rho[v]\\ \hat{f}(u+1,v) &= \hat{\tau}\left[u,\hat{f}\big(u,\hat{\sigma}_1[u,v]\big),\dots,\hat{f}\big(u,\hat{\sigma}_k[u,v]\big),v\right]\end{aligned}$$

for suitable terms $\hat{\tau}[u,\vec{w},v],\hat{\sigma}_1[u,v],\dots,\hat{\sigma}_k[u,v]$. Note that this is primitive recursion on $u$ with substitution in the parameter $v$ with $k$ recursive applications. We will take then the identity $f(x,y)=\hat{f}(2x,y)$ as an alternative, explicit definition of $f$.

The idea behind reduction of recursive applications is as follows. We would like to have $\hat{f}(2x,y)=f(x,y)$ and so it must be

$$T \vdash \hat{f}\big(2(x+1),y\big) = \tau\left[x,\hat{f}\big(2x,\sigma_1[x,y]\big),\dots,\hat{f}\big(2x,\sigma_{k+1}[x,y]\big),y\right]. \tag{1}$$

For the values of the form $\hat{f}(2x+1,v)$ we require

$$\begin{aligned}\hat{f}\big(2x+1,\langle 1,y\rangle\big) &= \Big\langle f\big(x,\sigma_1[x,y]\big),\dots,f\big(x,\sigma_k[x,y]\big)\Big\rangle\\ \hat{f}\big(2x+1,\langle 2,y\rangle\big) &= f\big(x,\sigma_{k+1}[x,y]\big).\end{aligned}$$

Note that the application $\hat{f}\big(2x+1,\langle 1,y\rangle\big)$ returns a number which codes $k$ values $f\big(x,\sigma_1[x,y]\big),\dots,f\big(x,\sigma_k[x,y]\big)$ of the function $f$.

These informal arguments can be rewritten without mentioning the function $f$ as follows:

$$T \vdash \hat{f}\big(2x+1,\langle 1,y\rangle\big) = \Big\langle \hat{f}\big(2x,\sigma_1[x,y]\big),\dots,\hat{f}\big(2x,\sigma_k[x,y]\big)\Big\rangle \tag{2}$$

$$T \vdash \hat{f}\big(2x+1,\langle 2,y\rangle\big) = \hat{f}\big(2x,\sigma_{k+1}[x,y]\big). \tag{3}$$

$$\begin{aligned}T \vdash \quad \hat{f}\big(2x+2,y\big) = \tau\Big[x,&\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]^k_1,\dots,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]^k_k,\\ &\hat{f}\big(2x+1,\langle 2,y\rangle\big),y\Big]\end{aligned} \tag{4}$$

This means that the terms $\hat{\tau},\hat{\sigma}_1,\dots,\hat{\sigma}_k$. satisfy the properties

$$T \vdash \hat{\tau}[2x+1,z,z_{k+1},\dots,y] = \tau\left[x,[z]^k_1,\dots,[z]^k_k,z_{k+1},y\right] \tag{5}$$

$$T \vdash \hat{\tau}\big[2x,z_1,\dots,z_k,\langle 1,y\rangle\big] = \langle z_1,\dots,z_k\rangle \tag{6}$$

$$T \vdash \hat{\tau}\big[2x,z_{k+1},\dots,\langle 2,y\rangle\big] = z_{k+1} \tag{7}$$

and

$$T \vdash \qquad \hat{\sigma}_1[2x+1,y] = \langle 1,y\rangle \tag{8}$$

$$T \vdash \qquad \hat{\sigma}_2[2x+1,y] = \langle 2,y\rangle \tag{9}$$

$$T \vdash \qquad \bigwedge_{i=1}^{k} \hat{\sigma}_i\big[2x, \langle 1, y\rangle\big] = \sigma_i[x, y] \tag{10}$$

$$T \vdash \qquad \hat{\sigma}_1\big[2x, \langle 2, y\rangle\big] = \sigma_{k+1}[x, y]. \tag{11}$$

For that it is sufficient to set

$$\begin{aligned}\hat{\tau}[u, w_1, \dots, w_k, v] \equiv D\Big(&u \bmod 2, \tau\big[u \div 2, [w_1]_1^k, \dots, [w_1]_k^k, w_2, v\big],\\ &D\big(\pi_1(v) =_* 1, \langle w_1, \dots, w_k\rangle, w_1\big)\Big)\end{aligned}$$

and

$$\begin{aligned}\hat{\sigma}_1[u, v] &\equiv D\Bigg(u \bmod 2, \langle 1, v\rangle,\\ &\qquad D\Big(\pi_1(v) =_* 1, \sigma_1\big[u \div 2, \pi_2(v)\big], \sigma_{k+1}\big[u \div 2, \pi_2(v)\big]\Big)\Bigg)\\ \hat{\sigma}_2[u, v] &\equiv D\Big(u \bmod 2, \langle 2, v\rangle, \sigma_2\big[u \div 2, \pi_2(v)\big]\Big)\\ \hat{\sigma}_i[u, v] &\equiv \sigma_i\big[u \div 2, \pi_2(v)\big] \quad \text{for } i = 3, \dots, k.\end{aligned}$$

*Proof.* (5)-(11): Directly from definition. (2): It follows from

$$\begin{aligned}&\hat{f}\big(2x+1, \langle 1, y\rangle\big) =\\ &= \hat{\tau}\Big[2x, \hat{f}\Big(2x, \hat{\sigma}_1\big[2x, \langle 1, y\rangle\big]\Big), \dots, \hat{f}\Big(2x, \hat{\sigma}_k\big[2x, \langle 1, y\rangle\big]\Big), \langle 1, y\rangle\Big] \overset{(10)}{=}\\ &= \hat{\tau}\big[2x, \hat{f}\big(2x, \sigma_1[x, y]\big), \dots, \hat{f}\big(2x, \sigma_k[x, y]\big), \langle 1, y\rangle\big] \overset{(6)}{=}\\ &= \Big\langle \hat{f}\big(2x, \sigma_1[x, y]\big), \dots, \hat{f}\big(2x, \sigma_k[x, y]\big)\Big\rangle.\end{aligned}$$

(3): It follows from

$$\begin{aligned}\hat{f}\big(2x+1, \langle 2, y\rangle\big) &= \hat{\tau}\Big[2x, \hat{f}\Big(2x, \hat{\sigma}_1\big[2x, \langle 2, y\rangle\big]\Big), \dots, \langle 2, y\rangle\Big] \overset{(11)}{=}\\ &= \hat{\tau}\big[2x, \hat{f}\big(2x, \sigma_{k+1}[x, y]\big), \dots, \langle 2, y\rangle\big] \overset{(7)}{=} \hat{f}\big(2x, \sigma_{k+1}[x, y]\big).\end{aligned}$$

(4): It follows from

$$\begin{aligned}&\hat{f}(2x+2, y) = \hat{f}(2x+1+1, y) =\\ &= \hat{\tau}\Big[2x+1, \hat{f}\big(2x+1, \hat{\sigma}_1[2x+1, y]\big), \hat{f}\big(2x+1, \hat{\sigma}_2[2x+1, y]\big), \dots, y\Big] \overset{(8),(9)}{=}\\ &= \hat{\tau}\Big[2x+1, \hat{f}\big(2x+1, \langle 1, y\rangle\big), \hat{f}\big(2x+1, \langle 2, y\rangle\big), \dots, y\Big] \overset{(5)}{=}\\ &= \tau\Big[x, \big[\hat{f}\big(2x+1, \langle 1, y\rangle\big)\big]_1^k, \dots, \big[\hat{f}\big(2x+1, \langle 1, y\rangle\big)\big]_k^k, \hat{f}\big(2x+1, \langle 2, y\rangle\big), y\Big].\end{aligned}$$

We are now in position to prove (1):

$$\begin{aligned}
&\hat{f}\big(2(x+1),y\big) = \hat{f}(2x+2,y) \overset{(4)}{=} \\
&= \tau\Big[x,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_1^k,\dots,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_k^k,\hat{f}\big(2x+1,\langle 2,y\rangle\big),y\Big] \overset{(2),(3)}{=} \\
&= \tau\big[x,\hat{f}\big(2x,\sigma_1[x,y]\big),\dots,\hat{f}\big(2x,\sigma_k[x,y]\big),\hat{f}\big(2x,\sigma_{k+1}[x,y]\big),y\big]. \qquad \square
\end{aligned}$$

**3.4.18 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by primitive recursion with parameter substitution for the case $n=1$ is an extension by definition.*

*Proof.* The claim is proved by (meta-)induction on the number $k$ of recursive applications in the defining axiom 3.4.16(2). The case $k=0$ is in fact explicit definition with monadic discrimination on the first argument and it follows from Thm. 3.1.10. The cases $k=1$ or $k=2$ follow from Thm. 3.4.14. So suppose that the claim holds for the case $k\geq 2$. We will prove that the claim holds also for the case $k+1$.

Let $T'$ be an extension of $T$ by primitive recursion with parameter substitution as in Par. 3.4.16 and $T''$ an extension of $T$ by the explicit definition

$$f(x,y) = \hat{f}(2x,y),$$

where $\hat{f}$ is from Par. 3.4.17. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$ by (meta-)IH. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show that $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.4.16(1)(2) of $f$. The first defining axiom follows from

$$f(0,y) = \hat{f}(2\times 0,y) = \hat{f}(0,y) = \rho[y].$$

The second defining axiom follows from

$$\begin{aligned}
f(x+1,y) = \hat{f}\big(2(x+1),y\big) &\overset{3.4.17(1)}{=} \\
&= \tau\big[x,\hat{f}\big(2x,\sigma_1[x,y]\big),\dots,\hat{f}\big(2x,\sigma_{k+1}[x,y]\big),y\big] = \\
&= \tau\big[x,f\big(x,\sigma_1[x,y]\big),\dots,f\big(x,\sigma_{k+1}[x,y]\big),y\big].
\end{aligned}$$

Now we show that $T' \vdash T''$. For that it suffices to prove

$$T' \vdash \forall y\, f(x,y) = \hat{f}(2x,y).$$

This is proved by induction on $x$. In the base case take any $y$ and we have

$$f(0,y) = \rho[y] = \hat{f}(0,y) = \hat{f}(2 \times 0, y).$$

In the induction step take any $y$ and we obtain

$$\begin{aligned} f(x+1,y) &= \tau\big[x, f\big(x, \sigma_1[x,y]\big), \dots, f\big(x, \sigma_{k+1}[x,y]\big), y\big] \overset{(k+1)\times\mathrm{IH}}{=} \\ &= \tau\big[x, \hat{f}\big(2x, \sigma_1[x,y]\big), \dots, \hat{f}\big(2x, \sigma_{k+1}[x,y]\big), y\big] \overset{3.4.17(1)}{=} \\ &= \hat{f}\big(2(x+1), y\big). \end{aligned}$$ □

**3.4.19 Theorem** *Primitive recursive functions are closed under primitive recursion with parameter substitution for the case $n = 1$.*

*Proof.* By inspection of the proof of Thm. 3.4.18. □

## *Primitive Recursion with Parameter Substitution*

**3.4.20 Introduction.** In this subsection we will show that the scheme of primitive recursion with substitution in arbitrary number of parameters is admissible in PA. This will be proved in Thm. 3.4.22 by reducing it to primitive recursion with substitution in one parameter.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by primitive recursion with parameter substitution with the defining axioms

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

$$f(x+1,\vec{y}) = \tau\left[x, f\big(x, \vec{\sigma}_1[x,\vec{y}]\big), \dots, f\big(x, \vec{\sigma}_k[x,\vec{y}]\big), \vec{y}\right]. \tag{2}$$

Here $f$ is a new $(n{+}1)$-ary function symbol ($n \geq 1$). We claim that $T'$ is an extension of $T$ by definitions.

Below we will consider the case when the definition has at least two parameters, i.e. $n \geq 2$. The case $n = 0$ is in fact parameterless primitive recursion for which the claim has been already proved in Thm. 3.1.10. The case with one parameter ($n = 1$) follows from Thm. 3.4.18.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.4.21 Contraction of parameters.** We will reduce the above scheme, where $n \geq 2$, to a new one for a binary function $\langle f\rangle(x,y)$ so that

$$\langle f\rangle(x,y) = f\left(x, [y]^n_1, \dots, [y]^n_n\right).$$

The $n$ parameters $\vec{y} \equiv y_1, \dots, y_n$ are replaced by a single parameter $y$. We will call the number $y = \langle\vec{y}\rangle \equiv \langle y_1, \dots, y_n\rangle$ the *contraction* of the numbers $\vec{y}$.

The *contraction* function $\langle f\rangle(x,y)$ is defined by primitive recursion on $x$ with substitution in the (only) parameter $y$ as a p.r. function by

$$\begin{aligned}\langle f\rangle(0,y) &= \rho\big[[y]^n_1,\dots,[y]^n_n\big]\\ \langle f\rangle(x+1,y) &= \tau\Big[x,\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_1\big[x,[y]^n_1,\dots,[y]^n_n\big]\Big\rangle\Big),\dots,\\ &\qquad\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_k\big[x,[y]^n_1,\dots,[y]^n_n\big]\Big\rangle\Big),[y]^n_1,\dots,[y]^n_n\Big].\end{aligned}$$

**3.4.22 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by primitive recursion with parameter substitution is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by primitive recursion with parameter substitution as in Par. 3.4.20, where the number of parameters is at least two $(n \geq 2)$.[2] Let further $T''$ be an extension of $T$ by the explicit definition

$$f(x,\vec y) = \langle f\rangle\big(x,\langle\vec y\rangle\big),$$

where $\langle f\rangle$ is the contraction function of $f$ from Par. 3.4.21. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$ by Thm. 3.4.18. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

We show $T'' \vdash T'$ first. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.4.20(1)(2) of $f$. The first defining axiom follows from

$$f(0,\vec y) = \big\langle f\big\rangle\big(0,\langle\vec y\rangle\big) = \rho\big[[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\big] \overset{2.3.5(1)}{=} \rho[\vec y].$$

The second defining axiom follows from

$$\begin{aligned}f(x+1,\vec y) &= \big\langle f\big\rangle\big(x+1,\langle\vec y\rangle\big) =\\ &= \tau\Bigg[x,\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_1\big[x,[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\big]\Big\rangle\Big),\dots,\\ &\qquad\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_k\big[x,[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\big]\Big\rangle\Big),[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\Bigg] \overset{2.3.5(1)}{=}\\ &= \tau\Big[x,\Big\langle f\Big\rangle\Big(x,\big\langle\vec\sigma_1[x,\vec y]\big\rangle\Big),\dots,\Big\langle f\Big\rangle\Big(x,\big\langle\vec\sigma_k[x,\vec y]\big\rangle\Big),\vec y\Big] =\\ &= \tau\Big[x,f\big(x,\vec\sigma_1[x,\vec y]\big),\dots,f\big(x,\vec\sigma_k[x,\vec y]\big),\vec y\Big].\end{aligned}$$

Now we show that $T' \vdash T''$. For that it suffices to prove

$$T' \vdash \forall\vec y\, f(x,\vec y) = \big\langle f\big\rangle\big(x,\langle\vec y\rangle\big).$$

[2] The cases $n = 0$ or $n = 1$ follow from Thm. 3.1.10 or Thm. 3.4.16, respectively.

This is proved by induction on $x$. In the base case take any $\vec{y}$ and we have

$$f(0,\vec{y}) = \rho[\vec{y}] \overset{2.3.5(1)}{=} \rho\big[[\langle\vec{y}\rangle]^n_1,\dots,[\langle\vec{y}\rangle]^n_n\big] = \big\langle f\big\rangle\big(0,\langle\vec{y}\rangle\big).$$

In the induction step take any $\vec{y}$ and we obtain

$$\begin{aligned} f(x+1,\vec{y}) &= \tau\big[x, f\big(x,\vec{\sigma}_1[x,\vec{y}]\big),\dots,f\big(x,\vec{\sigma}_k[x,\vec{y}]\big),\vec{y}\big] \overset{k\times\mathrm{IH}}{=} \\ &= \tau\Big[x,\Big\langle f\Big\rangle\Big(x,\big\langle\vec{\sigma}_1[x,\vec{y}]\big\rangle\Big),\dots,\Big\langle f\Big\rangle\Big(x,\big\langle\vec{\sigma}_k[x,\vec{y}]\big\rangle\Big),\vec{y}\Big] \overset{2.3.5(1)}{=} \\ &= \tau\Bigg[x,\Big\langle f\Big\rangle\Big(x,\Big\langle\vec{\sigma}_1\big[x,[\langle\vec{y}\rangle]^n_1,\dots,[\langle\vec{y}\rangle]^n_n\big]\Big\rangle\Big),\dots, \\ &\qquad \Big\langle f\Big\rangle\Big(x,\Big\langle\vec{\sigma}_k\big[x,[\langle\vec{y}\rangle]^n_1,\dots,[\langle\vec{y}\rangle]^n_n\big]\Big\rangle\Big),[\langle\vec{y}\rangle]^n_1,\dots,[\langle\vec{y}\rangle]^n_n\Bigg] = \\ &= \big\langle f\big\rangle\big(x+1,\langle\vec{y}\rangle\big). \end{aligned}$$

□

**3.4.23 Theorem** *Primitive recursive functions are closed under primitive recursion with parameter substitution.*

*Proof.* By inspection of the proof of Thm. 3.4.22. □

## *Course of Values Recursion with Parameter Substitution*

**3.4.24 Introduction.** In this subsection we will show that the general scheme of course of values recursion with parameter substitution is admissible in PA. This will be proved in Thm. 3.4.27 by it to primitive recursion with parameter substitution.

The notation used in this subsection is fixed as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by course of values recursion with parameter substitution with the defining axioms

$$f(0,\vec{y}) = \rho[\vec{y}] \tag{1}$$
$$f(x+1,\vec{y}) = \tau\big[x, f\big(\xi_1[x,\vec{y}],\vec{\sigma}_1[x,\vec{y}]\big),\dots,f\big(\xi_k[x,\vec{y}],\vec{\sigma}_k[x,\vec{y}]\big),\vec{y}\big], \tag{2}$$

where

$$T \vdash \xi_1[x,\vec{y}] \le x \quad \dots \quad T \vdash \xi_k[x,\vec{y}] \le x. \tag{3}$$

Here $f$ is a new $(n+1)$-ary function symbol. We claim that $T'$ is an extension of $T$ by definitions.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.4.25 Approximation function.** We will introduce the function $f(x,\vec{y})$ into the theory $T$ with the help of its *approximation* function $f^+(z,x,\vec{y})$. The additional argument $z$ plays the role of the depth of recursion counter. It estimates the depth of recursion needed to compute the value $f(x,\vec{y})$. If $z$ is sufficiently large then we have $f^+(z,x,\vec{y}) = f(x,\vec{y})$. As we will see below every number $z > x$ gives us sufficient estimation of the depth of recursion. This will allow us to introduce $f$ into PA explicitly by $f(x,\vec{y}) = f^+(x+1,x,\vec{y})$.

The $(n+2)$-ary function $f^+(z,x,\vec{y})$ satisfies

$$T \vdash \quad f^+(0,x,\vec{y}) = 0 \tag{1}$$

$$T \vdash \quad f^+(z+1,0,\vec{y}) = \rho[\vec{y}] \tag{2}$$

$$\begin{aligned} T \vdash f^+(z+1,x+1,\vec{y}) = \tau\Big[x, &f^+\big(z,\xi_1[x,\vec{y}],\vec{\sigma}_1[x,\vec{y}]\big),\dots, \\ &f^+\big(z,\xi_k[x,\vec{y}],\vec{\sigma}_k[x,\vec{y}]\big),\vec{y}\Big], \end{aligned} \tag{3}$$

and it is defined by primitive recursion on $z$ with substitution in the parameters $x,\vec{y}$ as a p.r. function by

$$\begin{aligned} f^+(0,x,\vec{y}) &= 0 \\ f^+(z+1,x,\vec{y}) &= D\Bigg(x,\tau\Big[x \dot{-} 1, f^+\big(z,\xi_1[x \dot{-} 1,\vec{y}],\vec{\sigma}_1[x \dot{-} 1,\vec{y}]\big),\dots, \\ &\qquad f^+\big(z,\xi_k[x \dot{-} 1,\vec{y}],\vec{\sigma}_k[x \dot{-} 1,\vec{y}]\big),\vec{y}\Big],\rho[\vec{y}]\Bigg). \end{aligned}$$

**3.4.26 Monotonicity of the approximation function.** We have

$$T \vdash x < z_1 \land x < z_2 \to f^+(z_1,x,\vec{y}) = f^+(z_2,x,\vec{y}). \tag{1}$$

The property asserts that the application $f^+(z,x,\vec{y})$ yields the same result for every number $z > x$.

*Proof.* The property is proved by induction on $z_1$ as $\forall x \forall \vec{y} \forall z_2(1)$. In the base case there is nothing to prove. In the induction step take any numbers $x,\vec{y},z_2$ such that $x < z_1 + 1$ and $x < z_2$. Then $z_2 = z_2' + 1$ for some $z_2'$. We consider two cases. If $x = 0$ then we have

$$f^+(z_1+1,0,\vec{y}) \overset{3.4.25(2)}{=} \rho[\vec{y}] \overset{3.4.25(2)}{=} f^+(z_2'+1,0,\vec{y}).$$

If $x = x' + 1$ for some $x'$ then $\xi_i[x',\vec{y}] \le x' < z_1$ and $\xi_i[x',\vec{y}] \le x' < z_2'$ for every $i = 1,\dots,k$ by 3.4.24(3), We obtain

$$\begin{aligned} &f^+(z_1+1,x'+1,\vec{y}) \overset{3.4.25(3)}{=} \\ &\quad = \tau\Big[x', f^+\big(z_1,\xi_1[x',\vec{y}],\vec{\sigma}_1[x',\vec{y}]\big),\dots,f^+\big(z_1,\xi_k[x',\vec{y}],\vec{\sigma}_k[x',\vec{y}]\big),\vec{y}\Big] \overset{k\times\text{IH}}{=} \end{aligned}$$

$$= \tau\Big[x', f^+\big(z_2', \xi_1[x',\vec{y}], \vec{\sigma}_1[x',\vec{y}]\big), \ldots, f^+\big(z_2', \xi_k[x',\vec{y}], \vec{\sigma}_k[x',\vec{y}]\big), \vec{y}\Big] \overset{3.4.25(3)}{=}$$
$$= f^+(z_2' + 1, x' + 1, \vec{y}). \qquad \square$$

**3.4.27 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by course of values recursion with parameter substitution is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by course of values recursion with parameter substitution as in Par. 3.4.24. Let further $T''$ be an extension of $T$ by the explicit definition

$$f(x,\vec{y}) = f^+(x+1, x, \vec{y}),$$

where $f^+$ is the approximation function of $f$ from Par. 3.4.25. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$ by Thm. 1.3.18. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show that $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.4.24(1)(2) of $f$. The first defining axiom follows from

$$f(0,\vec{y}) = f^+(0+1, 0, \vec{y}) \overset{3.4.25(2)}{=} \rho[\vec{y}].$$

The second defining axiom is proved as follows. Recall that by 3.4.24(3) we have $\xi_i[x,\vec{y}] < x + 1$ for every $i = 1, \ldots, k$. We then obtain

$$\begin{aligned}
&f(x+1,\vec{y}) = f^+(x+1+1, x+1, \vec{y}) \overset{3.4.25(3)}{=} \\
&= \tau\Big[x, f^+\big(x+1, \xi_1[x,\vec{y}], \vec{\sigma}_1[x,\vec{y}]\big), \ldots, \\
&\qquad f^+\big(x+1, \xi_k[x,\vec{y}], \vec{\sigma}_k[x,\vec{y}]\big), \vec{y}\Big] \overset{3.4.26(1)}{=} \\
&= \tau\Big[x, f^+\big(\xi_1[x,\vec{y}]+1, \xi_1[x,\vec{y}], \vec{\sigma}_1[x,\vec{y}]\big), \ldots, \\
&\qquad f^+\big(\xi_k[x,\vec{y}]+1, \xi_k[x,\vec{y}], \vec{\sigma}_k[x,\vec{y}]\big), \vec{y}\Big] = \\
&= \tau\Big[x, f\big(\xi_1[x,\vec{y}], \vec{\sigma}_1[x,\vec{y}]\big), \ldots, f\big(\xi_k[x,\vec{y}], \vec{\sigma}_k[x,\vec{y}]\big), \vec{y}\Big].
\end{aligned}$$

We now show that $T' \vdash T''$. For that it suffices to prove

$$T' \vdash \forall \vec{y}\, f(x,\vec{y}) = f^+(x+1, x, \vec{y}).$$

This is proved by complete induction on $x$. So take any $\vec{y}$ and consider two cases. If $x = 0$ then we have

$$f(0,\vec{y}) = \rho[\vec{y}] \overset{3.4.25(2)}{=} f^+(0+1,0,\vec{y}).$$

If $x = x'+1$ for some $x'$ then $\xi_i[x',\vec{y}] < x'+1$ for every $i = 1,\dots,k$ by 3.4.24(3), We then obtain

$$\begin{aligned}
f(x'+1,\vec{y}) &= \tau\Big[x', f\big(\xi_1[x',\vec{y}],\vec{\sigma}_1[x',\vec{y}]\big),\dots, f\big(\xi_k[x',\vec{y}],\vec{\sigma}_k[x',\vec{y}]\big),\vec{y}\Big] \overset{k\times\mathrm{IH}}{=} \\
&= \tau\Big[x', f^+\big(\xi_1[x',\vec{y}]+1,\xi_1[x',\vec{y}],\vec{\sigma}_1[x',\vec{y}]\big),\dots, \\
&\qquad f^+\big(\xi_k[x',\vec{y}]+1,\xi_k[x',\vec{y}],\vec{\sigma}_k[x',\vec{y}]\big),\vec{y}\Big] \overset{3.4.26(1)}{=} \\
&= \tau\Big[x', f^+\big(x'+1,\xi_1[x',\vec{y}],\vec{\sigma}_1[x',\vec{y}]\big),\dots, \\
&\qquad f^+\big(x'+1,\xi_k[x',\vec{y}],\vec{\sigma}_k[x',\vec{y}]\big),\vec{y}\Big] \overset{3.4.25(3)}{=} \\
&= f^+(x'+1+1,x'+1,\vec{y}).
\end{aligned}$$
□

**3.4.28 Theorem** *Primitive recursive functions are closed under course of values recursion with parameter substitution.*

*Proof.* By inspection of the proof of Thm. 3.4.27. □

## 3.5 Nested Simple Recursion

**3.5.1 Introduction.** In this section we will investigate recursive definitions for which parameters may be arbitrarily substituted for, even with nested recursive applications. For instance:

$$\begin{aligned}
f(0,y) &= g(y) \\
f(x+1,y) &= h\Big(x, f\big(x,\sigma[x,y,f(x,y)]\big),y\Big).
\end{aligned}$$

The function $f$ is an example of a 1-*recursive* function in the hierarchy of *multiply recursive* functions. Péter has proved in [39, 40] that primitive recursive functions are closed under 1-recursion. The scheme of 1-recursion is usually called nested simple recursion and we will also adopt this convention.

**3.5.2 Notation.** Let $\tau[f]$ be a term which may apply an $n$-ary function symbol $f$ and $\vec{x}$ pairwise different $n$ variables. We will use the *special lambda notation* $\tau[\dot{\lambda}\vec{x}.\rho[\vec{x}]]$, for the term obtained from $\tau$ by the replacement of all applications $f(\vec{\theta})$ in it by terms $\rho[\vec{\theta}]$.

**3.5.3 Extensions by nested simple recursion.** Let $T$ be an extension by definitions of PA and $f$ a new $(n+1)$-ary function symbol. Let further $\rho[\vec{y}]$ be a term of $\mathcal{L}_T$ and $\tau[f;x,\vec{y}]$ a term of the extended language $\mathcal{L}_T \cup \{f\}$ in which no other variables than the indicated ones are free. Consider the theory $T'$ obtained from $T$ by adding the symbol $f$, the defining axioms

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

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

and the scheme of mathematical induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of $T$ by nested simple recursion.*

We will show at the end of this section in Thm. 3.5.19 that the theory $T'$ is an extension by definition of $T$. The proof proceeds in stages. First, we prove the claim for the scheme with with two recursive applications ($k = 2$) and one parameter ($n = 1$). This is proved in Thm. 3.5.11 by reducing the scheme to course of values recursion with parameter substitution. Next, we extend this result to the scheme with arbitrary number of recursive applications (Thm. 3.5.15). Finally, we prove the claim for the scheme with arbitrary number of parameters (Thm. 3.5.19).

We may assume that the function $f$ is applied in $\tau$ at least once because otherwise there would be nothing to prove. In order to simplify our discussion we transform the equation (2) into equivalent one by 'unnesting' all recursive applications of $f$ in the term $\tau$:

$$\bigwedge_{i=1}^{k} f(x,\vec{\sigma}_i[x,\vec{y},\vec{z}_{i-1}]) = z_i \to f(x+1,\vec{y}) = \theta[x,\vec{z},\vec{y}]. \tag{3}$$

Here $\vec{z}_i$ abbreviates $z_1,\dots,z_i$ and the terms $\sigma_1,\dots,\sigma_k,\theta$ contain at most the indicated variables and do not apply $f$.

*Remark.* The definition can be viewed as a function operator which takes all auxiliary functions applied in the terms $\rho,\tau$ and yields the function $f$ as a result. We will prove in Thm. 3.5.20 that the class of primitive recursive functions is closed under the operator of nested simple recursion.

## *Nested Simple Recursion: Case $k = 2$ and $n = 1$*

**3.5.4 Introduction.** In this subsection we will investigate the scheme of nested simple recursion with two different recursive applications ($k = 2$) and one parameter ($n = 1$). The admissibility of the scheme in PA will be shown in Thm. 3.5.11.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by nested simple recursion with the defining axioms

$$f(0,y) = \rho[y] \tag{1}$$
$$f(x+1,y) = \theta\big[x, f\big(x,\sigma_1[x,y]\big), f\big(x,\sigma_2[x,y,f(x,\sigma_1[x,y])]\big), y\big]. \tag{2}$$

Here $f$ is a new binary function symbol. We claim that $T'$ is an extension of $T$ by definitions. We will prove this fact by reducing the above scheme to previous recursive schemes.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.5.5 The outline of the proof.** We will introduce the function $f$ into the theory $T$ by arithmetization of its computation trees in which we use as computational rules the defining axioms 3.5.4(1)(2). The evaluation of the application $f(x,y)$ can be visualized as a full binary tree of depth $x+1$ with labels consisting of all applications $f(x_i,y_i)$ which are needed to compute the value $f(x,y)$.

Binary trees are coded as follows. The empty tree is coded by the number 0. A non-empty tree is coded by the number $\langle z,l,r\rangle$, where $z$ is the label of its root node, and $l$ and $r$ are the codes of its left and right subtree, respectively. Note that if $t$ is the code of a non-empty tree then the label of its root node is the first projection of $t$, i.e. the number $\pi_1(t)$.

We intend to introduce $f$ with the help of its course of values function $\overline{f}$. The function $\overline{f}(x,y)$ yields the computation tree for the application $f(x,y)$:

$$\overline{f}(0,y) = \big\langle f(0,y),0,0\big\rangle$$
$$\overline{f}(x+1,y) = \Big\langle f(x+1,y), \overline{f}\big(x,\sigma_1[x,y]\big), \overline{f}\big(x,\sigma_2\big[x,y,f(x,\sigma_1[x,y])\big]\big)\Big\rangle.$$

We will introduce the course of values function into the theory $T$ as a p.r. function. Hence, the following explicit definition $f(x,y) = \pi_1\overline{f}(x,y)$. introduces $f$ into $T$ as a p.r. function.

Note that the natural evaluation strategy for calculating $f(x,y)$ corresponds to postorder traversal of the computation tree $\overline{f}(x,y)$ for $f(x,y)$. Consider, for instance, the computation tree from Fig. 3.2. The sequence

$$f(x_0,y_0), f(x_1,y_1), f(x_2,y_2), \ldots, f(x_i,y_i), \ldots \tag{1}$$

of its labels consists of all applications which are needed to compute its root value. Note that the parameter $y_i$ of each application $f(x_i,y_i)$ depends only on those values $f(x_j,y_j)$ which are directly before it ($j < i$). This means that the order in which the sequence (1) is sorted corresponds to postorder traversal of the computation tree. We will extend this indexing scheme also for binary trees (already shown in Fig. 3.2).

Let us now consider a finite sequence of full binary trees of depth $x+1$

$$t_0, t_1, t_2, \ldots, t_i, \ldots, t_{2^{x+1} \dot{-} 1}$$

where each tree $t_{i+1}$ satisfies the following condition: *the subtree of* $t_{i+1}$ *at position* $i$ *is a computation tree for* $f(x_i, y_i)$. We will call such trees *partial* computation trees for $f(x,y)$. Note that the last tree $t_{2^{x+1}\dot{-}1}$ is in fact a (full) computation tree for $f(x,y)$.

This suggests the following method for building the computation tree for $f(x,y)$. We start by creating a 'dummy' full binary tree $t_0$ of depth $x+1$. Suppose now that after $i < 2^{x+1}$-steps we have a partial computation tree $t_i$ for $f(x,y)$, The tree is updated at position $i$ by the value $f(x_i,y_i)$ whereby we obtain a new partial computation tree $t_{i+1}$ for $f(x,y)$. After $2^{x+1}$ steps we obtain a full computation tree for $f(x,y)$.

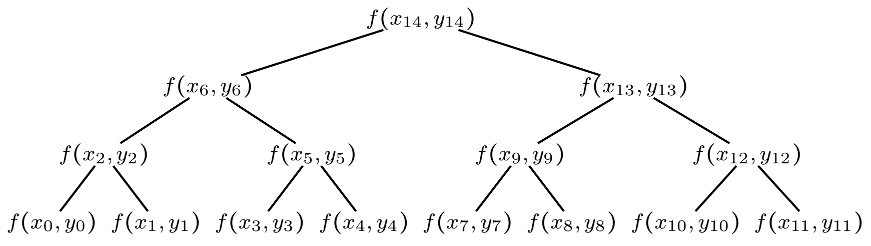

**Fig. 3.2** Postorder traversal of a computation tree of depth 4

**3.5.6 Full binary trees.** The function *Full*$(n)$ creates a full binary tree of the depth $n$. The function is defined by primitive recursion as a p.r. function:

$$\begin{aligned} \mathit{Full}(0) &= 0 \\ \mathit{Full}(n+1) &= \langle 0, \mathit{Full}(n), \mathit{Full}(n)\rangle. \end{aligned}$$

**3.5.7 Local node condition.** The application $V(x,y,l,r)$ determines the correct value $f(x,y)$ from the subtrees $l$ and $r$ of a partial computation tree $\langle z,l,r\rangle$ for $f(x,y)$. The function is primitive recursive by the following explicit definition (with monadic discrimination):

$$\begin{aligned} V(0,y,l,r) &= \rho[y] \\ V(x+1,y,l,r) &= \theta[x, \pi_1(l), \pi_1(r), y] \end{aligned}$$

**3.5.8 Local update.** The 4-ary function $U(t,i,x,y)$ updates the partial computation tree $t$ for $f(x,y)$ at position $i$ by the expected value $f(x_i,y_i)$. The function has the following basic properties

$$T \vdash i < 2^x \dot{-} 1 \rightarrow U\big(\langle z,l,r\rangle, i, x, y\big) = \Big\langle z, U\big(l, i, x \dot{-} 1, \sigma_1[x \dot{-} 1, y]\big), r\Big\rangle \qquad (1)$$

$$T \vdash j < 2^x \dot{-} 1 \to U\big(\langle z, l, r\rangle, 2^x \dot{-} 1 + j, x, y\big) = \Big\langle z, l, U\big(r, j, x \dot{-} 1, \sigma_2[x \dot{-} 1, y, \pi_1(l)]\big)\Big\rangle \tag{2}$$

$$T \vdash U\big(\langle z, l, r\rangle, 2^{x+1} \dot{-} 2, x, y\big) = \big\langle V(x, y, l, r), l, r\big\rangle. \tag{3}$$

Note that both 'recursive' applications of the function $U$ on the right-hand side of the conditional equations (1) and (2) are applied to lesser arguments $l < \langle z, l, r\rangle$ and $r < \langle z, l, r\rangle$ than the one on the left. We will use this observation to find a course of values recursive definition of $U$ as follows. The transformation of the specification properties into course of values derivation of $U$ is based on the following simple properties of the projection functions:

$$\vdash_{\mathrm{PA}} \exists z \exists l \exists r\, t = \langle z, l, r\rangle \leftrightarrow \pi_2(t) \neq 0$$
$$\vdash_{\mathrm{PA}} t = \langle z, l, r\rangle \to z = \pi_1(t) \land l = \pi_1\pi_2(t) \land r = \pi_2^2(t)$$
$$\vdash_{\mathrm{PA}} \pi_1(t+1) \le t \land \pi_1\pi_2(t+1) \le t \land \pi_2^2(t+1) \le t.$$

Now let $\xi$ be the term

$$\begin{aligned}\xi[t, l_1, r_1, i, x, y] \equiv D\Bigg(&\pi_2(t),\\ &D\Bigg(i{+}1 <_* 2^x,\\ &\quad \big\langle \pi_1(t), l_1, \pi_2^2(t)\big\rangle,\\ &\quad D\Bigg(i{+}2 <_* 2^{x+1},\\ &\quad\quad \big\langle \pi_1(t), \pi_1\pi_2(t), r_1\big\rangle,\\ &\quad\quad \Big\langle V\big(x, y, \pi_1\pi_2(t), \pi_2^2(t)\big), \pi_1\pi_2(t), \pi_2^2(t)\Big\rangle\Bigg)\Bigg), 0\Bigg).\end{aligned}$$

The function $U(t, i, x, y)$ is defined by course of values recursion on $t$ with substitution in parameters as a p.r. function by

$$\begin{aligned} U(0, i, x, y) &= 0\\ U(t+1, i, x, y) &= \xi\Big[t+1, U\big(\pi_1\pi_2(t+1), i, x \dot{-} 1, \sigma_1[x \dot{-} 1, y]\big),\\ &\qquad U\big(\pi_2^2(t+1), i \dot{-} (2^x \dot{-} 1), x \dot{-} 1, \sigma_2[x \dot{-} 1, y, \pi_1^2\pi_2(t+1)]\big),\\ &\qquad i, x, y\Big].\end{aligned}$$

It is clear that the function $U$ satisfies (1)-(3).

**3.5.9 Global update.** The 4-ary function $M_i(x, y, t)$ updates the partial computation tree $t$ for $f(x, y)$ at each position $j < i$ by the expected value

$f(x_i, y_i)$. The function is defined by primitive recursion on $i$ as a p.r. function:

$$\begin{aligned} M_0(x,y,t) &= t \\ M_{i+1}(x,y,t) &= U(M_i(x,y,t), i, x, y). \end{aligned}$$

It has the following properties which will be needed in the sequel:

$$T \vdash i+1 \le 2^{x+1} \to M_i\big(x+1, y, \langle z,l,r\rangle\big) = \Big\langle z, M_i\big(x, \sigma_1[x,y], l\big), r\Big\rangle \tag{1}$$

$$\begin{aligned} &T \vdash i+1 \le 2^{x+1} \wedge M_i\big(x, \sigma_1[x,y], l\big) = l_1 \to \\ &\quad M_{2^{x+1} \dot- 1+i}\big(x+1, y, \langle z,l,r\rangle\big) = \Big\langle z, l_1, M_i\big(x, \sigma_2[x,y,\pi_1(l_1)], r\big)\Big\rangle. \end{aligned} \tag{2}$$

*Proof.* (1): By induction on $i$. In the base case, clearly $0+1 \le 2^{x+1}$ and thus

$$M_0(x+1, y, \langle z,l,r\rangle) = \langle z,l,r\rangle = \big\langle z, M_0(x, \sigma_1[x,y], l), r\big\rangle.$$

In the induction step, if $(i+1)+1 \le 2^{x+1}$ then $i+1 \le 2^{x+1}$ and therefore

$$\begin{aligned} M_{i+1}(x+1,y,\langle z,l,r\rangle) &= U\big(M_i(x,y,\langle z,l,r\rangle), i,x,y\big) \overset{\text{IH}}{=} \\ &= U\big(\big\langle z, M_i(x,\sigma_1[x,y],l), r\big\rangle, i, x, y\big) \overset{3.5.8(1)}{=} \\ &= \Big\langle z, U\big(M_i(x,\sigma_1[x,y],l), i, x, y\big), r\Big\rangle = \big\langle z, M_{i+1}(x,\sigma_1[x,y],l), r\big\rangle. \end{aligned}$$

(2): By induction on $i$. In the base case suppose that $M_i(x, \sigma_1[x,y], l) = l_1$. We clearly have $0+1 \le 2^{x+1}$ and thus

$$\begin{aligned} M_{2^{x+1}\dot-1+0}(x+1,y,\langle z,l,r\rangle) &= M_{2^{x+1}\dot-1}(x+1,y,\langle z,l,r\rangle) \overset{(1)}{=} \\ = \big\langle z, M_{2^{x+1}\dot-1}(x,\sigma_1[x,y],l), r\big\rangle &= \langle z, l_1, r\rangle = \Big\langle z, l_1, M_0\big(x, \sigma_2[x,y,\pi_1(l_1)], r\big)\Big\rangle. \end{aligned}$$

In the induction step, assume $(i+1)+1 \le 2^{x+1}$ and $M_i(x,\sigma_1[x,y],l) = l_1$. Then $i+1 \le 2^{x+1}$ and we obtain

$$\begin{aligned} M_{2^{x+1}\dot-1+(i+1)}(x+1,y,\langle z,l,r\rangle) &= M_{2^{x+1}\dot-1+i+1}(x+1,y,\langle z,l,r\rangle) = \\ &= U\big(M_{2^{x+1}\dot-1+i}(x+1,y,\langle z,l,r\rangle), 2^{x+1}\dot-1+i, x, y\big) \overset{\text{IH}}{=} \\ &= U\Big(\Big\langle z, l_1, M_i\big(x,\sigma_2[x,y,\pi_1(l_1)],r\big)\Big\rangle, 2^{x+1}\dot-1+i, x, y\Big) \overset{3.5.8(2)}{=} \\ &= \Big\langle z, l_1, U\Big(M_i\big(x,\sigma_2[x,y,\pi_1(l_1)],r\big), 2^{x+1}\dot-1+i, x, y\Big)\Big\rangle = \\ &= \Big\langle z, l_1, M_{2^{x+1}\dot-1+i+1}\big(x,\sigma_2[x,y,\pi_1(l_1)],r\big)\Big\rangle = \\ &= \Big\langle z, l_1, M_{2^{x+1}\dot-1+(i+1)}\big(x,\sigma_2[x,y,\pi_1(l_1)],r\big)\Big\rangle. \end{aligned}$$

**3.5.10 Course of values function.** The binary function $\overline{f}(x,y)$ returns the computation tree for $f(x,y)$. The course of values function for $f$ satisfies

$$T \vdash \overline{f}(0,y) = \big\langle \rho[y], 0, 0 \big\rangle \tag{1}$$

$$\begin{aligned} T \vdash \overline{f}\big(x, \sigma_1[x,y]\big) = l \wedge \overline{f}\big(x, \sigma_2[x,y,\pi_1(l)]\big) = r \rightarrow \\ \overline{f}(x+1,y) = \big\langle \theta[x,\pi_1(l),\pi_1(r),y], l, r\big\rangle \end{aligned} \tag{2}$$

and it is defined explicitly as a p.r. function by

$$\overline{f}(x,y) = M_{2^{x+1} \dot{-} 1}\big(x, y, \mathit{Full}(x+1)\big).$$

*Proof.* (1): It follows from

$$\begin{aligned} \overline{f}(0,y) &= M_{2^{0+1} \dot{-} 1}\big(0,y,\mathit{Full}(0+1)\big) = M_1\big(0,y,\mathit{Full}(1)\big) = M_1\big(0,y,\langle 0,0,0\rangle\big) = \\ &= U\Big(M_0\big(0,y,\langle 0,0,0\rangle\big),0,0,y\Big) = U\big(\langle 0,0,0\rangle,0,0,y\big) \overset{3.5.8(3)}{=} \\ &= \big\langle V(0,y,0,0),0,0\big\rangle = \big\langle \rho[y],0,0\big\rangle. \end{aligned}$$

(2): Suppose that

$$\begin{aligned} \overline{f}\big(x,\sigma_1[x,y]\big) &= l \\ \overline{f}\big(x,\sigma_2[x,y,\pi_1(l)]\big) &= r. \end{aligned}$$

Then, by definition, we have

$$M_{2^{x+1} \dot{-} 1}\big(x,\sigma_1[x,y],\mathit{Full}(x+1)\big) = l \tag{$\dagger_1$}$$

$$M_{2^{x+1} \dot{-} 1}\big(x,\sigma_2[x,y,\pi_1(l)],\mathit{Full}(x+1)\big) = r \tag{$\dagger_2$}$$

and therefore

$$\begin{aligned} \overline{f}(x+1,y) &= M_{2^{x+1+1} \dot{-} 1}\big(x+1,y,\mathit{Full}(x+1+1)\big) = \\ &= M_{2^{x+2} \dot{-} 2+1}\big(x+1,y,\big\langle 0,\mathit{Full}(x+1),\mathit{Full}(x+1)\big\rangle\big) = \\ &= U\Bigg(M_{2^{x+2} \dot{-} 2}\big(x+1,y,\big\langle 0,\mathit{Full}(x+1),\mathit{Full}(x+1)\big\rangle\big), 2^{x+2} \dot{-} 2, x+1, y\Bigg) = \\ &= U\Bigg(M_{2^{x+1} \dot{-} 1 + (2^{x+1} \dot{-} 1)}\big(x+1,y,\big\langle 0,\mathit{Full}(x+1),\mathit{Full}(x+1)\big\rangle\big), \\ &\qquad 2^{x+2} \dot{-} 2, x+1, y\Bigg) \overset{(\dagger_1),\ 3.5.9(2)}{=} \\ &= U\Bigg(\Big\langle 0, M_{2^{x+1} \dot{-} 1}\big(x,\sigma_1[x,y],\mathit{Full}(x+1)\big), \\ &\qquad M_{2^{x+1} \dot{-} 1}\big(x,\sigma_2[x,y,\pi_1(l)],\mathit{Full}(x+1)\big)\Big\rangle, 2^{x+2} \dot{-} 2, x+1, y\Bigg) \overset{(\dagger_1),\ (\dagger_2)}{=} \end{aligned}$$

$$= U\big(\langle 0,l,r\rangle, 2^{x+2} \dot{-} 2, x+1, y\big) \overset{3.5.8(3)}{=} \big\langle V(x+1,y,l,r),l,r\big\rangle =$$
$$= \big\langle \theta[x,\pi_1(l),\pi_1(r),y],l,r\big\rangle. \qquad \square$$

**3.5.11 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by nested simple recursion for the case $k = 2$ and $n = 1$ is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by nested simple recursion as in Par. 3.5.4 and $T''$ an extension of $T$ by the explicit definition

$$f(x,\vec{y}) = \pi_1\overline{f}(x,\vec{y}),$$

where $\overline{f}$ is the course of values function for $f$ (see Par. 3.5.10). We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$ by Thm. 1.3.19. So it suffices to show that the theories $T'$ and $T''$ have the same theorems.

We prove $T'' \vdash T'$ first. The theory $T''$ is an extension by definitions of PA, and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing $f$. It remains to show that the theory $T''$ proves both defining axioms 3.5.4(1)(2) of $f$. The first defining axiom follows from

$$f(0,y) = \pi_1\overline{f}(0,y) \overset{3.5.10(1)}{=} \pi_1\big\langle \rho[y],0,0\big\rangle = \rho[y].$$

The second defining axiom follows from

$$\begin{aligned} f(x+1,y) &= \pi_1\overline{f}(x+1,y) \overset{3.5.10(2)}{=} \\ &= \theta\Big[x,\pi_1\overline{f}\big(x,\sigma_1[x,y]\big),\pi_1\overline{f}\Big(x,\sigma_2\big[x,y,\pi_1\overline{f}(x,\sigma_1[x,y])\big]\Big),y\Big] = \\ &= \theta\Big[x,f\big(x,\sigma_1[x,y]\big),\pi_1\overline{f}\Big(x,\sigma_2\big[x,y,f(x,\sigma_1[x,y])\big]\Big),y\Big] = \\ &= \theta\Big[x,f\big(x,\sigma_1[x,y]\big),f\Big(x,\sigma_2\big[x,y,f(x,\sigma_1[x,y])\big]\Big),y\Big]. \end{aligned}$$

Vice versa, in order to show $T' \vdash T''$ it suffices to prove

$$T' \vdash \forall y\, f(x,y) = \pi_1\overline{f}(x,y).$$

This is proved by induction on $x$. In the base case take any $y$ and we have

$$f(0,y) = \rho[y] = \pi_1\big\langle g(y),0,0\big\rangle \overset{3.5.10(1)}{=} \pi_1\overline{f}(0,y).$$

In the induction step take any $y$ and we obtain

$$\begin{aligned} f(x+1,y) &= \theta\Big[x,f\big(x,\sigma_1[x,y]\big),f\Big(x,\sigma_2\big[x,y,f(x,\sigma_1[x,y])\big]\Big),y\Big] \overset{\text{IH}}{=} \\ &= \theta\Big[x,f\big(x,\sigma_1[x,y]\big),\pi_1\overline{f}\Big(x,\sigma_2\big[x,y,f(x,\sigma_1[x,y])\big]\Big),y\Big] \overset{\text{IH}}{=} \end{aligned}$$

$$= \theta\Big[x, \pi_1\overline{f}\big(x, \sigma_1[x,y]\big), \pi_1\overline{f}\Big(x, \sigma_2\big[x, y, \pi_1\overline{f}(x, \sigma_1[x,y])\big]\Big), y\Big] \overset{3.5.10(2)}{=}$$
$$= \pi_1\overline{f}(x+1,y). \qquad \square$$

**3.5.12 Theorem** *Primitive recursive functions are closed under nested simple recursion for the case $k = 2$ and $n = 1$.*

*Proof.* By inspection of the proof of Thm. 3.5.11. $\square$

## *Nested Simple Recursion: Case $n = 1$*

**3.5.13 Introduction.** In this subsection we will show that the scheme of nested simple recursion with one parameter is admissible in PA. This will be proved in Thm. 3.5.15 by reducing the number of recursive applications. This leads eventually to nested simple recursion with one parameter and two recursive applications.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by primitive recursion with parameter substitution with the defining axioms

$$f(0,y) = \rho[y] \tag{1}$$
$$\bigwedge_{i=1}^{k+1} f\big(x, \sigma_i[x, y, \vec{z}_{i-1}]\big) = z_i \to f(x+1,y) = \theta[x, z_1, \ldots, z_{k+1}, y]. \tag{2}$$

Here $f$ is a new binary function symbol. We claim that $T'$ is an extension of $T$ by definitions.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.5.14 Reduction of the number of recursive applications.** We will reduce the above definition for $k \geq 2$ to a new one, where only $k$ recursive applications are allowed. This new definition is for a binary function $\hat{f}(u,v)$ such that $\hat{f}(2x,y) = f(x,y)$ and it is of the form

$$\hat{f}(0,v) = \rho[v]$$
$$\bigwedge_{i=1}^{k} \hat{f}\big(u, \hat{\sigma}_i[u, v, \vec{w}_{i-1}]\big) = w_i \to \hat{f}(u+1,v) = \hat{\theta}[u, w_1, \ldots, w_k, v]$$

for suitable terms $\hat{\theta}[u, \vec{w}, v], \hat{\sigma}_1[u, v, \vec{w}_0], \ldots, \hat{\sigma}_k[u, v, \vec{w}_{k-1}]$. Here $\vec{w}_i$ abbreviates $w_1, \ldots, w_i$. Note that this is nested simple recursion on $u$ with substitution in the parameter $v$ with $k$ recursive applications. We will take then the identity $f(x,y) = \hat{f}(2x,y)$ as an alternative, explicit definition of $f$.

The idea behind reduction of recursive applications is as follows. We would like to have $\hat{f}(2x, y) = f(x, y)$ and so it must be

$$T \vdash \bigwedge_{i=1}^{k+1} \hat{f}\big(2x, \sigma_i[x, y, \vec{z}_{i-1}]\big) = z_i \to \hat{f}\big(2(x+1), y\big) = \theta[x, z_1, \ldots, z_{k+1}, y]. \tag{1}$$

For the values of the form $\hat{f}(2x+1, v)$ we require

$$\bigwedge_{i=1}^{k} f\big(x, \sigma_i[x, y, \vec{z}_{i-1}]\big) = z_i \to \hat{f}\big(2x+1, \langle 1, y\rangle\big) = \langle z_1, \ldots, z_k\rangle$$
$$\hat{f}\big(2x+1, \langle 2, y, \vec{z}_k\rangle\big) = f\big(x, \sigma_{k+1}[x, y, \vec{z}_k]\big).$$

Note that the application $\hat{f}\big(2x+1, \langle 1, y\rangle\big)$ returns a number which codes $k$ values $f\big(x, \sigma_1[x, y, \vec{z}_0]\big), \ldots, f\big(x, \sigma_k[x, y, \vec{z}_{k-1}]\big)$ of the function $f$ for some $\vec{z}$.

These informal arguments can be rewritten without mentioning the function $f$ as follows:

$$T \vdash \bigwedge_{i=1}^{k} \hat{f}\big(2x, \sigma_i[x, y, \vec{z}_{i-1}]\big) = z_i \to \hat{f}\big(2x+1, \langle 1, y\rangle\big) = \langle z_1, \ldots, z_k\rangle \tag{2}$$
$$T \vdash \hat{f}\big(2x+1, \langle 2, y, \vec{z}_k\rangle\big) = \hat{f}\big(2x, \sigma_{k+1}[x, y, \vec{z}_k]\big) \tag{3}$$
$$T \vdash \hat{f}\big(2x+2, y\big) = \theta\Big[x, \big[\hat{f}\big(2x+1, \langle 1, y\rangle\big)\big]_1^k, \ldots, \big[\hat{f}\big(2x+1, \langle 1, y\rangle\big)\big]_k^k, \tag{4}$$
$$\hat{f}\Big(2x+1, \big\langle 2, y, \hat{f}(2x+1, \langle 1, y\rangle)\big\rangle\Big), y\Big].$$

This means that the terms $\hat{\tau}, \hat{\sigma}_1, \ldots, \hat{\sigma}_k$. satisfy the properties

$$T \vdash \hat{\theta}[2x+1, z, z_{k+1}, \ldots, y] = \theta\left[x, [z]_1^k, \ldots, [z]_k^k, z_{k+1}, y\right] \tag{5}$$
$$T \vdash \hat{\theta}\big[2x, z_1, \ldots, z_k, \langle 1, y\rangle\big] = \langle z_1, \ldots, z_k\rangle \tag{6}$$
$$T \vdash \hat{\theta}\big[2x, z_{k+1}, \ldots, \langle 2, y, z\rangle\big] = z_{k+1} \tag{7}$$

and

$$T \vdash \hat{\sigma}_1[2x+1, y] = \langle 1, y\rangle \tag{8}$$
$$T \vdash \hat{\sigma}_2[2x+1, y, z] = \langle 2, y, z\rangle \tag{9}$$
$$T \vdash \bigwedge_{i=1}^{k} \hat{\sigma}_i\big[2x, \langle 1, y\rangle, \vec{z}_{i-1}\big] = \sigma_i[x, y, \vec{z}_{i-1}] \tag{10}$$
$$T \vdash \hat{\sigma}_1\big[2x, \langle 2, y, \vec{z}_k\rangle\big] = \sigma_{k+1}[x, y, \vec{z}_k]. \tag{11}$$

For that it is sufficient to set

$$\begin{aligned}\hat\theta[u,w_1,\dots,w_k,v] \equiv D\Big(&u \bmod 2, \theta\Big[u \div 2, [w_1]^k_1,\dots,[w_1]^k_k, w_2, v\Big],\\ &D\big(\pi_1(v) =_* 1, \langle w_1,\dots,w_k\rangle, w_1\big)\Big)\end{aligned}$$

and

$$\begin{aligned}\hat\sigma_1[u,v] &\equiv D\Bigg(u \bmod 2, \langle 1,v\rangle,\\ &\qquad D\Bigg(\pi_1(v) =_* 1, \sigma_1\big[u\div 2, \pi_2(v)\big],\\ &\qquad\quad \sigma_{k+1}\Big[u\div 2, \pi_1\pi_2(v), \big[\pi_2^2(v)\big]^k_1,\dots,\big[\pi_2^2(v)\big]^k_k\Big]\Bigg)\Bigg)\\ \hat\sigma_2[u,v,w_1] &\equiv D\big(u \bmod 2, \langle 2,v,w_1\rangle, \sigma_2\,[u\div 2, \pi_2(v), w_1]\big)\\ \hat\sigma_i[u,v,\vec w_{i-1}] &\equiv \sigma_i\,[u\div 2, \pi_2(v), \vec w_{i-1}] \quad \text{for } i = 3,\dots,k.\end{aligned}$$

*Proof.* (5)-(11): Directly from definition. (2): Let us denote by $z_1,\dots,z_k$ the numbers such that $\hat f\big(2x, \sigma_i[x,y,\vec z_{i-1}]\big) = z_i$ for every $i = 1,\dots,k$. We then have

$$\hat f\Big(2x, \hat\sigma_i\big[2x, \langle 1,y\rangle, \vec z_{i-1}\big]\Big) \overset{(10)}{=} \hat f\big(2x, \sigma_i[x,y,\vec z_{i-1}]\big) = z_i \qquad (\dagger_1)$$

for every $i = 1,\dots,k$. From this we obtain

$$\hat f\big(2x+1, \langle 1,y\rangle\big) \overset{(\dagger_1)}{=} \hat\theta\big[2x, z_1,\dots,z_k, \langle 1,y\rangle\big] \overset{(6)}{=} \langle z_1,\dots,z_k\rangle.$$

(3): It follows from

$$\begin{aligned}\hat f\big(2x+1, \langle 2,y,\vec z_k\rangle\big) &= \hat\theta\Big[2x, \hat f\Big(2x, \hat\sigma_1\big[2x, \langle 2,y,\vec z_k\rangle\big]\Big),\dots,\langle 2,y\rangle\Big] \overset{(11)}{=}\\ &= \hat\theta\Big[2x, \hat f\big(2x, \sigma_{k+1}[x,y,\vec z_k]\big),\dots,\langle 2,y\rangle\Big] \overset{(7)}{=} \hat f\big(2x, \sigma_{k+1}[x,y,\vec z_k]\big).\end{aligned}$$

(4): It follows from

$$\begin{aligned}&\hat f(2x+2,y) = \hat f(2x+1+1,y) =\\ &= \hat\theta\Bigg[2x+1, \hat f\big(2x+1, \hat\sigma_1[2x+1,y]\big),\\ &\qquad \hat f\Big(2x+1, \hat\sigma_2\big[2x+1, y, \hat f(2x+1, \hat\sigma_1[2x+1,y])\big]\Big),\dots,y\Bigg] \overset{(8),(9)}{=}\end{aligned}$$

$$
\begin{aligned}
&= \hat{\theta}\Big[2x+1, \hat{f}\big(2x+1,\langle 1,y\rangle\big), \hat{f}\Big(2x+1,\big\langle 2,y,\hat{f}(2x+1,\langle 1,y\rangle)\big\rangle\Big),\dots,y\Big] \overset{(5)}{=} \\
&= \theta\Bigg[x,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_1^k,\dots,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_k^k, \\
&\qquad \hat{f}\Big(2x+1,\big\langle 2,y,\hat{f}(2x+1,\langle 1,y\rangle)\big\rangle\Big),y\Bigg].
\end{aligned}
$$

We are now in position to prove (1). Suppose that

$$\bigwedge_{i=1}^{k+1} \hat{f}\big(2x,\sigma_i[x,y,\vec{z}_{i-1}]\big) = z_i$$

Then by (2) and (3) we obtain

$$\hat{f}\big(2x+1,\langle 1,y\rangle\big) = \langle z_1,\dots,z_k\rangle \tag{$\dagger_2$}$$

$$\hat{f}\big(2x+1,\langle 2,y,\vec{z}_k\rangle\big) = z_{k+1}. \tag{$\dagger_3$}$$

We now have

$$
\begin{aligned}
\hat{f}\big(2(x+1),y\big) &= \hat{f}(2x+2,y) \overset{(4)}{=} \\
&= \theta\Bigg[x,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_1^k,\dots,\big[\hat{f}\big(2x+1,\langle 1,y\rangle\big)\big]_k^k, \\
&\qquad \hat{f}\Big(2x+1,\big\langle 2,y,\hat{f}(2x+1,\langle 1,y\rangle)\big\rangle\Big),y\Bigg] \overset{(\dagger_2),(\dagger_3)}{=} \\
&= \theta\Big[x,\big[\langle z_1,\dots,z_k\rangle\big]_1^k,\dots,\big[\langle z_1,\dots,z_k\rangle\big]_k^k, z_{k+1},y\Big] \overset{2.3.5(1)}{=} \\
&= \theta[x,z_1,\dots,z_k,z_{k+1},y]. \qquad \square
\end{aligned}
$$

**3.5.15 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by nested simple recursion for the case $n = 1$ is an extension by definition.*

*Proof.* The claim is proved by (meta-)induction on the number $k$ of recursive applications in the defining axiom 3.5.13(2). The case $k = 0$ is in fact explicit definition with monadic discrimination and it follows from Thm. 3.1.10. The cases $k = 1$ or $k = 2$ follow from Thm. 3.5.11. So suppose that the claim holds for the case $k \geq 2$. We will prove that the claim holds also for the case $k + 1$.

So let $T'$ be an extension of $T$ by primitive recursion with parameter substitution as in Par. 3.5.13 and $T''$ an extension of $T$ by the explicit definition

$$f(x,y) = \hat{f}(2x,y),$$

where $\hat{f}$ is from Par. 3.5.14. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$ by (meta-)IH. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show that $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.5.13(1)(2) of $f$. The first defining axiom follows from

$$f(0,y) = \hat{f}(2 \times 0, y) = \hat{f}(0,y) = \rho[y].$$

The second defining axiom is proved as follows. Let us denote by $z_1, \dots, z_{k+1}$ the numbers such that

$$\bigwedge_{i=1}^{k+1} f\big(x, \sigma_i[x,y,\vec{z}_{i-1}]\big) = z_i.$$

Directly from definition we have

$$\bigwedge_{i=1}^{k+1} \hat{f}\big(2x, \sigma_i[x,y,\vec{z}_{i-1}]\big) = z_i.$$

We then obtain

$$f(x+1,y) = \hat{f}\big(2(x+1),y\big) \overset{3.5.14(1)}{=} \theta[x, z_1, \dots, z_{k+1}, y].$$

Now we show that $T' \vdash T''$. For that it suffices to prove

$$T' \vdash \forall y\, f(x,y) = \hat{f}(2x,y).$$

This is proved by induction on $x$. In the base case take any $y$ and we have

$$f(0,y) = \rho[y] = \hat{f}(0,y) = \hat{f}(2 \times 0, y).$$

In the induction step take any $y$ and let us denote by $z_1, \dots, z_{k+1}$ the numbers such that

$$\bigwedge_{i=1}^{k+1} f\big(x, \sigma_i[x,y,\vec{z}_{i-1}]\big) = z_i$$

By $(k{+}1)$ applications of IH we obtain

$$\bigwedge_{i=1}^{k+1} \hat{f}\big(2x, \sigma_i[x,y,\vec{z}_{i-1}]\big) = z_i.$$

We then have

$$f(x+1,y) = \theta[x,z_1,\dots,z_{k+1},y] \overset{3.5.14(1)}{=} \hat{f}\big(2(x+1),y\big). \qquad \square$$

**3.5.16 Theorem** *Primitive recursive functions are closed under nested simple recursion for the case $n = 1$.*

*Proof.* By inspection of the proof of Thm. 3.5.15. □

## *Nested Simple Recursion*

**3.5.17 Introduction.** In this subsection we will show that the scheme of nested simple recursion with arbitrary number of parameters is admissible in PA. This will be proved in Thm. 3.5.19 by reducing it to nested simple recursion with one parameter.

We will fix the notation used in this subsection as follows. Let $T$ be an extension by definitions of PA and $T'$ an extension of $T$ by nested simple recursion with the defining axioms

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

$$\bigwedge_{i=1}^{k} f\big(x,\vec{\sigma}_i[x,\vec{y},\vec{z}_{i-1}]\big) = z_i \to f(x+1,\vec{y}) = \theta[x,\vec{z},\vec{y}]. \tag{2}$$

Here $f$ is a new $(n{+}1)$-ary function symbol. We claim that $T'$ is an extension of $T$ by definitions.

Below we will consider the case when the definition has at least two parameters, i.e. $n \geq 2$. The case $n = 0$ is in fact parameterless primitive recursion for which the claim has been already proved in Thm. 3.1.10. The case with one parameter ($n = 1$) follows from Thm. 3.5.15.

We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

**3.5.18 Contraction of parameters.** We will reduce the above scheme, where $n \geq 2$, to a new one for a binary function $\langle f\rangle(x,y)$ so that

$$\langle f\rangle(x,y) = f\left(x,[y]^n_1,\dots,[y]^n_n\right).$$

The $n$ parameters $\vec{y} \equiv y_1,\dots,y_n$ are replaced by a single parameter $y$. We will call the number $y = \langle\vec{y}\rangle \equiv \langle y_1,\dots,y_n\rangle$ the *contraction* of the numbers $\vec{y}$.

The *contraction* function $\langle f\rangle(x,y)$ is defined by nested simple recursion on $x$ with one parameter $y$ as a p.r. function by

$$\langle f\rangle(0,y) = \rho\big[[y]^n_1,\dots,[y]^n_n\big] \tag{1}$$

$$\bigwedge_{i=1}^{k}\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_i\big[x,[y]^n_1,\dots,[y]^n_n,\vec z_{i-1}\big]\Big\rangle\Big) = z_i \rightarrow \\ \Big\langle f\Big\rangle\big(x+1,[y]^n_1,\dots,[y]^n_n\big) = \theta\big[x,\vec z,[y]^n_1,\dots,[y]^n_n\big]. \tag{2}$$

**3.5.19 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by nested simple recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by nested simple recursion as in Par. 3.5.17, where the number of parameters is at least two ($n \geq 2$).[3] Let further $T''$ be an extension of $T$ by the following explicit definition

$$f(x,\vec y) = \langle f\rangle\big(x,\langle\vec y\rangle\big),$$

where $\langle f\rangle$ is the contraction function of $f$ from Par. 3.5.18. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$ by Thm. 3.5.15. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show that $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA and therefore, by Thm. 1.4.9, it proves the principle of mathematical induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that $T''$ proves both defining axioms 3.5.17(1)(2) of $f$. The first defining axiom follows from

$$f(0,\vec y) = \big\langle f\big\rangle\big(0,\langle\vec y\rangle\big) = \rho\big[[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\big] \overset{2.3.5(1)}{=} \rho[\vec y].$$

The second defining axiom is proved as follows. Let us denote by $\vec z$ the numbers such that the following holds

$$\bigwedge_{i=1}^{k} f\big(x,\vec\sigma_i[x,\vec y,\vec z_{i-1}]\big) = z_i.$$

Then from definition we obtain

$$\bigwedge_{i=1}^{k}\Big\langle f\Big\rangle\Big(x,\big\langle\vec\sigma_i[x,\vec y,\vec z_{i-1}]\big\rangle\Big) = z_i$$

and thus, by 2.3.5(1), we have

$$\bigwedge_{i=1}^{k}\Big\langle f\Big\rangle\Big(x,\Big\langle\vec\sigma_i\big[x,[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n,\vec z_{i-1}\big]\Big\rangle\Big) = z_i.$$

From this we obtain

$$f(x+1,\vec y) = \big\langle f\big\rangle\big(x+1,\langle\vec y\rangle\big) = \theta\big[x,\vec z,[\langle\vec y\rangle]^n_1,\dots,[\langle\vec y\rangle]^n_n\big] \overset{2.3.5(1)}{=} \theta[x,\vec z,\vec y].$$

[3] The cases $n = 0$ or $n = 1$ follow from Thm. 3.1.10 or Thm. 3.5.13, respectively.

This proves the second defining axiom.

Now we show that $T' \vdash T''$. For that it suffices to prove

$$T' \vdash \forall \vec{y}\, f(x,\vec{y}) = \big\langle f \big\rangle \big(x, \langle \vec{y} \rangle\big).$$

This is proved by induction on $x$. In the base case take any $\vec{y}$ and we have

$$f(0,\vec{y}) = \rho[\vec{y}] \overset{2.3.5(1)}{=} \rho\big[[\langle\vec{y}\rangle]^n_1, \dots, [\langle\vec{y}\rangle]^n_n\big] = \big\langle f \big\rangle \big(0, \langle \vec{y} \rangle\big).$$

In the induction step take any $\vec{y}$ and let us denote by $\vec{z}$ the numbers such that the following holds

$$\bigwedge_{i=1}^{k} f\big(x, \vec{\sigma}_i[x,\vec{y},\vec{z}_{i-1}]\big) = z_i.$$

By $k$ applications of IH we obtain

$$\bigwedge_{i=1}^{k} \big\langle f \big\rangle \Big(x, \big\langle \vec{\sigma}_i[x,\vec{y},\vec{z}_{i-1}] \big\rangle \Big) = z_i$$

and thus, by 2.3.5(1), we have

$$\bigwedge_{i=1}^{k} \big\langle f \big\rangle \Big(x, \Big\langle \vec{\sigma}_i\big[x, [\langle\vec{y}\rangle]^n_1, \dots, [\langle\vec{y}\rangle]^n_n, \vec{z}_{i-1}\big] \Big\rangle \Big) = z_i.$$

From this we obtain

$$f(x+1,\vec{y}) = \theta[x,\vec{z},\vec{y}] \overset{2.3.5(1)}{=} \theta\big[x,\vec{z},[\langle\vec{y}\rangle]^n_1, \dots, [\langle\vec{y}\rangle]^n_n\big] = \big\langle f \big\rangle \big(x+1, \langle \vec{y} \rangle\big). \qquad \square$$

**3.5.20 Theorem** *Primitive recursive functions are closed under nested simple recursion.*

*Proof.* By inspection of the proof of Thm. 3.5.19. □

# Chapter 4
# General Recursive Schemes

For efficient computation computer programming requires definitions of functions with almost arbitrary recursion. Since we do not wish the extended theories to be inconsistent we restrict ourselves to *regular* recursive definitions. *Conditions of regularity* for a recursive definition of the form $f(\vec{x}) = \tau[f;\vec{x}]$ means that there must be a *well-founded relation* $\prec$ in which the recursion goes down; i.e. for each recursive application $f(\vec{\rho})$ in $\tau$ we have $\vec{\rho} \prec \vec{x}$ under the assumption of all conditions *governing* that recursive application.

Conditions of regularity are strong semantic conditions which cannot be turned into syntactic ones on accord of the incompleteness theorem of Gödel. For that reason we start by considering a restrictive form of recursive definitions, called definitions by *well-founded recursion.* We require that every recursive application $f(\vec{\rho})$ in $\tau$ is surrounded by the *guard* $D(\vec{\rho} \prec_* \vec{x}, f(\vec{\rho}), 0)$, which guarantees that we go into recursion only if $\vec{\rho} \prec \vec{x}$.

First we consider a special case of well-founded recursion, where the well-founded relation $\prec$ is of the form $\vec{x} \prec \vec{y} \leftrightarrow \mu[\vec{x}] < \mu[\vec{y}]$ for some *measure* $\mu[\vec{x}]$. This is called *recursion by measure* and it is the most general scheme of recursion which does not lead outside of primitive recursive functions. The admissibility of measure and well-founded recursion in PA is proved in Thm. 4.2.7 and Thm. 4.3.10, respectively.

For regular recursive definitions we can drop the guarded conditions $\vec{\rho} \prec \vec{x}$ which surround the recursive applications of $f$ in these restrictive forms of recursion. The closure of PA under *regular recursion* is proved in Thm. 4.4.3. Such definitions can be used as programs where we use the defining equalities from left to right as reduction rules. This is shown in the next section.

In the last section we will discuss the computational model based on reduction of terms. Every program $P$ is a property of some function $f$ which can be used as a computational rule to calculate this function. The program $P$ has assigned a *precondition* describing which elements can be used as inputs. Regularity conditions for the program $P$ guarantee that computation terminates for every input $\vec{x}$ which satisfies its precondition yielding the correct value $f(\vec{x})$. This is proved in Thm. 4.5.6.

## 4.1 Introduction

**4.1.1 Introduction.** In this section we give an overview of a simple programming language for the definition of computable functions. We impose strong semantic conditions called conditions of regularity on the correct form of recursive definitions. In this language a correct functional equation of the form $f(\vec{x}) = \tau[f;\vec{x}]$ plays a dual role. Firstly, it defines a function and we can use the definition to reason about the properties of that function. Secondly, the defining equation can be used as a computation rule from left to right to evaluate the applications of the defined function.

**4.1.2 Partial functions.** Some of the programs discussed in this section do not halt for all inputs. By allowing non-termination, we have to consider, at least for a while, partial functions as the meaning of computation. In this paragraph we give a brief overview of partial functions.

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

Partial functions with the same arity are ordered by the set inclusion relation $\subseteq$. If $f \subseteq g$ then we say that the partial function $g$ is an *extension* of the partial function $f$ and also that $f$ is a *restriction* of $g$. Note that the least $n$-ary partial function is the nowhere defined $n$-ary partial function $\varnothing^{(n)}$, i.e. the partial function with the empty domain $\varnothing \subseteq \mathrm{N}^n$. Note that maximal elements coincide with total partial functions.

The assignment of denotation to terms with partial functions is *strict* for all partial functions $f$ except for conditionals (see Par. 4.1.3). This means that the term $f(\vec{\tau})$ is defined if all terms $\vec{\tau}$ are defined and $\vec{\tau}$ is in the domain of $f$. In the presence of partial functions, we will use the non-strict identity $\tau \simeq \rho$ to express the fact that either both expressions are defined and denote the same number or both are undefined.

**4.1.3 Conditionals.** The terms of defining equations are built-up non only from variables, constants, previously defined functions and recursive applications, but also from *conditionals*. These are distinguished applications of the ternary case discrimination function $D$.

The assignment of denotation to conditionals with partial functions is *non-strict*. This means that a conditional $D(\tau_1, \tau_2, \tau_3)$ is defined only if $\tau_1$ is defined and then if the value $\tau_1$ is non-zero then the conditional denotes the same number as the term $\tau_2$ and otherwise the conditional denotes the same number as the term $\tau_3$. Note that in the first case the term $\tau_3$, and in the second case the $\tau_2$, may be undefined without affecting the denotation of the conditional.

The evaluation of each conditional $D(\tau_1, \tau_2, \tau_3)$ is also *non-strict*; first we evaluate the subexpression $\tau_1$ and according to its value we evaluate either $\tau_2$ if the value of $\tau_1$ is non-zero, or $\tau_3$ otherwise.

In order to improve the readability of definitions we will visualize the conditional $D(\tau_1, \tau_2, \tau_3)$ by notation common in computer programming

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

Conditionals of the form

$$\textbf{if } P_*(\vec{\tau}_1) \neq 0 \textbf{ then } \tau_2 \textbf{ else } \tau_3$$

will be further abbreviated to

$$\textbf{if } P(\vec{\tau}_1) \textbf{ then } \tau_2 \textbf{ else } \tau_3,$$

or even to

$$\begin{array}{l}\textbf{case}\\ \quad P(\vec{\tau}_1) \Rightarrow \tau_2\\ \quad \neg P(\vec{\tau}_1) \Rightarrow \tau_3\\ \textbf{end}.\end{array}$$

Similar abbreviation will be used also for complex formulas as those for predicate applications.

**4.1.4 Example.** Consider the following explicit definition of the binary function $\min(x, y)$ yielding the minimum of two numbers $x$ and $y$:

$$\min(x, y) = \textbf{if } (x \leq_* y) \neq 0 \textbf{ then } x \textbf{ else } y.$$

This can be abbreviated to

$$\min(x, y) = \textbf{if } x \leq y \textbf{ then } x\textbf{else } y$$

and further to

$$\begin{array}{rl}\min(x, y) = & \textbf{case}\\ & \quad x \leq y \Rightarrow x\\ & \quad x \nleq y \Rightarrow y\\ & \textbf{end}.\end{array}$$

Note that $x \nleq y \leftrightarrow x > y$ and so the last identity can be written as

$$\begin{array}{rl}\min(x, y) = & \textbf{case}\\ & \quad x \leq y \Rightarrow x\\ & \quad x > y \Rightarrow y\\ & \textbf{end}.\end{array}$$

The **case**-construct used in the definition is usually called dichotomy discrimination on whether or not $x \leq y$.

The following is the explicit definition the maximum function

```
max(x, y) = case
                $x \geq y \Rightarrow x$
                $x < y \Rightarrow y$
            end,
```

where we have adopted all abbreviations mentioned so far.

**4.1.5 Example.** For a more complex example consider the problem of finding the median $\mathit{Median}(x,y,z)$ of three numbers. The function satisfies

$$\vdash_{\mathrm{PA}} x \leq y \leq z \rightarrow \mathit{Median}(x,y,z) = y$$
$$\vdash_{\mathrm{PA}} \mathit{Median}(x,y,z) = \mathit{Median}(y,x,z)$$
$$\vdash_{\mathrm{PA}} \mathit{Median}(x,y,z) = \mathit{Median}(x,z,y)$$

and it is defined explicitly by

```
$Median(x, y, z)$ = case
        $x \leq y \Rightarrow$ case
                $y \leq z \Rightarrow y$
                $y > z \Rightarrow \max(x, z)$
            end
        $x > y \Rightarrow$ case
                $y \leq z \Rightarrow y$
                $y > z \Rightarrow \min(x, z)$
            end
    end.
```

Note that **case**-constructs are nested in the definition.

**4.1.6 Unrestricted recursion.** Many modern declarative programming languages allowed arbitrary forms of recursive programs. With unrestricted recursion the best we can do is to compute partial functions as shown in the following three simple examples. This means, if we allow unrestricted recursion we have to deal with partial functions not only with (total) functions.

**4.1.7 Example.** First, consider the functional equation of the form

$$f(x) = f(x) + 1.$$

This is not a legal definition since there is no unary function satisfying the identity. Using the equation as a rewriting rule, we obtain a program which does not terminate for any input. For instance, the following reduction sequence for evaluation of the application $f(0)$ never ends:

$$f(0) = f(0) + 1 = f(0) + 1 + 1 = f(0) + 1 + 1 + 1 = \cdots.$$

This means that the program computes the nowhere defined unary partial function $\varnothing^{(1)}$. On the other hand, the same partial function is the unique (hence minimal) solution of the functional equation

$$f(x) \simeq f(x) + 1$$

when solved in the class of unary partial functions.

**4.1.8 Example.** Next, consider the functional equation

$$f(x) = f(x).$$

Again, this is not a legal definition since it has infinitely many solutions: in fact every unary function satisfies the identity. Using the equation as a rewriting rule, we obtain a program which computes the partial function $\varnothing^{(1)}$. For instance, the following reduction sequence for evaluation of $f(0)$ does not terminate:

$$f(0) = f(0) = f(0) = \cdots$$

On the other hand, the partial function $\varnothing^{(1)}$ is the minimal (but not unique) solution of the functional equation:

$$f(x) \simeq f(x).$$

**4.1.9 Example.** Finally, consider the functional equation

$$f(x) = f(x) \times 0$$

This is a legal definition since the zero function $Z(x) = 0$ is its only solution. On the other hand, using the equation as a program, we obtain an algorithm which computes the partial function $\varnothing^{(1)}$ again. For instance, the following reduction sequence for evaluation of $f(0)$ does not terminate:

$$f(0) = f(0) \times 0 = f(0) \times 0 \times 0 = f(0) \times 0 \times 0 \times 0 = \cdots$$

As before, the partial function $\varnothing$ is the minimal (but not unique) solution of the corresponding functional equation

$$f(x) \simeq f(x) \times 0.$$

**4.1.10 Kleene's first recursion theorem.** Functional equations from the previous three examples have one thing in common from the computational view: each one when taken as a program computes the same partial function $\varnothing^{(1)}$. On the other hand, there seems to be no correlation between the solutions of these equations and the partial function they compute. Closer scrutiny reveals that the partial function $\varnothing^{(1)}$ is the minimal solution of each functional equation in the class of unary partial functions.

This is not a coincidence. In fact one can prove that every functional equation of the form $f(\vec{x}) \simeq \tau[f;\vec{x}]$ has a minimal solution in the class of partial functions and that this distinguished solution can be computed using the identity as a rewriting rule. This is called First Recursion Theorem and it is due to Kleene (see [21]). It establishes the equivalence of the definitional semantics (minimal solution in this case) with the computational.

**4.1.11 Regular recursion with measure.** In order to achieve the equivalence of the definitional semantics with the computational within the class of total functions, we restrict ourselves to *regular* recursive definitions with measure. The condition of regularity for the functional equation of a form

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

means that there must be a *measure* $\mu[\vec{x}]$ in which the recursion goes down; i.e. for each recursive application $f(\vec{\rho})$ in $\tau$ we have $\mu[\vec{\rho}] < \mu[\vec{x}]$ under the assumption of all conditions governing that recursive application. The regularity conditions guarantees that the equation has a unique solution and that the solution is computable using the defining identity as a rewriting rule from left to right.

Below we give several examples of regular recursive definitions. The justification of this kind of recursion within PA is the subject of study in the following sections.

**4.1.12 Example.** Consider the binary function $f$ obtained from the functions $g$ and $h$ by primitive recursion:

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

The recurrences can be written in the equivalent form using just one equation:

$$f(x,y) = \textbf{if } x \neq 0 \textbf{ then } h\big(x \dot{-} 1, f(x \dot{-} 1, y), y\big) \textbf{ else } g(y).$$

This is an example of regular recursion where recursion goes down in the first argument $x$. The following is the condition of regularity for the (only) recursive application of $f$ on the right-hand side of the identity:

$$\vdash_{\mathrm{PA}} x \neq 0 \rightarrow x \dot{-} 1 < x.$$

The condition is trivially satisfied.

**4.1.13 Example.** Consider the following recursive definition of the integer division function:

```
$x \div y =$ if $y \neq 0$ then
              case
                $x < y \Rightarrow 0$
                $x \geq y \Rightarrow (x \dot{-} y) \div y + 1$
              end
            else
              0
```

This is a definition by regular recursion in which the first argument $x$ goes down. Its condition of regularity

$$\vdash_{\mathrm{PA}} y \neq 0 \land x \geq y \rightarrow x \dot{-} y < x$$

is trivially satisfied.

**4.1.14 Example.** The program for $x \div y$ from Par. 4.1.13 is less optimal than it should be due to repeated test $y \neq 0$ in each recursive call. We would obtain a better one by computing the function using the identity

```
$x \div y =$ case
               $x < y \Rightarrow 0$
               $x \geq y \Rightarrow (x \dot{-} y) \div y + 1$
             end
```

as a reduction rule. The program works correctly only for those inputs that satisfy the property $y \neq 0$. This is is called the *precondition* of the program. Note that for the input $y = 0$ the evaluation of $x \div 0$ does not terminate.

Note that if we write the above program together with its precondition as the following conditional equation

```
$\vdash_{\mathrm{PA}} y \neq 0 \rightarrow x \div y =$ case
                              $x < y \Rightarrow 0$
                              $x \geq y \Rightarrow (x \dot{-} y) \div y + 1$
                            end
```

we obtain an assertion which is, in fact, a property of the integer division function. The following is its (extended) condition of regularity:

$$\vdash_{\mathrm{PA}} y \neq 0 \land x \geq y \rightarrow x \dot{-} y < x \land y \neq 0.$$

It consists of two parts. The first one is the same as before:

$$\vdash_{\mathrm{PA}} y \neq 0 \land x \geq y \rightarrow x \dot{-} y < x.$$

The second one is the so-called *applicability* condition (see [56]):

$$\vdash_{\mathrm{PA}} y \neq 0 \land x \geq y \rightarrow y \neq 0$$

It says that the arguments of the recursive application satisfy the precondition of the program. Both parts of the regularity condition are trivially satisfied.

**4.1.15 Example.** Consider now the recursive definition of the greatest divisor function of the form

```
gcd(x,y) = if $x \neq 0 \land y \neq 0$ then
               case
                 $x > y \Rightarrow \mathrm{gcd}(x \dot{-} y, y)$
                 $x = y \Rightarrow x$
                 $x < y \Rightarrow \mathrm{gcd}(x, y \dot{-} x)$
               end
             else
               $\max(x,y)$
```

based on the following property of the divisibility predicate:

$$\vdash_{\mathrm{PA}} x > y \land z \mid y \to z \mid x \leftrightarrow z \mid x \dot{-} y.$$

The definition is an example of regular recursion where recursion goes down in the measure $\max(x,y)$. Its conditions of regularity

$$\vdash_{\mathrm{PA}} x \neq 0 \land y \neq 0 \land x > y \to \max(x \dot{-} y, y) < \max(x,y)$$
$$\vdash_{\mathrm{PA}} x \neq 0 \land y \neq 0 \land x < y \to \max(x, y \dot{-} x) < \max(x,y)$$

follow from

$$\vdash_{\mathrm{PA}} a > b > 0 \to a \dot{-} b < a.$$

**4.1.16 Example.** The program for $\mathrm{gcd}(x,y)$ from Par. 4.1.15 is less optimal than it should be due to repeated test $x \neq 0 \land y \neq 0$ in each recursive call. We would obtain a better one by computing the function using its property

```
$\vdash_{\mathrm{PA}} x \neq 0 \land y \neq 0 \to \mathrm{gcd}(x,y) =$ case
                                   $x > y \Rightarrow \mathrm{gcd}(x \dot{-} y, y)$
                                   $x = y \Rightarrow x$
                                   $x < y \Rightarrow \mathrm{gcd}(x, y \dot{-} x)$
                                 end
```

as a reduction rule with the precondition $x \neq 0 \land y \neq 0$. The following are its conditions of regularity

$$\vdash_{\mathrm{PA}} x \neq 0 \land y \neq 0 \land x > y \to \max(x \dot{-} y, y) < \max(x,y) \land x \dot{-} y \neq 0 \land y \neq 0$$
$$\vdash_{\mathrm{PA}} x \neq 0 \land y \neq 0 \land x < y \to \max(x, y \dot{-} x) < \max(x,y) \land x \neq 0 \land y \dot{-} x \neq 0.$$

**4.1.17 Example.** Violating the condition of applicability may leads to a program that does not terminate for every input satisfying its precondition. Indeed, consider the following property of the zero function:

$$\vdash_{\mathrm{PA}} x \neq 0 \to Z(x) = Z(x \dot{-} 1)$$

Clearly, the recursion goes down in its argument as we have

$$\vdash_{\mathrm{PA}} x \neq 0 \to x \dot{-} 1 < x.$$

On the other, the applicability property

$$\vdash_{\mathrm{PA}} x \neq 0 \to x \dot{-} 1 \neq 0.$$

does not hold for $x = 1$. Note that the following reduction sequence for evaluation of the application $Z(1)$ does not terminate:

$$Z(1) = Z(1 \dot{-} 1) = Z(0) = Z(0 \dot{-} 0) = Z(0) = \cdots.$$

**4.1.18 Example.** Violating the condition of applicability may leads to a program which computes wrong results even for inputs satisfying its precondition. Consider the following property of the function $x \div 2$:

$$\begin{aligned}\vdash_{\mathrm{PA}} \exists y\, x = 2y \to x \div 2 = \textbf{if } & x \neq 0 \textbf{ then}\\ & (x \dot{-} 1) \div 2 + 1\\ & \textbf{else}\\ & 0.\end{aligned}$$

Clearly, the recursion goes down in its argument as we have

$$\vdash_{\mathrm{PA}} \exists y\, x = 2y \land x \neq 0 \to x \dot{-} 1 < x.$$

On the other, the following property

$$\vdash_{\mathrm{PA}} \exists y\, x = 2y \land x \neq 0 \to \exists y\, x \dot{-} 1 = 2y.$$

does not hold for even numbers. Note that the following reduction sequence for evaluation of $2 \div 2$ yields wrong answer:

$$\begin{aligned}2 \div 2 &= (2 \dot{-} 1) \div 2 + 1 = 1 \div 2 + 1 = (1 \dot{-} 1) \div 2 + 1 + 1 =\\ &= 0 \div 2 + 1 + 1 = 0 + 1 + 1 = 2.\end{aligned}$$

But the expected value is $1 = 2 \div 2$.

**4.1.19 Simultaneous recursion.** Consider the following recursive definition of two binary functions $f_1$ and $f_2$:

$$\begin{aligned}f_1(0,y) &= g_1(y)\\ f_1(x+1,y) &= h_1\big(x, f_1(x,y), f_2(x,y), y\big)\\ f_2(0,y) &= g_2(y)\\ f_2(x+1,y) &= h_2\big(x, f_1(x,y), f_2(x,y), y\big).\end{aligned}$$

Such a recursion is called *simultaneous* recursion. It can be reduced to ordinary non-simultaneous recursion as follows.

For that we need a binary function $f$ such that $f(x,y) = \langle f_1(x,y), f_2(x,y)\rangle$. First we define explicitly two auxiliary functions $g$ and $h$ by

$$\begin{aligned} g(y) &= \big\langle g_1(y), g_2(y)\big\rangle \\ h(x,z,y) &= \Big\langle h_1\big(x,[z]^2_1,[z]^2_2,y\big), h_2\big(x,[z]^2_1,[z]^2_2,y\big)\Big\rangle. \end{aligned}$$

The function $f$ is then obtained from $g$ and $h$ by primitive recursion:

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

or equivalently by

$$f(x,y) = \textbf{if } x \neq 0 \textbf{ then } h\big(x \dot{-} 1, f(x \dot{-} 1, y), y\big) \textbf{ else } g(y).$$

Finally, the functions $f_1$ and $f_2$ are defined explicitly by

$$\begin{aligned} f_1(x,y) &= [f(x,y)]^2_1 \\ f_2(x,y) &= [f(x,y)]^2_1 . \end{aligned}$$

General schemes of simultaneous recursion can be reduced to ordinary one using a method similar to that above.

**4.1.20 Example.** Some schemes of recursion lead beyond primitive recursion. For instance, consider the well-known Ackermann-Péter function (see [38]) which grows faster than any primitive recursive function:

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

The recurrences can be written in a more compact form by

```
$A(x,y) =$ if $x \neq 0$ then
             if $y \neq 0$ then
               $A\big(x \dot{-} 1, A(x, y \dot{-} 1)\big)$
             else
               $A(x \dot{-} 1, 1)$
           else
             $1.$
```

The definition is an example of a definition by *well-founded* recursion which is into the *lexicographic* ordering $<_{\text{lex}}$of the set of pairs of natural numbers:

$$(x_1,y_1) <_{\text{lex}} (x_2,y_2) \leftrightarrow x_1 < x_2 \lor x_1 = x_2 \land y_1 < y_2 .$$

This is because either the first argument of each recursive application decreases or it remains the same and the second argument decreases:

$$\begin{aligned} x \neq 0 \land y = 0 &\to (x \dot{-} 1, 1) <_{\mathrm{lex}} (x, y) \\ x \neq 0 \land y \neq 0 &\to (x, y \dot{-} 1) <_{\mathrm{lex}} (x, y) \\ x \neq 0 \land y \neq 0 &\to (x \dot{-} 1, A(x, y \dot{-} 1)) <_{\mathrm{lex}} (x, y). \end{aligned}$$

We call these properties the conditions of regularity of the definition.

The lexicographic ordering is a *well-founded* relation, i.e. there is no infinite descending chain of pairs of numbers:

$$(x_1, y_1) >_{\mathrm{lex}} (x_2, y_2) >_{\mathrm{lex}} \cdots >_{\mathrm{lex}} (x_n, y_n) >_{\mathrm{lex}} \cdots.$$

This means that the calculation of the application $A(x, y)$ always terminates.

## 4.2 Recursion with Measure

**4.2.1 Introduction.** Consider the functional equation of the form

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

in which every recursive application $f(\vec{\rho})$ in $\tau$ is surrounded by the guard

$$\textbf{if } \mu[\vec{\rho}] < \mu[\vec{x}] \textbf{ then } f(\vec{\rho}) \textbf{ else } 0.$$

Here, the $\mu[\vec{x}]$ is an arbitrary but fixed expression called measure. This is called *recursion with measure*. That PA admits definitions by such recursion is proved in Thm. 4.2.7.

Note that the guard guarantees that we go into recursion only if we have $\mu[\vec{\rho}] < \mu[\vec{x}]$. By satisfying these conditions we obtain a regular recursive definition. These are discussed in Sect. 4.4.

**4.2.2 The principle of measure induction.** For every formula $\varphi[\vec{x}]$ and term $\mu[\vec{x}]$, the formula of *induction on $\vec{x}$ with measure $\mu[\vec{x}]$ for $\varphi$* is the following one:

$$\forall \vec{x}\Big(\forall \vec{y}\big(\mu[\vec{y}] < \mu[\vec{x}] \to \varphi[\vec{y}]\big) \to \varphi[\vec{x}]\Big) \to \forall \vec{x} \varphi[\vec{x}]. \tag{1}$$

We assume here that the variables $\vec{y}$ are different from $\vec{x}$ and that they do not occur freely in $\varphi$. The formula $\varphi$ and the term $\mu$ may contain additional variables as parameters.

Note that for $\vec{x} \equiv x$ and $\mu[x] \equiv x$, the scheme of measure induction coincides with the scheme of complete induction.

**4.2.3 Theorem** *If $T$ is an extension by definitions of* PA *then it proves the principle of measure induction for each formula of $\mathcal{L}_T$.*

*Proof.* The principle of measure induction 4.2.2(1) of $\mathcal{L}_T$ is reduced to mathematical induction as follows. Under the assumption that $\varphi$ is *$\mu$-progressive*:

$$\forall\vec{x}\Big(\forall\vec{y}\big(\mu[\vec{y}] < \mu[\vec{x}] \to \varphi[\vec{y}]\big) \to \varphi[\vec{x}]\Big), \tag{$\dagger_1$}$$

we first prove, by induction on $n$, the following auxiliary property

$$\forall\vec{z}\big(\mu[\vec{z}] < n \to \varphi[\vec{z}]\big). \tag{$\dagger_2$}$$

In the base case there is nothing to prove. In the induction step take any $\vec{z}$ such that $\mu[\vec{z}] < n+1$ and consider two cases. If $\mu[\vec{z}] < n$ then we obtain $\varphi[\vec{z}]$ by IH. If $\mu[\vec{z}] = n$ then by instantiating of ($\dagger_1$) with $\vec{x} := \vec{z}$ we obtain

$$\forall\vec{y}\big(\mu[\vec{y}] < n \to \varphi[\vec{y}]\big) \to \varphi[\vec{z}].$$

Now we apply IH to get $\varphi[\vec{z}]$.

With the auxiliary property proved we obtain that $\varphi[\vec{x}]$ holds for every $\vec{x}$ by instantiating of $\forall n(\dagger_2)$ with $n := \mu[\vec{x}] + 1$ and $\vec{z} := \vec{x}$. □

**4.2.4 Extensions by recursion with measure.** Let $T$ be an extension by definitions of PA and $f$ a new $n$-ary function symbol. Let further $\mu[\vec{x}]$ be a term of $\mathcal{L}_T$ and $\tau[f;\vec{x}]$ a term of the extended language $\mathcal{L}_T \cup \{f\}$ in which no other variables than the $n$ indicated ones are free. Finally, let $[f]^{\mu}_{\vec{x}}(\vec{y})$ be the restriction of $f$ to the numbers $\vec{y}$ s.t. $\mu[\vec{y}] < \mu[\vec{x}]$, i.e.

$$[f]^{\mu}_{\vec{x}}(\vec{y}) \equiv D\big(\mu[\vec{y}] <_* \mu[\vec{x}], f(\vec{y}), 0\big).$$

Consider the theory $T'$ obtained from the theory $T$ by adding the function symbol $f$, the defining axiom

$$f(\vec{x}) = \tau\big[\dot{\lambda}\vec{y}.[f]^{\mu}_{\vec{x}}(\vec{y});\vec{x}\big], \tag{1}$$

and the scheme of measure induction for all formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of $T$ by (course of values) recursion with measure.*

Note the use of special lambda notation (see Par. 3.5.2) in the defining axiom (1). This means that every recursive application $f(\vec{\rho})$ in $\tau$ is replaced by the restriction of $f$ to the numbers $\vec{\rho}$ s.t. $\mu[\vec{\rho}] < \mu[\vec{x}]$, i.e. by the term

$$[f]^{\mu}_{\vec{x}}(\vec{\rho}) \equiv D\big(\mu[\vec{\rho}] <_* \mu[\vec{x}], f(\vec{\rho}), 0\big).$$

In the sequel we will use the notation $\tau[[f]^{\mu}_{\vec{x}};\vec{x}]$ (or even $\tau[[f];\vec{x}]$) as an abbreviation for the term on the right-hand side of the identity (1).

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 4.2.7 that the theory $T'$ is an extension by definition of the theory $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

*Remark.* The definition can be viewed as a function operator which takes all auxiliary functions applied in the terms $\mu, \tau$ and yields the function $f$ as a result. We will prove in Thm. 4.2.8 that the class of primitive recursive functions is closed under the operator of course of values recursion with measure.

**4.2.5 Approximation function.** We wish to introduce the function $f$ into the theory $T$ with the help of its *approximation* function $f^+(z,\vec{x})$. The additional argument $z$ plays the role of the depth of recursion counter. It estimates the depth of recursion needed to compute the value $f(\vec{x})$. If $z$ is sufficiently large then we have $f(\vec{x}) = f^+(z,\vec{x})$. As we will see below every number $z > \mu[\vec{x}]$ gives us sufficient estimation of the depth of recursion. This will allow us to introduce $f$ into PA explicitly by $f(\vec{x}) = f^+(\mu[\vec{x}]+1,\vec{x})$.

The approximation function is introduced with the help of *approximation* terms $\rho^+[f^+;z,\vec{x}]$ which are defined for all subterms $\rho$ of $\tau$ to satisfy:

$$\begin{aligned} x_i^+ &\equiv x_i && (\textit{variable}) \\ g(\rho_1,\dots,\rho_k)^+ &\equiv g(\rho_1^+,\dots,\rho_k^+) && (\textit{auxiliary function}) \\ f(\rho_1,\dots,\rho_n)^+ &\equiv f^+(z,\rho_1^+,\dots,\rho_n^+). && (\textit{recursive application}) \end{aligned}$$

Let us denote by $[f^+]^{\mu}_{z,\vec{x}}(\vec{y})$ the restriction of $f^+$ to $\vec{y}$ s.t. $\mu[\vec{y}] < \mu[\vec{x}]$, i.e.

$$[f^+]^{\mu}_{z,\vec{x}}(\vec{y}) \equiv D\big(\mu[\vec{y}] <_* \mu[\vec{x}], f^+(z,\vec{y}), 0\big).$$

We define the approximation function $f^+$ by nested simple recursion:

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

$$f^+(z+1,\vec{x}) = \tau^+\Big[\dot{\lambda}\vec{y}.[f^+]^{\mu}_{z,\vec{x}}(\vec{y}); z, \vec{x}\Big]. \tag{2}$$

Below we will use the notation $\tau^+[[f^+]^{\mu}_{z,\vec{x}}; z, \vec{x}]$ (or even $\tau^+[[f^+]; z, \vec{x}]$) as an abbreviation for the term on the right-hand side of the equation (2). We will also use the notation $(\rho_1,\dots,\rho_m)^+$ as an abbreviation for $(\rho_1^+,\dots,\rho_m^+)$.

**4.2.6 Monotonicity of the approximation function.** We have

$$T \vdash \mu[\vec{x}] < z_1 \le z_2 \to f^+(z_1,\vec{x}) = f^+(z_2,\vec{x}). \tag{1}$$

The property asserts that the application $f^+(z,\vec{x})$ yields the same result for all numbers $z > \mu[\vec{x}]$.

*Proof.* The property is proved by induction on $z_2$ as $\forall\vec{x}\forall z_1(1)$. In the base case there is nothing to prove. In the induction step, take any numbers $\vec{x}, z_1$ such that $\mu[\vec{x}] < z_1 \leq z_2 + 1$ and prove by inner induction of subterms $\rho[f;\vec{x}]$ of the term $\tau$ the following identity

$$\rho^+\big[[f^+]; z_1 \dot{-} 1, \vec{x}\big] = \rho^+\big[[f^+]; z_2, \vec{x}\big]. \tag{$\dagger_1$}$$

We continue by the case analysis of $\rho$. If $\rho \equiv f(\vec{\theta})$ then by inner IH there are numbers $\vec{y} \equiv y_1, \dots, y_n$ such that

$$\theta_i^+\big[[f^+]; z_1 \dot{-} 1, \vec{x}\big] = y_i = \theta_i^+\big[[f^+]; z_2, \vec{x}\big]$$

for every $i = 1, \dots, n$. We consider two subcases. The subcase $\mu[\vec{y}] \geq \mu[\vec{x}]$ is obvious. In the subcase $\mu[\vec{y}] < \mu[\vec{x}]$ we have $\mu[\vec{y}] < z_1 \dot{-} 1 \leq z_2$ and thus

$$D\big(\mu[\vec{y}] <_* \mu[\vec{x}], f^+(z_1 \dot{-} 1, \vec{y}), 0\big) = f^+(z_1 \dot{-} 1, \vec{y}) \overset{\text{outer IH}}{=} f^+(z_2, \vec{y}) =$$
$$= D\big(\mu[\vec{y}] <_* \mu[\vec{x}], f^+(z_2, \vec{y}), 0\big).$$

The remaining cases when $\rho \equiv x_i$ or $\rho \equiv g(\vec{\theta})$ are straightforward.

With the auxiliary property proved the induction step of the outer induction follows from

$$f^+(z_1, \vec{x}) = \tau^+\big[[f^+]; z_1 \dot{-} 1, \vec{x}\big] \overset{(\dagger_1)}{=} \tau^+\big[[f^+]; z_2, \vec{x}\big] = f^+(z_2 + 1, \vec{x}). \qquad \square$$

**4.2.7 Theorem** *If* $T$ *is an extension by definitions of* PA *then any extension of* $T$ *by recursion with measure is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by recursion

$$f(\vec{x}) = \tau\big[[f]^{\mu}_{\vec{x}}; \vec{x}\big] \tag{$\dagger_1$}$$

as in Par. 4.2.4 and $T''$ an extension of $T$ by explicit definition

$$f(\vec{x}) = f^+(\mu[\vec{x}] + 1, \vec{x}), \tag{$\dagger_2$}$$

where $f^+$ is the approximation function from Par. 4.2.5. We have $\mathcal{L}_{T'} = \mathcal{L}_{T''}$ and $T''$ is an extension by definition of $T$. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we prove $T'' \vdash T'$. The theory $T''$ is an extension by definitions of PA, and therefore, by Thm. 4.2.3, it proves the principle of measure induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that

$$T'' \vdash f(\vec{x}) = \tau\big[[f]^{\mu}_{\vec{x}}; \vec{x}\big]\ . \tag{$\dagger_3$}$$

For that we need the following property which is proved by induction on the structure of subterms $\rho[f;\vec{x}]$ of $\tau$:

$$T'' \vdash \rho^+\big[[f^+];\mu[\vec{x}],\vec{x}\big] = \rho\big[[f];\vec{x}\big]. \tag{$\dagger_4$}$$

Take any $\vec{x}$, any subterm $\rho$ of $\tau$, and continue by case analysis of $\rho$. The case when $\rho \equiv f(\vec{\theta})$ follows from

$$\begin{aligned}
&D\Big(\mu\big[\vec{\theta}^+[[f^+];\mu[\vec{x}],\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu[\vec{x}],\vec{\theta}^+[[f^+];\mu[\vec{x}],\vec{x}]\big),0\Big) \overset{\text{IH's}}{=} \\
&= D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu[\vec{x}],\vec{\theta}[[f];\vec{x}]\big),0\Big) \overset{4.2.6(1)}{=} \\
&= D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu\big[\vec{\theta}[[f];\vec{x}]\big]+1,\vec{\theta}[[f];\vec{x}]\big),0\Big) \overset{(\dagger_2)}{=} \\
&= D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f\big(\vec{\theta}[[f];\vec{x}]\big),0\Big).
\end{aligned}$$

The remaining cases when $\rho \equiv x_i$ or $\rho \equiv g(\vec{\theta})$ are straightforward. With the auxiliary property proved the equality ($\dagger_3$) is obtained from

$$f(\vec{x}) \overset{(\dagger_2)}{=} f^+(\mu[\vec{x}]+1,\vec{x}) \overset{4.2.5(2)}{=} \tau^+\big[[f^+];\mu[\vec{x}],\vec{x}\big] \overset{(\dagger_4)}{=} \tau\big[[f];\vec{x}\big].$$

Now we prove $T' \vdash T''$. For that it suffices to show

$$T' \vdash f(\vec{x}) = f^+(\mu[\vec{x}]+1,\vec{x}). \tag{$\dagger_5$}$$

This is proved by induction on $\vec{x}$ with measure $\mu[\vec{x}]$. So take any $\vec{x}$ and prove by (the inner) induction on the structure of subterms $\rho[f;\vec{x}]$ of $\tau$ the property

$$T' \vdash \rho\big[[f];\vec{x}\big] = \rho^+\big[[f^+];\mu[\vec{x}],\vec{x}\big]. \tag{$\dagger_6$}$$

We continue by case analysis of $\rho$. The case when $\rho \equiv f(\vec{\theta})$ follows from

$$\begin{aligned}
&D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f\big(\vec{\theta}[[f];\vec{x}]\big),0\Big) \overset{\text{outer IH}}{=} \\
&= D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu\big[\vec{\theta}[[f];\vec{x}]\big]+1,\vec{\theta}[[f];\vec{x}]\big),0\Big) \overset{4.2.6(1)}{=} \\
&= D\Big(\mu\big[\vec{\theta}[[f];\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu[\vec{x}],\vec{\theta}[[f];\vec{x}]\big),0\Big) \overset{\text{inner IH's}}{=} \\
&= D\Big(\mu\big[\vec{\theta}^+[[f^+];\mu[\vec{x}],\vec{x}]\big] <_* \mu[\vec{x}], f^+\big(\mu[\vec{x}],\vec{\theta}^+[[f^+];\mu[\vec{x}],\vec{x}]\big),0\Big).
\end{aligned}$$

The remaining cases when $\rho \equiv x_i$ or $\rho \equiv g(\vec{\theta})$ are straightforward. With the auxiliary property proved the equality ($\dagger_5$) is obtained from

$$f(\vec{x}) \overset{(\dagger_1)}{=} \tau\big[[f];\vec{x}\big] \overset{(\dagger_6)}{=} \tau^+\big[[f^+];\mu[\vec{x}],\vec{x}\big] \overset{4.2.5(2)}{=} f^+(\mu[\vec{x}]+1,\vec{x}). \qquad \square$$

**4.2.8 Theorem** *Primitive recursive functions are closed under recursion with measure.*

*Proof.* By inspection of the proof of Thm. 4.2.7. □

## 4.3 Well-Founded Recursion

**4.3.1 Introduction.** Consider the functional equation of the form

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

in which every recursive application $f(\vec{\rho})$ in $\tau$ is surrounded by the guard

$$\textbf{if } \vec{\rho} \prec \vec{x} \textbf{ then } f(\vec{\rho}) \textbf{ else } 0.$$

Here, the $\prec$ is an arbitrary but fixed well-founded relation over natural numbers. This is called well-founded recursion. That PA admits definitions by such recursion is proved in Thm. 4.3.10.

The guard guarantees that we go into recursion only if $\vec{\rho} \prec \vec{x}$. By satisfying these conditions we obtain a regular recursive definition. These are discussed in Sect. 4.4.

**4.3.2 Well-founded relations.** Let $\vec{x} \prec \vec{y}$ be a $2n$-ary relation over N. We say that the relation $\prec$ is *well-founded* if there is no infinite descending chain of the $n$-tuples of numbers: $\vec{x}_1 \succ \vec{x}_2 \succ \cdots \succ \vec{x}_n \succ \cdots$. It is well-known that a predicate $\vec{x} \prec \vec{y}$ is well-founded if and only if the following holds

$$\forall\vec{x}\big(\forall\vec{y}(\vec{y} \prec \vec{x} \to P(\vec{y})) \to P(\vec{x})\big) \to \forall\vec{x}P(\vec{x})$$

for every $n$-ary predicate over N.

*Example.* The standard ordering $x < y$ of natural numbers is obviously a well-founded relation. Another well-founded relation is the lexicographic ordering $(x_1, x_2) <_{\text{lex}} (y_1, y_2)$ of the set of pairs of natural numbers (see Par. 4.1.20). In fact, each measure term $\mu[\vec{x}]$ gives us a well-founded relation over N: namely the relation $\prec$ explicitly defined by $\vec{x} \prec \vec{y} \leftrightarrow \mu[\vec{x}] < \mu[\vec{y}]$.

**4.3.3 The principle of well-founded induction.** Let $\vec{x} \prec \vec{y}$ be a $2n$-ary predicate of a theory $T$. The formula of *$\prec$-well-founded induction on $\vec{x}$ for a formula $\varphi$* of $\mathcal{L}_T$ is the following one:

$$\forall\vec{x}\big(\forall\vec{y}(\vec{y} \prec \vec{x} \to \varphi[\vec{y}]) \to \varphi[\vec{x}]\big) \to \forall\vec{x}\varphi[\vec{x}]. \tag{1}$$

We assume here that the variables $\vec{y}$ are different from $\vec{x}$ and that they do not occur in $\varphi$. The formula $\varphi$ may contain additional variables as parameters.

We say that the predicate $\prec$ is *provably well-founded in $T$* if the theory $T$ proves the principle of $\prec$-well-founded induction for each formula of $\mathcal{L}_T$. The following claim asserts that the relation $\prec$ is provably well-founded in every extension by definitions of $T$.

**4.3.4 Theorem** *If $T'$ is an extension by definitions of $T$ then it proves the principle of $\prec$-well-founded induction for each formula of $\mathcal{L}_{T'}$.*

*Proof.* Translation of the principle of $\prec$-well-founded induction for a formula of $\mathcal{L}_{T'}$ into the theory $T$ yields a formula which is again the principle of $\prec$-well-founded induction but now for a formula of $\mathcal{L}_T$. □

**4.3.5 Extensions by well-founded recursion.** Let $T$ be an extension by definitions of PA and $f$ a new $n$-ary function symbol. Let $\tau[f;\vec{x}]$ be a term of the extended language $\mathcal{L}_T \cup \{f\}$ in which no other variables than the $n$ indicated are free. Let further $\vec{\rho} \prec \vec{x}$ be a provably well-founded relation in $T$. Let finally $[f]^{\prec}_{\vec{x}}(\vec{y})$ be the restriction of $f$ to the numbers $\vec{y}$ s.t. $\vec{y} \prec \vec{x}$, i.e.

$$[f]^{\prec}_{\vec{x}}(\vec{y}) \equiv D\big(\vec{y} \prec_* \vec{x}, f(\vec{y}), 0\big).$$

Consider the theory $T'$ obtained from the theory $T$ by adding the $n$-ary function symbol $f$, the defining axiom

$$f(\vec{x}) = \tau\big[\dot{\lambda}\vec{y}.[f]^{\prec}_{\vec{x}}(\vec{y}); \vec{x}\big], \tag{1}$$

and the scheme of $\prec$-well-founded induction for all formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of $T$ by $\prec$-well-founded recursion.*

Note the use of special lambda notation (see Par. 3.5.2) in the defining axiom (1). This means that every recursive application $f(\vec{\rho})$ in $\tau$ is replaced by the restriction of $f$ to the numbers $\vec{\rho}$ s.t. $\vec{\rho} \prec \vec{x}$, i.e. by the term

$$[f]^{\prec}_{\vec{x}}(\vec{\rho}) \equiv D\big(\vec{\rho} \prec_* \vec{x}, f(\vec{\rho}), 0\big).$$

In the sequel we will use the notation $\tau[[f]^{\prec}_{\vec{x}};\vec{x}]$ (or even $\tau[[f];\vec{x}]$) as an abbreviation for the term on the right-hand side of the identity (1).

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 4.3.10 that the theory $T'$ is an extension by definition of the theory $T$. We will be working in an extension by definitions of the theory $T$. We will keep the notation $T$ also for this extension of $T$.

*Remark.* The definition can be viewed as a function operator which takes all auxiliary functions applied in the term $\tau$ and yields the function $f$ as a result. We will prove in Thm. 4.3.11 that the class of μ-recursive functions is closed under the operator of well-founded recursion.

**4.3.6 Approximation function.** We wish to introduce the function $f$ into the theory $T$ with the help of its *approximation* function $f^{+}(z,\vec{x})$. The additional argument $z$ plays the role of the depth of recursion counter. If the value $f(\vec{x})$ can be computed with the depth $z$ then

$$f^+(z, x_1 + 1, \ldots, x_n + 1) = f(x_1, \ldots, x_n) + 1,$$

where $\vec{x} + 1$ is an abbreviation for $x_1 + 1, \ldots, x_n + 1$. Otherwise, i.e. if the depth $z$ does not suffice, then we have $f^+(z, x_1 + 1, \ldots, x_n + 1) = 0$.

First we define the ternary approximation function $D^+$ of $D$:

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

The approximation function $D^+$ clearly satisfies:

$$T \vdash D^+(0, y, z) = 0 \tag{1}$$
$$T \vdash D^+(x + 1, y, z) = D(x, y, z) \tag{2}$$
$$T \vdash D^+(x + 1, y + 1, z + 1) = D(x, y, z) + 1 \tag{3}$$

Next, for each $k$-ary function $g$ ($k \geq 0$) occurring in the term $\tau$ we have its $k$-ary approximation function $g^+$ defined as

$$g^+(x_1, \ldots, x_k) = \begin{cases} g(x_1 \dot{-} 1, \ldots, x_k \dot{-} 1) + 1 & \text{if } x_1 \neq 0, \ \ldots, \ x_k \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

More formally

$$g^+(x_1, \ldots, x_k) = D\Big(\bigwedge_{*\,i=1}^{k} (x_i \neq_* 0), g(x_1 \dot{-} 1, \ldots, x_k \dot{-} 1) + 1, 0\Big),$$

where $\wedge_{*}{}_{i=1}^{k}(x_i \neq_* 0)$ stands for $(x_1 \neq_* 0) \wedge_* \cdots \wedge_* (x_k \neq_* 0)$. We have

$$T \vdash \bigvee_{i=1}^{k} x_i = 0 \rightarrow g^+(x_1, \ldots, x_k) = 0 \tag{4}$$
$$T \vdash g^+(x_1 + 1, \ldots, x_k + 1) = g(x_1, \ldots, x_k) + 1 \tag{5}$$
$$T \vdash 0^+ = 1. \tag{6}$$

The approximation function $\vec{x} <_*^+ \vec{y}$ of the characteristic function $\vec{x} <_* \vec{y}$ of the predicate $\vec{x} < \vec{y}$ is defined similarly as $g^+$. We then have

$$T \vdash \bigvee_{i=1}^{n} x_i = 0 \vee \bigvee_{i=1}^{n} y_i = 0 \rightarrow (x_1, \ldots, x_n <_*^+ y_1, \ldots, y_n) = 0 \tag{7}$$

$$\begin{aligned} T \vdash {} & (x_1 + 1, \ldots, x_n + 1 <_*^+ y_1 + 1, \ldots, y_n + 1) = \\ & = (x_1, \ldots, x_n <_*^+ y_1, \ldots, y_n) + 1. \end{aligned} \tag{8}$$

Note that by combining (2) and (8) we get

$$\begin{aligned} T \vdash {} & D^+\big((u_1 + 1, \ldots, u_n + 1 <_*^+ v_1 + 1, \ldots, v_n + 1), y, z\big) = \\ & = D\big((u_1, \ldots, u_n <_* v_1, \ldots, v_n), y, z\big). \end{aligned} \tag{9}$$

Approximation terms $\rho^+[f^+; z, \vec{x}]$ are defined for all subterms $\rho$ of $\tau$ to satisfy:

$$
\begin{aligned}
x_i^+ &\equiv x_i + 1 && (\textit{variable})\\
g(\rho_1, \dots, \rho_k)^+ &\equiv g^+(\rho_1^+, \dots, \rho_k^+) && (\textit{auxiliary function})\\
f(\rho_1, \dots, \rho_n)^+ &\equiv f^+(z, \rho_1^+, \dots, \rho_n^+). && (\textit{recursive application})
\end{aligned}
$$

Let us denote by $[f^+]^{<^+}_{z,\vec{x}}(\vec{y})$ the restriction of $f^+$ to $\vec{y}$ s.t. $\vec{y} <^+ \vec{x}$, i.e.

$$[f^+]^{<^+}_{z,\vec{x}}(\vec{y}) \equiv D\big(\vec{y} <^+_* \vec{x}, f^+(z, \vec{y}), 0\big).$$

We define the approximation function $f^+$ by nested simple recursion:

$$f^+(0, \vec{x}) = 0 \tag{10}$$

$$f^+(z+1, \vec{x}) = \tau^+[\dot{\lambda}\vec{y}.[f^+]^{<^+}_{z,\vec{x}}(\vec{y}); z, \vec{x}]. \tag{11}$$

By Thm. 3.5.19, extensions by nested simple recursion are extensions by definition. Hence we may assume without loss of generality that the theory $T$ already contains the approximation function $f^+$ and that its defining axioms are provable in $T$.

Below we will use the notation $\tau^+[[f^+]^{<^+}_{z,\vec{x}}; z, \vec{x}]$ (or even $\tau^+[[f^+]; z, \vec{x}]$) as an abbreviation for the term on the right-hand side of the equation (11).

*Remark.* In order to simplify the discussion below, we will often write $\vec{\tau} + 1$ as an abbreviation defined as

$$\vec{\tau} + 1 \equiv \tau_1 + 1, \dots, \tau_n + 1,$$

where $\vec{\tau} \equiv \tau_1, \dots, \tau_n$. We will need some of the following maximum functions

$$\max(x_1, \dots, x_m) = y \leftrightarrow \bigwedge_{i=1}^{m} x_i \le y \land \bigvee_{i=1}^{m} x_i = y,$$

where $m = 1, 2, \dots$.

**4.3.7 Monotonicity of the approximation function.** The next property asserts that once evaluation of the application $f(\vec{x})$ succeeds with the depth of recursion $z_1$ then it will succeed with the same result for every larger depth of recursion counter $z_2$:

$$T \vdash z_1 \le z_2 \land f^+(z_1, \vec{x}+1) = y + 1 \to f^+(z_2, \vec{x}+1) = y + 1. \tag{1}$$

The property can be generalized for approximation terms as follows. For every subterm $\rho[f; \vec{x}]$ of the term $\tau$ we have

$$T \vdash z_1 \le z_2 \land \rho^+\big[[f^+]; z_1, \vec{x}+1\big] = y + 1 \to \rho^+\big[[f^+]; z_2, \vec{x}+1\big] = y + 1. \tag{2}$$

*Proof.* (1): This is proved by induction on $z_2$ as $\forall z_1 \forall \vec{x} \forall y(1)$. The base case is trivial. In the induction step take any $z_1, \vec{x}, y$ such that $z_1 \le z_2 + 1$ and

$$f^+(z_1, \vec{x} + 1) = y + 1. \tag{$\dagger_1$}$$

Note that it must be $z_1 \neq 0$. First we prove by the inner induction of subterms $\rho[f; \vec{x}]$ of the term $\tau$ that we have

$$\forall v\big(\rho^+\big[[f^+]; z_1 \dot{-} 1, \vec{x}\big] = v + 1 \rightarrow \rho^+\big[[f^+]; z_2, \vec{x}\big] = v + 1\big). \tag{$\dagger_2$}$$

So take any number $v$ such that $\rho^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1] = v + 1$ and continue by the case analysis of the term $\rho$.

The case when $\rho \equiv x_i$ is obvious. If $\rho \equiv g(\vec{\theta})$, where $g$ is a $k$-ary function symbol, then we have

$$g^+\big(\vec{\theta}^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1]\big) = v + 1.$$

By 4.3.6(4), there are $k$ numbers $u_1, \dots, u_k$ s.t. $\theta_i^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1] = u_i + 1$ for every $i = 1, \dots, k$. From the inner IH we obtain $\theta_i^+[[f^+]; z_2, \vec{x} + 1] = u_i + 1$ for every $i = 1, \dots, k$. Consequently

$$g^+\big(\vec{\theta}^+[[f^+]; z_2, \vec{x} + 1]\big) = v + 1.$$

If $\rho \equiv f(\vec{\theta})$ then we have

$$D^+\Big(\vec{\theta}^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1] <_*^+ \vec{x} + 1, f^+\big(z_1 \dot{-} 1, \vec{\theta}^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1]\big), 0^+\Big) = v + 1.$$

By 4.3.6(1), it must be $(\vec{\theta}^+[[f^+]; z_1 \dot{-} 1, \vec{x} + 1] <_*^+ \vec{x} + 1) \neq 0$ and therefore, by 4.3.6(7), there are $n$ numbers $\vec{u} \equiv u_1, \dots, u_n$ such that

$$\theta_i^+\big[[f^+]; z_1 \dot{-} 1, \vec{x} + 1\big] = u_i + 1$$

for every $i = 1, \dots, n$. By the inner IH we obtain

$$\theta_i^+\big[[f^+]; z_2, \vec{x} + 1\big] = u_i + 1$$

for every $i = 1, \dots, n$. Therefore

$$D^+\big(\vec{u} + 1 <_*^+ \vec{x} + 1, f^+(z_1 \dot{-} 1, \vec{u} + 1), 0^+\big) = v + 1$$

and thus, by 4.3.6(9), we have

$$D\big(\vec{u} <_* \vec{x}, f^+(z_1 \dot{-} 1, \vec{u} + 1), 0^+\big) = v + 1.$$

By a simple case analysis on whether $\vec{u} < \vec{x}$ or not and by applying the outer IH with $z_1 \dot{-} 1$ in place of $z_1$, $\vec{u}$ in place of $\vec{x}$, and $v$ in place of $y$, we obtain

$$D\big(\vec{u} \prec_* \vec{x}, f^+(z_2, \vec{u}+1), 0^+\big) = v+1.$$

Working backwards we have

$$D^+\big(\vec{u}+1 \prec_*^+ \vec{x}+1, f^+(z_2, \vec{u}+1), 0^+\big) = v+1$$

and thus

$$D^+\Big(\vec{\theta}^+[[f^+]; z_2, \vec{x}+1] \prec_*^+ \vec{x}+1, f^+\big(z_2, \vec{\theta}^+[[f^+]; z_2, \vec{x}+1]\big), 0^+\Big) = v+1.$$

This concludes the proof of the inner induction step.

With the auxiliary property proved we finish the induction step of the outer induction as follows:

$$\begin{aligned} y+1 \overset{(\dagger_1)}{=} f^+(z_1, \vec{x}+1) = \tau^+[[f^+]; z_1 \dot{-} 1, \vec{x}+1] \overset{(\dagger_2)}{=} \\ = \ \tau^+[[f^+]; z_2, \vec{x}+1] = f^+(z_2+1, \vec{x}+1). \end{aligned}$$

This proves (1).

(2): By structural induction on subterms $\rho[f; \vec{x}]$ of the term $\tau$ as $\forall y(2)$. Take any number $y$, assume $z_1 \le z_2$ and $\rho^+[[f^+]; z_1, \vec{x}+1] = y+1$, and continue by the case analysis of the term $\rho$.

The case when $\rho \equiv x_i$ is obvious; the case $\rho \equiv g(\vec{\theta})$ is proved by similar arguments as above. So suppose that $\rho \equiv f(\vec{\theta})$. We then have

$$D^+\Big(\vec{\theta}^+[[f^+]; z_1, \vec{x}+1] \prec_*^+ \vec{x}+1, f^+\big(z_1, \vec{\theta}^+[[f^+]; z_1, \vec{x}+1]\big), 0^+\Big) = v+1.$$

A similar argument as above yields that there are numbers $\vec{u} \equiv u_1, \ldots, u_n$ such that the following holds for every $i = 1, \ldots, n$:

$$\theta_i^+\big[[f^+]; z_1, \vec{x}+1\big] = u_i + 1.$$

We apply IH $n$-times to obtain

$$\theta_i^+\big[[f^+]; z_2, \vec{x}+1\big] = u_i + 1$$

for every $i = 1, \ldots, n$. Therefore

$$D^+\big(\vec{u}+1 \prec_*^+ \vec{x}+1, f^+(z_1, \vec{u}+1), 0^+\big) = v+1$$

and thus, by 4.3.6(9), we have

$$D\big(\vec{u} \prec_* \vec{x}, f^+(z_1, \vec{u}+1), 0^+\big) = v+1.$$

Now, by using (1), a simple case analysis on whether $\vec{u} \prec \vec{x}$ or not shows that

$$D\big(\vec{u} <_* \vec{x}, f^+(z_2, \vec{u}+1), 0^+\big) = v+1$$

Working backwards we obtain

$$D^+\big(\vec{u}+1 <_*^+ \vec{x}+1, f^+(z_2, \vec{u}+1), 0^+\big) = v+1$$

and hence

$$D^+\Big(\vec{\theta}^+[[f^+]; z_2, \vec{x}+1] <_*^+ \vec{x}+1, f^+\big(z_2, \vec{\theta}^+[[f^+]; z_2, \vec{x}+1]\big), 0^+\Big) = v+1.$$

This concludes the proof of the induction step. □

**4.3.8 The existence condition for the approximation function.** The following property states that for every input $\vec{x}$ there exists a value of the depth of recursion for the evaluation of the application $f(\vec{x})$:

$$T \vdash \exists y \exists z\, f^+(z, \vec{x}+1) = y+1. \tag{1}$$

Note that the value $y$ for which the recursion counter $z$ exists is uniquely determined by the monotonicity of the approximation function.

*Proof.* In the proof we tacitly use the monotone properties of the approximation function and approximation terms. Property (1) is proved by $<$-well-founded induction on $\vec{x}$. So take any numbers $\vec{x}$ and prove first by the inner induction on the structure of subterms $\rho[f;\vec{x}]$ of term $\tau$ the following:

$$\exists y \exists z\, \rho^+\big[[f^+]; z, \vec{x}+1\big] = y+1\ . \tag{$\dagger_1$}$$

We continue by the case analysis of $\rho$.

If $\rho \equiv x_i$ then we have $x_i^+[[f^+]; z, \vec{x}+1] = x_i + 1$ and therefore it suffices to take $y := x_i$ and $z := 0$.

If $\rho \equiv g(\vec{\theta})$, where $g$ is a $k$-ary function symbol, then by the inner IH there are numbers $u_1, \ldots, u_k, z_1, \ldots, z_k$ such that for every $i = 1, \ldots, k$ we have

$$\forall z\big(z \ge z_i \to \theta_i^+\big[[f^+]; z, \vec{x}+1\big] = u_i+1\big). \tag{$\dagger_2$}$$

Thus for every $z \ge \max(z_1, \ldots, z_k)$ we have ($\vec{u} \equiv u_1, \ldots, u_k$)

$$g^+\big(\vec{\theta}^+[[f^+]; z, \vec{x}+1]\big) \overset{(\dagger_2)}{=} g^+(\vec{u}+1) \overset{4.3.6(5)}{=} g(\vec{u})+1.$$

It suffices to take $y := g(\vec{u})$ and $z := \max(x_1, \ldots, x_k)$.

If $\rho \equiv f(\vec{\theta})$ then by the inner IH there are numbers $u_1, \ldots, u_n, z_1, \ldots, z_n$ such that for every $i = 1, \ldots, n$ we have

$$\forall z\big(z \ge z_i \to \theta_i^+\big[[f^+]; z, \vec{x}+1\big] = u_i+1\big). \tag{$\dagger_3$}$$

Thus we have the following for every $z \ge \max(z_1, \ldots, z_n)$ :

$$\begin{aligned}
&D^+\Big(\vec{\theta}^+[[f^+];z,\vec{x}+1] <^+_* \vec{x}+1, f^+\big(z,\vec{\theta}^+[[f^+];z,\vec{x}+1]\big),0^+\Big) \overset{(\dagger_3)}{=}\\
&\quad = D^+\big(\vec{u}+1 <^+_* \vec{x}+1, f^+(z,\vec{u}+1),0^+\big) \overset{4.3.6(9)}{=}\\
&\quad = D\big(\vec{u} <_* \vec{x}, f^+(z,\vec{u}+1),0^+\big).
\end{aligned}$$

Now if $\vec{u} < \vec{x}$ then from the outer IH we obtain numbers $u_0$ and $z_0$ such that

$$\forall z(z \geq z_0 \rightarrow f^+(z,\vec{u}+1) = u_0+1). \qquad (\dagger_4)$$

Thus for every $z \geq \max(z_0,z_1,\ldots,z_n)$ we have

$$D\big(\vec{u} <_* \vec{x}, f^+(z,\vec{u}+1),0^+\big) = f^+(z,\vec{u}+1) \overset{(\dagger_4)}{=} u_0+1.$$

It suffices to take $y := u_0$ and $z := \max(z_0,z_1,\ldots,z_n)$. Otherwise we have $\vec{u} \not< \vec{x}$ and thus for every $z \geq \max(z_1,\ldots,z_n)$ we get

$$D\big(\vec{u} <_* \vec{x}, f^+(z,\vec{u}+1),0^+\big) = 0^+ \overset{4.3.6(6)}{=} 0+1.$$

It suffices to take $y := 0$ and $z = \max(z_1,\ldots,z_n)$.

We are now in position to finish the proof of the outer induction. By $(\dagger_1)$ applied to the term $\tau$ we obtain that the following holds for some $y$ and $z$:

$$\tau^+\big[[f^+];z,\vec{x}+1\big] = y+1\ . \qquad (\dagger_5)$$

The induction step of the outer induction follows from

$$f^+(z+1,\vec{x}+1) = \tau^+\big[[f^+];z,\vec{x}+1\big] \overset{(\dagger_5)}{=} y+1.$$

This proves (1). □

**4.3.9 Depth of recursion counter.** Consider the $n$-ary function $\delta(\vec{x})$ defined by the following minimalization

$$\delta(\vec{x}) = \mu z[f^+(z+1,\vec{x}+1) \neq 0].$$

Note that its condition of regularity

$$T \vdash \exists z\, f^+(z+1,\vec{x}+1) \neq 0.$$

follows from 4.3.8(1). The value $\delta(\vec{x})+1$ is the depth of recursion needed to evaluate the application $f(\vec{x})$.

The function satisfies

$$T \vdash f^+(z, \vec{x}+1) = f^+(\delta(\vec{x})+1, \vec{x}+1) \leftrightarrow \delta(\vec{x}) < z \tag{1}$$
$$T \vdash \rho^+[[f^+]; \delta(\vec{x}), \vec{x}+1] \neq 0 \tag{2}$$
$$\begin{aligned} T \vdash D\big(\vec{y} \prec_* \vec{x}, f^+(z, \vec{y}+1), 0^+\big) \neq 0 \to \\ D\big(\vec{y} \prec_* \vec{x}, f^+(z, \vec{y}+1), 0^+\big) = D\big(\vec{y} \prec_* \vec{x}, f^+(\delta(\vec{y})+1, \vec{y}+1), 0^+\big), \end{aligned} \tag{3}$$

where the property (2) holds for every subterm $\rho[f; \vec{x}]$ of $\tau$.

*Proof.* First note that directly from the definition we obtain that

$$T \vdash f^+(\delta(\vec{x})+1, \vec{x}+1) \neq 0 \tag{$\dagger_1$}$$
$$T \vdash f^+(z+1, \vec{x}+1) \neq 0 \to \delta(\vec{x}) \leq z. \tag{$\dagger_2$}$$

The last property can be written in the following equivalent form

$$T \vdash f^+(z, \vec{x}+1) \neq 0 \to \delta(\vec{x}) < z \tag{$\dagger_3$}$$

because $f^+(0, \vec{x}+1) = 0$ by definition.

(1): Suppose that $f^+(z, \vec{x}+1) = f^+(\delta(\vec{x})+1, \vec{x}+1)$. Then by ($\dagger_1$), we have $f^+(z, \vec{x}+1) \neq 0$ and thus $\delta(\vec{x}) < z$ by ($\dagger_3$). This proves the ($\to$)-direction of the claim. Vice versa, if $\delta(\vec{x}) < z$ then $\delta(\vec{x}) \leq z \dot{-} 1$ and thus

$$0 \overset{(\dagger_1)}{\neq} f^+(\delta(\vec{x})+1, \vec{x}+1) \overset{4.3.7(1)}{=} f^+(z \dot{-} 1 + 1, \vec{x}+1) = f^+(z, \vec{x}+1).$$

This proves the reverse direction.

(2): This is proved by backward induction on the structure of subterms $\rho[f; \vec{x}]$ of the term $\tau$. If $\rho \equiv \tau$ then the claim follows from

$$\tau^+\big[[f^+]; \delta(\vec{x}), \vec{x}+1\big] = f^+(\delta(\vec{x})+1, \vec{x}+1) \overset{(\dagger_1)}{\neq} 0.$$

The case when $\rho \equiv x_i$ is obvious. If $\rho \equiv g(\vec{\theta})$, where $g$ is an auxiliary $k$-ary function, then we obtain from IH

$$g^+\big(\vec{\theta}^+[[f^+]; \delta(\vec{x}), \vec{x}+1]\big) \neq 0,$$

and thus, by 4.3.6(5), we have

$$\theta_i^+\big[[f^+]; \delta(\vec{x}), \vec{x}+1\big] \neq 0$$

for every $i = 1, \ldots, k$. If $\rho \equiv f(\vec{\theta})$ then we obtain from IH

$$D^+\Big(\vec{\theta}^+[[f^+]; \delta(\vec{x}), \vec{x}+1] \prec_*^+ \vec{x}+1, f^+\big(\delta(\vec{x}) \dot{-} 1, \vec{\theta}^+[[f^+]; \delta(\vec{x}), \vec{x}+1]\big), 0^+\Big) \neq 0.$$

By 4.3.6(1), it must be

$$\big(\vec{\theta}^{+}[[f^{+}];\delta(\vec{x}),\vec{x}+1] <^{+}_{*} \vec{x}+1\big) \neq 0$$

and thus, by 4.3.6(7), we have

$$\theta^{+}_{i}\big[[f^{+}];\delta(\vec{x}),\vec{x}+1\big] \neq 0$$

for every $i = 1,\dots,n$.

(3): Under the antecedent of the claim we consider two cases. If $\vec{y} \prec \vec{x}$ then

$$f^{+}(z,\vec{y}+1) = D\big(\vec{y} \prec_{*} \vec{x}, f^{+}(z,\vec{y}+1), 0^{+}\big) \neq 0$$

and thus $\delta(\vec{y}) < z$ by ($\dagger_3$). We then obtain

$$\begin{aligned} D\big(\vec{y} \prec_{*} \vec{x}, f^{+}(z,\vec{y}+1), 0^{+}\big) = f^{+}(z,\vec{y}+1) \overset{(1)}{=} f^{+}(\delta(\vec{y})+1,\vec{y}+1) = \\ = D\big(\vec{y} \prec_{*} \vec{x}, f^{+}(\delta(\vec{y})+1,\vec{y}+1), 0^{+}\big). \end{aligned}$$

The case when $\vec{y} \nprec \vec{x}$ is straightforward. □

**4.3.10 Theorem** *If T is an extension by definitions of* PA *then any extension of T by well-founded recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by the $\prec$-well-founded recursion

$$f(\vec{x}) = \tau\big[[f]^{\prec}_{\vec{x}};\vec{x}\big]$$

as in Par. 4.3.5 and $T''$ an extension of $T$ by the explicit definition

$$f(\vec{x}) = f^{+}(\delta(\vec{x})+1,\vec{x}+1) \dot{-} 1,$$

where $f^{+}$ and $\delta$ are as in Par. 4.3.6 and Par. 4.3.9, respectively. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ have the same theorems.

First we show $T'' \vdash T'$. The theory $T''$ is an extension by definition of $T$, and therefore, by Thm. 4.3.4, it proves the principle of $\prec$-well-founded induction for each formula of $\mathcal{L}_{T'}$ containing the symbol $f$. It remains to show that

$$T'' \vdash f(\vec{x}) = \tau\big[[f];\vec{x}\big]. \qquad (\dagger_1)$$

For that we need the following auxiliary property which is proved by induction on the structure of subterms $\rho[f;\vec{x}]$ of the term $\tau$:

$$T'' \vdash \rho^{+}\big[[f^{+}];\delta(\vec{x}),\vec{x}+1\big] = \rho\big[[f];\vec{x}\big] + 1. \qquad (\dagger_2)$$

We continue by the case analysis of $\rho$.

The case when $\rho \equiv x_i$ is obvious. If $\rho \equiv g(\vec{\theta})$, where $g$ is an auxiliary function, then we have

$$g^+\big(\vec{\theta}^+[[f^+];\delta(\vec{x}),\vec{x}+1]\big) \overset{\text{IH's}}{=} g^+\big(\vec{\theta}[[f];\vec{x}]+1\big) \overset{4.3.6(5)}{=} g\big(\vec{\theta}[[f];\vec{x}]\big)+1.$$

If $\rho \equiv f(\vec{\theta})$ then

$$D^+\Big(\vec{\theta}^+[[f^+];\delta(\vec{x}),\vec{x}+1] <_*^+ \vec{x}+1, f^+\big(\delta(\vec{x}),\vec{\theta}^+[[f^+];z,\vec{x}+1]\big),0^+\Big) \neq 0 \qquad (\dagger_3)$$

by 4.3.9(2) and thus

$$\begin{aligned}
&D^+\Big(\vec{\theta}^+[[f^+];\delta(\vec{x}),\vec{x}+1] <_*^+ \vec{x}+1, f^+\big(\delta(\vec{x}),\vec{\theta}^+[[f^+];z,\vec{x}+1]\big),0^+\Big) \overset{\text{IH's}}{=}\\
&= D^+\Big(\vec{\theta}[[f];\vec{x}]+1 <_*^+ \vec{x}+1, f^+\big(\delta(\vec{x}),\vec{\theta}[[f];\vec{x}]+1\big),0^+\Big) \overset{4.3.6(9)}{=}\\
&= D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f^+\big(\delta(\vec{x}),\vec{\theta}[[f];\vec{x}]+1\big),0^+\Big) \overset{(\dagger_3),4.3.9(3)}{=}\\
&= D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f^+\big(\delta(\vec{\theta}[[f];\vec{x}])+1,\vec{\theta}[[f];\vec{x}]+1\big),0^+\Big) =\\
&= D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f\big(\vec{\theta}[[f];\vec{x}]\big)+1,0^+\Big) \overset{4.3.6(6)}{=}\\
&= D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f\big(\vec{\theta}[[f];\vec{x}]\big),0\Big)+1.
\end{aligned}$$

This concludes the proof of $(\dagger_2)$.

With the auxiliary property proved we obtain $(\dagger_1)$ from

$$\begin{aligned}
f(\vec{x}) &= f^+(\delta(\vec{x})+1,\vec{x}+1) \dot{-} 1 = \tau^+\big[[f^+];\delta(\vec{x}),\vec{x}+1\big] \dot{-} 1 \overset{(\dagger_2)}{=}\\
&= \tau\big[[f];\vec{x}\big]+1 \dot{-} 1 = \tau\big[[f];\vec{x}\big].
\end{aligned}$$

Now we prove $T' \vdash T''$. For that it suffices to show

$$T' \vdash f(\vec{x}) = f^+(\delta(\vec{x})+1,\vec{x}+1) \dot{-} 1. \qquad (\dagger_4)$$

This is proved by $<$-well-founded induction on $\vec{x}$. Take any $\vec{x}$ and prove first by (the inner) induction on the structure of subterms $\rho[f;\vec{x}]$ of $\tau$ the property

$$\rho\big[[f];\vec{x}\big]+1 = \rho^+\big[[f^+];\delta(\vec{x}),\vec{x}+1\big]. \qquad (\dagger_5)$$

We continue by the case analysis of $\rho$.

The case when $\rho \equiv x_i$ is obvious. If $\rho \equiv g(\vec{\theta})$, where $g$ is an auxiliary function, then we have

$$g\big(\vec{\theta}[[f];\vec{x}]\big)+1 \overset{4.3.6(5)}{=} g^+\big(\vec{\theta}[[f];\vec{x}]+1\big) \overset{\text{IH's}}{=} g^+\big(\vec{\theta}^+[[f^+];\delta(\vec{x}),\vec{x}+1]\big).$$

If $\rho \equiv f(\vec{\theta})$ then

$$D^+\Big(\vec{\theta}^+[[f^+];\delta(\vec{x}),\vec{x}+1] <_*^+ \vec{x}+1, f^+\big(\delta(\vec{x}),\vec{\theta}^+[[f^+];z,\vec{x}+1]\big),0^+\Big) \neq 0 \qquad (\dagger_6)$$

by 4.3.9(2) and thus

$$\begin{aligned}
&D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f\big(\vec{\theta}[[f];\vec{x}]\big), 0\Big) + 1 \overset{4.3.6(6)}{=} \\
&\quad = D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f\big(\vec{\theta}[[f];\vec{x}]\big) + 1, 0^+\Big) \overset{\text{outer IH}}{=} \\
&\quad = D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f^+\big(\delta(\vec{\theta}[[f];\vec{x}]) + 1, \vec{\theta}[[f];\vec{x}] + 1\big), 0^+\Big) \overset{(\dagger_6),4.3.9(3)}{=} \\
&\quad = D\Big(\vec{\theta}[[f];\vec{x}] <_* \vec{x}, f^+\big(\delta(\vec{x}), \vec{\theta}[[f];\vec{x}] + 1\big), 0^+\Big) \overset{4.3.6(9)}{=} \\
&\quad = D^+\Big(\vec{\theta}[[f];\vec{x}] + 1 <_*^+ \vec{x} + 1, f^+\big(\delta(\vec{x}), \vec{\theta}[[f];\vec{x}] + 1\big), 0^+\Big) \overset{\text{inner IH's}}{=} \\
&\quad = D^+\Big(\vec{\theta}^+[[f^+];\delta(\vec{x}), \vec{x}+1] <_*^+ \vec{x} + 1, f^+\big(\delta(\vec{x}), \vec{\theta}^+[[f^+];\delta(\vec{x}), \vec{x}+1]\big), 0^+\Big).
\end{aligned}$$

This ends the proof of the inner induction formula $(\dagger_5)$.

We are now in position to finish the proof of the induction step of the outer induction. We have

$$\begin{aligned}
f(\vec{x}) = \tau\big[[f];\vec{x}\big] = \tau\big[[f];\vec{x}\big] + 1 \dot{-} 1 \overset{(\dagger_5)}{=} \tau^+\big[[f^+];\delta(\vec{x}), \vec{x}+1\big] \dot{-} 1 = \\
= f^+(\delta(\vec{x}) + 1, \vec{x} + 1) \dot{-} 1.
\end{aligned}$$ □

**4.3.11 Theorem** *μ-Recursive functions are closed under well-founded recursion.*

*Proof.* By inspection of the proof of Thm. 4.3.10. □

## 4.4 Regular Recursion

**4.4.1 Introduction.** Consider the following recursive definition of the function $f$, which is into the well-founded relation $\prec$:

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

where $[f]^{\prec}_{\vec{x}}$ is the restriction of $f$:

$$[f]^{\prec}_{\vec{x}}(\vec{y}) \equiv \textbf{if } \vec{y} \prec \vec{x} \textbf{ then } f(\vec{y}) \textbf{ else } 0.$$

We can drop the guards $\vec{\rho} \prec \vec{x}$ around recursive applications of $f$ in the definition if we restrict them to *regular* applications. The condition of regularity means that for every recursive application $f(\vec{\rho})$ we have

$$\Gamma^{\tau}_{f(\vec{\rho})}[f;\vec{x}] \rightarrow \vec{\rho}[f;\vec{x}] \prec \vec{x}.$$

Here, the $\Gamma^{\tau}_{f(\vec{\rho})}$ is the *governing condition* of the term $f(\vec{\rho})$ in $\tau$: it is the conjunction of all conditions surrounding the recursive application $f(\vec{\rho})$ in the term $\tau$. Such definitions are called *regular* definitions. The definition (1) can be then written in the equivalent form

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

with the governing conditions removed. For regular definitions we have not only the extensional property:

$$\tau\big[[f]^{\prec}_{\vec{x}};\vec{x}\big] = \tau[f;\vec{x}]$$

but also a stronger intensional property that we can use the defining equation (2) as a computational rule from left to right for the evaluation of $f$. This is shown in the next section.

**4.4.2 Extensions by regular recursion.** The concept of regular recursive definitions is formalized as follows. Let $T$ be an extension by definitions of PA and $\prec$ a well-founded relation of $T$.. Let further $f$ be a new $n$-ary function symbol and $\tau[f;\vec{x}]$ a term in $\vec{x}$ of the theory $T_f$ obtained from $T$ by adding the function symbol $f$.

To every occurrence of a subterm $\rho$ in $\tau$ we define inductively on $\|\tau\| - \|\rho\|$ the *governing condition* $\Gamma^{\tau}_{\rho}$ *of* $\rho$ *in* $\tau$. Here, the $\|\tau\|$ is the *size*, i.e. the number of operations, of the term $\tau$. If $\rho \equiv \tau$ or $\rho$ is without conditionals then $\Gamma^{\tau}_{\rho} \equiv \top$ (i.e. always true). If $\rho \equiv D(\theta_1,\theta_2,\theta_3)$ then $\Gamma^{\tau}_{\theta_1} \equiv \Gamma^{\tau}_{\rho}$, $\Gamma^{\tau}_{\theta_2} \equiv \Gamma^{\tau}_{\rho} \wedge \theta_1 \neq 0$ and $\Gamma^{\tau}_{\theta_3} \equiv \Gamma^{\tau}_{\rho} \wedge \theta_1 = 0$.

We assign to each occurrence of the recursive application $f(\vec{\rho})$ in $\tau$ with governing condition $\Gamma^{\tau}_{f(\vec{\rho})}$ the following *condition of regularity*:

$$\Gamma^{\tau}_{f(\vec{\rho})} \to \vec{\rho} \prec \vec{x}.$$

We denote by $Reg^{\prec}_{\tau}[f]$ the conjunction of universal closures of all its conditions of regularity.

We say that the term $\tau$ *is regular in the well-founded relation* $\prec$ if the extension $T''$ of $T$ by (bounded) well-founded recursion $f(\vec{x}) = \tau\big[[f]^{\prec}_{\vec{x}};\vec{x}\big]$ proves all its condition of regularity, i.e. we have $T'' \vdash Reg^{\prec}_{\tau}[f]$.

Consider the theory $T'$ obtained from the theory $T$ by adding the $n$-ary function symbol $f$, the defining axiom

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

where $\tau$ is a regular term in $\prec$, the conditions of regularity $Reg^{\prec}_{\tau}[f]$, and the scheme of $\prec$-well-founded induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of* $T$ *by regular (well-founded) recursion.*

*Remark.* The definition can be viewed as a function operator which takes all auxiliary functions applied in the term $\tau$ and yields the function $f$ as a result. We will prove in Thm. 4.4.5 that the class of μ-recursive functions is closed under the operator of regular well-founded recursion.

*Remark.* The definition is said to be *regular recursion with measure* if

$$\vec{x} < \vec{y} \leftrightarrow \mu[\vec{x}] < \mu[\vec{y}]$$

for a suitable measure $\mu[\vec{x}]$ of $\mathcal{L}_T$. In this case we will use the following notation $Reg^{\mu}_{\tau}[f]$ to denote its conditions of regularity.

Such definition can be viewed as a function operator which takes all auxiliary functions applied in the terms $\mu, \tau$ and yields the function $f$ as a result. We will prove in Thm. 4.4.4 that the class of primitive recursive functions is closed under the operator of regular recursion with measure.

**4.4.3 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by regular recursion is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by regular recursion

$$f(\vec{x}) = \tau[f;\vec{x}] \tag{$\dagger_1$}$$

and $T''$ an extension of $T$ by (bounded) well-founded recursion

$$f(\vec{x}) = \tau\big[[f]^{<}_{\vec{x}};\vec{x}\big] \tag{$\dagger_2$}$$

as in Par. 4.4.2. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and the theory $T''$ is an extension by definition of the theory $T$ by Thm. 4.3.10. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ are equivalent.

First we prove $T'' \vdash T'$. First of all, the theory $T''$ contains the principle of $<$-well-founded induction for every formula of $\mathcal{L}_{T'}$ with the symbol $f$. Next the term $\tau$ is regular and therefore $T'' \vdash Reg^{<}_{\tau}[f]$. So it remains to show that

$$T'' \vdash f(\vec{x}) = \tau[f;\vec{x}]. \tag{$\dagger_3$}$$

For that we need the following auxiliary property which is proved by induction on the structure of subterms $\rho[f;\vec{x}]$ of $\tau$:

$$T'' \vdash \mathit{\Gamma}^{\tau}_{\rho}[f;\vec{x}] \rightarrow \rho\big[[f]^{<}_{\vec{x}};\vec{x}\big] = \rho[f;\vec{x}]. \tag{$\dagger_4$}$$

So take any numbers $\vec{x}$ and subterm $\rho$ of $\tau$ such that $\mathit{\Gamma}^{\tau}_{\rho}[f;\vec{x}]$ holds, and continue by case analysis of $\rho$.

The case when $\rho \equiv x_i$ is trivial. If $\rho \equiv g(\vec{\theta})$ then the terms $\vec{\theta}$ are governed in $\tau$ by $\mathit{\Gamma}^{\tau}_{\rho}$ and we obtain $g(\vec{\theta}[[f];\vec{x}]) = g(\vec{\theta}[f;\vec{x}])$ directly from IH.

If $\rho \equiv f(\vec{\theta})$ then the terms $\vec{\theta}$ are governed in $\tau$ by $\Gamma^{\tau}_{\rho}$. By regularity $\vec{\theta}[f;\vec{x}] \prec \vec{x}$ and thus

$$D\bigl(\vec{\theta}[[f];\vec{x}] \prec_* \vec{x}, f(\vec{\theta}[[f];\vec{x}]),0\bigr) \overset{\text{IH}}{=} D\bigl(\vec{\theta}[f;\vec{x}] \prec_* \vec{x}, f(\vec{\theta}[f;\vec{x}]),0\bigr) =$$
$$= D\bigl(1, f(\vec{\theta}[f;\vec{x}]),0\bigr) = f(\vec{\theta}[f;\vec{x}]).$$

If $\rho \equiv D(\theta_1,\theta_2,\theta_3)$ then the terms $\theta_1,\theta_2,\theta_3$ are governed in $\tau$ respectively by $\Gamma^{\tau}_{\rho}$, $\Gamma^{\tau}_{\rho} \wedge \theta_1 \neq 0$ and $\Gamma^{\tau}_{\rho} \wedge \theta_1 = 0$. We have by IH:

$$\theta_1[[f];\vec{x}] = \theta_1[f;\vec{x}] \tag{$\dagger_5$}$$
$$\theta_1[f;\vec{x}] \neq 0 \to \theta_2[[f];\vec{x}] = \theta_2[f;\vec{x}] \tag{$\dagger_6$}$$
$$\theta_1[f;\vec{x}] = 0 \to \theta_3[[f];\vec{x}] = \theta_3[f;\vec{x}]. \tag{$\dagger_7$}$$

Thus

$$D\bigl(\theta_1[[f];\vec{x}],\theta_2[[f];\vec{x}],\theta_3[[f];\vec{x}]\bigr) \overset{(\dagger_5)}{=}$$
$$= D\bigl(\theta_1[f;\vec{x}],\theta_2[[f];\vec{x}],\theta_3[[f];\vec{x}]\bigr) \overset{(\dagger_6),(\dagger_7)}{=} D\bigl(\theta_1[f;\vec{x}],\theta_2[f;\vec{x}],\theta_3[f;\vec{x}]\bigr).$$

With the auxiliary property proved we obtain $(\dagger_3)$ from

$$f(\vec{x}) = \tau\bigl[[f]^{\prec}_{\vec{x}};\vec{x}\bigr] \overset{(\dagger_4)}{=} \tau[f;\vec{x}]$$

by noting that $\tau$ is governed in $\tau$ by $\top$.

Now we prove $T' \vdash T''$. The theory $T'$ contains the principle of $\prec$-well-founded induction for every formula of $\mathcal{L}_{T''}$ with the symbol $f$. It remains to show

$$T' \vdash f(\vec{x}) = \tau\bigl[[f]^{\prec}_{\vec{x}};\vec{x}\bigr] \ . \tag{$\dagger_8$}$$

For that we need the following auxiliary property which is proved by induction on the structure of subterms $\rho[f;\vec{x}]$ of $\tau$:

$$T' \vdash \Gamma^{\tau}_{\rho}[f;\vec{x}] \to \rho[f;\vec{x}] = \rho\bigl[[f]^{\prec}_{\vec{x}};\vec{x}\bigr], \tag{$\dagger_9$}$$

Take any numbers $\vec{x}$ any subterm $\rho$ of $\tau$ such that $\Gamma^{\tau}_{\rho}[f;\vec{x}]$ holds, and continue by case analysis of the term $\rho$.

The case when $\rho \equiv x_i$ is trivial. If $\rho \equiv g(\vec{\theta})$ then the terms $\vec{\theta}$ are governed in $\tau$ by $\Gamma^{\tau}_{\rho}$ and we obtain $g(\vec{\theta}[f;\vec{x}]) = g(\vec{\theta}[[f];\vec{x}])$ directly from IH.

If $\rho \equiv f(\vec{\theta})$ then the terms $\vec{\theta}$ are governed in $\tau$ by $\Gamma^{\tau}_{\rho}$. We have $\vec{\theta}[f;\vec{x}] \prec \vec{x}$ from the conditions of regularity $Reg^{\prec}_{\tau}[f]$ and thus

$$
\begin{aligned}
f(\vec{\theta}[f;\vec{x}]) = D\big(1, f(\vec{\theta}[f;\vec{x}]),0\big) = D\big(\vec{\theta}[f;\vec{x}] \prec_* \vec{x}, f(\vec{\theta}[f;\vec{x}]),0\big) &\overset{\text{IH}}{=} \\
= D\big(\vec{\theta}[[f];\vec{x}] \prec_* \vec{x}, f(\vec{\theta}[[f];\vec{x}]),0\big)&.
\end{aligned}
$$

If $\rho \equiv D(\theta_1,\theta_2,\theta_3)$ then the terms $\theta_1,\theta_2,\theta_3$ are governed in $\tau$ respectively by $\varGamma^\tau_\rho$, $\varGamma^\tau_\rho \land \theta_1 \neq 0$ and $\varGamma^\tau_\rho \land \theta_1 = 0$. We have by IH:

$$\theta_1[f;\vec{x}] = \theta_1[[f];\vec{x}] \tag{$\dagger_{10}$}$$

$$\theta_1[f;\vec{x}] \neq 0 \to \theta_2[f;\vec{x}] = \theta_2[[f];\vec{x}] \tag{$\dagger_{11}$}$$

$$\theta_1[f;\vec{x}] = 0 \to \theta_3[f;\vec{x}] = \theta_3[[f];\vec{x}]. \tag{$\dagger_{12}$}$$

Thus

$$
\begin{aligned}
&D\big(\theta_1[f;\vec{x}],\theta_2[f;\vec{x}],\theta_3[f;\vec{x}]\big) \overset{(\dagger_{11}),(\dagger_{12})}{=} D\big(\theta_1[f;\vec{x}],\theta_2[[f];\vec{x}],\theta_3[[f];\vec{x}]\big) \overset{(\dagger_{10})}{=} \\
&\quad = D\big(\theta_1[[f];\vec{x}],\theta_2[[f];\vec{x}],\theta_3[[f];\vec{x}]\big).
\end{aligned}
$$

With the auxiliary property proved we obtain $(\dagger_8)$ from

$$f(\vec{x}) = \tau[f;\vec{x}] \overset{(\dagger_9)}{=} \tau\big[[f]^{\prec}_{\vec{x}};\vec{x}\big]$$

by noting that $\tau$ is governed in $\tau$ by $\top$. □

**4.4.4 Theorem** *Primitive recursive functions are closed under regular recursion with measure.*

*Proof.* By inspection of the proof of Thm. 4.4.3 using Thm. 4.2.8. □

**4.4.5 Theorem** μ-*Recursive functions are closed under regular well-founded recursion.*

*Proof.* By inspection of the proof of Thm. 4.4.3 using Thm. 4.3.11. □

**4.4.6 Remark.** We have already encountered regular recursive definitions in Sect. 4.1. More elaborated examples will be studied in the next chapters. Below you will find some examples of non-trivial nested regular recursive definitions. Proving conditions of regularity in such cases usually requires knowing something about the function defined in advance. The bounded version of regular recursion comes here in rescue.

**4.4.7 Example.** We begin with the functional equation of the form (see [13, 29, 47])

$$f(x) = \textbf{if } x \neq 0 \textbf{ then } f\,f(x \mathbin{\dot-} 1) \textbf{ else } 0. \tag{1}$$

We claim that this is a recursive definition of the zero function regular in $x$. The following formulas are its conditions of regularity:

$$\vdash_{\text{PA}} \; x \neq 0 \to x \dot{-} 1 < x \tag{2}$$
$$\vdash_{\text{PA}} \; x \neq 0 \to f(x \dot{-} 1) < x. \tag{3}$$

We need to check only the second property for the first one holds trivially. For that, consider the following recursive definition associated with (1):

$$f(x) = \textbf{if } x \neq 0 \textbf{ then } [f]_x[f]_x(x \dot{-} 1) \textbf{ else } 0. \tag{4}$$

Here, the $[f]_x(y)$ is the restriction of $f$ to $y < x$, i.e.

$$[f]_x(y) \equiv \textbf{if } y < x \textbf{ then } f(y) \textbf{ else } 0.$$

We will show that the function $f$ defined by (4) satisfies the second condition of regularity. We provide two proofs of this fact.

One method is to prove directly (3) by complete induction on $x$. So suppose that $x \neq 0$ and consider two cases. If $x = 1$ then we have

$$f(1 \dot{-} 1) = f(0) = 0 < 1.$$

If $x \geq 2$ then $x \dot{-} 1 < x$ and therefore

$$f(x \dot{-} 2) < x \dot{-} 1. \tag{5}$$

by IH applied to the number $x \dot{-} 1$. We now have

$$\begin{aligned} f(x \dot{-} 1) &= [f]_{x \dot{-} 1}[f]_{x \dot{-} 1}(x \dot{-} 2) = [f]_{x \dot{-} 1} f(x \dot{-} 2) \overset{(5)}{=} \\ &= f\,f(x \dot{-} 2) = f\big(f(x \dot{-} 2) + 1 \dot{-} 1\big) \overset{(5),\text{IH}}{<} f(x \dot{-} 2) + 1 \overset{(5)}{<} x. \end{aligned}$$

Note the second use of IH applied to the number $f(x \dot{-} 2) + 1 < x$.

For an alternative proof of (3) we will use the following auxiliary property, which is is proved by induction on $x$:

$$\vdash_{\text{PA}} \; f(x) = 0. \tag{6}$$

The base is obvious. The induction step follows from

$$\begin{aligned} f(x+1) &= [f]_{x+1}[f]_{x+1}(x + 1 \dot{-} 1) = [f]_{x+1}[f]_{x+1}(x) = \\ &= [f]_{x+1} f(x) \overset{\text{IH}}{=} [f]_{x+1}(0) = f(0) = 0. \end{aligned}$$

We are now in position to prove (3). Suppose that $x \neq 0$. We then have

$$f(x \dot{-} 1) \overset{(6)}{=} 0 \leq x \dot{-} 1 < x.$$

**4.4.8 Example.** Next we consider the functional equation of the form

$$f(x) = \textbf{if } x < 101 \textbf{ then } f\,f(x+11) \textbf{ else } x \dot{-} 10. \tag{1}$$

We claim this is a recursive definition of the celebrated McCarthy 91 function:

$$f_{91}(x) = \begin{cases} 91 & \text{if } x < 101, \\ x \dot{-} 10 & \text{if } x \geq 101. \end{cases}$$

The definition is regular in the measure $101 \dot{-} x$ and the following formulas are its conditions of regularity:

$$\vdash_{\mathrm{PA}} \; x < 101 \rightarrow 101 \dot{-} (x+11) < 101 \dot{-} x \tag{2}$$

$$\vdash_{\mathrm{PA}} \; x < 101 \rightarrow 101 \dot{-} f(x+11) < 101 \dot{-} x. \tag{3}$$

We need to check only the second condition, for the first one follows from

$$\vdash_{\mathrm{PA}} \; a < 101 \rightarrow 101 \dot{-} b < 101 \dot{-} a \leftrightarrow b > a.$$

For that consider the following recursive definition associated with (1):

$$f(x) = \textbf{if } x < 101 \textbf{ then } [f]_x[f]_x(x+11) \textbf{ else } x \dot{-} 10. \tag{4}$$

Here, the $[f]_x(y)$ is the restriction of $f$ to $y$ s.t. $101 \dot{-} y < 101 \dot{-} x$, i.e.

$$[f]_x(y) \equiv \textbf{if } 101 \dot{-} y < 101 \dot{-} x \textbf{ then } f(y) \textbf{ else } 0.$$

We will show that the function $f$ defined by (4) satisfies the second condition of regularity by proving its equivalent formulation

$$\vdash_{\mathrm{PA}} \; x < 101 \rightarrow f(x+11) > x. \tag{5}$$

We give here two proofs of this fact.

One method is to prove (5) directly by induction with measure $101 \dot{-} x$. So suppose that $x < 101$ and consider two cases. If $x + 11 \geq 101$ then we have

$$f(x+11) = x + 11 \dot{-} 10 = x + 1 > x.$$

If $x + 11 < 101$ then $x + 11 + 11 > x + 11 > x$ and thus

$$101 \dot{-} (x+11+11) < 101 \dot{-} (x+11) < 101 \dot{-} x.$$

By applying IH to the number $x + 11$ we obtain

$$f(x+11+11) > x+11. \tag{6}$$

Hence

$$101 \dot{-} f(x+11+11) < 101 \dot{-} (x+11). \tag{7}$$

From (6) we also obtain

$$101 \dot{-} \big(f(x+11+11) \dot{-} 11\big) < 101 \dot{-} x. \tag{8}$$

Consequently

$$\begin{aligned} f(x+11) &= [f]_{x+11}[f]_{x+11}(x+11+11) = [f]_{x+11} f(x+11+11) \overset{(7)}{=} \\ &= f\,f(x+11+11) \overset{(6)}{=} f\big(f(x+11+11) \dot{-} 11 + 11\big) \overset{(8),\mathrm{IH}}{>} \\ &> f(x+11+11) \dot{-} 11 \overset{(6)}{>} x + 11 \dot{-} 11 = x. \end{aligned}$$

Note the second use of IH, now for the number $f(x+11+11) \dot{-} 11$.

For an alternative proof of (5) we will use the following auxiliary property of the function $f$ defined by (4):

$$\vdash_{\mathrm{PA}} x < 101 \to f(x) = 91. \tag{9}$$

This is proved by induction with measure $101 \dot{-} x$. So suppose that $x < 101$. We then have $101 \dot{-} (x+11) < 101 \dot{-} x$ and thus

$$f(x) = [f]_x[f]_x(x+11) = [f]_x f(x+11).$$

Consider now two cases. If $x+11 < 101$ then $91 > x$ and thus $101 \dot{-} 91 < 101 \dot{-} x$. We now proceed

$$[f]_x f(x+11) \overset{\mathrm{IH}}{=} [f]_x(91) = f(91) \overset{\mathrm{IH}}{=} 91.$$

If $x + 11 \geq 101$ then $101 \dot{-} (x+1) < 101 \dot{-} x$ since $x+1 > x$ and thus

$$[f]_x f(x+11) = [f]_x(x+11 \dot{-} 10) = [f]_x(x+1) = f(x+1) \overset{\mathrm{IH}}{=} 91.$$

This proves (9). We are now in position to prove (5). Suppose that $x < 101$. Consider two cases. If $x + 11 < 101$ then $90 > x$ and we obtain

$$f(x+11) \overset{(9)}{=} 91 > 90 > x.$$

Otherwise $x + 11 \geq 101$ and then

$$f(x+11) = x + 11 \dot{-} 10 = x + 1 > x.$$

In either case $f(x+11) > x$. Consequently $101 \dot{-} f(x+11) < 101 \dot{-} x$.

## 4.5 Computation Model

**4.5.1 Introduction.** In this section we will introduce computational model based on reduction of terms. Every program $P$ is a property of some function $f$ which can be used as a rewriting rule to evaluate this function. The program $P$ has assigned a precondition describing $\varphi[\vec{x}]$ which elements $\vec{x}$ can be used as inputs. Regularity conditions for the program $P$ guarantees that computation terminates for every input $\vec{x}$ which satisfies the precondition $\varphi$ yielding the correct value $f(\vec{x})$. Regular recursive definitions are special cases of such programs with always satisfied precondition $\top$.

**4.5.2 Regular programs.** Consider the following property of a function $f$:

$$\vdash_{\mathrm{PA}} \varphi[\vec{x}] \to f(\vec{x}) = \tau[f; \vec{x}]. \tag{1}$$

We assign to each occurrence of the recursive application $f(\vec{\rho})$ in $\tau$ with the governing condition $\Gamma^{\tau}_{f(\vec{\rho})}$ the following *(extended) condition of regularity*:

$$\varphi[\vec{x}] \land \Gamma^{\tau}_{f(\vec{\rho})} \to \vec{\rho} \prec \vec{x} \land \varphi[\vec{\rho}].$$

We say that the property (1) is a *program regular in the well-founded relation* $\prec$ if

$$\vdash_{\mathrm{PA}} \varphi[\vec{x}] \land \Gamma^{\tau}_{f(\vec{\rho})} \to \vec{\rho} \prec \vec{x} \land \varphi[\vec{\rho}]$$

for every condition of regularity of $f$ in $\tau$. The formula $\varphi$ is called the *precondition* of the program.

*Fixing notation.* We keep the notation introduced in this paragraph fixed until the end of this section where we prove in Thm. 4.5.6 that the regular program (1) computes the function $f$ for every input $\vec{x}$ which satisfies its precondition $\varphi[\vec{x}]$. By $g_1, \ldots, g_k$ we denote below are all auxiliary functions occurring in the term $\tau$.

**4.5.3 Computational model.** *Computational* terms (*C-terms* for short) for the above regular program are closed terms which contains only the symbols occurring in the program and the successor function $S$. We take $\underline{x}$ as an abbreviation for the numeral $S^x(0)$. We will need also a notation for $n$-tuple of monadic numerals and define $\underline{x_1, \ldots, x_n}$ as an abbreviation for $\underline{x_1}, \ldots, \underline{x_n}$.

We now describe a possibly infinite process by which we can *reduce* a closed C-term until we obtain a numeral which cannot be further reduced. If a closed C-term $\rho$ is not a numeral then it must contain at least one occurrence, called the *redex* (for *reducible expression*), of one of the closed C-terms listed below on the left-hand side:

$$\begin{aligned} D(\underline{0}, \theta_2, \theta_3) &\rhd_1 \theta_3 \\ D(\underline{x+1}, \theta_2, \theta_3) &\rhd_1 \theta_2 \\ g_i(\underline{\vec{y}}) &\rhd_1 \underline{g_i(\vec{y})} \\ f(\underline{\vec{x}}) &\rhd_1 \tau[f; \underline{\vec{x}}]. \end{aligned}$$

Note that the first occurrence of $g_i$ is a function symbol whereas the second one is the corresponding denotation of this symbol (in the standard model).

*One step reduction* consists of locating the *leftmost* redex in $\rho_1$ and replacing it by its *contraction*, which is the closed term on the corresponding right-hand side. By the replacement we obtain again a closed C-term $\rho_2$. We note that the term $\rho_2$ is uniquely determined by $\rho_1$.

We say that $\rho_1$ *reduces to* $\rho_2$ *in* $k$ *steps*, in symbols $\rho_1 \rhd_k \rho_2$, if there is a finite one step reduction sequence of length $k \geq 0$ of closed C-terms $\theta_0, \theta_1, \ldots, \theta_k$ such that $\theta_0 \equiv \rho_1$, $\theta_k \equiv \rho_2$, and for each $i < k$ we obtain the term $\theta_{i+1}$ by one step reduction from the term $\theta_i$. We write $\rho_1 \rhd \rho_2$ if $\rho_1 \rhd_k \rho_2$ for some $k$.

It is not difficult to see that for every closed C-terms $\rho$, $\rho_1$ and $\rho_2$ we have

$$\text{if } \rho \rhd \rho_1 \text{ and } \rho \rhd \rho_2 \text{ then } \rho_1 \rhd \rho_2 \text{ or } \rho_2 \rhd \rho_1.$$

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

**4.5.4 Lemma** *For every $\vec{x}$ such that $\varphi[\vec{x}]$, if $f(\underline{\vec{x}}) \rhd \underline{y}$ then $f(\vec{x}) = y$.*

*Proof.* For that we need the following auxiliary property which holds for every number $k$ and for every subterm $\rho[f; \vec{x}]$ of the term $\tau$:

$$\text{for every } \vec{x} \text{ and } y, \text{ if } \varphi[\vec{x}] \wedge \Gamma^\tau_\rho[f; \vec{x}] \text{ and } \rho[f; \underline{\vec{x}}] \rhd_k \underline{y} \text{ then } \rho[f; \vec{x}] = y. \qquad (\dagger_1)$$

Here $\Gamma^\tau_\rho$ is the governing condition of $\rho$ in $\tau$.

The auxiliary property is proved by double induction: the outer one is by complete induction on $k$ and the inner by induction on the structure of the subterm $\rho[f; \vec{x}]$ of $\tau$. So take any $\vec{x}$ and $y$ such that $\varphi[\vec{x}] \wedge \Gamma^\tau_\rho[f; \vec{x}]$ and

$$\rho[f; \underline{\vec{x}}] \rhd_k \underline{y}\ , \qquad (\dagger_2)$$

and continue by the case analysis of $\rho$.

If $\rho \equiv x_i$ then from $(\dagger_2)$ we obtain $k = 0$ and $\underline{x_i} \equiv \underline{y}$. Consequently $x_i = y$.

If $\rho \equiv 0$ then from $(\dagger_2)$ we obtain $k = 0$ and $\underline{0} \equiv \underline{y}$. Consequently $0 = y$.

If $\rho \equiv S(\theta)$ then from $(\dagger_2)$ we can conclude that there is a number $v$ such that $\theta[f; \underline{\vec{x}}] \rhd_k \underline{v}$ and $S(\underline{v}) \equiv \underline{y}$. By the inner IH we have $\theta[f; \vec{x}] = v$ and therefore $S(\theta[f; \vec{x}]) = S(v) = y$.

If $\rho \equiv D(\theta_1, \theta_2, \theta_3)$ then the terms $\theta_1, \theta_2, \theta_3$ are governed in $\tau$ respectively by $\Gamma^\tau_\rho$, $\Gamma^\tau_\rho \wedge \theta_1 \neq 0$ and $\Gamma^\tau_\rho \wedge \theta_1 = 0$. From ($\dagger_2$) we obtain that there are numbers $k_1 < k$ and $v_1$ such that $\theta_1[f; \underline{\vec{x}}] \rhd_{k_1} \underline{v_1}$. By the outer IH $\theta_1[f; \vec{x}] = v_1$. We consider two subcases. If $v_1 \neq 0$ then the governing condition $\Gamma^\tau_{\theta_2}[f; \vec{x}]$ is trivially satisfied and therefore we have the following for some number $k_2$:

$$D\big(\theta_1[f; \underline{\vec{x}}], \theta_2[f; \underline{\vec{x}}], \theta_3[f; \underline{\vec{x}}]\big) \rhd_{k_1} D\big(\underline{v_1}, \theta_2[f; \underline{\vec{x}}], \theta_3[f; \underline{\vec{x}}]\big) \rhd_1 \theta_2[f; \underline{\vec{x}}] \rhd_{k_2} \underline{y}.$$

Also $k_1 + 1 + k_2 = k$. By another outer IH $\theta_2[f; \vec{x}] = y$ and hence

$$D\big(\theta_1[f; \vec{x}], \theta_2[f; \vec{x}], \theta_3[f; \vec{x}]\big) = D\big(v_1, \theta_2[f; \vec{x}], \theta_3[f; \vec{x}]\big) = \theta_2[f; \vec{x}] = y.$$

The subcase $v_1 = 0$ is similar.

If $\rho \equiv g(\vec{\theta})$, where $g$ is an auxiliary $m$-ary function symbol, then the terms $\vec{\theta} \equiv \theta_1, \dots, \theta_m$ are governed in $\tau$ by $\Gamma^\tau_\rho$. From ($\dagger_2$) we obtain that there are numbers $\vec{k} \equiv k_1, \dots, k_m$ and $\vec{v} \equiv v_1, \dots, v_m$ such that $\theta_i[f; \underline{\vec{x}}] \rhd_{k_i} \underline{v_i}$ for every $i = 1, \dots, m$. Moreover

$$g(\vec{\theta}[f; \underline{\vec{x}}]) \rhd_{\sum_i k_i} g(\underline{\vec{v}}) \rhd_1 \underline{g(\vec{v})} \equiv \underline{y}$$

and $\sum_i k_i + 1 = k$. For every $i = 1, \dots, m$ we have $k_i < k$ and thus $\theta_i[f; \vec{x}] = v_i$ by the outer IH. Consequently $g(\vec{\theta}[f; \vec{x}]) = g(\vec{v}) = y$.

If $\rho \equiv f(\vec{\theta})$ then the terms $\vec{\theta} \equiv \theta_1, \dots, \theta_n$ are governed in $\tau$ by $\Gamma^\tau_\rho$. From ($\dagger_2$) we obtain that there are numbers $\vec{k} \equiv k_1, \dots, k_n$ and $\vec{v} \equiv v_1, \dots, v_n$ such that $\theta_i[f; \underline{\vec{x}}] \rhd_{k_i} \underline{v_i}$ for every $i = 1, \dots, n$. Moreover

$$f(\vec{\theta}[f; \underline{\vec{x}}]) \rhd_{\sum_i k_i} f(\underline{\vec{v}}) \rhd_1 \tau[f; \underline{\vec{v}}] \rhd_l \underline{y}$$

for some number $l$ such that $\sum_i k_i + 1 + l = k$. Clearly $k_i < k$ and $l < k$. By the outer IH's we have $\theta_i[f; \vec{x}] = v_i$ for every $i = 1, \dots, n$ and $\tau[f; \vec{v}] = y$. Consequently $f(\vec{\theta}[f; \vec{x}]) = f(\vec{v}) = \tau[f; \vec{v}] = y$.

With the auxiliary property ($\dagger_1$) proved we are now ready to prove the desired claim. If $\varphi[\vec{x}]$ and $f(\underline{\vec{x}}) \rhd \underline{y}$ then

$$f(\underline{\vec{x}}) \rhd_1 \tau[f; \underline{\vec{x}}] \rhd_k \underline{y}$$

for some number $k$. From ($\dagger_1$) we obtain $\tau[f; \vec{x}] = y$ by noting that the term $\tau$ is governed in $\tau$ by $\top$. Consequently $f(\vec{x}) = \tau[f; \vec{x}] = y$. □

**4.5.5 Lemma** *For every $\vec{x}$ such that $\varphi[\vec{x}]$, if $f(\vec{x}) = y$ then $f(\underline{\vec{x}}) \rhd \underline{y}$.*

*Proof.* By $\prec$-well-founded induction on $\vec{x}$ we prove that

$$\text{for every } y, \text{ if } \varphi[\vec{x}] \text{ and } f(\vec{x}) = y \text{ then } f(\underline{\vec{x}}) \rhd_k \underline{y} \text{ for some } k. \qquad (\dagger_1)$$

Take any $y$ such that $\varphi[\vec{x}]$ and $f(\vec{x}) = y$, and prove by the inner induction on the structure of subterms $\rho[f;\vec{x}]$ of the term $\tau$ the following property:

$$\text{for every } z \text{, if } \mathit{\Gamma}^{\tau}_{\rho}[f;\vec{x}] \text{ and } \rho[f;\vec{x}] = z \text{ then } \rho[f;\underline{\vec{x}}] \rhd_k \underline{z} \text{ for some } k. \qquad (\dagger_2)$$

Here $\mathit{\Gamma}^{\tau}_{\rho}$ is the governing condition of $\rho$ in $\tau$. So take any subterm $\rho[f;\vec{x}]$ of $\tau$ such that $\mathit{\Gamma}^{\tau}_{\rho}[f;\vec{x}]$ holds and any number $z$ such that

$$\rho[f;\vec{x}] = z, \qquad (\dagger_3)$$

and continue by the case analysis of $\rho$.

If $\rho \equiv x_i$ then $x_i = z$ from $(\dagger_3)$ and thus $\underline{x_i} \equiv \underline{z} \rhd_0 \underline{z}$. It suffices to take $k = 0$.

If $\rho \equiv 0$ then $0 = z$ from $(\dagger_3)$ and thus $\underline{0} \equiv \underline{z} \rhd_0 \underline{z}$. It suffices to take $k = 0$.

If $\rho \equiv S(\theta)$ then the term $\theta$ is governed in $\tau$ by $\mathit{\Gamma}^{\tau}_{\rho}$. From $(\dagger_3)$ we obtain that there is a number $v$ such that $\theta[f;\vec{x}] = v$ and $S(v) = z$. By the inner IH, there exists a number $k$ such that $\theta[f;\underline{\vec{x}}] \rhd_k \underline{v}$. Hence $S(\theta[f;\underline{\vec{x}}]) \rhd_k S(\underline{v}) \equiv \underline{z}$.

If $\rho \equiv D(\theta_1,\theta_2,\theta_3)$ then the terms $\theta_1,\theta_2,\theta_3$ are governed in $\tau$ respectively by $\mathit{\Gamma}^{\tau}_{\rho}$, $\mathit{\Gamma}^{\tau}_{\rho} \wedge \theta_1 \neq 0$ and $\mathit{\Gamma}^{\tau}_{\rho} \wedge \theta_1 = 0$. From the assumption $(\dagger_3)$ we obtain that there is a number $v_1$ such that

$$\theta_1[f;\vec{x}] = v_1 \wedge \big(v_1 \neq 0 \wedge \theta_2[f;\vec{x}] = z \vee v_1 = 0 \wedge \theta_3[f;\vec{x}] = z\big).$$

By the inner IH, there exists a number $k_1$ such that $\theta_1[f;\underline{\vec{x}}] \rhd_{k_1} \underline{v_1}$. Now we consider two subcases. If $v_1 \neq 0$ then the governing condition $\mathit{\Gamma}^{\tau}_{\theta_2}[f;\vec{x}]$ is trivially satisfied and therefore, by the inner IH, there exists a number $k_2$ such that $\theta_2[f;\underline{\vec{x}}] \rhd_{k_2} \underline{z}$. We thus obtain

$$D\big(\theta_1[f;\underline{\vec{x}}],\theta_2[f;\underline{\vec{x}}],\theta_3[f;\underline{\vec{x}}]\big) \rhd_{k_1} D\big(\underline{v_1},\theta_2[f;\underline{\vec{x}}],\theta_3[f;\underline{\vec{x}}]\big) \rhd_1 \theta_2[f;\underline{\vec{x}}] \rhd_{k_2} \underline{z}.$$

It suffices to take $k = k_1 + 1 + k_2$. The subcase $v_1 = 0$ is similar.

If $\rho \equiv g(\vec{\theta})$, where $g$ is an auxiliary $m$-ary function symbol, then the terms $\vec{\theta} \equiv \theta_1,\dots,\theta_m$ are governed in $\tau$ by $\mathit{\Gamma}^{\tau}_{\rho}$. From $(\dagger_3)$ we obtain that there are numbers $\vec{v} \equiv v_1,\dots,v_m$ such that

$$\bigwedge_{i=1}^{m} \theta_i[f;\vec{x}] = v_i \wedge g(\vec{v}) = z.$$

By the inner IH, there exist numbers $\vec{k} \equiv k_1,\dots,k_m$ such that $\theta_i[f;\underline{\vec{x}}] \rhd_{k_i} \underline{v_i}$ for every $i = 1,\dots,m$. We thus have

$$g\big(\vec{\theta}[f;\underline{\vec{x}}]\big) \rhd_{\sum_i k_i} g(\underline{\vec{v}}) \rhd_1 \underline{g(\vec{v})} \equiv \underline{z}.$$

It suffices to take $k = \sum_i k_i + 1$.

If $\rho \equiv f(\vec{\theta})$ then the terms $\vec{\theta} \equiv \theta_1,\dots,\theta_n$ are governed in $\tau$ by $\mathit{\Gamma}^{\tau}_{\rho}$. From $(\dagger_3)$ we obtain that there are numbers $\vec{v} \equiv v_1,\dots,v_n$ such that

$$\bigwedge_{i=1}^{n} \theta_i[f;\vec{x}] = v_i \wedge f(\vec{v}) = z.$$

By the inner IH there exist numbers $\vec{k} \equiv k_1, \dots, k_n$ such that $\theta_i[f;\underline{\vec{x}}] \rhd_{k_i} \underline{v_i}$ for every $i = 1, \dots, n$. By regularity $\vec{v} \prec \vec{x}$ and thus, by the outer IH applied to the numbers $\vec{v}$, there exists a number $l$ such that $f(\underline{\vec{v}}) \rhd_l \underline{z}$. We obtain

$$f\big(\vec{\theta}[f;\underline{\vec{x}}]\big) \rhd_{\sum_i k_i} f(\underline{\vec{v}}) \rhd_l \underline{z}.$$

It suffices to take $k = \sum_i k_i + l$.

With the auxiliary property proved we can now finish the proof of the inductive step of the outer induction of $(\dagger_1)$. From $f(\vec{x}) = y$ we get $\tau[f;\vec{x}] = y$ and thus, by $(\dagger_2)$ and by noting that the term $\tau$ is governed in $\tau$ by $\top$, there is a number $l$ such that $\tau[f;\underline{\vec{x}}] \rhd_l \underline{y}$. We then obtain

$$f(\underline{\vec{x}}) \rhd_1 \tau[f;\underline{\vec{x}}] \rhd_l \underline{y}.$$

It suffices to take $k = 1 + l$. □

**4.5.6 Theorem** *For every $\vec{x}$ and $y$ s.t. $\varphi[\vec{x}]$, $f(\vec{x}) = y$ if and only if $f(\underline{\vec{x}}) \rhd \underline{y}$.*

*Proof.* Directly from Lemma 4.5.4 and Lemma 4.5.5. □

**4.5.7 Example.** For given a function $f(x)$ and predicate $P(x)$, the *unlimited iteration* of $f$ is the unary function $f^*(x)$ satisfying

$$f^*(x) = \begin{cases} f^k(x) & \text{if } Pf^k(x) \text{ and } k \text{ is the least such number,} \\ 0 & \text{if there is no such number.} \end{cases}$$

We iterate the function $f$ until the condition $Pf^k(x)$ is met. The iteration can be introduced into PA with the following contextual definition:

$$\begin{aligned} f^*(x) = y \leftrightarrow {} & \exists k\big(Pf^k(x) \wedge \forall l < k \neg Pf^k(x) \wedge y = f^k(x)\big) \vee \\ & \neg\exists k\, Pf^k(x) \wedge y = 0. \end{aligned}$$

Note that its existence and uniqueness conditions are trivially satisfied.

Computable functions are not closed under the operator of unlimited iteration. That is, the function $f^*$ might not be computable even in cases when both $f$ and $P$ are. Indeed, consider an interpreter of a simple programming language over N. If the language is sufficiently strong then the interpreter is a partial function with no total computable completion. On the other hand, one can easily easily express a total completion of such interpreter with the help of unlimited iteration $f^*$ for suitable $f$ and $P$; the function $f$ realizes one computation step and the predicates $P$ decides when the computation terminates.

Nevertheless, it is still possible to compute the application $f^*(x)$ for some inputs. Namely, the following property

$$\vdash_{\mathrm{PA}} \exists k\, P f^k(x) \rightarrow f^*(x) = \textbf{if } P(x) \textbf{ then } x \textbf{ else } f^*\, f(x)$$

can be taken as a program for evaluating $f^*(x)$ for the inputs $x$ satisfying the precondition $\exists k\, P f^k(x)$. The program is regular in the measure $m(x)$, which is defined by regular minimalization

$$m(x) = \mu k\big[\exists l\, P f^l(x) \rightarrow P f^k(x)\big].$$

The following is the condition of regularity of the program

$$\vdash_{\mathrm{PA}} \exists k\, P f^k(x) \land \neg P(x) \rightarrow m\, f(x) < m(x) \land \exists k\, P f^k f(x).$$

The property follows from

$$\vdash_{\mathrm{PA}} \exists k\, P f^k(x) \land \neg P(x) \rightarrow m\, f(x) + 1 = m(x).$$

# Chapter 5
# Programming Language

In this chapter we describe a programming language with extensible syntax of programming constructs. The language of expressions is extended with a powerful generalization of case constructs and pattern matching known from declarative programming languages. These new constructs, called *case discrimination terms*, have flexible syntax which legality must be certified by a formal proof in PA. Each case discrimination term has assigned certain *precondition* prescribing for which inputs the case analysis must be provably *pairwise-disjoint* and *exhaustive.*

In Sect. 5.1 we illustrate the language by giving several examples of definitions of the integer square root function. In Sect. 5.2 we describe the language of *generalized terms*, which are expressions with flexible syntax of case discrimination. In Sect. 5.3 we show that PA admits a very flexible kind of extensions by *regular recursive definitions* of functions. This is our most expressive scheme of recursive definitions formalized within PA. Such definitions serve a dual purpose. They described extensionally the properties of the defined functions and at the same time they serve intentionally as rules for the computation of these objects. We can significantly improve the readability of recursive definitions by writing them in *clausal form* (see Sect. 5.4). The language of regular recursive definitions is then extended to include definitions of predicates.

## 5.1 Introduction

**5.1.1 Introduction.** In this section we illustrate case discrimination terms (conditionals) with some examples. These new constructs are generalizations of case constructs and pattern matching known from declarative programming languages. We do not dwell here on the syntax of these constructs as we consider them almost self-explanatory.

Our conditionals have flexible syntax which legality must be certified by a formal proof. Consider, for instance, the following explicit definition *by cases*:

$$\begin{aligned} f(x) = \mathbf{case} \\ & R(x) \Rightarrow 0 \\ & Q(x) \Rightarrow 1 \\ \mathbf{end}. \end{aligned}$$

For the definition to be correct it is sufficient and necessary that the predicates $R$ and $Q$ are mutually *disjoint* and *complete*, i.e. $\neg\big(R(x) \land Q(x)\big)$ and $R(x) \lor Q(x)$ for every number $x$. These conditions are strong semantic conditions which can be checked in general by a formal proof. It may be even the case that the conditions are undecidable in PA.

**5.1.2 Case study: integer square root.** We start our presentation of the language by considering the problem of computing the integer square root $\lfloor\sqrt{x}\rfloor$ of a natural number $x$. The function $\lfloor\sqrt{x}\rfloor$ can be introduced into PA by the following contextual definition:

$$\lfloor\sqrt{x}\rfloor = y \leftrightarrow y^2 \leq x < (y+1)^2.$$

We also intend to demonstrate that simple recursion such as primitive recursion does not always lead to a definition which, when used as rewriting rules, is efficient. A computationally optimal definition usually needs a more complex recursion/discrimination. Some of these examples are from [26, 28].

**5.1.3 Primitive recursion.** We can find a primitive recursive derivation of the integer square root function by assuming as IH that

$$\lfloor\sqrt{x}\rfloor^2 \leq x < (\lfloor\sqrt{x}\rfloor + 1)^2$$

holds and then considering the relation between $x+1$ and $(\lfloor\sqrt{x}\rfloor + 1)^2$. If

$$\lfloor\sqrt{x}\rfloor^2 \leq x < x+1 < (\lfloor\sqrt{x}\rfloor + 1)^2$$

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

$$\lfloor\sqrt{x}\rfloor^2 \leq x < (\lfloor\sqrt{x}\rfloor + 1)^2 = x+1 < (\lfloor\sqrt{x}\rfloor + 2)^2$$

and it suffices to set $\lfloor\sqrt{x+1}\rfloor := \lfloor\sqrt{x}\rfloor + 1$.

The idea is expressed by the following regular recursive definition:

```
$\lfloor\sqrt{x}\rfloor$ = case
            $x = 0 \Rightarrow 0$
            $x = y + 1 \Rightarrow_y$
              case
                $x < (\lfloor\sqrt{y}\rfloor + 1)^2 \Rightarrow \lfloor\sqrt{y}\rfloor$
                $x \geq (\lfloor\sqrt{y}\rfloor + 1)^2 \Rightarrow \lfloor\sqrt{y}\rfloor + 1$
              end
            end.
```

Its conditions of regularity

$$\vdash_{\mathrm{PA}} x = y + 1 \land x < (\lfloor\sqrt{y}\rfloor + 1)^2 \to y < x$$
$$\vdash_{\mathrm{PA}} x = y + 1 \land x \geq (\lfloor\sqrt{y}\rfloor + 1)^2 \to y < x$$

are trivially satisfied.

The expression on the right side of the defining equation applies two conditionals. The first one is *monadic discrimination* on whether $x = 0$ or $\exists y\, x = y + 1$. If the latter then there must be a unique number $y$ s.t. $x = y + 1$. We say that the *local variable* $y$ gets its value by *pattern matching*, where we match the input $x$ against the *pattern* $y + 1$. The locality of the variable $y$ is indicated by the subscript $y$ in the implication $\Rightarrow_y$. The second conditional is *dichotomy discrimination* on whether or not $x < (\lfloor\sqrt{y}\rfloor + 1)^2$.

We can significantly improve the readability of recursive definitions by writing them in clausal form; in this case by *unfolding* of two conditionals in the above definition we obtain the following three clauses

$$\begin{aligned}
&\lfloor\sqrt{x}\rfloor = 0 \leftarrow x = 0\\
&\lfloor\sqrt{x}\rfloor = \lfloor\sqrt{y}\rfloor \leftarrow x = y + 1 \land x < (\lfloor\sqrt{y}\rfloor + 1)^2\\
&\lfloor\sqrt{x}\rfloor = \lfloor\sqrt{y}\rfloor + 1 \leftarrow x = y + 1 \land x \geq (\lfloor\sqrt{y}\rfloor + 1)^2.
\end{aligned}$$

The clauses are just ordinary formulas even though their implications are customarily written in the converse form. This can be further simplified by eliminating the variable $x$ by substituting it for 0 in the first clause and for $y + 1$ in the next two clauses. After simplification we obtain

$$\begin{aligned}
&\lfloor\sqrt{0}\rfloor = 0\\
&\lfloor\sqrt{y+1}\rfloor = \lfloor\sqrt{y}\rfloor \leftarrow y + 1 < (\lfloor\sqrt{y}\rfloor + 1)^2\\
&\lfloor\sqrt{y+1}\rfloor = \lfloor\sqrt{y}\rfloor + 1 \leftarrow y + 1 \geq (\lfloor\sqrt{y}\rfloor + 1)^2.
\end{aligned}$$

Finally, we rename the variable $y$ by $x$ in the last two clauses to obtain

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

Note that the last set of clauses have the form of a primitive recursive definition with dichotomy discrimination in the recursive case.

**5.1.4 Assignments.** The program for $\lfloor\sqrt{x}\rfloor$ from Par. 5.1.3, though quite pleasant mathematically, is infeasible in practice for many reasons. One of them is the twofold occurrence of $\lfloor\sqrt{x}\rfloor$ in the recursive clauses: once in the test and once in the result. This leads to the exponential explosion of computation time. The explosion can be prevented by an *assignment* $\lfloor\sqrt{x}\rfloor = r$ in the following definition:

```
⌊√x⌋ = case
          $x = 0 \Rightarrow 0$
          $x = y + 1 \Rightarrow_y$ let $\lfloor\sqrt{y}\rfloor = r$ in
            case
              $x < (r+1)^2 \Rightarrow r$
              $x \geq (r+1)^2 \Rightarrow r + 1$
            end
        end.
```

Note that only $x$ tests are needed. The following is its clausal form:

$$\lfloor\sqrt{0}\rfloor = 0$$
$$\lfloor\sqrt{x+1}\rfloor = r \leftarrow \lfloor\sqrt{x}\rfloor = r \land x + 1 < (r+1)^2$$
$$\lfloor\sqrt{x+1}\rfloor = r + 1 \leftarrow \lfloor\sqrt{x}\rfloor = r \land x + 1 \leq (r+1)^2.$$

**5.1.5 Bottom-up program.** The last program for $\lfloor\sqrt{x}\rfloor$ which uses primitive recursive and assignments can be improved as follows. Primitive recursion is an example of top-down approach for solving problems; in this case the computation of $\lfloor\sqrt{x}\rfloor$ goes down from $x$ to $x - 1$ until it reaches 0 and then it does the comparisons and the incrementation by one on the way back. We will shorten the computation of $\lfloor\sqrt{x}\rfloor$ by using a bottom-up approach in which we search (starting from 0) the smallest number $r$ such that $x < (r+1)^2$.

This is done with the help the binary function $f(x, r)$ defined by

```
$f(x,r)$ = case
             $x < (r+1)^2 \Rightarrow r$
             $x \geq (r+1)^2 \Rightarrow f(x, r+1)$
           end.
```

This is an example of definition by *backward recursion*, where $r^2$ grows towards $x$, i.e. by recursion with measure $x \dot{-} r^2$. Its condition of regularity

$$\vdash_{\mathrm{PA}} x \geq (r+1)^2 \to x \dot{-} (r+1)^2 < x \dot{-} r^2$$

is trivially satisfied. The following is the clausal form of the definition:

$$f(x,r) = r \leftarrow x < (r+1)^2$$
$$f(x,r) = f(x, r+1) \leftarrow x \geq (r+1)^2.$$

The auxiliary function satisfies

$$\vdash_{\mathrm{PA}} r^2 \leq x \to f(x,r)^2 \leq x < (f(x,r)+1)^2$$

and thus we can take the following identity

$$\vdash_{\text{PA}} \lfloor\sqrt{x}\rfloor = f(x,0)$$

as an alternative program for computing the integer square root function. Note that computation of $\lfloor\sqrt{x}\rfloor$ now takes order $\lfloor\sqrt{x}\rfloor$ time.

**5.1.6 Bottom-up program revisited.** We can improve the bottom-up program for $\lfloor\sqrt{x}\rfloor$ by saving the squaring operation $(r+1)^2$ in the test $x < (r+1)^2$ which is repeatedly done for every recursive call. This can be done with the help of a ternary function $f(x,r,s)$ with the additional accumulator $s$ such that we have $f(x,r,r^2) = \lfloor\sqrt{x}\rfloor$ provided $r^2 \le x$. As the second argument goes from $r$ to $r+1$ the accumulator goes from $s = r^2$ to $s_1 = (r+1)^2 = s + 2r + 1$. This arrangement reduces the squaring operation to the increment $2r+1$ which is very fast in binary representation of natural numbers.

The auxiliary accumulator function $f(x,r,s)$ is defined by backward recursion on the difference $x \dot{-} r$:

```
$f(x,r,s)$ = let $s + 2r + 1 = s_1$ in
             case
               $x < s_1 \Rightarrow r$
               $x \ge s_1 \Rightarrow f(x, r+1, s_1)$
             end
```

Its condition of regularity

$$\vdash_{\text{PA}} s + 2r + 1 = s_1 \land x \ge s_1 \to x \dot{-} (r+1) < x \dot{-} r$$

is trivially satisfied. The following is its clausal form

$$f(x,r,s) = r \leftarrow s + 2r + 1 = s_1 \land x < s_1$$
$$f(x,r,s) = f(x,r+1,s_1) \leftarrow s + 2r + 1 = s_1 \land x \ge s_1.$$

The function satisfies

$$\vdash_{\text{PA}} r^2 \le x \to f(x,r,r^2)^2 \le x < (f(x,r,r^2)+1)^2$$

and thus we can take the following identity

$$\vdash_{\text{PA}} \lfloor\sqrt{x}\rfloor = f(x,0,0)$$

as an alternative program for computing the integer square root function.

**5.1.7 A fast program by recursion on notation.** All programs considering so far share the same shortcomings: recursion goes exponentially longer that it should. We can obtain a fast $\mathcal{O}(\lg(x))$ program for $\lfloor\sqrt{x}\rfloor$ 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 $\lfloor\sqrt{10x}\rfloor \approx \lfloor\sqrt{10}\rfloor\lfloor\sqrt{x}\rfloor$

while it works well in the centennial notation because $\lfloor\sqrt{100x}\rfloor \approx 10\lfloor\sqrt{x}\rfloor$. The same works in the 4-*ary representation* of natural numbers, where for every $x$ there are unique numbers $y$ and $z$ such that $x = 4y + z$ and $z < 4$ holds. Note that we then have $\lfloor\sqrt{4x}\rfloor \approx 2\lfloor\sqrt{x}\rfloor$.

Thus assume as IH that we have for $y \neq 0$

$$\lfloor\sqrt{y}\rfloor^2 \leq y < (\lfloor\sqrt{y}\rfloor + 1)^2.$$

From the last we obtain $y \leq \lfloor\sqrt{y}\rfloor^2 + 2\lfloor\sqrt{y}\rfloor$ and so we get for $z < 4$:

$$\begin{aligned}(2\lfloor\sqrt{y}\rfloor)^2 \leq 4y \leq 4y + z \leq 4\lfloor\sqrt{y}\rfloor^2 + 8\lfloor\sqrt{y}\rfloor + z <&\\ < 4\lfloor\sqrt{y}\rfloor^2 + 8\lfloor\sqrt{y}\rfloor + 4 = (2\lfloor\sqrt{y}\rfloor + 2)^2.&\end{aligned}$$

This means that we have

$$2\lfloor\sqrt{y}\rfloor \leq \left\lfloor\sqrt{4y+z}\right\rfloor \leq 2\lfloor\sqrt{y}\rfloor + 1$$

and so the following identity is a fast program for $\lfloor\sqrt{x}\rfloor$:

```
$\lfloor\sqrt{x}\rfloor$ = let $x = 4y + z \land z < 4$ in
        case
          $y = 0 \Rightarrow (z \neq_* 0)$
          $y \neq 0 \Rightarrow$ let $2\lfloor\sqrt{y}\rfloor = s$ in
            case
              $x < (s+1)^2 \Rightarrow s$
              $x \geq (s+1)^2 \Rightarrow s + 1$
            end
        end.
```

The following is its condition of regularity

$$\vdash_{\mathrm{PA}} x = 4y + z \land z < 4 \land y \neq 0 \rightarrow y < x,$$

which is trivially satisfied. Note that the expression on the right-hand side of the defining equation applies two assignments and two conditionals.

The first construct is assignment and uses the fast pattern matching with the *numeric pattern* $x = 4y + z \land z < 4$. The pattern is satisfied for every $x$ since there exist (unique) numbers $y, z$ satisfying the identity. It can be viewed as a generalization of *let*-constructs from functional programming languages. Note that the *local* variables $y, z$ are then referred in the subterm. This is indicated by the subscript in $\mathbf{in}_{y,z}$. The pattern matching is fast, just consider the last two binary digits of the binary representation of the number $x$.

The reader will also note that the second assignment $2\lfloor\sqrt{y}\rfloor = s$ is crucial to the speed of the program because without it we would have two recursive invocations: once in the test and once in the result. Consequently, as the depth of recursion is $\mathcal{O}(\lg(x))$, we would have $\mathcal{O}(2^{\lg(x)}) = \mathcal{O}(x)$ recursive invocations and the definition would not be any faster than the one by primitive recursion.

The reader should have no difficulties to understand the intended meaning of the remaining constructs: the test on whether or not $y = 0$, and finally the dichotomy discrimination on whether or not $x < (s+1)^2$.

The following is the clausal form of the above definition:

$$\begin{aligned}
&\lfloor\sqrt{x}\rfloor = (z \neq_* 0) \leftarrow x = 4y + z \land z < 4 \land y = 0\\
&\lfloor\sqrt{x}\rfloor = s \leftarrow x = 4y + z \land z < 4 \land y \neq 0 \land 2\lfloor\sqrt{y}\rfloor = s \land x < (s+1)^2\\
&\lfloor\sqrt{x}\rfloor = s + 1 \leftarrow x = 4y + z \land z < 4 \land y \neq 0 \land 2\lfloor\sqrt{y}\rfloor = s \land x \geq (s+1)^2.
\end{aligned}$$

## 5.2 Syntax

**5.2.1 Introduction.** In this section we extend the language of terms with a new construct – case discrimination terms. Case discrimination terms are powerful generalizations of case constructs and pattern matching known from declarative programming languages. These new constructs have flexible syntax which legality must be certified by a formal proof. Some of them may bind local variables and so they are instances of *variable binding operators*.

### *Patterns*

**5.2.2 Introduction.** Examples of recursive definitions presented in Sect. 5.1 show a form of definitions known in the modern programming languages as *pattern matching*. Pattern matching was introduced by Burstall in HOPE [6] and Turner in SASL [50], and it enormously simplifies the form of definitions by giving them a mathematical look where we can directly read the definition as a direct assertion about the object being defined.

In the next paragraph we introduce the concept of pattern matching far surpassing the one permitted in the current functional languages. Patterns are usually just terms whereas we permit formulas. We can thus use the full logical apparatus of propositional connectives and quantifiers in the definitions of functions and predicates.

In order to shorten our discussion we will often say that a *term* $\tau$ *is in* $\vec{x}$ if all free variables of $\tau$ are among $\vec{x}$.

**5.2.3 Patterns.** Let $T$ be an extension of PA, and let $\vec{x}$ and $\vec{y}$ be respectively an $n$-tuple and $m$-tuple of variables $(n, m \geq 0)$. Let further $\Gamma[\vec{x}]$ and $\varphi[\vec{x}, \vec{y}]$ be formulas of $T$ with all their free variables among the indicated ones such that

$$T \vdash \Gamma[\vec{x}] \to \varphi[\vec{x},\vec{y}] \wedge \varphi[\vec{x},\vec{z}] \to \bigwedge_{i=1}^{m} y_i = z_i. \tag{1}$$

The formula $\varphi$ is called a *pattern* (or *single-valued* formula), $\Gamma$ is called a *guard* of the pattern, and the variables $\vec{x}$ and $\vec{y}$ are called respectively the *input* and *output* variables of the pattern. Property (1) is called the *pattern's uniqueness condition.*

The pattern $\varphi$ is *absolute* if the guard $\Gamma$ is *trivial*, i.e. if $\Gamma \equiv \top$. In order to distinguish between the input and output variables of the pattern $\varphi[\vec{x},\vec{y}]$, we will use the semicolon instead of the comma and write $\varphi[\vec{x};\vec{y}]$.

**5.2.4 Annotation of patterns.** For a given pattern $\varphi[\vec{x};\vec{y}]$, we wish effectively decide whether $\exists\vec{y}\,\varphi[\vec{x};\vec{y}]$ holds or not, and then find effectively $\vec{y}$ if there are such numbers. Computation with a pattern is done with the help of its characteristic and witnessing terms. A term $\chi[\vec{x}]$ in $\vec{x}$ is called a *characteristic term* of the pattern $\varphi[\vec{x};\vec{y}]$ guarded by $\Gamma[\vec{x}]$ if we have

$$T \vdash \Gamma[\vec{x}] \to \chi[\vec{x}] = 0 \vee \chi[\vec{x}] = 1$$
$$T \vdash \Gamma[\vec{x}] \to \exists\vec{y}\,\varphi[\vec{x};\vec{y}] \leftrightarrow \chi[\vec{x}] = 1.$$

Terms $\vec{\omega}[\vec{x}] \equiv \omega_1[\vec{x}],\ldots,\omega_m[\vec{x}]$ in $\vec{x}$ are called *witnessing terms* of the pattern if the following holds

$$T \vdash \Gamma[\vec{x}] \to \exists\vec{y}\,\varphi[\vec{x};\vec{y}] \leftrightarrow \varphi\big[\vec{x};\vec{\omega}[\vec{x}]\big].$$

The characteristic and witnessing terms of the pattern clearly satisfy

$$T \vdash \Gamma[\vec{x}] \to \varphi[\vec{x};\vec{y}] \leftrightarrow \chi[\vec{x}] = 1 \wedge \bigwedge_{i=1}^{m} y_i = \omega_i[\vec{x}]. \tag{1}$$

**5.2.5 Examples of patterns.** We give here some examples of typical patterns we will use later.

**5.2.6 Numeric patterns.** A *monadic pattern* is a pattern of the form $\tau[\vec{x}] = y + 1$. The pattern's uniqueness condition holds trivially. Clearly $\tau \neq_* 0$ and $\tau \dot{-} 1$ are respectively its characteristic and witnessing terms.

Let $p$ be a constant such that $\vdash_{\mathrm{PA}} p > 1$. A *p-ary numeric pattern* is a pattern of the form $\tau[\vec{x}] = py_1 + y_2 \wedge y_2 < p$. The pattern's uniqueness condition follows from the uniqueness of quotients and remainders. The characteristic term of the pattern is the constant 1; this means that the pattern can be always satisfied. The sole purpose of the pattern is to introduce local variables. The witnessing terms of the pattern are $\tau \div p$ for $y_1$ and $\tau \bmod p$ for $y_2$.

**5.2.7 Assignments.** An *assignment* is a pattern of the form $\tau[\vec{x}] = y$. The pattern's uniqueness condition is trivially satisfied. The terms 1 and $\tau$ are respectively its characteristic and witnessing terms.

**5.2.8 Tests.** A *tests* is a pattern of the form $\varphi[\vec{x};]$ with empty set of output variables. The pattern's uniqueness condition is trivially satisfied. Note that tests do not have witnessing terms.

**5.2.9 Pair patterns.** A *pair pattern* is a pattern of the form $\tau[\vec{x}] = \langle y_1, y_2\rangle$. The pattern's uniqueness condition follows from the pairing property 2.1.7(1). The characteristic term of the pattern is $\tau \neq_* 0$; its witnessing terms are $\pi_1(\tau)$ for $y_1$ and $\pi_2(\tau)$ for $y_2$.

**5.2.10 Pair constructor patterns.** A *constant pair constructor* pattern is a pattern of the form $\tau[\vec{x}] = K_c$, where $K_c$ is a constant such that $\vdash_{\mathrm{PA}} K_c = \langle c, 0\rangle$. The $K_c$ is called *constant pair constructor* and $c$ is a constant called the *tag* of the constructor. Note that we have

$$\vdash_{\mathrm{PA}} \tau = K_c \leftrightarrow \tau \neq 0 \land \pi_1(\tau) = c \land \pi_2(\tau) = 0.$$

Therefore $\tau \neq_* 0 \land_* \pi_1(\tau) =_* c \land_* \pi_2(\tau) =_* 0$ is its characteristic term.

A *functional pair constructor* pattern is a pattern of the form $\tau[\vec{x}] = K_c(\vec{y})$, where $K_c(\vec{y})$ is an $m$-ary function $\vdash_{\mathrm{PA}} K_c(\vec{y}) = \big\langle c, \langle \vec{y}\rangle\big\rangle$. The $K_c$ is called *functional pair constructor* and $c$ is a constant called the *tag* of the constructor. Note that we have

$$\vdash_{\mathrm{PA}} \exists \vec{y}\, \tau = K_c(\vec{y}) \leftrightarrow \tau \neq 0 \land \pi_1(\tau) = c \land \mathit{Tuple}\big(m, \pi_2(\tau)\big).$$

Therefore $\tau \neq_* 0 \land_* \pi_1(\tau) =_* c \land_* \mathit{Tuple}_*\big(m, \pi_2(\tau)\big)$ is its characteristic term of the pattern. Note also that

$$\vdash_{\mathrm{PA}} \exists \vec{y}\, \tau = K_c(\vec{y}) \leftrightarrow \tau = K_c\big([\pi_2(\tau)]^m_1, \dots, [\pi_2(\tau)]^m_m\big)$$

and so the terms $[\pi_2(\tau)]^m_1, \dots, [\pi_2(\tau)]^m_m$ are witnessing terms of the pattern.

## *Generalized Terms*

**5.2.11 Generalized terms.** We now describe the syntax of generalized terms, which are expressions with flexible syntax of case constructs and pattern matching – called case discrimination terms. We will use $\alpha, \beta, \dots$ as syntactic variables ranging over generalized terms.

Let $T$ be an extension of PA. For an $n$-tuple of variables $\vec{x}$ and a *guard* formula $\Gamma[\vec{x}]$ in $\vec{x}$ we define the set of *generalized terms in $\vec{x}$ of the theory $T$ guarded by $\Gamma[\vec{x}]$* as the smallest set of expressions satisfying the following:

- Every term in $\vec{x}$ of $T$ is a generalized term guarded by $\Gamma[\vec{x}]$.
- The expression of the form ($m \geq 1$):

$$\mathcal{D}^{\vec{\omega}_1,\dots,\vec{\omega}_m}_{\chi_1,\dots,\chi_m}(\varphi_1[\vec{x};\vec{y}_1],\beta_1[\vec{x},\vec{y}_1],\dots,\varphi_m[\vec{x};\vec{y}_m],\beta_m[\vec{x},\vec{y}_m]) \tag{1}$$

is a generalized term in $\vec{x}$ of $T$ guarded by $\mathit{\Gamma}[\vec{x}]$ if $\varphi_i[\vec{x};\vec{y}_i]$ are patterns of $T$ guarded by $\mathit{\Gamma}[\vec{x}]$ with characteristic terms $\chi_i$ and witnessing terms $\vec{\omega}_i$, and $\beta_i[\vec{x},\vec{y}_i]$ are generalized terms in $\vec{x},\vec{y}_i$ of $T$ guarded by $\mathit{\Gamma}[\vec{x}] \land \varphi_i[\vec{x};\vec{y}_i]$. Moreover, the patterns satisfies the following *disjointness* and *completeness* conditions:

$$T \vdash \mathit{\Gamma}[\vec{x}] \to \bigwedge_{\substack{i,j=1\\ i\neq j}}^{m} \neg(\exists \vec{y}_i\, \varphi_i[\vec{x};\vec{y}_i] \land \exists \vec{y}_j\, \varphi_j[\vec{x};\vec{y}_j]) \tag{2}$$

$$T \vdash \mathit{\Gamma}[\vec{x}] \to \bigvee_{i=1}^{m} \exists \vec{y}_i\, \varphi_i[\vec{x};\vec{y}_i]. \tag{3}$$

Generalized terms of the form (1) are called *case discrimination* terms or *conditionals*. The *local* variables $\vec{y}_i$ are said to be *bound* in the term (1). The generalized term $\alpha$ guarded by $\mathit{\Gamma}$ is *absolute* if the guard $\mathit{\Gamma}$ is trivial, i.e. if we have $\mathit{\Gamma} \equiv \top$.

**5.2.12 Translation of generalized terms.** We describe here the translation of generalized terms of $T$ to ordinary terms of $T$ by defining a meta-theoretic function $\alpha^\star$ yielding a term of $T$ whose interpretation can be understood as the intended interpretation of $\alpha$. In other words, the meaning (denotation) of generalized terms is obtained by translation.

The translation is defined by induction on the construction of generalized terms. If $\alpha$ is without conditionals then $\alpha^\star \equiv \alpha$. Otherwise, we set

$$\mathcal{D}^{\vec{\omega}_1,\dots,\vec{\omega}_m}_{\chi_1,\dots,\chi_m}(\varphi_1,\beta_1[\vec{y}_1],\dots,\varphi_m,\beta_m[\vec{y}_m])^\star \equiv$$
$$\equiv \overbrace{D(\chi_1,\beta_1^\star[\vec{\omega}_1],\dots D(}^{(m-1)\text{-times}}\chi_{m-1},\beta_{m-1}^\star[\vec{\omega}_{m-1}],\beta_m^\star[\vec{\omega}_m])\dots).$$

From the properties of patterns and case discrimination terms we get

$$T \vdash \mathit{\Gamma} \to \mathcal{D}^{\vec{\omega}_1,\dots,\vec{\omega}_m}_{\chi_1,\dots,\chi_m}(\dots,\varphi_i,\beta_i[\vec{y}_i],\dots)^\star = v \leftrightarrow \bigvee_{i=1}^{m}(\chi_i = 1 \land \beta_i^\star[\vec{\omega}_i] = v) \tag{1}$$

$$T \vdash \mathit{\Gamma} \to \mathcal{D}^{\vec{\omega}_1,\dots,\vec{\omega}_m}_{\chi_1,\dots,\chi_m}(\dots,\varphi_i,\beta_i[\vec{y}_i],\dots)^\star = v \leftrightarrow \bigvee_{i=1}^{m}\exists \vec{y}_i\big(\varphi_i \land \beta_i^\star[\vec{y}_i] = v\big) \tag{2}$$

provided the conditional is guarded by $\mathit{\Gamma}$.

By giving the meaning to generalized terms, we will use them from now on as ordinary terms of PA.

**5.2.13 Notational conventions.** In the sequel, we will often omit characteristic and witnessing terms from the notation of conditionals and write

$$\mathcal{D}(\varphi_1[\vec{x};\vec{y}_1],\beta_1[\vec{x},\vec{y}_1],\ldots,\varphi_m[\vec{x};\vec{y}_m],\beta_m[\vec{x},\vec{y}_m]) \tag{1}$$

as an abbreviation for the conditional 5.2.11(1).

We will visualize the conditional (1) by the notation known from functional programming languages as it is shown here on the left side:

```
case                                   case
  $\varphi_1[\vec{x};\vec{y}_1] \Rightarrow_{\vec{y}_1} \beta_1[\vec{x},\vec{y}_1]$                  $\varphi_1[\vec{x};\vec{y}_1] \Rightarrow_{\vec{y}_1} \beta_1[\vec{x},\vec{y}_1]$
  $\vdots$                                     $\vdots$
  $\varphi_m[\vec{x};\vec{y}_m] \Rightarrow_{\vec{y}_m} \beta_m[\vec{x},\vec{y}_m]$                  $\varphi_{m-1}[\vec{x};\vec{y}_{m-1}] \Rightarrow_{\vec{y}_{m-1}} \beta_{m-1}[\vec{x},\vec{y}_{m-1}]$
end                                      otherwise $\Rightarrow \beta_m[\vec{x}]$
                                       end.
```

We write the conditional (1) even as it is shown on the right if the last pattern $\varphi_m$ is of the form $\bigwedge_{i=1}^{m-1} \neg\exists\vec{y}_i\varphi_i$ and hence without output variables.

In the next paragraphs we give some examples of typical case discrimination terms we will use later.

**5.2.14 Negation discrimination.** *Negation discrimination* terms are conditionals of the form as it is shown here on the left side

```
case                              case
  $R(\vec{\tau}[\vec{x}]) \Rightarrow \beta_1[\vec{x}]$             $\tau_1[\vec{x}] = \tau_2[\vec{x}] \Rightarrow \beta_1[\vec{x}]$
  $\neg R(\vec{\tau}[\vec{x}]) \Rightarrow \beta_2[\vec{x}]$           $\tau_1[\vec{x}] \neq \tau_2[\vec{x}] \Rightarrow \beta_2[\vec{x}]$
end                               end
```

*Equality tests* are special cases of negation discrimination as it is shown here on the right side.

**5.2.15 Dichotomy and trichotomy discrimination.** *Dichotomy discrimination* and *trichotomy discrimination* terms are conditionals of the form

```
case                                  case
  $\tau_1[\vec{x}] \leq \tau_2[\vec{x}] \Rightarrow \beta_1[\vec{x}]$         $\tau_1[\vec{x}] < \tau_2[\vec{x}] \Rightarrow \beta_1[\vec{x}]$
  $\tau_1[\vec{x}] > \tau_2[\vec{x}] \Rightarrow \beta_2[\vec{x}]$         $\tau_1[\vec{x}] = \tau_2[\vec{x}] \Rightarrow \beta_2[\vec{x}]$
end                                     $\tau_1[\vec{x}] > \tau_2[\vec{x}] \Rightarrow \beta_3[\vec{x}]$
                                      end
```

listed in that order.

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

```
case
  $\tau[\vec{x}] = c_1 \Rightarrow \beta_1[\vec{x}]$
      $\vdots$
  $\tau[\vec{x}] = c_k \Rightarrow \beta_k[\vec{x}]$
  otherwise $\Rightarrow \beta_{k+1}[\vec{x}]$
end,
```

where $c_1, \dots, c_k$ are *provably* pairwise different constants, i.e. for every $i \neq j$ we have $\vdash_{\mathrm{PA}} c_i \neq c_j$.

**5.2.17 Numeric discrimination.** *Monadic discrimination* terms are conditionals of the form

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

**5.2.18 Assignments.** *Assignments* are case discrimination terms with only one alternative and their sole purpose is to introduce local variables. For instance, the following are the most typical assignments used in computer programming

$$\begin{array}{l} \textbf{case} \\ \quad \tau[\vec{x}] = y \Rightarrow_y \beta[\vec{x}, y] \\ \textbf{end}. \end{array}$$

A similar construct is known in the functional programming languages as the *let*-construct. For that reason we will often visualize the above assignment as

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

by the notation known from functional languages.

Our assignments can be viewed as a generalization of the *let*-constructs as they can introduce more than one local variable. For instance

$$\begin{array}{l} \textbf{case} \\ \quad \tau[\vec{x}] = 4y_1 + y_2 \wedge y_2 < 4 \Rightarrow_{y_1, y_2} \beta[\vec{x}, y_1, y_2] \\ \textbf{end}. \end{array}$$

**5.2.19 Pair discrimination.** *Pair discrimination* terms

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

**5.2.20 Pair constructor discrimination.** *Pair constructor discrimination* terms are conditionals of a form

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

where $K_{c_i}(\vec{y}_i)$ are constant and/or functional pair constructors with provably pairwise different tags $c_i$.

## 5.3 Regular Recursion

**5.3.1 Introduction.** Consider a functional equation of the form

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

with a (absolute) generalized term $\alpha$. In order to achieve the equivalence of the definitional semantics with the computational within the class of total functions, we restrict ourselves to regular recursive definitions. This means that there must be a well-founded relation $\prec$ in which the recursion goes down; i.e. for each recursive application $f(\vec{\rho})$ in $\alpha$ we have $\vec{\rho} \prec \vec{x}$ under the assumption of all conditions governing that recursive application. For such functional equations we not only have the extensional property that (1) has a unique solution but we also have a stronger intensional property that we can use the identity as a rewriting rule from left to right for evaluation of that solution.

**5.3.2 Extensions by regular recursion.** The concept of regular recursive definitions is formalized as follows. Let $T$ be an extension by definitions of PA and $\prec$ a well-founded relation of $T$. Let further $f$ be a new $n$-ary function symbol and $\alpha[f;\vec{x}]$ a (absolute) generalized term in $\vec{x}$ of the theory $T_f$ obtained from $T$ by adding the function symbol $f$.

To every occurrence of a subterm $\beta$ in $\alpha$ we define inductively on $\|\alpha\| - \|\beta\|$ the *governing condition* $\Gamma^\alpha_\beta$ *of* $\beta$ *in* $\alpha$. Here we denote by $\|\alpha\|$ the *size*, i.e. the number of operations, of the term $\alpha$. If $\beta \equiv \alpha$ or $\beta$ is without conditionals then $\Gamma^\alpha_\beta \equiv \top$. If $\beta \equiv \mathcal{D}^{\vec{\omega}_1,\dots,\vec{\omega}_m}_{\chi_1,\dots,\chi_m}(\varphi_1,\beta_1,\dots,\varphi_m,\beta_m)$ then $\Gamma^\alpha_{\chi_i} \equiv \Gamma^\alpha_\beta$, $\Gamma^\alpha_{\omega^j_i} \equiv \Gamma^\alpha_\beta$ and $\Gamma^\alpha_{\beta_i} \equiv \Gamma^\alpha_\beta \wedge \varphi_i$, where $\vec{\omega}_i \equiv \omega^1_i,\dots,\omega^{k_i}_i$.

We assign to each occurrence of the recursive application $f(\vec{\rho})$ in $\alpha$ with governing condition $\Gamma^\alpha_{f(\vec{\rho})}$ the following *condition of regularity*:

$$\Gamma^\alpha_{f(\vec{\rho})} \rightarrow \vec{\rho} \prec \vec{x}.$$

We denote by $Reg^\prec_\alpha$ the conjunction of universal closures of all its conditions of regularity.

We say that the term $\alpha$ *is regular in the well-founded relation* $\prec$ if the extension $T''$ of $T$ by (bounded) well-founded recursion $f(\vec{x}) = \alpha^\star\big[[f]^\prec_{\vec{x}};\vec{x}\big]$ proves all its condition of regularity, i.e. we have $T'' \vdash Reg^\prec_\alpha[f]$.

Consider the theory $T'$ obtained from $T$ by adding the symbol $f$, the defining axiom

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

where $\alpha$ is regular in $\prec$, the conditions of regularity $Reg^{\prec}_{\alpha}[f]$, and the scheme of $\prec$-well-founded induction for the formulas of $\mathcal{L}_{T'}$ containing the symbol $f$. We say that $T'$ is an *extension of* $T$ *by regular recursion.*

*Remark.* The definition can be viewed as a function operator which takes all auxiliary functions applied in the term $\alpha$ and yields the function $f$ as a result. We will prove in Thm. 5.3.6 that the class of μ-recursive functions is closed under the operator of regular well-founded recursion.

*Remark.* The definition is said to be *regular recursion with measure* if

$$\vec{x} \prec \vec{y} \leftrightarrow \mu[\vec{x}] < \mu[\vec{y}]$$

for a suitable measure $\mu[\vec{x}]$ of $\mathcal{L}_T$. In this case we will use the following notation $Reg^{\mu}_{\alpha}[f]$ to denote its conditions of regularity.

Such definition can be viewed as a function operator which takes all auxiliary functions applied in the terms $\mu, \alpha$ and yields the function $f$ as a result. We will prove in Thm. 5.3.5 that the class of primitive recursive functions is closed under the operator of regular recursion with measure.

**5.3.3 Lemma** *We have*

$$T_f \vdash Reg^{\prec}_{\alpha}[f] \leftrightarrow Reg^{\prec}_{\alpha^\star}[f].$$

*Proof outlined.* Below we indicate only the local variables of subterms of $\alpha$. To every subterm $\beta[\vec{y}]$ of $\alpha$ we can find terms $\chi[\vec{y}], \vec{\omega}$ such that

$$\begin{aligned} &T_f \vdash \chi[\vec{y}] = 1 \lor \chi[\vec{y}] = 0 \\ &T_f \vdash \Gamma^{\alpha}_{\beta}[\vec{y}] \leftrightarrow \chi[\vec{y}] = 1 \land \bigwedge_i y_i = \omega_i. \end{aligned} \qquad (\dagger_1)$$

Moreover, the term which corresponds to $\beta[\vec{y}]$ in the translation of $\alpha$ is of the form $\beta^\star[\vec{\omega}]$ with the governing condition such that

$$T_f \vdash \Gamma^{\alpha^\star}_{\beta^\star[\vec{\omega}]} \leftrightarrow \chi[\vec{\omega}] = 1. \qquad (\dagger_2)$$

Now for the recursive application $\beta[\vec{y}] \equiv f(\vec{\rho}[\vec{y}])$, its condition of regularity

$$\Gamma^{\alpha}_{\beta}[\vec{y}] \to \vec{\rho}[\vec{y}] \prec \vec{x}$$

is equivalent, by $(\dagger_1)$, to the formula

$$\chi[\vec{y}] = 1 \land \bigwedge_i y_i = \omega_i \to \vec{\rho}[\vec{y}] \prec \vec{x}$$

and therefore also to the formula

$$\chi[\vec{\omega}] = 1 \to \vec{\rho}[\vec{\omega}] \prec \vec{x}.$$

By ($\dagger_2$), the last assertion is equivalent to

$$\Gamma^{\alpha^\star}_{\beta^\star[\vec{\omega}]} \to \vec{\rho}[\vec{\omega}] \prec \vec{x}.$$

This means that we have

$$T_f \vdash \forall \vec{y}(\Gamma^{\alpha}_{\beta}[\vec{y}] \to \vec{\rho}[\vec{y}] \prec \vec{x}) \leftrightarrow \Gamma^{\alpha^\star}_{\beta^\star[\vec{\omega}]} \to \vec{\rho}[\vec{\omega}] \prec \vec{x}. \qquad \square$$

**5.3.4 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by regular recursion is an extension by definition.*

*Proof.* By Lemma 5.3.3, the following functional equations $f(\vec{x}) = \alpha[f;\vec{x}]$ and $f(\vec{x}) = \alpha^\star[f;\vec{x}]$ have equivalent conditions of regularity. The claim now follows from Thm. 4.4.3. □

**5.3.5 Theorem** *Primitive recursive functions are closed under regular recursion with measure.*

*Proof.* By inspection of the proof of Thm. 5.3.4 using Thm. 4.4.4. □

**5.3.6 Theorem** $\mu$-*Recursive functions are closed under regular well-founded recursion.*

*Proof.* By inspection of the proof of Thm. 5.3.4 using Thm. 4.4.5. □

**5.3.7 Regular programs.** Consider the following property of a function $f$:

$$\vdash_{\mathrm{PA}} \varphi[\vec{x}] \to f(\vec{x}) = \alpha[f;\vec{x}]. \qquad (1)$$

We assign to each occurrence of the recursive application $f(\vec{\rho})$ in $\alpha$ with the governing condition $\Gamma^{\alpha}_{f(\vec{\rho})}$ the following *(extended) condition of regularity*:

$$\varphi[\vec{x}] \land \Gamma^{\alpha}_{f(\vec{\rho})} \to \vec{\rho} \prec \vec{x} \land \varphi[\vec{\rho}].$$

We say the property (1) is a *program regular in the well-founded relation* $\prec$ if

$$\vdash_{\mathrm{PA}} \varphi[\vec{x}] \land \Gamma^{\alpha}_{f(\vec{\rho})} \to \vec{\rho} \prec \vec{x} \land \varphi[\vec{\rho}]$$

for every condition of regularity of $f$ in $\alpha$. The formula $\varphi$ is called the *precondition* of the program. Note that regular recursive definitions are examples of regular programs.

By inspection of the proof of Lemma 5.3.3, we can easily see that (1) and the following program with simple conditionals

$$\vdash_{\mathrm{PA}} \varphi[\vec{x}] \to f(\vec{x}) = \alpha^\star[f;\vec{x}] \tag{2}$$

have equivalent conditions of regularity. This means that for a regular program (1) we can take the identity (2) as a computational rule for evaluation of the function $f$ satisfying (1). In other words, the operational (computational) semantics of the regular program (1) is obtained by translation.

**5.3.8 Efficiency of reductions.** In order to facilitate not only effective but also *efficient* computation we introduce *mixed numerals* as terms composed by the successor function, the binary successor functions $x\mathbf{0} = 2x$ and $x\mathbf{1} = 2x + 1$, and pairing. It is now possible that two different mixed numerals may denote the same number, e.g. the following mixed numerals

$$S(0) = 0\mathbf{1} = \langle 0, 0\rangle.$$

denote the number 1. Mixed numerals have a simple representation in computers. For instance, the mixed numeral $\langle \tau_1, \tau_2 \rangle$ is represented by a pointer to a Lisp-cell with pointers to the representations of mixed numerals $\tau_1$ and $\tau_2$. The *conversions* between mixed numerals are effective because the basic arithmetic operations such as addition, multiplication, division or remainder, as well as the pairing function and its projections are primitive recursive functions. The reader interested in more details in efficient computation with generalized terms and mixed numerals should consult the text [28].

## 5.4 Clausal Definitions

**5.4.1 Introduction.** We can significantly improve the readability of explicit and recursive definitions of functions by writing them in clausal form. A clausal definition of a function $f$ is a set of clauses which can be unfolded by a set of equivalent transformations from an explicit or recursive definition of $f$.

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

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

where $\alpha$ is a generalized term and the formulas $\psi_1, \ldots, \psi_k$ do not apply conditionals. Clauses used in definitions are customarily written with converse implications:

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

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

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

**5.4.3 Unfolding.** Let $T$ be an extension of PA and $\alpha$ the generalized term of $T$. We now describe the *unfolding* transformation, which leads from an equation

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

where $\alpha$ may apply $f$, to a finite set of terminal clauses

$$\{\varphi_1, \dots, \varphi_m\} \tag{2}$$

satisfying the following property

$$T \vdash \forall \vec{x}\, f(\vec{x}) = \alpha[\vec{x}] \leftrightarrow \forall \varphi_1 \wedge \cdots \wedge \forall \varphi_m. \tag{3}$$

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

In one unfolding step we take a non-terminal clause $\psi$ of the form

$$f(\vec{x}) = \mathcal{D}(\theta_1, \beta_1, \dots, \theta_k, \beta_k) \leftarrow \xi_1 \wedge \cdots \wedge \xi_l$$

and unfold the clause to the set of clauses $\{\psi_1, \dots, \psi_k\}$, where every its clause $\psi_i$ is of the form $(1 \le i \le k)$:

$$f(\vec{x}) = \beta_i \leftarrow \xi_1 \wedge \cdots \wedge \xi_l \wedge \theta_i.$$

Clearly, the clauses $\{\psi_1, \dots, \psi_k\}$ satisfies the following *unfolding invariant*:

$$T \vdash \forall \psi \leftrightarrow \forall \psi_1 \wedge \cdots \wedge \forall \psi_k. \tag{4}$$

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

**5.4.4 Extensions by clausal definitions.** Let $T$ be an extension by definitions of PA and $<$ a well-founded order of $T$. Let further $f$ be a new $n$-ary function symbol and $\alpha[f; \vec{x}]$ a generalized term in $\vec{x}$ regular in the well-founded order $<$. Consider the theory $T'$ obtained from $T$ by adding the symbol $f$, the set of clauses (defining axioms):

$$\{\varphi_1, \dots, \varphi_m\} \tag{1}$$

which is a clausal form of the equation

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

and the conditions of regularity $Reg^{<}_{\alpha}[f; \vec{x}]$. We say that $T'$ is an *extension of $T$ by clausal definition.* The identity (2) is called the *closed form* of the clausal definition (1).

**5.4.5 Theorem** *If $T$ is an extension by definitions of* PA *then any extension of $T$ by clausal definitions is an extension by definition.*

*Proof.* Let $T'$ be an extension of $T$ by clausal definition as in Par. 5.4.4. Let further $T''$ be an extension of $T$ by recursive definition

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

regular in the well-founded order $<$. We have $\mathcal{L}_{T''} = \mathcal{L}_{T'}$ and $T''$ is an extension by definition of $T$ by Thm. 5.3.4. In order to prove the claim it suffices to show that the theories $T'$ and $T''$ are equivalent. But the last follows directly from 5.4.3(3). □

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

$$\varphi[x] \leftarrow \varphi_1[x] \wedge \tau = x \wedge \varphi_2[x]$$

is simplified into the equivalent clause

$$\varphi[\tau] \leftarrow \varphi_1[\tau] \wedge \varphi_2[\tau].$$

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

Clausal definitions are probably best explained with examples. Chapters 6–7 contain a lot of programs written in clausal form. The programs deal with list and tree processing, as well as with symbolic problems. The reader will note that the conditions of regularity of clausal definitions can be easily read off from their recursive clauses. We recommend that readers interested in more details in programming and proving with clausal language should download the text [28].

**5.4.7 Example.** Consider the following recursive definition:

```
$f(x,y) =$ **case**
    $x = 0 \Rightarrow g(y)$
    $x = x_1 + 1 \Rightarrow_{x_1} h\big(x_1, f(x_1,y), y\big)$
  **end**,
```

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

$$\vdash_{\mathrm{PA}} x = x_1 + 1 \rightarrow x_1 < x$$

is trivially satisfied.

We obtain the clausal form of the above definition as follows. First, we unfold the defining equation to the clauses

$$\begin{aligned} &f(x,y) = g(y) \leftarrow x = 0 \\ &f(x,y) = h\big(x_1, f(x_1,y), y\big) \leftarrow x = x_1 + 1. \end{aligned}$$

Next, we simplify these two clauses by eliminating the local variable $x$ from their bodies: the variable $x$ is substituted for 0 in the first clause and for $x_1 + 1$ in the second clause. After simplification we obtain

$$\begin{aligned} &f(0,y) = g(y) \\ &f(x_1+1,y) = h\big(x_1, f(x_1,y), y\big). \end{aligned}$$

Finally, we rename the variable $x_1$ by $x$ in the second clause whereby we obtain the clausal form of the above recursive definition:

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

The condition of regularity of the clausal definition, which is of the form

$$\vdash_{\mathrm{PA}} x < x + 1,$$

can be easily read off from its second clause.

**5.4.8 Example.** Consider now the recursive definition of integer divisor:

```
$x \div y =$ **if** $y \neq 0$ **then**
    **case**
      $x < y \Rightarrow 0$
      $x \geq y \Rightarrow (x \dot{-} y) \div y + 1$
    **end**
  **else**
    $0.$
```

The definition is regular in the second argument since its regularity condition

$$\vdash_{\mathrm{PA}} y \neq 0 \land x \geq y \to x \mathbin{\dot-} y < x$$

is trivially satisfied.

Its clausal form is obtained as follows. First, we unfold the definition to the clauses

$$\begin{aligned}&x \div y = 0 \leftarrow y = 0\\&x \div y = 0 \leftarrow y \neq 0 \land x < y\\&x \div y = (x \mathbin{\dot-} y) \div y + 1 \leftarrow y \neq 0 \land x \geq y.\end{aligned}$$

Then the first clause is simplified by eliminating the local variable $y$ from its body. After simplification we obtain

$$\begin{aligned}&x \div 0 = 0\\&x \div y = 0 \leftarrow y \neq 0 \land x < y\\&x \div y = (x \mathbin{\dot-} y) \div y + 1 \leftarrow y \neq 0 \land x \geq y.\end{aligned}$$

Finally, we omit the first clause by default whereby we obtain

$$\begin{aligned}&x \div y = 0 \leftarrow y \neq 0 \land x < y\\&x \div y = (x \mathbin{\dot-} y) \div y + 1 \leftarrow y \neq 0 \land x \geq y.\end{aligned}$$

This is clausal form of the above recursive definition. Note that conditions of regularity of both forms of recursive definition are the same.

Note also that this is a typical example where we wish to use the default clauses, in this case

$$x \div 0 = 0$$

in order not to clutter the definition. We do not care what value is yielded by the application $x \div y$ for $y = 0$.

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

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

```
$Even_*(x) =$ case
                $x = 0 \Rightarrow 1$
                $x = y + 1 \Rightarrow_y$
                    case
                        $Even_*(y) \neq 0 \Rightarrow 0$
                        $Even_*(y) = 0 \Rightarrow 1$
                    end
                end.
```

The definition is regular since its condition of regularity

$$x = y + 1 \to y < x$$

is trivially satisfied.

Clausal form of the definition for the predicate $Even(x)$ is obtained as follows. First, we unfold the equation to the clauses

$Even_*(x) = 1 \leftarrow x = 0$
$Even_*(x) = 0 \leftarrow x = y + 1 \land Even_*(y) \neq 0$
$Even_*(x) = 1 \leftarrow x = y + 1 \land Even_*(y) = 0.$

We simplify the clauses by eliminating the variable $x$ from their bodies by substituting it for 0 in the first clause and for $y + 1$ in the next two clauses. After simplification we obtain the following clauses

$Even_*(0) = 1$
$Even_*(y + 1) = 0 \leftarrow Even_*(y) \neq 0$
$Even_*(y + 1) = 1 \leftarrow Even_*(y) = 0.$

The clauses are then transformed into predicate form:

$Even(0)$
$\neg Even(y + 1) \leftarrow Even(y)$
$Even(y + 1) \leftarrow \neg Even(y).$

Finally, we omit the second clause by default whereby we obtain the clausal form of the above recursive definition in predicate form:

$Even(0)$
$Even(y + 1) \leftarrow \neg Even(y).$

The final clauses forms the clausal definition of the predicate $Even(x)$. Note that the condition of regularity

$$\vdash_{\text{PA}} y < y + 1,$$

can be easily read off from the last clause. Note also that the default clause

$\neg Even(y + 1) \leftarrow Even(y)$

though omitted in the clausal definition, is important as a property of the predicate. Without the default clause as a property we would not be able to derive the following property of *Even*:

$$\vdash_{\mathrm{PA}} \; Even(x) \leftrightarrow x = 0 \lor \exists y\big(x = y + 1 \land \neg Even(y)\big).$$

# Chapter 6
# Programs Operating on Lists

One of the most fundamental structures in computer programming are lists. We begin by showing how lists together with their basic operations can be arithmetized. We show also how structural recursion and induction on lists can be formalized within PA. In Sects. 6.1 and 6.2 we give examples of various simple list operations together with proofs of their specification properties.

In Sect. 6.3 we will study several combinatorial problems over lists. Such problems are good examples for program synthesis. We begin by bootstrapping of the specification predicate and then synthesize an algorithm that satisfies it. By reverting the process we obtain the program correctness.

Finally, in Sect. 6.4, we will consider the problem of sorting of lists. We will demonstrate the verification of two sorting algorithms: insertion sort and merge sort. This requires again non-trivial bootstrapping of specification predicates (ordered lists, permutations).

Most programs are by structural recursion but in some cases the problem requires the use of more general recursion (recursion with measure). Their properties can be proved by the corresponding induction principle.

## 6.1 Lists

**6.1.1 Introduction.** In this section we will show how to arithmetize *lists of natural numbers*. In most functional programming languages the type *Ln* of such lists can be defined by a *union type*:

$$Ln = Nil \mid Cons(\mathrm{N}, Ln).$$

A value of type *Ln* is therefore either the *empty* list *Nil* or a non-empty list of the form $Cons(v, w)$, where $v$ is its first *element* of of type N and $w$ is a value of type *Ln* called the *tail* of that non-empty list. The constant *Nil* and the function *Cons* are called *constructors*.

**6.1.2 Constructors of lists.** Arithmetization of lists is done with the help of the following two constructors: the first one is the number 0 and the second is the pairing function $\langle v, w\rangle$. From the properties of the pairing function we obtain

$$\vdash_{\mathrm{PA}} 0 \neq \langle v, w\rangle$$
$$\vdash_{\mathrm{PA}} \langle v_1, w_1\rangle = \langle v_2, w_2\rangle \rightarrow v_1 = v_2 \wedge w_1 = w_2 .$$

The first property says that the constructors are pairwise disjoint and the second that the functional constructor $\langle v, w\rangle$ is an injective mapping.

We obtain the pattern matching style of definitions of functions operating over lists with conditionals of the form

```
case
  $x = 0 \Rightarrow \beta_1$
  $x = \langle v, w\rangle \Rightarrow_{v,w} \beta_2[x, v, w]$.
end
```

This is called *discrimination on the constructors of lists.* Recall that such conditionals are instances of pair discrimination terms discussed in Par. 5.2.9.

The above conditional is evaluated as follows. First note that the expression $x =_* 0$ is the characteristic term of its first variant, and the expression $x \neq_* 0$ is the characteristic term of its second variant as we have

$$\vdash_{\mathrm{PA}} \exists v \exists w\, x = \langle v, w\rangle \leftrightarrow x \neq 0.$$

Note also that we have

$$\vdash_{\mathrm{PA}} x = \langle v, w\rangle \rightarrow v = \pi_1(x) \wedge w = \pi_2(x)$$

and therefore, the terms $\pi_1(x)$ and $\pi_2(x)$ are the witnessing terms for the output variables $v, w$ of the second variant of the conditional.

**6.1.3 List representation of** N. Recall that the pairing function $\langle x, y\rangle$ permits an extremely simple uniform coding of finite sequences over natural numbers (see Sect. 2.4). We assign the code 0 to the empty sequence $\emptyset$. A non-empty sequence $x_1, \ldots, x_n$ is coded by the number $\langle x_1, x_2, \ldots, x_n, 0\rangle$ as shown in Fig. 6.1

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. This is called *list representation* of numbers. Codes of finite sequences are called *lists* in computer science and this is how we will be calling them from now on.

**6.1.4 Case analysis on lists.** From properties of the pairing function we can see that every list $x$ is either the empty list 0 or can be uniquely be

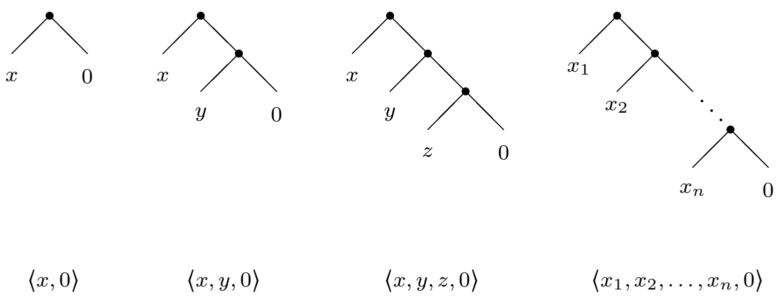

**Fig. 6.1** List representation of natural numbers

written in the form $\langle v, w\rangle$, where the number $v$ is called the *head* of the list $x$ and the number $w$ is called the *tail* of the list $x$. In particular

$$\vdash_{\mathrm{PA}} x = 0 \lor \exists v \exists w\, x = \langle v, w\rangle.$$

This is called the principle of *structural case analysis on the constructors of the list $x$.*

**6.1.5 Structural induction on lists.** The principle of structural induction over lists can be informally stated as follows. To prove by list induction that a property holds for every list it suffices to prove:

*Base case:* the property holds for the empty list 0.
*Induction step:* if the property holds for the list $w$ then it holds also for the list $\langle v, w\rangle$.

This is expressed formally in PA as follows. Let $\varphi[x]$ be a formula of PA with the indicated variable $x$ free. The principle of *list induction on $x$ for $\varphi[x]$* is the following one:

$$\vdash_{\mathrm{PA}} \varphi[0] \land \forall v \forall w(\varphi[w] \to \varphi[\langle v, w\rangle]) \to \varphi[x].$$

Note that the formula $\varphi[x]$ may contain additional variables as parameters.

*Proof.* The principle of list induction is proved as follows. Under the assumptions $\varphi[0]$ and $\forall v \forall w(\varphi[w] \to \varphi[\langle v, w\rangle])$ we prove that $\varphi[x]$ holds for every $x$ by complete induction on $x$. So take any $x$ and consider two cases. If $x = 0$ then the claim follows directly from the first assumption. Otherwise, $x$ is of the form $\langle v, w\rangle$ for some $v, w$. By 2.1.6(2), we have $w < \langle v, w\rangle$ and thus $\varphi[w]$ by IH. We obtain $\varphi[\langle v, w\rangle]$ from the second assumption.

**6.1.6 Structural recursion on lists.** List induction is used to prove properties of functions defined by the scheme of *list recursion.* In its simplest form, the operator of list recursion introduces a function $f$ from two functions $g$ and $h$ satisfying

```
$f(x,y) =$ case
        $x = 0 \Rightarrow g(y)$
        $x = \langle v,w\rangle \Rightarrow h\big(v,w,f(w,y),y\big)$
    end.
```

Note that this is a recursive definition regular in the first argument with discrimination on the constructors of lists (output variables of the second variant are omitted). The following identities form the clausal form of the above definition

$$f(0,y) = g(y)$$
$$f(\langle v,w\rangle,y) = h(v,w,f(w,y),y).$$

Note that this is a recursive definition regular in the first argument. Similar schemes, when we allow terms with arbitrary number of parameters on the right-hand side of the above identities, substitution in parameters, or even nested recursive applications, will be also called definitions by list recursion.

**6.1.7 List length.** The function $L(x)$ yields the length of the list $x$:

$$L\,\langle x_1,x_2,\dots,x_n,0\rangle = n.$$

It is defined by parameterless structural list recursion as a p.r. function:

$$L(0) = 0$$
$$L\,\langle v,w\rangle = L(w) + 1.$$

**6.1.8 List indexing.** The binary function $x[i]$ yields the $(i+1)$-st element of the list $x$ (counting from 0):

$$\langle x_0,\dots,x_i,\dots,x_{n-1},0\rangle[i] = \begin{cases} x_i & \text{if } i < n, \\ 0 & \text{otherwise.} \end{cases}$$

The function is defined by primitive recursion on $i$ with substitution in parameter as p.r. function:

$$\langle v,w\rangle[0] = v$$
$$\langle v,w\rangle[i+1] = w[i].$$

Note that $0[i] = 0$ by default.

Usually we intend to apply the operations $x[i]$ only in cases when $i < L(x)$. We can take the following property as alternative programs for computing the function in such cases:

```
$\vdash_{\mathrm{PA}} i < L(x) \to x[i] =$ case
        $i = 0 \Rightarrow$ let $x = \langle v,w\rangle$ in $v$
        $i = j+1 \Rightarrow$ let $x = \langle v,w\rangle$ in $w[j]$
    end.
```

Its condition of regularity

$$\vdash_{\mathrm{PA}} i < L(x) \wedge i = j + 1 \wedge x = \langle v, w\rangle \to w < x \wedge j < L(w)$$

is trivially satisfied.

**6.1.9 Remark.** The following property can be used as an alternative definition of list indexing:

$$\vdash_{\mathrm{PA}} x[i] = \pi_1\, \pi_2^i(x). \tag{1}$$

This is proved by (mathematical) induction on $i$ as $\forall x(1)$ In the base case take any $x$ and consider two cases. If $x = 0$ then $0[0] = 0 = \pi_1(0) = \pi_1\, \pi_2^0(0)$; if $x = \langle v, w\rangle$ for some $v, w$ then $\langle v, w\rangle[0] = v = \pi_1\langle v, w\rangle = \pi_1\, \pi_2^0\langle v, w\rangle$. In the induction step take any $x$ and consider the same two cases. If $x = 0$ then

$$0[i+1] = 0 = \pi_1(0) \overset{2.2.19(5)}{=} \pi_1\, \pi_2^{i+1}(0).$$

Otherwise $x = \langle v, w\rangle$ for some $v, w$ and we obtain

$$\langle v, w\rangle[i+1] = w[i] \overset{\mathrm{IH}}{=} \pi_1\, \pi_2^i(w) = \pi_1\, \pi_2^i\, \pi_2\langle v, w\rangle = \pi_1\, \pi_2^{i+1}\langle v, w\rangle.$$

Note that the induction hypothesis is applied with $w$ in place of $x$.

**6.1.10 List concatenation.** The binary function $x \oplus y$ concatenates two lists together to form a new one:

$$\langle x_1, \ldots, x_n, 0\rangle \oplus \langle y_1, \ldots, y_m, 0\rangle = \langle x_1, \ldots, x_n, y_1, \ldots, y_m, 0\rangle.$$

The function is defined by structural list recursion on $x$ as a p.r. function by

$$\begin{aligned} &0 \oplus y = y \\ &\langle v, w\rangle \oplus y = \langle v, w \oplus y\rangle. \end{aligned}$$

We can use the recurrences directly for computation. For example:

$$\begin{aligned} \langle 1,2,3,0\rangle \oplus \langle 4,5,0\rangle = \big\langle 1, \langle 2,3,0\rangle \oplus \langle 4,5,0\rangle\big\rangle = \big\langle 1,2, \langle 3,0\rangle \oplus \langle 4,5,0\rangle\big\rangle = \\ = \big\langle 1,2,3, 0 \oplus \langle 4,5,0\rangle\big\rangle = \langle 1,2,3,4,5,0\rangle \end{aligned}$$

Note that during the computation there is no need to convert the values into monadic (or binary) notation.

**6.1.11 Basic properties of list concatenation.** We have

$$\vdash_{\mathrm{PA}} x \oplus y = 0 \leftrightarrow x = 0 \wedge y = 0 \tag{1}$$

$$\vdash_{\mathrm{PA}} x \oplus 0 = x \tag{2}$$

$$\vdash_{\mathrm{PA}} x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{3}$$

$$\vdash_{\mathrm{PA}}\ x \oplus y = x \oplus z \rightarrow y = z \tag{4}$$
$$\vdash_{\mathrm{PA}}\ x \oplus \langle a,0\rangle = y \oplus \langle b,0\rangle \rightarrow x = y \wedge a = b \tag{5}$$
$$\vdash_{\mathrm{PA}}\ x \oplus z = y \oplus z \rightarrow x = y \tag{6}$$
$$\vdash_{\mathrm{PA}}\ L(x \oplus y) = L(x) + L(y) \tag{7}$$
$$\vdash_{\mathrm{PA}}\ i < L(x) \rightarrow (x \oplus y)[i] = x[i] \tag{8}$$
$$\vdash_{\mathrm{PA}}\ i < L(y) \rightarrow (x \oplus y)[L(x) + i] = y[i]. \tag{9}$$

In the sequel we will use these properties without explicitly referring to them. Note that (3) says that list concatenation is an associative operation. For this reason we will not be using any parentheses in expressions like $\tau_1 \oplus \tau_2 \oplus \tau_3$.

*Proof.* (1): By case analysis on whether or not the list $x$ is empty.

(2): By a straightforward structural list induction.

(3): This is proved by structural induction on the list $x$. The base case follows from $0 \oplus (y \oplus z) = y \oplus z = (0 \oplus y) \oplus z$. In the induction step we have

$$\langle v,w\rangle \oplus (y \oplus z) = \langle v, w \oplus (y \oplus z)\rangle \overset{\mathrm{IH}}{=} \langle v, (w \oplus y) \oplus z\rangle =$$
$$= \langle v, w \oplus y\rangle \oplus z = (\langle v,w\rangle \oplus y) \oplus z.$$

(4): By structural induction on the list $x$. The base case is obvious. The induction step follows from

$$\langle v,w\rangle \oplus y = \langle v,w\rangle \oplus z \Rightarrow \langle v, w \oplus y\rangle = \langle v, w \oplus z\rangle \Rightarrow w \oplus y = w \oplus z \overset{\mathrm{IH}}{\Rightarrow} y = z.$$

(5): By structural induction on the list $x$ as $\forall y(5)$. In the base case take any $y$ and consider two cases. If $y = 0$ then

$$0 \oplus \langle a,0\rangle = 0 \oplus \langle b,0\rangle \Rightarrow \langle a,0\rangle = \langle b,0\rangle \Rightarrow a = b \Rightarrow 0 = 0 \wedge a = b.$$

The case when $y = \langle v_2, w_2\rangle$ for some $v_2, w_2$ leads to contradiction:

$$0 \oplus \langle a,0\rangle = \langle v_2,w_2\rangle \oplus \langle b,0\rangle \Rightarrow \langle a,0\rangle = \big\langle v_2, w_2 \oplus \langle b,0\rangle\big\rangle \Rightarrow$$
$$\Rightarrow 0 = w_2 \oplus \langle b,0\rangle \overset{(1)}{\Rightarrow} 0 = \langle b,0\rangle.$$

In the induction step, when $x = \langle v_1, w_1\rangle$ for some $v_1, w_1$, take any $y$ and consider two cases. The case $y = 0$ leads to contradiction by similar arguments as above. So it must be $y = \langle v_2, w_2\rangle$ for some $v_2, w_2$. We then have

$$\langle v_1,w_1\rangle \oplus \langle a,0\rangle = \langle v_2,w_2\rangle \oplus \langle b,0\rangle \Rightarrow \big\langle v_1, w_1 \oplus \langle a,0\rangle\big\rangle = \big\langle v_2, w_2 \oplus \langle b,0\rangle\big\rangle \Rightarrow$$
$$\Rightarrow v_1 = v_2 \wedge w_1 \oplus \langle a,0\rangle = w_2 \oplus \langle b,0\rangle \overset{\mathrm{IH}}{\Rightarrow} v_1 = v_2 \wedge w_1 = w_2 \wedge a = b \Rightarrow$$
$$\Rightarrow \langle v_1,w_1\rangle = \langle v_2,w_2\rangle \wedge a = b.$$

Note that the induction hypothesis is applied with $w_2$ in place of $y$.

(6): By structural induction on the list $z$ as $\forall x \forall y(6)$. The base case is follows from (2). In the induction step take any $x, y$ and we have

$$
\begin{aligned}
&x \oplus \langle v, w\rangle = y \oplus \langle v, w\rangle \Rightarrow x \oplus \langle v, 0 \oplus w\rangle = y \oplus \langle v, 0 \oplus w\rangle \Rightarrow \\
&\quad\Rightarrow x \oplus (\langle v, 0\rangle \oplus w) = y \oplus (\langle v, 0\rangle \oplus w) \overset{(3)}{\Rightarrow} \\
&\quad\Rightarrow (x \oplus \langle v, 0\rangle) \oplus w = (y \oplus \langle v, 0\rangle) \oplus w \overset{\mathrm{IH}}{\Rightarrow} \\
&\quad\Rightarrow x \oplus \langle v, 0\rangle = y \oplus \langle v, 0\rangle \overset{(5)}{\Rightarrow} x = y.
\end{aligned}
$$

Note that the induction hypothesis is applied with $x \oplus \langle v, 0\rangle$ in place of $x$ and with $y \oplus \langle v, 0\rangle$ in place of $y$.

(7): By a straightforward structural induction on the list $x$.

(8): By structural induction on the list $x$ as $\forall i(8)$. In the base case there is nothing to prove. In the induction step, when $x = \langle v, w\rangle$ for some $v, w$, take any $i$ s.t. $i < L\langle v, w\rangle = L(w) + 1$, and consider two cases. If $i = 0$ then

$$(\langle v, w\rangle \oplus y)[0] = \langle v, w \oplus y\rangle[0] = v = \langle v, w\rangle[0].$$

If $i = j + 1$ for some $j$ then $j < L(w)$ and thus we obtain

$$(\langle v, w\rangle \oplus y)[j+1] = \langle v, w \oplus y\rangle[j+1] = (w \oplus y)[j] \overset{\mathrm{IH}}{=} w[j] = \langle v, w\rangle[j+1].$$

Note that the induction hypothesis is applied with $j$ in place of $i$.

(9): By a straightforward structural induction on the list $x$. □

**6.1.12 List membership.** The binary predicate $x \,\varepsilon\, y$ holds if the number $x$ is an element of the list $y$:

$$x \,\varepsilon\, \langle y_1, \dots, y_n, 0\rangle \quad \text{if } x = y_i \text{ for some } 1 \le i \le n.$$

The list membership predicate is defined explicitly as primitive recursive by

$$x \,\varepsilon\, y \leftrightarrow \exists i (i < L(y) \land x = y[i]).$$

Note that from the property 2.1.6(2) of the pairing function we get

$$\vdash_{\mathrm{PA}} x \,\varepsilon\, y \to x < y$$

and therefore

$$\vdash_{\mathrm{PA}} \forall x (x \,\varepsilon\, y \to \varphi[x]) \leftrightarrow \forall x \le y (x \,\varepsilon\, y \to \varphi[x])$$

for every formula $\varphi[x]$ of PA. The universal quantifier $\forall x$ in the contexts like $\forall x(x \,\varepsilon \ldots \to \cdots)$ can be bounded and thus it can be used in explicit definitions of primitive recursive predicates. Similarly for existential quantifiers.

**6.1.13 Basic properties of list membership.** We have

$$\vdash_{\mathrm{PA}} x \not\varepsilon\, 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} x \,\varepsilon\, \langle v, w\rangle \leftrightarrow x = v \lor x \,\varepsilon\, w \tag{2}$$
$$\vdash_{\mathrm{PA}} x \,\varepsilon\, y \oplus z \leftrightarrow x \,\varepsilon\, y \lor x \,\varepsilon\, z \tag{3}$$
$$\vdash_{\mathrm{PA}} x \,\varepsilon\, y \leftrightarrow \exists z_1 \exists z_2\, y = z_1 \oplus \langle x, z_2\rangle. \tag{4}$$

In the sequel we will use the properties (1)–(3) without explicitly referring to them. Note also that the last property (4) can be used as alternative definition of the list membership predicate.

*Proof.* (1): Obvious. (2): This follows from

$$\begin{aligned}
&x \,\varepsilon\, \langle v, w\rangle \Leftrightarrow \exists i\big(i < L\,\langle v, w\rangle \land x = \langle v, w\rangle[i]\big) \overset{(*_1)}{\Leftrightarrow}\\
&0 < L(w) + 1 \land x = \langle v, w\rangle[0] \lor \exists j\big(j + 1 < L(w) + 1 \land x = \langle v, w\rangle[j+1]\big) \Leftrightarrow\\
&x = v \lor \exists j(j < L(w) \land x = w[j]) \Leftrightarrow x = v \lor x \,\varepsilon\, w.
\end{aligned}$$

The step marked by $(*_1)$ is by case analysis on whether or not $i = 0$.

(3): By structural induction on the list $y$. The base case is trivial and the induction step follows from

$$\begin{aligned}
x \,\varepsilon\, \langle v, w\rangle \oplus z \Leftrightarrow x \,\varepsilon\, \langle v, w \oplus z\rangle \overset{(2)}{\Leftrightarrow} x = v \lor x \,\varepsilon\, w \oplus z \overset{\mathrm{IH}}{\Leftrightarrow}&\\
\Leftrightarrow x = v \lor x \,\varepsilon\, w \lor x \,\varepsilon\, z \overset{(2)}{\Leftrightarrow} x \,\varepsilon\, \langle v, w\rangle \lor x \,\varepsilon\, z.&
\end{aligned}$$

(4): By structural induction on the list $y$. The base case follows from (1) and 6.1.11(1). In the induction step we have

$$\begin{aligned}
x \,\varepsilon\, \langle v, w\rangle &\overset{(2)}{\Leftrightarrow} x = v \lor x \,\varepsilon\, w \overset{\mathrm{IH}}{\Leftrightarrow} x = v \lor \exists z_1 \exists z_2\, w = z_1 \oplus \langle x, z_2\rangle \Leftrightarrow\\
&\Leftrightarrow \langle v, w\rangle = 0 \oplus \langle x, w\rangle \lor \exists z_1 \exists z_2\, \langle v, w\rangle = \langle v, z_1\rangle \oplus \langle x, z_2\rangle \overset{(*_2)}{\Leftrightarrow}\\
&\Leftrightarrow \exists z_1 \exists z_2\, \langle v, w\rangle = z_1 \oplus \langle x, z_2\rangle.
\end{aligned}$$

The step $(*_2)$ is by case analysis on whether or not the list $z_1$ is empty. □

**6.1.14 List reversal.** We wish to introduce into PA the function $Rev(x)$ which reverses the elements of the list $x$:

$$Rev\,\langle x_1, x_2, \dots, x_n, 0\rangle = \langle x_n, \dots, x_2, x_1, 0\rangle.$$

The list reversal is defined by structural list recursion as a p.r. function:

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

**6.1.15 Basic properties of list reversal.** We have

$$\vdash_{\mathrm{PA}} Rev(x) = 0 \leftrightarrow x = 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} Rev(x \oplus y) = Rev(y) \oplus Rev(x) \tag{2}$$
$$\vdash_{\mathrm{PA}} Rev\,Rev(x) = x \tag{3}$$
$$\vdash_{\mathrm{PA}} Rev(x) = Rev(y) \rightarrow x = y \tag{4}$$
$$\vdash_{\mathrm{PA}} \exists y\, x = Rev(y) \tag{5}$$
$$\vdash_{\mathrm{PA}} L\,Rev(x) = L(x) \tag{6}$$
$$\vdash_{\mathrm{PA}} y \mathrel{\varepsilon} Rev(x) \leftrightarrow y \mathrel{\varepsilon} x. \tag{7}$$

In the sequel we will use these properties without explicitly referring to them.

*Proof.* (1): By case analysis on whether or not the list $x$ is empty.

(2): By structural induction on the list $x$. The base case is obvious and the induction step follows from

$$Rev(\langle v, w\rangle \oplus y) = Rev\,\langle v, w \oplus y\rangle = Rev(w \oplus y) \oplus \langle v, 0\rangle \overset{\mathrm{IH}}{=}$$
$$= Rev(y) \oplus Rev(w) \oplus \langle v, 0\rangle = Rev(y) \oplus Rev\,\langle v, w\rangle.$$

(3): By structural list induction. The base case is obvious and the induction step follows from

$$Rev\,Rev(\langle v, w\rangle \oplus y) = Rev\,Rev\,\langle v, w \oplus y\rangle = Rev\big(Rev(w \oplus y) \oplus \langle v, 0\rangle\big) \overset{(2)}{=}$$
$$= Rev\,\langle v, 0\rangle \oplus Rev\,Rev(w \oplus y) \overset{\mathrm{IH}}{=} \langle v, 0\rangle \oplus w \oplus y = \langle v, w\rangle \oplus y.$$

(4): This follows from

$$Rev(x) = Rev(y) \Rightarrow Rev\,Rev(x) = Rev\,Rev(y) \overset{(3)}{\Rightarrow} x = y.$$

(5): This follows from (3) by setting $y := Rev(x)$.

(6),(7): By a straightforward structural induction on the list $x$. □

**6.1.16 Fast reversal.** The application $Rev(x)$ repeatedly invokes list concatenation to append an element to the end of a list. Consequently, it takes $\mathcal{O}(L(x)^2)$ operations to compute $Rev(x)$. This is clearly wasteful and we can ask the question whether $Rev(x)$ cannot be computed in $\mathcal{O}(L(x))$ steps. By accumulating the reversed list into an *accumulator* $a$ we can perform the reversal of $x$ in $\mathcal{O}(L(x))$ operations with the help of the binary accumulator function $f(x, a)$ defined by

$$f(0, a) = a$$
$$f(\langle v, w\rangle, a) = f(w, \langle v, a\rangle).$$

The reader will note that this is a structural recursion on the list $x$ with substitution in the parameter $a$.

The auxiliary function $f$ satisfies the property

$$\vdash_{\mathrm{PA}} \forall a\, f(x,a) = Rev(x) \oplus a, \tag{1}$$

from which, by instantiating $a := 0$, we get the relation between $Rev$ and its accumulator version:

$$\vdash_{\mathrm{PA}} Rev(x) = f(x,0).$$

Now we can take the last identity as a program computing $Rev(x)$ with a number of reduction steps proportional to the length of $x$.

It remains to show that (1) holds. The proof is by structural induction on the list $x$. The base case is trivial. In the induction step take any $a$ and we obtain

$$\begin{aligned} f(\langle v,w\rangle,a) = f(w,\langle v,a\rangle) &\overset{\mathrm{IH}}{=} Rev(w) \oplus \langle v,a\rangle = \\ &= Rev(w) \oplus \langle v,0\rangle \oplus a = Rev\,\langle v,w\rangle \oplus a. \end{aligned}$$

## 6.2 Operations on Lists

**6.2.1 Introduction.** In this section we are concerned with the problem of specification and verification of various useful simple operations over lists. We will show how the algorithms can be implemented by using structural recursion and how their specification properties can be proved by the corresponding induction principles.

**6.2.2 Map.** The operation $Map_f(x)$ applies an unary function $f$ to each element of the list $x$:

$$\vdash_{\mathrm{PA}} L\,Map_f(x) = L(x) \tag{1}$$

$$\vdash_{\mathrm{PA}} i < L(x) \rightarrow Map_f(x)[i] = f(x[i]) \tag{2}$$

The mapping is defined by structural list recursion as a p.r. function in $f$ by

$$\begin{aligned} &Map_f(0) = 0 \\ &Map_f\langle v,w\rangle = \big\langle f(v), Map_f(w)\big\rangle. \end{aligned}$$

*Verification.* (1): By a straightforward structural list induction.

(2): By structural induction on the list $x$ as $\forall i(2)$. In the base case there is nothing to prove. In the induction step, when $x = \langle v,w\rangle$ for some $v,w$, take any $i$ s.t. $i < L\,\langle v,w\rangle = L(w) + 1$ and consider two cases. If $i = 0$ then

$$Map_f\langle v,w\rangle[0] = \big\langle f(v), Map_f(w)\big\rangle[0] = f(v) = f(\langle v,w\rangle[0]).$$

If $i = j + 1$ for some $j$ then $j < L(w)$ and we thus obtain

$$Map_f\langle v,w\rangle[j+1] = \big\langle f(v), Map_f(w)\big\rangle[j+1] = Map_f(w)[j] \overset{\mathrm{IH}}{=}$$
$$= f(w[j]) = f(\langle v,w\rangle[j+1]).$$

Note that the induction hypothesis is applied with $j$ in place of $i$. □

**6.2.3 Take and drop.** The function $Take(n,x)$ yields the initial segment of a list $x$ of the length $n$ provided $n \le L(x)$. The function satisfies

$$\vdash_{\mathrm{PA}} n \le L(x) \to L\,Take(n,x) = n \tag{1}$$
$$\vdash_{\mathrm{PA}} n \le L(x) \to \exists y\, x = Take(n,x) \oplus y \tag{2}$$

and it is defined by primitive recursion on $n$ with substitution in parameter as primitive recursive by

$Take(0,x) = 0$
$Take(n+1,\langle v,w\rangle) = \big\langle v, Take(n,w)\big\rangle.$

Note the default $Take(n+1,0) = 0$.

The function $Drop(n,x)$ removes the initial segment of a list $x$ of the length $n$ provided $n \le L(x)$. The function satisfies

$$\vdash_{\mathrm{PA}} n \le L(x) \to L\,Drop(n,x) = L(x) \dot{-} n \tag{3}$$
$$\vdash_{\mathrm{PA}} n \le L(x) \to \exists y\, x = y \oplus Drop(n,x) \tag{4}$$

and it is defined by primitive recursion on $n$ with substitution in parameter as primitive recursive by

$Drop(0,x) = x$
$Drop(n+1,\langle v,w\rangle) = Drop(n,w).$

Note the default $Drop(n+1,0) = 0$.

Usually we intend to apply both operations $Take(n,x)$ and $Drop(n,x)$ only in cases when $n \le L(x)$. We can take the following properties as alternative programs for computing the functions in such cases:

```
⊢PA $n \le L(x) \to Take(n,x) =$ case
        $n = 0 \Rightarrow 0$
        $n = m+1 \Rightarrow$ let $x = \langle v,w\rangle$ in $\big\langle v, Take(n,w)\big\rangle$
    end.
```

```
⊢PA $n \le L(x) \to Drop(n,x) =$ case
        $n = 0 \Rightarrow x$
        $n = m+1 \Rightarrow$ let $x = \langle v,w\rangle$ in $Drop(n,w)$
    end.
```

Note that both programs share the same condition of regularity

$$\vdash_{\mathrm{PA}} n \le L(x) \land n = m+1 \land x = \langle v,w\rangle \to w < x \land m \le L(w)$$

which is trivially satisfied.

*Verification.* (1): This is proved by induction on $n$ as $\forall x(1)$. The base case is obvious. In the induction step take any $x$ such that $n+1 \leq L(x)$. Then $x = \langle v, w\rangle$ for some $v, w$, where $n \leq L(w)$. We obtain

$$L\, \mathit{Take}(n+1, \langle v, w\rangle) = L\left\langle v, \mathit{Take}(n, w)\right\rangle = L\, \mathit{Take}(n, w) + 1 \overset{\mathrm{IH}}{=} n + 1.$$

Note that the induction hypothesis is applied with $w$ in place of $x$.

(2): By induction on $n$ as $\forall x(2)$. In the base case it suffices to take $y := x$ since $\mathit{Take}(0, x) \oplus x = 0 \oplus x = x$. In the induction step assume $n + 1 \leq L(x)$. Then $x = \langle v, w\rangle$ for some $v, w$. Since $n \leq L(w)$ we get from IH applied with $w$ in place of $x$ that $w = \mathit{Take}(n, w) \oplus y$ for some $y$. We then have

$$\langle v, w\rangle = \langle v, \mathit{Take}(n, w) \oplus y\rangle = \langle v, \mathit{Take}(n, w)\rangle \oplus y = \mathit{Take}(n+1, \langle v, w\rangle) \oplus y.$$

The remaining properties (3) and (4) are proved similarly. □

**6.2.4 Interval.** The binary function $[m \mathinner{..} n)$ returns the list of numbers from $m$ to $n-1$ if $m < n$; the list is empty if $m \geq n$. The function satisfies

$$\vdash_{\mathrm{PA}} L\,[m \mathinner{..} n) = n \dot{-} m \tag{1}$$

$$\vdash_{\mathrm{PA}} i + m < n \to [m \mathinner{..} n)[i] = m + i \tag{2}$$

and it is defined by recursion with measure $n \dot{-} m$ as a p.r. function by

$$\begin{aligned}
&[m \mathinner{..} n) = 0 \leftarrow m \geq n\\
&[m \mathinner{..} n) = \left\langle m, [m+1 \mathinner{..} n)\right\rangle \leftarrow m < n.
\end{aligned}$$

Note that this is an example of function definition by backward recursion.

We usually intend to apply the operation $[m \mathinner{..} n)$ only in cases when $m \leq n$. For that we can take the following property as an alternative (conditional) program for computing the function:

$$\begin{aligned}
\vdash_{\mathrm{PA}} m \leq n \to [m \mathinner{..} n) = {}&\mathbf{case}\\
&\quad m = n \Rightarrow 0\\
&\quad m \neq n \Rightarrow \left\langle m, [m+1 \mathinner{..} n)\right\rangle\\
&\mathbf{end}
\end{aligned}$$

Its condition of regularity

$$\vdash_{\mathrm{PA}} m \leq n \wedge m \neq n \to n \dot{-} (m+1) < n \dot{-} m \wedge m + 1 \leq n$$

is trivially satisfied. Note that the program does not terminate for $m > n$.

*Verification.* (1): By induction with measure $n \dot{-} m$. Take any $m, n$ and consider two cases. If $m \geq n$ then $L\,[m \mathinner{..} n) = L(0) = 0 = n \dot{-} m$. If $m < n$ then $m < m + 1 \leq n$ and we obtain

$$L\,[m\,..\,n) = L\,\big\langle m, [m+1\,..\,n)\big\rangle = L\,[m+1\,..\,n) + 1 \overset{\text{IH}}{=} n \mathbin{\dot-} (m+1) + 1 = n \mathbin{\dot-} m.$$

(2): This is proved by induction with measure $n \mathbin{\dot-} m$ as $\forall i(2)$. Take any $m, n, i$ such that $i + m < n$ and consider two cases. If $i = 0$ then we have

$$[m\,..\,n)[0] = \big\langle m, [m+1\,..\,n)\big\rangle[0] = m = m + 0.$$

If $i = j + 1$ for some $j$ then $j + (m+1) = j + 1 + m < n$ and thus

$$\begin{aligned}[m\,..\,n)[j+1] = \big\langle m, [m+1\,..\,n)\big\rangle[j+1] = [m+1\,..\,n)[j] \overset{\text{IH}}{=} \\ = m + 1 + j = m + (j+1).\end{aligned}$$

Note that the induction hypothesis is applied with $j$ in place of $i$. □

**6.2.5 Filter.** Let $A(x)$ be arbitrary but fixed unary predicate. The function $\mathit{Filter}_A(x)$ removes all elements from a list which do not satisfy the predicate. The function satisfies

$$\vdash_{\text{PA}} \; a \mathrel{\varepsilon} \mathit{Filter}_A(x) \leftrightarrow a \mathrel{\varepsilon} x \wedge A(a) \tag{1}$$

and it is defined by structural list recursion as a p.r. predicate in $A$:

$\mathit{Filter}_A(0) = 0$
$\mathit{Filter}_A\,\langle v, w\rangle = \big\langle v, \mathit{Filter}_A(w)\big\rangle \leftarrow A(v)$
$\mathit{Filter}_A\,\langle v, w\rangle = \mathit{Filter}_A(w) \leftarrow \neg A(v).$

*Verification.* Property (1) is proved by structural induction on the list $x$. The base case is obvious. In the induction step, when $x = \big\langle v, w\big\rangle$ for some $v, w$, we consider two cases. If $A(v)$ then we have

$$\begin{aligned}&a \mathrel{\varepsilon} \mathit{Filter}_A\,\langle v, w\rangle \Leftrightarrow a \mathrel{\varepsilon} \big\langle v, \mathit{Filter}_A(w)\big\rangle \Leftrightarrow a = v \vee a \mathrel{\varepsilon} \mathit{Filter}_A(w) \overset{\text{IH}}{\Leftrightarrow} \\ &a = v \vee a \mathrel{\varepsilon} w \wedge A(a) \overset{(*)}{\Leftrightarrow} (a = v \vee a \mathrel{\varepsilon} w) \wedge A(a) \Leftrightarrow a \mathrel{\varepsilon} \langle v, w\rangle \wedge A(a).\end{aligned}$$

The equivalence marked by $(*)$ is by case analysis on whether or not $a = v$. The case when $A(v)$ does not hold is similar. □

**6.2.6 Removal of duplicates from lists.** The function $\mathit{Nodoubles}(x)$ removes duplicates from a list. The function satisfies

$$\vdash_{\text{PA}} \; a \mathrel{\varepsilon} \mathit{Nodoubles}(x) \leftrightarrow a \mathrel{\varepsilon} x \tag{1}$$

$$\vdash_{\text{PA}} \; a \mathrel{\varepsilon} \mathit{Nodoubles}(x) \rightarrow \#_a \mathit{Nodoubles}(x) = 1 \tag{2}$$

and it is defined by structural list recursion as a p.r. function:

$\mathit{Nodoubles}(0) = 0$
$\mathit{Nodoubles}\,\langle v, w\rangle = \mathit{Nodoubles}(w) \leftarrow v \mathrel{\varepsilon} w$

$Nodoubles\,\langle v,w\rangle = \big\langle v, Nodoubles(w)\big\rangle \leftarrow v \not\varepsilon\, w.$

*Verification.* (1): By a straightforward structural list induction.

(2): By structural induction on the list $x$. In the base case there is nothing to prove. In the induction step, when $x = \langle v,w\rangle$ for some $v,w$, assume $a \varepsilon\, Nodoubles\,\langle v,w\rangle$ and consider two cases. If $v \varepsilon w$ then, by definition, $a \varepsilon\, Nodoubles(w)$. We obtain

$$\#_a Nodoubles\,\langle v,w\rangle = \#_a Nodoubles(w) \overset{\mathrm{IH}}{=} 1.$$

If $v \not\varepsilon\, w$ then by definition either $a = v$ or $a \varepsilon\, Nodoubles(w)$, and also

$$\#_a Nodoubles\,\langle v,w\rangle = (a =_* v) + \#_a Nodoubles(w). \qquad (\dagger_1)$$

Now consider two subcases. If $a = v$ then $v \not\varepsilon\, Nodoubles(w)$ by (1) and thus

$$\#_v Nodoubles\,\langle v,w\rangle \overset{(\dagger_1)}{=} (v =_* v) + \#_v Nodoubles(w) \overset{6.2.8(3)}{=} 1 + 0 = 1.$$

If $a \neq v$ then it must be $a \varepsilon\, Nodoubles(w)$ and therefore

$$\#_a Nodoubles\,\langle v,w\rangle \overset{(\dagger_1)}{=} 0 + \#_a Nodoubles(w) \overset{\mathrm{IH}}{=} 0 + 1 = 1. \qquad \square$$

**6.2.7 List minimum.** The function $Minl(x)$ yields the minimal element of a non-empty list. The function satisfies

$$\vdash_{\mathrm{PA}} x \neq 0 \rightarrow Minl(x) \,\varepsilon\, x \qquad (1)$$
$$\vdash_{\mathrm{PA}} x \neq 0 \wedge a \,\varepsilon\, x \rightarrow Minl(x) \leq a \qquad (2)$$

and it is defined by list recursion as a p.r. function:

$Minl\,\langle v,0\rangle = v$
$Minl\,\langle v,w\rangle = \min(v, Minl(w)) \leftarrow w \neq 0.$

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

We usually intend to apply the operation $Minl(\mathrm{x})$ only in cases when input lists are non-empty. For that we can take the following property as an alternative (conditional) program for computing list minimum:

```
⊢PA  $x \neq 0 \rightarrow Minl(x) =$ let $x = \langle v,w\rangle$ in
                              case
                                $w = 0 \Rightarrow v$
                                $w \neq 0 \Rightarrow \min(v, Minl(w))$
                              end.
```

Its condition of regularity

$$\vdash_{\mathrm{PA}} x \neq 0 \wedge x = \langle v,w\rangle \wedge w \neq 0 \rightarrow w < x \wedge w \neq 0$$

is trivially satisfied.

*Verification.* (1),(2): By a straightforward structural list induction on $x$. □

**6.2.8 Multiplicity.** The binary function $\#_a(x)$ counts the number of occurrences of the element $a$ in the list $x$. This is called the multiplicity of $a$ in $x$. The function satisfies

$$\vdash_{\mathrm{PA}} \#_a\langle b, 0\rangle = (a =_* b) \tag{1}$$
$$\vdash_{\mathrm{PA}} \#_a(x \oplus y) = \#_a(x) + \#_a(y) \tag{2}$$

and it is defined by structural list recursion as a p.r. function:

$$\#_a(0) = 0$$
$$\#_a\langle v, w\rangle = (a =_* v) + \#_a(w).$$

We have also

$$\vdash_{\mathrm{PA}} a \,\varepsilon\, x \leftrightarrow \#_a(x) \neq 0. \tag{3}$$
$$\vdash_{\mathrm{PA}} x = 0 \leftrightarrow \forall a\, \#_a(x) = 0. \tag{4}$$

*Verification.* (1): Directly from definition.
(2): By structural induction on the list $x$. The base case is obvious. The induction step follows from

$$\begin{aligned}\#_a(\langle v, w\rangle \oplus y) &= \#_a\langle v, w \oplus y\rangle = (a =_* v) + \#_a(w \oplus y) \overset{\mathrm{IH}}{=} \\ &= (a =_* v) + \#_a(w) + \#_a(y) = \#_a\langle v, w\rangle + \#_a(y)\,.\end{aligned}$$

(3): By a straightforward structural induction on the list $x$.
(4): By a simple case analysis on whether or not the list $x$ is empty. □

**6.2.9 Permutations.** We wish to introduce into PA the binary predicate $x \sim y$ holding if the list $x$ is a permutation of the list $y$. For example:

$$\langle 1,2,3,0\rangle \quad \langle 2,1,3,0\rangle \quad \langle 2,3,1,0\rangle \quad \langle 1,3,2,0\rangle \quad \langle 3,1,2,0\rangle \quad \langle 3,2,1,0\rangle$$

are all permutations of the three-element list $\langle 1,2,3,0\rangle$. The standard mathematical definition uses a second-order concept (bijections over finite sets) which is not expressible directly in first-order arithmetic. Our definition of the predicate in PA is based on the following simple observation:

two lists are permutations precisely when every number has the same multiplicity in either list.

Thus we can define the predicate explicitly by

$$x \sim y \leftrightarrow \forall a\, \#_a(x) = \#_a(y)\,.$$

Note that from 6.2.8(3) we get

$$\vdash_{\text{PA}} x \sim y \leftrightarrow \forall a\big(a \mathrel{\varepsilon} x \to \#_a(x) = \#_a(y)\big) \land \forall a\big(a \mathrel{\varepsilon} y \to \#_a(x) = \#_a(y)\big).$$

Consequently, the predicate $x \sim y$ is primitive recursive.

**6.2.10 Basic properties of permutations.** First note the predicate $x \sim y$ constitutes an equivalence relation which is reflexive, symmetric and transitive. This is expressed in that order by

$$\vdash_{\text{PA}} x \sim x \tag{1}$$
$$\vdash_{\text{PA}} x \sim y \to y \sim x \tag{2}$$
$$\vdash_{\text{PA}} x \sim y \land y \sim z \to x \sim z. \tag{3}$$

Congruence properties of permutations are expressed by

$$\vdash_{\text{PA}} x \sim y \to \langle a, x\rangle \sim \langle a, y\rangle \tag{4}$$
$$\vdash_{\text{PA}} x \sim y \to L(x) = L(y) \tag{5}$$
$$\vdash_{\text{PA}} x_1 \sim y_1 \land x_2 \sim y_2 \to x_1 \oplus x_2 \sim y_1 \oplus y_2 \tag{6}$$
$$\vdash_{\text{PA}} x \sim y \land a \mathrel{\varepsilon} x \to a \mathrel{\varepsilon} y. \tag{7}$$

There is one cancellation law, namely:

$$\vdash_{\text{PA}} x_1 \oplus \langle a, x_2\rangle \sim y_1 \oplus \langle a, y_2\rangle \leftrightarrow x_1 \oplus x_2 \sim y_1 \oplus y_2. \tag{8}$$

Finally, we have also the following recurrent properties of permutations:

$$\vdash_{\text{PA}} x \sim 0 \leftrightarrow x = 0 \tag{9}$$
$$\vdash_{\text{PA}} x \sim \langle v, w\rangle \leftrightarrow \exists z_1 \exists z_2 (x = z_1 \oplus \langle v, z_2\rangle \land w \sim z_1 \oplus z_2). \tag{10}$$

In the sequel we will use these properties without explicitly referring to them.

*Proof.* Properties (1)–(3) hold trivially. Property (4) follows directly from the definition. Properties (6)–(9) follow from the properties of the multiplicity function (see Par. 6.2.8).

(10): In the direction ($\to$) assume $x \sim \langle v, w\rangle$. Then $v \mathrel{\varepsilon} x$ by (7) and thus, by 6.1.13(4), we have $x = z_1 \oplus \langle v, z_2\rangle$ for some $z_1, z_2$. Now it suffices to apply (8) to get $w \sim z_1 \oplus z_2$. The reverse direction ($\leftarrow$) follows from (8).

(5): This is proved as $\forall y(5)$ by structural induction on the list $x$. The base case is straightforward. In the induction step, when $x = \langle v, w\rangle$ for some $v, w$, take any $y$ such that $\langle v, w\rangle \sim y$. By (10), there are lists $z_1, z_2$ such that $y = z_1 \oplus \langle v, z_2\rangle$ and $w \sim z_1 \oplus z_2$. We then obtain

$$\begin{aligned} L\langle v, w\rangle = L(w) + 1 &\overset{\text{IH}}{=} L(z_1 \oplus z_2) + 1 = L(z_1) + L(z_2) + 1 = \\ &= L(z_1) + L\langle v, z_2\rangle = L(z_1 \oplus \langle v, z_2\rangle). \end{aligned}$$

Note that the induction hypothesis is applied with $z_1 \oplus z_2$ in place of $y$. □

## 6.3 Combinatorial Functions over Lists

**6.3.1 Introduction.** In this section we will study several combinatorial problems over lists. Such problems are good examples for program synthesis. We begin by bootstrapping of a specification and then synthesize an algorithm that satisfies it. By reverting the process we obtain the program correctness.

**6.3.2 Suffixes.** Let us start by considering the problem of generating the last segments (suffixes) of a list. For example, the following lists are all suffixes of the four-element list $\langle 1,2,3,4,0\rangle$:

$$\langle 1,2,3,4,0\rangle \quad \langle 2,3,4,0\rangle \quad \langle 3,4,0\rangle \quad \langle 4,0\rangle \quad 0.$$

The specification predicate $x \sqsupset y$ (*$x$ is suffix of $y$*) is defined explicitly by

$$x \sqsupset y \leftrightarrow \exists z\, x = z \oplus y.$$

Note that the existential quantifier in the definition can be bounded by $\leq x$ and thus the predicate is primitive recursive. Our task is to find a program for the function $\mathit{Suffixes}(x)$ which returns a list of all suffixes of the list $x$:

$$\vdash_{\mathrm{PA}} y \mathrel{\varepsilon} \mathit{Suffixes}(x) \leftrightarrow x \sqsupset y, \tag{1}$$

Note that the order of the elements in the list $\mathit{Suffixes}(x)$ is irrelevant.

The implementation of $\mathit{Suffixes}(x)$ is based on the following recurrent properties of the specification predicate:

$$\vdash_{\mathrm{PA}} 0 \sqsupset y \leftrightarrow y = 0 \tag{2}$$

$$\vdash_{\mathrm{PA}} \langle v,w\rangle \sqsupset y \leftrightarrow y = \langle v,w\rangle \lor w \sqsupset y. \tag{3}$$

Synthesizing both equivalences together leads to the following definition:

$$\mathit{Suffixes}(0) = \langle 0,0\rangle$$
$$\mathit{Suffixes}\,\langle v,w\rangle = \big\langle \langle v,w\rangle, \mathit{Suffixes}(w)\big\rangle.$$

Note that this is a definition by structural list recursion of a p.r. function.

*Verification.* (2): Directly from 6.1.11(1). (3): We have

$$\begin{aligned}\langle v,w\rangle \sqsupset y \Leftrightarrow \exists z\,\langle v,w\rangle = z \oplus y &\overset{(*)}{\Leftrightarrow}\\ \Leftrightarrow \langle v,w\rangle = 0 \oplus y \lor \exists z_1 \exists z_2\, \langle v,w\rangle = \langle z_1,z_2\rangle \oplus y &\Leftrightarrow\\ \Leftrightarrow y = \langle v,w\rangle \lor \exists z_2\, w = z_2 \oplus y \Leftrightarrow y = \langle v,w\rangle \lor w \sqsupset y.&\end{aligned}$$

The step $(*)$ is by case analysis on whether or not the list $z$ is empty.

(1): By structural induction on the list $x$. The base case is trivial:

$$y \mathrel{\varepsilon} \mathit{Suffixes}(0) \Leftrightarrow y \mathrel{\varepsilon} \langle 0,0\rangle \Leftrightarrow y = 0 \overset{(2)}{\Leftrightarrow} 0 \sqsupset y.$$

The induction step follows from

$$y \mathrel{\varepsilon} \mathit{Suffixes}\,\langle v,w\rangle \Leftrightarrow y \mathrel{\varepsilon} \big\langle \langle v,w\rangle, \mathit{Suffixes}(w)\big\rangle \Leftrightarrow$$
$$\Leftrightarrow y = \langle v,w\rangle \lor y \mathrel{\varepsilon} \mathit{Suffixes}(w) \overset{\text{IH}}{\Leftrightarrow} y = \langle v,w\rangle \lor w \sqsupset y \overset{(3)}{\Leftrightarrow} \langle v,w\rangle \sqsupset y. \qquad \square$$

**6.3.3 Auxiliary function.** Consider the binary function $\mathit{Mape}(a,x)$ defined by structural recursion on the list $x$ as a p.r. function:

$\mathit{Mape}(a,0) = 0$
$\mathit{Mape}(a,\langle v,w\rangle) = \big\langle \langle a,v\rangle, \mathit{Mape}(a,w)\big\rangle.$

The function satisfies:

$$\vdash_{\text{PA}} y \mathrel{\varepsilon} \mathit{Mape}(a,x) \leftrightarrow \exists z(z \mathrel{\varepsilon} x \land y = \langle a,z\rangle). \tag{1}$$

*Verification.* Property (1) is proved by structural induction on the list $x$. The base case is obvious. The induction step follows from

$$y \mathrel{\varepsilon} \mathit{Mape}(a,\langle v,w\rangle) \Leftrightarrow y \mathrel{\varepsilon} \big\langle \langle a,v\rangle, \mathit{Mape}(a,w)\big\rangle \Leftrightarrow$$
$$\Leftrightarrow y = \langle a,v\rangle \lor y \mathrel{\varepsilon} \mathit{Mape}(a,w) \overset{\text{IH}}{\Leftrightarrow} y = \langle a,v\rangle \lor \exists z(z \mathrel{\varepsilon} w \land y = \langle a,z\rangle) \Leftrightarrow$$
$$\Leftrightarrow \exists z\big((z = v \lor z \mathrel{\varepsilon} w) \land y = \langle a,z\rangle\big) \Leftrightarrow \exists z(z \mathrel{\varepsilon} \langle v,w\rangle \land y = \langle a,z\rangle). \qquad \square$$

**6.3.4 Prefixes.** Our next example is the problem of generating the initial segments (prefixes) of a list. For example:

$$0 \quad \langle 1,0\rangle \quad \langle 1,2,0\rangle \quad \langle 1,2,3,0\rangle \quad \langle 1,2,3,4,0\rangle$$

are all prefixes of the list $\langle 1,2,3,4,0\rangle$. The specification predicate $y \sqsubset x$, which is read as $y$ *is prefix of* $x$, is defined explicitly as primitive recursive by

$$y \sqsubset x \leftrightarrow \exists z\, x = y \oplus z.$$

The aim is to find an algorithm for the function $\mathit{Prefixes}(x)$ which returns a list of all prefixes of the list $x$:

$$\vdash_{\text{PA}} y \mathrel{\varepsilon} \mathit{Prefixes}(x) \leftrightarrow y \sqsubset x. \tag{1}$$

The order of the elements in the resulting list is irrelevant.

The program for the operation $\mathit{Prefixes}(x)$ is based on the following recurrent properties of the specification predicate:

$$\vdash_{\text{PA}} y \sqsubset 0 \leftrightarrow y = 0 \tag{2}$$
$$\vdash_{\text{PA}} y \sqsubset \langle v, w\rangle \leftrightarrow y = 0 \lor \exists z(z \sqsubset w \land y = \langle v, z\rangle). \tag{3}$$

By combining these two equivalences together we obtain:

$$\begin{aligned}&\mathit{Prefixes}(0) = \langle 0, 0\rangle\\ &\mathit{Prefixes}\,\langle v, w\rangle = \big\langle 0, \mathit{Mape}(v, \mathit{Prefixes}(w))\big\rangle.\end{aligned}$$

This is a definition by structural list recursion of a p.r. function.

*Verification.* (2): Directly from 6.1.11(1). (3): We have

$$\begin{aligned}&y \sqsubset \langle v, w\rangle \Leftrightarrow \exists z\,\langle v, w\rangle = y \oplus z \overset{(*)}{\Leftrightarrow}\\ &\Leftrightarrow y = 0 \lor \exists y_1 \exists y_2\big(y = \langle y_1, y_2\rangle \land \exists z\,\langle v, w\rangle = \langle y_1, y_2\rangle \oplus z\big) \Leftrightarrow\\ &\Leftrightarrow y = 0 \lor \exists y_2\big(y = \langle v, y_2\rangle \land \exists z\, w = y_2 \oplus z\big) \Leftrightarrow\\ &\Leftrightarrow y = 0 \lor \exists y_2(y = \langle v, y_2\rangle \land y_2 \sqsubset w) \Leftrightarrow y = 0 \lor \exists z(z \sqsubset w \land y = \langle v, z\rangle).\end{aligned}$$

The step $(*)$ is by case analysis on whether or not the list $y$ is empty.

(1): By structural induction on the list $x$ as $\forall y(1)$. The base case follows from (2). In the induction step when $x = \langle v, w\rangle$ take any $y$ and we have

$$\begin{aligned}y \mathrel{\varepsilon} \mathit{Prefixes}\,\langle v, w\rangle &\Leftrightarrow y \mathrel{\varepsilon} \big\langle 0, \mathit{Mape}(v, \mathit{Prefixes}(w))\big\rangle \Leftrightarrow\\ &\Leftrightarrow y = 0 \lor y \mathrel{\varepsilon} \mathit{Mape}(v, \mathit{Prefixes}(w)) \overset{6.3.3(1)}{\Leftrightarrow}\\ &\Leftrightarrow y = 0 \lor \exists z(z \mathrel{\varepsilon} \mathit{Prefixes}(w) \land y = \langle v, z\rangle) \overset{\text{IH}}{\Leftrightarrow}\\ &\Leftrightarrow y = 0 \lor \exists z(z \sqsubset w \land y = \langle v, z\rangle) \overset{(3)}{\Leftrightarrow} y \sqsubset \langle v, w\rangle.\end{aligned}$$

Note that the induction hypothesis is applied with $z$ in place of $y$. □

**6.3.5 Segments.** In this paragraph we consider the problem of generating the (contiguous) segments of a list. For example, the following lists are all segments of the four-element list $\langle 1, 2, 3, 4, 0\rangle$:

$$\begin{array}{llllll}0 & \langle 1,0\rangle & \langle 1,2,0\rangle & \langle 1,2,3,0\rangle & \langle 1,2,3,4,0\rangle & \\ \langle 2,0\rangle & \langle 2,3,0\rangle & \langle 2,3,4,0\rangle & \langle 3,0\rangle & \langle 3,4,0\rangle & \langle 4,0\rangle.\end{array}$$

The specification predicate $y \subset x$ ($y$ *is segment of* $x$) is defined explicitly as primitive recursive by

$$y \subset x \leftrightarrow \exists z_1 \exists z_2\, x = z_1 \oplus y \oplus z_2.$$

Our task is to find a program for the function $\mathit{Segments}(x)$ which returns a list of all segments of the list $x$:

$$\vdash_{\text{PA}} y \mathrel{\varepsilon} \mathit{Segments}(x) \leftrightarrow y \subset x. \tag{1}$$

Note that the order of the elements in the list *Segments*$(x)$ is irrelevant.

Our implementation of *Segments*$(x)$ is based on the following recurrent properties of the specification predicate:

$$\vdash_{\mathrm{PA}} y \subset 0 \leftrightarrow y = 0 \tag{2}$$

$$\vdash_{\mathrm{PA}} y \subset \langle v,w\rangle \leftrightarrow \exists z(z \sqsubset w \wedge y = \langle v,z\rangle) \vee y \subset w. \tag{3}$$

Synthesizing both equivalences together leads to the following definition:

$$\begin{aligned}&\mathit{Segments}(0) = \langle 0,0\rangle\\ &\mathit{Segments}\,\langle v,w\rangle = \mathit{Mape}(v,\mathit{Prefixes}(w)) \oplus \mathit{Segments}(w).\end{aligned}$$

Note that this is a definition by structural list recursion of a p.r. function.

*Verification.* (2): Directly from 6.1.11(1). (3): First note that

$$\vdash_{\mathrm{PA}} 0 \subset x. \tag{$\dagger_1$}$$

We then have

$$\begin{aligned}&y \subset \langle v,w\rangle \Leftrightarrow \exists z_1\exists z_2\,\langle v,w\rangle = z_1 \oplus y \oplus z_2 \overset{(*)}{\Leftrightarrow}\\
&\quad\Leftrightarrow \exists z_2\,\langle v,w\rangle = 0 \oplus y \oplus z_2 \vee \exists z_3\exists z_4\exists z_2\,\langle v,w\rangle = \langle z_3,z_4\rangle \oplus y \oplus z_2 \Leftrightarrow\\
&\quad\Leftrightarrow \exists z_2\,\langle v,w\rangle = y \oplus z_2 \vee \exists z_4\exists z_2\, w = z_4 \oplus y \oplus z_2 \Leftrightarrow\\
&\quad\Leftrightarrow \exists z_2\,\langle v,w\rangle = y \oplus z_2 \vee y \subset w \overset{(*)}{\Leftrightarrow}\\
&\quad\Leftrightarrow y = 0 \vee \exists z_5\exists z\big(y = \langle z_5,z\rangle \wedge \exists z_2\,\langle v,w\rangle = \langle z_5,z\rangle \oplus z_2\big) \vee y \subset w \overset{(\dagger_1)}{\Leftrightarrow}\\
&\quad\Leftrightarrow \exists z\big(y = \langle v,z\rangle \wedge \exists z_2\, w = z \oplus z_2\big) \vee y \subset w \Leftrightarrow\\
&\quad\Leftrightarrow \exists z(z \sqsubset w \wedge y = \langle v,z\rangle) \vee y \subset w.\end{aligned}$$

The steps $(*)$ are by case analysis on whether or not the list $z_1$ or $y$ is empty.

(1): By structural induction on the list $x$. The base case follows from (2). In the induction step we have

$$\begin{aligned}&y \mathrel{\varepsilon} \mathit{Segments}\,\langle v,w\rangle \Leftrightarrow y \mathrel{\varepsilon} \mathit{Mape}(v,\mathit{Prefixes}(w)) \oplus \mathit{Segments}(w) \Leftrightarrow\\
&\quad\Leftrightarrow y \mathrel{\varepsilon} \mathit{Mape}(v,\mathit{Prefixes}(w)) \vee y \mathrel{\varepsilon} \mathit{Segments}(w) \overset{6.3.3(1),\mathrm{IH}}{\Leftrightarrow}\\
&\quad\Leftrightarrow \exists z(z \mathrel{\varepsilon} \mathit{Prefixes}(w) \wedge y = \langle v,z\rangle) \vee y \subset w \overset{6.3.4(1)}{\Leftrightarrow}\\
&\quad\Leftrightarrow \exists z(z \sqsubset w \wedge y = \langle v,z\rangle) \vee y \subset w \overset{(3)}{\Leftrightarrow} y \subset \langle v,w\rangle.\end{aligned}$$

□

**6.3.6 Interleave.** In this paragraph we consider the problem of finding of all possible ways of inserting an element into a list. For example:

$$\langle 1,2,3,4,0\rangle \quad \langle 2,1,3,4,0\rangle \quad \langle 2,3,1,4,0\rangle \quad \langle 2,3,4,1,0\rangle$$

are all possible ways of inserting the number 1 into the list $\langle 2,3,4,0\rangle$. The specification predicate $y \approx x[\downarrow a]$, which holds if the list $y$ is obtained from the list $x$ by inserting the element $a$ into it, has the following explicit definition:

$$y \approx x[\downarrow a] \leftrightarrow \exists z_1 \exists z_2 (x = z_1 \oplus z_2 \land y = z_1 \oplus \langle a, z_2\rangle).$$

Both existential quantifiers can be bounded by $\leq x$ and thus the predicate is primitive recursive. The goal is to construct a program for the function *Interleave*$(a,x)$ satisfying

$$\vdash_{\mathrm{PA}}\ y \mathrel{\varepsilon} \mathit{Interleave}(a,x) \leftrightarrow y \approx x[\downarrow a]. \tag{1}$$

Note that the order of the elements of the list *Interleave*$(a,x)$ is irrelevant.

The implementation is based on the following two properties of the specification predicate:

$$\vdash_{\mathrm{PA}}\ y \approx 0[\downarrow a] \leftrightarrow y = \langle a,0\rangle \tag{2}$$

$$\vdash_{\mathrm{PA}}\ y \approx \big\langle v,w\big\rangle[\downarrow a] \leftrightarrow y = \langle a,v,w\rangle \lor \exists z (z \approx w[\downarrow a] \land y = \langle v,z\rangle). \tag{3}$$

By synthesizing both equivalences together we obtain the following definition:

$$\begin{aligned}&\mathit{Interleave}(a,0) = \big\langle \langle a,0\rangle, 0\big\rangle\\ &\mathit{Interleave}(a,\langle v,w\rangle) = \big\langle \langle a,v,w\rangle, \mathit{Mape}(v,\mathit{Interleave}(a,w))\big\rangle.\end{aligned}$$

This is a definition of by structural list recursion and hence the function *Interleave*$(a,x)$ is primitive recursive.

*Verification.* (2): Directly from 6.1.11(1). (3): We have

$$\begin{aligned}&y \approx \big\langle v,w\big\rangle[\downarrow a] \Leftrightarrow \exists z_1\exists z_2\big(\langle v,w\rangle = z_1 \oplus z_2 \land y = z_1 \oplus \langle a,z_2\rangle\big) \overset{(*)}{\Leftrightarrow}\\ &\exists z_2\big(\langle v,w\rangle = 0 \oplus z_2 \land y = 0 \oplus \langle a,z_2\rangle\big) \lor\\ &\quad \lor \exists z_3\exists z_4\exists z_2\big(\langle v,w\rangle = \langle z_3,z_4\rangle \oplus z_2 \land y = \langle z_3,z_4\rangle \oplus \langle a,z_2\rangle\big) \Leftrightarrow\\ &y = \langle a,v,w\rangle \lor \exists z_4\exists z_2\big(w = z_4 \oplus z_2 \land y = \langle v,z_4\rangle \oplus \langle a,z_2\rangle\big) \Leftrightarrow\\ &y = \langle a,v,w\rangle \lor \exists z\big(\exists z_4\exists z_2(w = z_4 \oplus z_2 \land z = z_4 \oplus \langle a,z_2\rangle) \land y = \langle v,z\rangle\big) \Leftrightarrow\\ &y = \langle a,v,w\rangle \lor \exists z(z \approx w[\downarrow a] \land y = \langle v,z\rangle).\end{aligned}$$

The step $(*)$ is by case analysis on whether or not the list $z_1$ is empty.

(1): By structural induction on the list $x$ as $\forall y(1)$ The base case follows from (2). In the induction step when $x = \langle v,w\rangle$ take any $y$ and we have

$$\begin{aligned}&y \mathrel{\varepsilon} \mathit{Interleave}(a,\langle v,w\rangle) \Leftrightarrow y \mathrel{\varepsilon} \big\langle \langle a,v,w\rangle, \mathit{Mape}(v,\mathit{Interleave}(a,w))\big\rangle \Leftrightarrow\\ &\quad \Leftrightarrow y = \langle a,v,w\rangle \lor y \mathrel{\varepsilon} \mathit{Mape}(v,\mathit{Interleave}(a,w)) \overset{6.3.3(1)}{\Leftrightarrow}\end{aligned}$$

$$\Leftrightarrow y = \langle a, v, w\rangle \lor \exists z(z \,\varepsilon\, \mathit{Interleave}(a, w) \land y = \langle v, z\rangle) \overset{\text{IH}}{\Leftrightarrow}$$

$$\Leftrightarrow y = \langle a, v, w\rangle \lor \exists z(z \approx w[\downarrow a] \land y = \langle v, z\rangle) \overset{(3)}{\Leftrightarrow} y \approx \langle v, w\rangle[\downarrow a].$$

Note that the induction hypothesis is applied with $z$ in place of $y$. □

**6.3.7 Auxiliary function.** Consider the binary function $\mathit{Mapi}(a, x)$ defined by structural recursion on the list $x$ as a p.r. function:

$\mathit{Mapi}(a, 0) = 0$
$\mathit{Mapi}(a, \langle v, w\rangle) = \mathit{Interleave}(a, v) \oplus \mathit{Mapi}(a, w).$

The function satisfies:

$$\vdash_{\text{PA}} y \,\varepsilon\, \mathit{Mapi}(a, x) \leftrightarrow \exists z(z \,\varepsilon\, x \land y \approx z[\downarrow a]). \tag{1}$$

*Verification.* Property (1) is proved by structural induction on the list $x$. The base case is obvious. The induction step follows from

$$y \,\varepsilon\, \mathit{Mapi}(a, \langle v, w\rangle) \Leftrightarrow y \,\varepsilon\, \mathit{Interleave}(a, v) \oplus \mathit{Mapi}(a, w) \Leftrightarrow$$

$$\Leftrightarrow y \,\varepsilon\, \mathit{Interleave}(a, v) \lor y \,\varepsilon\, \mathit{Mapi}(a, w) \overset{\text{6.3.6(1),IH}}{\Leftrightarrow}$$

$$\Leftrightarrow y \approx v[\downarrow a] \lor \exists z(z \,\varepsilon\, w \land y \approx z[\downarrow a]) \Leftrightarrow$$

$$\Leftrightarrow \exists z\big((z = v \lor z \,\varepsilon\, w) \land y \approx z[\downarrow a]\big) \Leftrightarrow \exists z(z \,\varepsilon\, \langle v, w\rangle \land y \approx z[\downarrow a]).$$ □

**6.3.8 Permutations.** In our last example of this section we will consider the problem of generating of all permutations of a list. Our aim is find a program for the function $\mathit{Perms}(x)$ such that

$$\vdash_{\text{PA}} y \,\varepsilon\, \mathit{Perms}(x) \leftrightarrow y \sim x. \tag{1}$$

Here, the $y \sim x$ is the permutation relation defined in Par. 6.2.9. Note that the order of elements in the list $\mathit{Perms}(x)$ is irrelevant.

The implementation of $\mathit{Perms}(x)$ is based on the following recurrent properties of the permutation relation:

$$\vdash_{\text{PA}} y \sim 0 \leftrightarrow y = 0 \tag{2}$$

$$\vdash_{\text{PA}} y \sim \langle v, w\rangle \leftrightarrow \exists z(z \sim w \land y \approx z[\downarrow v]). \tag{3}$$

The operation $\mathit{Perms}(x)$ is then defined by structural list recursion as a p.r. function by

$\mathit{Perms}(0) = \langle 0, 0\rangle$
$\mathit{Perms}\,\langle v, w\rangle = \mathit{Mapi}(v, \mathit{Perms}(w)).$

*Verification.* (2): This is 6.2.10(9). (3): We have

$$y \sim \langle v, w\rangle \overset{6.2.10(10)}{\Leftrightarrow} \exists z_1 \exists z_2 (z_1 \oplus z_2 \sim w \wedge y = z_1 \oplus \langle v, z_2\rangle) \Leftrightarrow$$
$$\Leftrightarrow \exists z_3 \exists z_4 \big(z_3 \oplus z_4 \sim w \wedge \exists z_1 \exists z_2 (z_3 \oplus z_4 = z_1 \oplus z_2 \wedge y = z_1 \oplus \langle v, z_2\rangle)\big) \Leftrightarrow$$
$$\Leftrightarrow \exists z_3 \exists z_4 \big(z_3 \oplus z_4 \sim w \wedge y \approx (z_3 \oplus z_4)[\downarrow v]\big) \Leftrightarrow \exists z (z \sim w \wedge y \approx z[\downarrow v]).$$

(1): By structural induction on the list $x$ as $\forall y(1)$. The base case follows from (2). In the induction step when $x = \langle v, w\rangle$ take any $y$ and we have

$$y \mathrel{\varepsilon} \mathit{Perms}\,\langle v, w\rangle \Leftrightarrow y \mathrel{\varepsilon} \mathit{Mapi}(v, \mathit{Perms}(w)) \overset{6.3.7(1)}{\Leftrightarrow}$$
$$\exists z (z \mathrel{\varepsilon} \mathit{Perms}(w) \wedge y \approx z[\downarrow v]) \overset{\mathrm{IH}}{\Leftrightarrow} \exists z (z \sim w \wedge y \approx z[\downarrow v]) \overset{(3)}{\Leftrightarrow} y \sim \langle v, w\rangle.$$

Note that the induction hypothesis is applied with $z$ in place of $y$. □

## 6.4 Sorting of Lists

**6.4.1 Introduction.** In this section we will consider the problem of sorting of lists. We will demonstrate the verification of two sorting algorithms: insertion sort and merge sort. We start by introducing into PA some of the specification predicates which are needed to specify and verify sorting algorithms. Recall that the properties of one of these predicates, the permutation predicate $x \sim y$, has already been investigated (see Par. 6.2.9 for details).

**6.4.2 Lower bounds of lists.** The predicate $a \preceq x$ holds if the number $a$ is a *lower bound* of the list $x$, i.e. we have $a \leq b$ for every element $b$ of $x$. The predicate is defined explicitly as primitive recursive by

$$a \preceq x \leftrightarrow \forall b (b \mathrel{\varepsilon} x \to a \leq b).$$

The predicate satisfies

$$\vdash_{\mathrm{PA}} a \preceq 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} a \preceq \langle v, w\rangle \leftrightarrow a \leq v \wedge a \preceq w \tag{2}$$
$$\vdash_{\mathrm{PA}} a \leq b \wedge b \preceq x \to a \preceq x \tag{3}$$
$$\vdash_{\mathrm{PA}} a \preceq x \oplus y \leftrightarrow a \preceq x \wedge a \preceq y \tag{4}$$
$$\vdash_{\mathrm{PA}} x \sim y \to a \preceq x \leftrightarrow a \preceq y. \tag{5}$$

*Proof.* (1): Obvious. (2): This follows from

$$a \preceq \langle v, w\rangle \Leftrightarrow \forall b (b \mathrel{\varepsilon} \langle v, w\rangle \to a \leq b) \overset{6.1.13(2)}{\Leftrightarrow} \forall b (b = v \vee b \mathrel{\varepsilon} w \to a \leq b) \Leftrightarrow$$
$$\Leftrightarrow a \leq v \wedge \forall b (b \mathrel{\varepsilon} w \to a \leq b) \Leftrightarrow a \leq v \wedge a \preceq w.$$

(3): Obvious. (4): This follows from

$$a \le x \oplus y \Leftrightarrow \forall b\big(b \,\varepsilon\, x \oplus y \to a \le b\big) \overset{6.1.13(3)}{\Leftrightarrow} \forall b\big(b \,\varepsilon\, x \vee b \,\varepsilon\, y \to a \le b\big) \Leftrightarrow$$
$$\Leftrightarrow \forall b\big(b \,\varepsilon\, x \to a \le b\big) \wedge \forall b\big(b \,\varepsilon\, y \to a \le b\big) \Leftrightarrow a \le x \wedge a \le y.$$

(5) Suppose that $x \sim y$. We have

$$a \le x \Leftrightarrow \forall b\big(b \,\varepsilon\, x \to a \le b\big) \overset{6.2.10(7)}{\Leftrightarrow} \forall b\big(b \,\varepsilon\, y \to a \le b\big) \Leftrightarrow a \le y. \qquad \square$$

**6.4.3 Ordered lists.** The predicate $Ord(x)$ holds if $x$ is an ordered list, i.e. the elements of the list $x$ are stored in $x$ in increasing order. The predicate is explicitly defined as primitive recursive by

$$Ord(x) \leftrightarrow \forall i \forall j\big(i < j < L(x) \to x[i] \le x[j]\big).$$

We list here some properties of ordered lists which we will use in sequel:

$$\vdash_{\text{PA}} Ord(0) \tag{1}$$
$$\vdash_{\text{PA}} Ord\,\langle v, w\rangle \leftrightarrow v \le w \wedge Ord(w). \tag{2}$$
$$\vdash_{\text{PA}} Ord\,\langle v, w\rangle \to a \le \langle v, w\rangle \leftrightarrow a \le v. \tag{3}$$

*Proof.* (1): Obvious. (2): This follows from

$$\begin{aligned}
&Ord\,\langle v,w\rangle \Leftrightarrow \forall i \forall j\big(i < j < L\,\langle v,w\rangle \to \langle v,w\rangle[i] \le \langle v,w\rangle[j]\big) \overset{(*)}{\Leftrightarrow}\\
&\forall j\big(0 < j < L(w)+1 \to \langle v,w\rangle[0] \le \langle v,w\rangle[j]\big) \wedge\\
&\quad \wedge \forall i_1 \forall j\big(i_1+1 < j < L(w)+1 \to \langle v,w\rangle[i_1+1] \le \langle v,w\rangle[j]\big) \Leftrightarrow\\
&\forall j_1\big(j_1+1 < L(w)+1 \to v \le \langle v,w\rangle[j_1+1]\big) \wedge\\
&\quad \wedge \forall i_1 \forall j_1\big(i_1+1 < j_1+1 < L(w)+1 \to w[i_1] \le \langle v,w\rangle[j_1+1]\big) \Leftrightarrow\\
&\forall j_1\big(j_1 < L(w) \to v \le w[j_1]\big) \wedge\\
&\quad \wedge \forall i_1 \forall j_1\big(i_1 < j_1 < L(w) \to w[i_1] \le w[j_1]\big) \Leftrightarrow v \le w \wedge Ord(w).
\end{aligned}$$

The step marked by $(*)$ is by case analysis on whether or not $i = 0$.

(3): If $Ord\,\langle v,w\rangle$ then $v \le w$ by (2) and thus, by 6.4.2(3), we have

$$a \le v \to a \le w. \tag{$\dagger_1$}$$

We then obtain

$$a \le \langle v,w\rangle \overset{6.4.2(2)}{\Leftrightarrow} a \le v \wedge a \le w \overset{(\dagger_1)}{\Leftrightarrow} a \le v. \qquad \square$$

## *Insertion Sort*

**6.4.4 Introduction.** The simplest sorting algorithm is *insertion sort* which takes order $\mathcal{O}(L(x)^2)$ time to sort a list $x$. Insertion sort works on a non-empty list by recursively sorting its tail and then inserts its first element into the sorted list.

**6.4.5 Insertion.** At the heart of insertion sort algorithm is the insertion function $Insert(a,x)$ which takes an ordered list $x$ and yields a new one by inserting the element $a$ into it. The function satisfies

$$\vdash_{\mathrm{PA}} Insert(a,x) \sim \langle a,x\rangle \tag{1}$$

$$\vdash_{\mathrm{PA}} Ord(x) \to Ord\, Insert(a,x) \tag{2}$$

and it is defined by structural recursion on the list $x$ as a p.r. function:

$$\begin{aligned}
&Insert(a,0) = \langle a,0\rangle\\
&Insert(a,\langle v,w\rangle) = \langle a,v,w\rangle \leftarrow a \le v\\
&Insert(a,\langle v,w\rangle) = \big\langle v, Insert(a,w)\big\rangle \leftarrow a > v.
\end{aligned}$$

*Verification.* (1): By structural induction on the list $x$. The base case is obvious. In the induction step when $x = \langle v,w\rangle$ we consider two cases. If $a \le v$ then the claim follows directly from the definition. Otherwise $a > v$ and then

$$Insert(a,\langle v,w\rangle) \sim \big\langle v, Insert(a,w)\big\rangle \overset{\mathrm{IH}}{\sim} \langle v,a,w\rangle \sim \langle a,v,w\rangle.$$

As a simple consequence of (1) and 6.4.2(5) we get the following

$$\vdash_{\mathrm{PA}} b \le Insert(a,x) \leftrightarrow b \le a \wedge b \le x. \tag{$\dagger_1$}$$

(2): By structural induction on the list $x$. The base case is straightforward. In the induction step, when $x = \big\langle v,w\big\rangle$ for some $v,w$, assume $Ord\,\langle v,w\rangle$ and consider two cases. If $a \le v$ then we have

$$Ord\, Insert(a,\langle v,w\rangle) \Leftrightarrow Ord\,\langle a,v,w\rangle \overset{6.4.3(2)}{\Leftrightarrow} Ord\,\langle v,w\rangle \wedge a \le \langle v,w\rangle.$$

The last follows from assumptions by 6.4.3(3). If $a > v$ then we have

$$\begin{aligned}
&Ord\, Insert(a,\langle v,w\rangle) \Leftrightarrow Ord\,\big\langle v, Insert(a,w)\big\rangle \overset{6.4.3(2)}{\Leftrightarrow}\\
&Ord\, Insert(a,w) \wedge v \le Insert(a,w) \overset{(\dagger_1)}{\Leftrightarrow} Ord\, Insert(a,w) \wedge v \le a \wedge v \le w.
\end{aligned}$$

The last follows from assumptions and IH. □

**6.4.6 Insertion sort.** The function $Isort(x)$ recursively sorts the tail of an non-empty list and then inserts its first element into the sorted one. The function satisfies

$$\vdash_{\mathrm{PA}} \mathit{Isort}(x) \sim x \tag{1}$$
$$\vdash_{\mathrm{PA}} \mathit{Ord}\,\mathit{Isort}(x) \tag{2}$$

and it is defined by structural list recursion as a p.r. function:

$\mathit{Isort}(0) = 0$
$\mathit{Isort}\,\langle v, w\rangle = \mathit{Insert}(v, \mathit{Isort}(w))$.

*Verification.* (1): By structural list induction. The base case is straightforward and the induction step follows from

$$\mathit{Isort}\,\langle v, w\rangle \sim \mathit{Insert}(v, \mathit{Isort}(w)) \overset{6.4.5(1)}{\sim} \big\langle v, \mathit{Isort}(w)\big\rangle \overset{\mathrm{IH}}{\sim} \langle v, w\rangle.$$

(2): By structural list induction. The base case follows from 6.4.3(1). In the induction step, when $x = \big\langle v, w\big\rangle$ for some $v, w$, assume $\mathit{Ord}\,\langle v, w\rangle$. Then $\mathit{Ord}(w)$ by 6.4.3(2) and we get from IH:

$$\mathit{Ord}\,\mathit{Isort}(w) \overset{6.4.5(2)}{\Rightarrow} \mathit{Ord}\,\mathit{Insert}(v, \mathit{Isort}(w)) \Rightarrow \mathit{Ord}\,\mathit{Isort}\,\langle v, w\rangle. \qquad \square$$

## *Merge Sort*

**6.4.7 Introduction.** More efficient sorting algorithm than insertion sort is *merge sort* which takes order $\mathcal{O}(L(x) \lg L(x))$ time to sort a list $x$ The algorithm sorts a list by dividing it into two roughly equal parts. Each part is then recursively sorted and the resulting lists are merged into one list.

Our implementation uses the discrimination on whether or not $L(x) \le 1$. As we have

$$\vdash_{\mathrm{PA}} L(x) \le 1 \leftrightarrow (\pi_2(x) =_* 0) = 1,$$

the evaluation of the variant $L(x) \le 1$ takes constant time provided the expression $\pi_2(x) =_* 0$ is taken as its characteristic term.

**6.4.8 Splitting the list into two halves.** The function $\mathit{Split}(x)$ divides a list into two lists: the length of the first one is at most one more than the length of the second. The function satisfies

$$\vdash_{\mathrm{PA}} \exists y \exists z\, \mathit{Split}(x) = \langle y, z\rangle \tag{1}$$
$$\vdash_{\mathrm{PA}} \mathit{Split}(x) = \langle y, z\rangle \to x \sim y \oplus z \tag{2}$$
$$\vdash_{\mathrm{PA}} \mathit{Split}(x) = \langle y, z\rangle \to (L(y) = L(z) \lor L(y) = L(z) + 1) \tag{3}$$

and it is defined by course of values recursion with measure $L(x)$ as a p.r. function by

$\mathit{Split}(x) = \langle x, 0\rangle \leftarrow L(x) \le 1$

$$\mathit{Split}(x) = \big\langle \langle u, y\rangle, \langle v, z\rangle \big\rangle \leftarrow L(x) > 1 \land x = \langle u, v, w\rangle \land \mathit{Split}(w) = \langle y, z\rangle.$$

*Verification.* (2): By induction with measure $L(x)$ as $\forall y \forall z(2)$. Take any $y, z$ such that $\mathit{Split}(x) = \langle y, z\rangle$ and consider two cases. The case when $L(x) \leq 1$ is obvious. So suppose that $L(x) > 1$. Then $x = \langle u, v, w\rangle$ for some $u, v, w$. By (1) there are $y_1, z_1$ such that $\mathit{Split}(w) = \langle y_1, z_1\rangle$. By definition $\langle u, y_1\rangle = y$ and $\langle v, z_1\rangle = z$. We then obtain

$$\langle u, v, w\rangle \overset{\mathrm{IH}}{\sim} \langle u, v, y_1 \oplus z_1\rangle \sim \langle u, y_1\rangle \oplus \langle v, z_1\rangle \sim y \oplus z.$$

(1),(3): This is proved similarly. □

**6.4.9 Merging two ordered lists into one.** The function $\mathit{Merge}(x, y)$ merges two ordered lists into one ordered list. The function satisfies

$$\vdash_{\mathrm{PA}} \mathit{Merge}(x, y) \sim x \oplus y \tag{1}$$

$$\vdash_{\mathrm{PA}} \mathit{Ord}(x) \land \mathit{Ord}(y) \to \mathit{Ord}\,\mathit{Merge}(x, y) \tag{2}$$

and it is defined by course of values recursion with measure $L(x) + L(y)$ as a p.r. function by

$$\begin{aligned}
&\mathit{Merge}(0, y) = y\\
&\mathit{Merge}(\langle v_1, w_1\rangle, 0) = \langle v_1, w_1\rangle\\
&\mathit{Merge}(\langle v_1, w_1\rangle, \langle v_2, w_2\rangle) = \big\langle v_1, \mathit{Merge}(w_1, \langle v_2, w_2\rangle)\big\rangle \leftarrow v_1 \leq v_2\\
&\mathit{Merge}(\langle v_1, w_1\rangle, \langle v_2, w_2\rangle) = \big\langle v_2, \mathit{Merge}(\langle v_1, w_1\rangle, w_2)\big\rangle \leftarrow v_1 > v_2.
\end{aligned}$$

*Verification.* (1): By course of values induction with measure $L(x) + L(y)$. We consider two cases. The case when either $x = 0$ or $y = 0$ is straightforward. So suppose $x = \langle v_1, w_1\rangle$ and $y = \langle v_2, w_2\rangle$ for some $v_1, w_1, v_2, w_2$. If $v_1 \leq v_2$ then we have

$$\begin{aligned}
\mathit{Merge}(\langle v_1, w_1\rangle, \langle v_2, w_2\rangle) \sim \big\langle v_1, \mathit{Merge}(w_1, \langle v_2, w_2\rangle)\big\rangle &\overset{\mathrm{IH}}{\sim}\\
\sim \big\langle v_1, w_1 \oplus \langle v_2, w_2\rangle\big\rangle &\sim \langle v_1, w_1\rangle \oplus \langle v_2, w_2\rangle.
\end{aligned}$$

The subcase when $v_1 < v_2$ has a similar proof.

As a simple consequence of (1) and 6.4.2(5) we get

$$\vdash_{\mathrm{PA}} a \leq \mathit{Merge}(x, y) \leftrightarrow a \leq x \land a \leq y. \tag{$\dagger_1$}$$

(2): By course of values induction with measure $L(x) + L(y)$. Assume $\mathit{Ord}(x)$ and $\mathit{Ord}(y)$, and consider two cases. If $x = 0$ or $y = 0$ then the property holds trivially. So suppose $x = \langle v_1, w_1\rangle$ and $y = \langle v_2, w_2\rangle$ for some $v_1, w_1, v_2, w_2$. If $v_1 \leq v_2$ then we have

$$\begin{aligned}
&\mathit{Ord}\,\mathit{Merge}(\langle v_1,w_1\rangle,\langle v_2,w_2\rangle) \Leftrightarrow \mathit{Ord}\,\big\langle v_1,\mathit{Merge}(w_1,\langle v_2,w_2\rangle)\big\rangle \overset{6.4.3(2)}{\Leftrightarrow}\\
&\mathit{Ord}\,\mathit{Merge}(w_1,\langle v_2,w_2\rangle)\wedge v_1 \le \mathit{Merge}(w_1,\langle v_2,w_2\rangle) \overset{(\dagger_1)}{\Leftrightarrow}\\
&\mathit{Ord}\,\mathit{Merge}(w_1,\langle v_2,w_2\rangle)\wedge v_1 \le w_1 \wedge v_1 \le \langle v_2,w_2\rangle.
\end{aligned}$$

The last follows from IH and from assumptions by 6.4.3(2) and 6.4.3(3). The subcase when $v_1 < v_2$ is similar. □

**6.4.10 Merge sort.** The function $\mathit{Msort}(x)$ sorts a list by dividing it into two equal parts. Each part is then recursively sorted and the resulting lists are merged together. The function $\mathit{Msort}(x)$ satisfies

$$\vdash_{\mathrm{PA}} \mathit{Msort}(x) \sim x \tag{1}$$

$$\vdash_{\mathrm{PA}} \mathit{Ord}\,\mathit{Msort}(x) \tag{2}$$

and it is defined by course of values recursion with measure $L(x)$ as a p.r. function by

$$\begin{aligned}
&\mathit{Msort}(x) = x \leftarrow L(x) \le 1\\
&\mathit{Msort}(x) = \mathit{Merge}(\mathit{Msort}(y),\mathit{Msort}(z)) \leftarrow L(x) > 1 \wedge \mathit{Split}(x) = \langle y,z\rangle.
\end{aligned}$$

Its conditions of regularity

$$\vdash_{\mathrm{PA}} L(x) > 1 \wedge \mathit{Split}(x) = \langle y,z\rangle \to L(y) < L(x) \tag{3}$$

$$\vdash_{\mathrm{PA}} L(x) > 1 \wedge \mathit{Split}(x) = \langle y,z\rangle \to L(z) < L(x) \tag{4}$$

follows from 6.2.10(5) and 6.4.8(2)(3).

*Verification.* (1): By course of values induction with measure $L(x)$. We consider two cases. The case when $L(x) \le 1$ is obvious. So suppose $L(x) > 1$. By 6.4.8(1) there are $y,z$ such that $\mathit{Split}(x) = \langle y,z\rangle$. Note that $L(y) < L(x)$ and $L(z) < L(x)$ by (3),(4). We have

$$\begin{aligned}
\mathit{Msort}(x) \sim \mathit{Merge}(\mathit{Msort}(y),\mathit{Msort}(z)) &\overset{6.4.9(1)}{\sim}\\
\sim \mathit{Msort}(y) \oplus \mathit{Msort}(z) \overset{\mathrm{IH}}{\sim} y \oplus z &\overset{6.4.8(2)}{\sim} x.
\end{aligned}$$

(2): By course of values induction with measure $L(x)$. We consider two cases. The case when $L(x) \le 1$ is obvious. So suppose $L(x) > 1$. By 6.4.8(1) there are $y,z$ such that $\mathit{Split}(x) = \langle y,z\rangle$. Note that $L(y) < L(x)$ and $L(z) < L(x)$ by (3),(4). We have by IH

$$\begin{aligned}
\mathit{Ord}\,\mathit{Msort}(y) \wedge \mathit{Ord}\,\mathit{Msort}(z) &\overset{6.4.9(2)}{\Rightarrow} \mathit{Ord}\,\mathit{Merge}(\mathit{Msort}(y),\mathit{Msort}(z)) \Rightarrow\\
&\Rightarrow \mathit{Ord}\,\mathit{Msort}(x).
\end{aligned}$$

□

# Chapter 7
# Programs Operating on Trees

In this chapter we will study several kinds of branching data structures, called trees in computer science. We will consider two of them: binary trees and symbolic expressions.

Our discussion starts in Sect. 7.1 with *binary trees.* We use the so-called constructors to code binary trees into natural numbers. Both the principle of structural induction and structural recursion over binary trees are easily formalized within PA. The rest of this section is devoted the specification, implementation and verification of basic operations on binary trees.

We give two applications of binary trees. The first one is discussed in Sect. 7.2 where we study binary search trees which are very useful for representing finite sets. The cost of most operations involving a single element is linear to the depth of a binary search tree. If it is reasonably balanced the cost of each such operation is logarithmic to its size.

In Sect. 7.3 we will study another kind of binary trees, called Braun trees, which are suitable for efficient implementation of flexible arrays. As they are size-balanced, the cost of most operations (subscription, updating, adding and removing of the first or last element) is logarithmic to the size.

Another example of non-linear branching structures are symbolic expressions. Section 7.4 shows the arithmetization of numeric terms of first-order arithmetic. We design a compiler transforming the terms to programs for a simple stack machine operating in polish postfix form. We also give an example of a program which goes beyond structural recursion – it is the problem of rearranging terms into expressions with left associated addition.

In Sect. 7.5 we consider the problem of the implementation of universal function for the class of primitive recursive functions. That will be done by arithmetization of primitive recursive derivations. The proposed encoding is an example of a coding scheme for general trees. This is because primitive recursive derivations can be visualized as multiway branching structures. We finish the section by providing effective operations operating on the codes of primitive recursive derivations. Our last example will be the construction of a self-reproducing program.

## 7.1 Binary Trees

**7.1.1 Introduction.** In this section we will show how to arithmetize *binary trees labelled by natural numbers* (see Fig. 7.1). In most functional programming languages the type $Bt$ of binary trees can be defined by a *union type*:

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

A value of type $Bt$ is therefore either the *empty* tree $E$ or a non-empty tree of the form $Nd(x, l, r)$, where $x$ is the *label* of its root node and $l, r$ are values of type $Bt$ called the *left* and *right* subtrees of that non-empty tree. The constant $E$ and the function $Nd$ are called *constructors*.

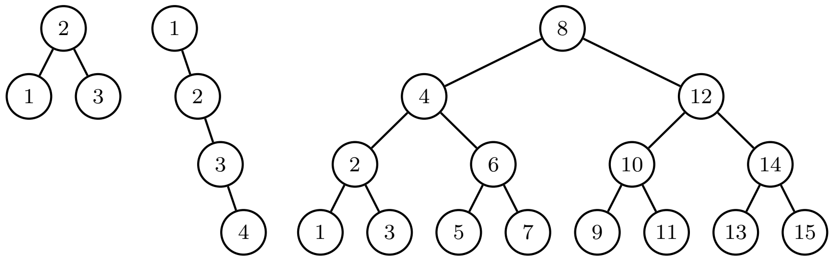

**Fig. 7.1** Examples of binary trees

**7.1.2 Constructors of binary trees.** Arithmetization of binary trees is done with the help of the following two pair constructors with pairwise different tags (see Par. 5.2.10 for details):

$$\langle\rangle = \langle 0, 0\rangle \qquad \text{(empty tree)}$$
$$\langle l \mid x \mid r\rangle = \langle 1, x, l, r\rangle. \qquad \text{(node)}$$

From the properties of the pairing function we obtain

$$\vdash_{\mathrm{PA}} \langle\rangle \neq \langle l \mid x \mid r\rangle$$
$$\vdash_{\mathrm{PA}} \langle l_1 \mid x_1 \mid r_1\rangle = \langle l_2 \mid x_2 \mid r_2\rangle \rightarrow x_1 = x_2 \wedge l_1 = l_2 \wedge r_1 = r_2$$

The first property says that the constructors are pairwise disjoint and the second that the functional constructor $\langle l \mid x \mid r\rangle$ is an injective mapping.

We obtain the pattern matching style of definitions of functions operating over the codes of binary trees with the conditionals of the form

```
case
  $t = \langle\rangle \Rightarrow \beta_1$
  $t = \langle l \mid x \mid r\rangle \Rightarrow_{x,l,r} \beta_2[x,l,r]$
  otherwise $\Rightarrow \beta_3$.
end
```

This is called *discrimination on the constructors of binary trees.* Recall that such conditionals are instances of pair constructors discrimination terms discussed in Par. 5.2.10.

The above conditional is evaluated as follows. Note that the expression

$$Tuple_*(2,t) \wedge_* [t]^2_1 =_* 0$$

is the characteristic term of its first variant since

$$\vdash_{\mathrm{PA}} t = \langle\rangle \leftrightarrow Tuple(2,t) \wedge [t]^2_1 = 0.$$

Similarly, the expression

$$Tuple_*(4,t) \wedge_* [t]^4_1 =_* 1$$

is the characteristic term of its second variant as we have

$$\vdash_{\mathrm{PA}} \exists x \exists l \exists r t = \langle l \mid x \mid r\rangle \leftrightarrow Tuple(4,t) \wedge [t]^4_1 = 1.$$

Finally note that we have

$$\vdash_{\mathrm{PA}} t = \langle l \mid x \mid r\rangle \rightarrow x = [t]^4_2 \wedge l = [t]^4_3 \wedge r = [t]^4_4$$

and therefore, the terms $[t]^4_2$, $[t]^4_3$ and $[t]^4_4$ are the witnessing terms for the output variables $x, l, r$ of the second variant of the conditional.

**7.1.3 Arithmetization of binary trees.** We wish to assign to every binary tree $T$ a unique number $\ulcorner T\urcorner$, called *the code of* $T$. The mapping is defined inductively on the structure of binary trees:

- If $T$ is the empty tree then $\ulcorner T\urcorner$ is the number $\langle\rangle$.
- If $T$ is a non-empty tree with the root label $x$, the left son $T_1$, and the right son $T_2$, then $\ulcorner T\urcorner$ is the number $\langle \ulcorner T_1\urcorner \mid x \mid \ulcorner T_2\urcorner\rangle$.

For instance, the code of the first binary tree from Fig. 7.1 is the number

$$\langle\langle\langle\rangle \mid 1 \mid \langle\rangle\rangle \mid 2 \mid \langle\langle\rangle \mid 3 \mid \langle\rangle\rangle\rangle = 26\,646\,277\,093\,331\,868\,372\,637.$$

Clearly, the mapping $\ulcorner T\urcorner$ is injective.

We will use the discrimination on the constructors of binary trees in the definition of the predicate $Bt(t)$ holding of the codes of binary trees. The predicate is defined by course of values recursion as primitive recursive by

$Bt\,\langle\rangle$
$Bt\,\langle l \mid x \mid r\rangle \leftarrow Bt(l) \land Bt(r).$

Note that the following are the omitted default clauses of the definition:

$\neg Bt\,\langle l \mid x \mid r\rangle \leftarrow \neg Bt(l) \lor \neg Bt(r)$
$\neg Bt(t) \leftarrow t \neq \langle\rangle \land \neg\exists x \exists l \exists r\, t = \langle l \mid x \mid r\rangle.$

Consequently, the clausal definition is equivalent to

$$\vdash_{\text{PA}} Bt(t) \leftrightarrow t = \langle\rangle \lor \exists x \exists l \exists r\big(t = \langle l \mid x \mid r\rangle \land Bt(l) \land Bt(r)\big).$$

The embedding of binary trees into natural numbers is so straightforward that we will henceforth identify binary trees with their codes, i.e. with the subset $Bt$ of natural numbers. In our case we will say *the binary tree $t$* instead of *the code $t$ of a binary tree.*

**7.1.4 Case analysis on binary trees.** Directly from the definition of the predicate $Bt$ we obtain the following property

$$\vdash_{\text{PA}} Bt(t) \rightarrow t = \langle\rangle \lor \exists x \exists l \exists r\, t = \langle l \mid x \mid r\rangle.$$

This is called the principle of *structural case analysis on the constructors of the binary tree $t$.*

We can use the above principle of structural case analysis in order to establish the admissibility of a certain kind of conditional discriminations on the constructors of binary trees. These are of the form

```
$Bt(t) \rightarrow$ **case**
        $t = \langle\rangle \Rightarrow \beta_1$
        $t = \langle l \mid x \mid r\rangle \Rightarrow_{x,l,r} \beta_2[x,l,r]$
    **end**
```

Because of the precondition $Bt(t)$ we have to evaluate only two alternatives instead of three. Moreover, as we have

$$\vdash_{\text{PA}} Bt(t) \rightarrow t = \langle\rangle \leftrightarrow [t]_1^2 = 0$$
$$\vdash_{\text{PA}} Bt(t) \rightarrow \exists x \exists l \exists r t = \langle l \mid x \mid r\rangle \leftrightarrow [t]_1^4 = 1,$$

we can use $[t]_1^2 =_* 0$ and $[t]_1^4 =_* 1$ as the characteristic terms of its two variants which are much simpler expressions than those from Par. 7.1.2.

**7.1.5 Structural induction for binary trees.** The principle of structural induction for binary trees can be informally stated as follows. To prove by tree induction that a property holds for every binary tree it suffices to prove:

*Base case:* the property holds for the empty tree $\langle\rangle$.
*Induction step:* if the property holds for the subtrees $l, r$ then it holds also for the whole tree $\langle l \mid x \mid r\rangle$.

This is expressed formally in PA by

$$\vdash_{\mathrm{PA}} \varphi[\langle\rangle] \land \forall x \forall l \forall r(\varphi[l] \land \varphi[r] \to \varphi[\langle l \mid {}^{x} \mid r\rangle]) \to Bt(t) \to \varphi[t],$$

where $\varphi[t]$ is a formula of PA. The property is called the principle of *structural induction on the binary tree t for* $\varphi[t]$.

*Proof.* The principle of structural induction for binary trees is derived in PA as follows. Assume $\varphi[\langle\rangle]$ and $\forall x \forall l \forall r(\varphi[l] \land \varphi[r] \to \varphi[\langle l \mid {}^{x} \mid r\rangle])$ take any binary tree $t$ and prove that $\varphi[t]$ holds by complete induction on $t$. We consider two cases. If $t$ is the empty tree then the claim follows directly from the first assumption. Otherwise, $t$ is a non-empty tree of a form $\langle l \mid {}^{x} \mid r\rangle$ for some $x, l, r$. By 2.1.6(2) we have $l < \langle l \mid {}^{x} \mid r\rangle$ and $r < \langle l \mid {}^{x} \mid r\rangle$. By applying two IH's we get $\varphi[l]$ and $\varphi[r]$, and thus $\varphi[\langle l \mid {}^{x} \mid r\rangle]$ by the second assumption.

**7.1.6 Structural recursion on binary trees.** Structural induction over binary trees is used to prove properties of functions defined by the scheme of *structural recursion on binary trees.* In its simplest form, the operator of structural recursion introduces a function $f$ from two functions $g$ and $h$ satisfying

$$\begin{aligned} f(t,y) = {} & \mathbf{case} \\ & \quad t = \langle\rangle \Rightarrow g(y) \\ & \quad t = \langle l \mid {}^{x} \mid r\rangle \Rightarrow h\big(x,l,r,f(l,y),f(r,y),y\big) \\ & \quad \mathbf{otherwise} \Rightarrow 0 \\ & \mathbf{end}. \end{aligned}$$

Note that this is a recursive definition regular in the first argument with discrimination on the constructors of binary tree (output variables of the second variant are omitted).

The following identities form the clausal form of the above definition

$$\begin{aligned} & f(\langle\rangle,y) = g(y) \\ & f(\langle l \mid {}^{x} \mid r\rangle,y) = h\big(x,l,r,f(l,y),f(r,y),y\big). \end{aligned}$$

Note here that this is a typical example where we wish to use the default clauses – in this case

$$f(t,y) = 0 \leftarrow t \neq \langle\rangle \land \neg\exists x \exists l \exists r\, t = \langle l \mid {}^{x} \mid r\rangle,$$

in order not to clutter the definition. We do not care what value is yielded by the application $f(t,y)$ if $t$ is not the code of a binary tree.

The above definition for the function $f$ can be easily rewritten to a conditional program for the same function as we have

$$\begin{aligned} \vdash_{\mathrm{PA}} Bt(t) \to f(t,y) = {} & \mathbf{case} \\ & \quad t = \langle\rangle \Rightarrow g(y) \\ & \quad t = \langle l \mid {}^{x} \mid r\rangle \Rightarrow h\big(x,l,r,f(l,y),f(r,y),y\big) \\ & \mathbf{end} \end{aligned}$$

Its conditions of regularity

$$\vdash_{\text{PA}} Bt(t) \land t = \langle l \mid x \mid r\rangle \to l < t \land Bt(l)$$
$$\vdash_{\text{PA}} Bt(t) \land t = \langle l \mid x \mid r\rangle \to r < t \land Bt(r)$$

are trivially satisfied.

Similar schemes, when we allow terms with arbitrary number of parameters on the right-hand side of the above identities, substitution in parameters, or even nested recursive applications, will be also called definitions/programs by structural recursion on binary trees.

**7.1.7 Depth and size of binary trees.** The *depth* function $d(t)$ yields the length of the longest path from the root to a leaf in the binary tree $t$. The function is defined by *parameterless* structural recursion on the binary tree $t$ as a primitive recursive function:

$$d\,\langle\rangle = 0$$
$$d\,\langle l \mid x \mid r\rangle = \max\bigl(d(l), d(r)\bigr) + 1.$$

The *size* function $|t|$ counts the number of labels in the binary tree $t$. The function is defined by parameterless structural recursion on the binary tree $t$ as a primitive recursive function:

$$|\langle\rangle| = 0$$
$$|\langle l \mid x \mid r\rangle| = |l| + |r| + 1.$$

The next property relates the number of labels in a binary tree to its depth:

$$\vdash_{\text{PA}} Bt(t) \to d(t) \leq |t| < 2^{d(t)}. \tag{1}$$

There are trees for which the second inequality is tight, i.e. $|t| + 1 = 2^{d(t)}$. Such trees are called *full binary trees.*

*Proof.* The property is proved by structural induction on the binary tree $t$. The base case is obvious. The induction step when $t = \langle l \mid x \mid r\rangle$ follows from

$$\begin{aligned} d\,\langle l \mid x \mid r\rangle &= \max(d(l), d(r)) + 1 \overset{\text{IH}}{\leq} \max(|l|, |r|) + 1 \leq \\ &\leq |l| + |r| + 1 = |\langle l \mid x \mid r\rangle| \end{aligned}$$

and

$$\begin{aligned} |\langle l \mid x \mid r\rangle| &= |l| + |r| + 1 \overset{\text{IH}}{<} 2^{d(l)} + 2^{d(r)} \leq 2 \times 2^{\max(d(l),d(r))} = \\ &= 2^{\max(d(l),d(r))+1} = 2^{d\,\langle l|x|r\rangle}. \end{aligned}$$ □

**7.1.8 Membership in binary trees.** The predicate $x \in t$ holds if $x$ is a label of the binary tree $t$. The predicate is defined by structural recursion on the binary tree $t$ as a primitive recursive predicate:

$$\begin{aligned}&x \in \langle l \mid {}^{y} \mid r\rangle \leftarrow x = y\\ &x \in \langle l \mid {}^{y} \mid r\rangle \leftarrow x \neq y \land x \in l\\ &x \in \langle l \mid {}^{y} \mid r\rangle \leftarrow x \neq y \land x \notin l \land x \in r.\end{aligned}$$

The following are the basic properties of the tree membership predicate:

$$\begin{aligned}&\vdash_{\mathrm{PA}} \; x \notin \langle\rangle\\ &\vdash_{\mathrm{PA}} \; x \in \langle l \mid {}^{y} \mid r\rangle \leftrightarrow x = y \lor x \in l \lor x \in r.\end{aligned}$$

From the property 2.1.6(2) of the pairing function we get $\vdash_{\mathrm{PA}} \; x \in t \rightarrow x < t$. Consequently, the universal quantifier $\forall x$ in a context like $\forall x(x \in \ldots \rightarrow \cdots)$ can be bounded. Similarly for existential quantifiers.

**7.1.9 Subtree relation.** Given the binary trees $t_1$ and $t_2$, the predicate $t_1 \trianglelefteq t_2$ holds if the tree $t_1$ is a subtree of the tree $t_2$. The predicate is defined by structural recursion on the binary tree $t_2$ as primitive recursive by

$$\begin{aligned}&t_1 \trianglelefteq t_2 \leftarrow t_1 = t_2\\ &t_1 \trianglelefteq t_2 \leftarrow t_1 \neq t_2 \land t_2 = \langle l_2 \mid {}^{x_2} \mid r_2\rangle \land t_1 \trianglelefteq l_2\\ &t_1 \trianglelefteq t_2 \leftarrow t_1 \neq t_2 \land t_2 = \langle l_2 \mid {}^{x_2} \mid r_2\rangle \land t_1 \not\trianglelefteq l_2 \land t_1 \trianglelefteq r_2.\end{aligned}$$

The following is the basic property of the subtree predicate:

$$\vdash_{\mathrm{PA}} \; t_1 \trianglelefteq t_2 \leftrightarrow t_1 = t_2 \lor \exists x_2 \exists l_2 \exists r_2\big(t_2 = \langle l_2 \mid {}^{x_2} \mid r_2\rangle \land (t_1 \trianglelefteq l_2 \lor t_1 \trianglelefteq r_2)\big). \quad (1)$$

As a straightforward consequence we obtain that

$$\begin{aligned}\vdash_{\mathrm{PA}} \; &\langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq \langle l_2 \mid {}^{x_2} \mid r_2\rangle \leftrightarrow x_1 = x_2 \land l_1 = l_2 \land r_2 = r_2 \lor\\ &\lor \langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq l_2 \lor \langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq r_2.\end{aligned} \quad (2)$$

Finally note that $\vdash_{\mathrm{PA}} \; t_1 \trianglelefteq t_2 \rightarrow t_1 \leq t_2$ by the property 2.1.6(2) of the pairing function. Therefore universal quantifiers in contexts like $\forall t_1(t_1 \trianglelefteq \ldots \rightarrow \cdots)$ can be bounded. Similarly for existential quantifiers.

**7.1.10 Subsorts of binary trees.** In the following sections we will study various kinds of binary trees where the sorted predicate $R(t)$ for each particular variety of binary trees has the following explicit definition

$$P(t) \leftrightarrow Bt(t) \land \forall x \forall l \forall r(\langle l \mid {}^{x} \mid r\rangle \trianglelefteq t \rightarrow \varphi[x, l, r]) \quad (1)$$

for a suitable $\varphi[x, l, r]$. For instance:

- perfectly size-balanced trees are defined by (1) with $\varphi \equiv |l| = |r|$;
- perfectly depth-balanced trees are defined by (1) with $\varphi \equiv d(l) = d(r)$.

The predicate $P$ defined by (1) has the following basic properties:

$$\vdash_{\mathrm{PA}} \; P\langle\rangle \quad (2)$$

$$\vdash_{\mathrm{PA}} \; P\langle l \mid {}^{x} \mid r\rangle \leftrightarrow \varphi[x, l, r] \land P(l) \land P(r). \quad (3)$$

*Proof.* (2): This is obvious. (3): It follows from

$$
\begin{aligned}
&P\langle l \mid {}^{x} \mid r\rangle \Leftrightarrow \\
&Bt\langle l \mid {}^{x} \mid r\rangle \land \forall x_1 \forall l_1 \forall r_1 \big(\langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq \langle l \mid {}^{x} \mid r\rangle \to \varphi[x_1, l_1, r_1]\big) \overset{7.1.9(2)}{\Leftrightarrow} \\
&Bt(l) \land Bt(r) \land \varphi[x, l, r] \land \forall x_1 \forall l_1 \forall r_1 \big(\langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq l \to \varphi[x_1, l_1, r_1]\big) \land \\
&\qquad \land \forall x_1 \forall l_1 \forall r_1 \big(\langle l_1 \mid {}^{x_1} \mid r_1\rangle \trianglelefteq r \to \varphi[x_1, l_1, r_1]\big) \Leftrightarrow \\
&\varphi[x, l, r] \land P(l) \land P(r). \qquad \square
\end{aligned}
$$

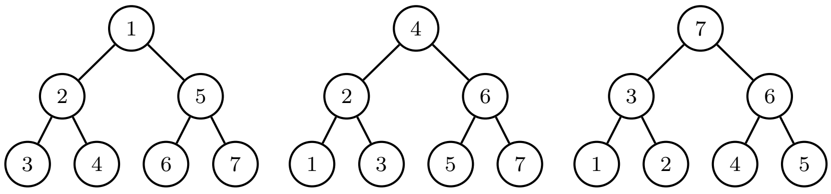

**Fig. 7.2** Three basic traversals of binary trees – preorder, inorder and postorder

**7.1.11 Depth-first traversal of binary trees.** Consider the function *Preorder*$(t)$ collecting the labels of a binary tree into a list in the order which corresponds to depth-first traversal of binary trees (see Fig. 7.2). The function is defined by structural recursion as primitive recursive by

$$
\begin{aligned}
&Preorder\,\langle\rangle = 0 \\
&Preorder\,\langle l \mid {}^{x} \mid r\rangle = \big\langle x, 0\big\rangle \oplus Preorder(l) \oplus Preorder(r).
\end{aligned}
$$

Note that the program runs in time $\mathcal{O}(|t|^2)$ due to repeated concatenation.

We obtain more efficient algorithm by keeping the labels of the visiting tree in an accumulator. For that we need a binary *accumulator* function $f(t, a)$ defined by *nested* structural recursion on the binary tree $t$:

$$
\begin{aligned}
&f(\langle\rangle, a) = a \\
&f(\langle l \mid {}^{x} \mid r\rangle, a) = \big\langle x, f\big(l, f(r, a)\big)\big\rangle
\end{aligned}
$$

Then we can take the following property

$$\vdash_{\mathrm{PA}} Bt(t) \to Preorder(t) = f(t, 0).$$

as an alternative (conditional) program of the preorder traversal function with time complexity $\mathcal{O}(|t|)$.

The property is a straightforward consequence of a more general property of the accumulator function:

$$\vdash_{\mathrm{PA}} Bt(t) \to \forall a\, f(t, a) = Preorder(t) \oplus a,$$

which is proved structural induction on the binary tree $t$. The base case is trivial. In the induction step when $t = \langle l \mid x \mid r\rangle$ take any $a$ and we get

$$f(\langle l \mid x \mid r\rangle, a) = \big\langle x, f\big(l, f(r,a)\big)\big\rangle \overset{\text{IH}}{=} \big\langle x, f(l, \mathit{Preorder}(r) \oplus a)\big\rangle \overset{\text{IH}}{=}$$
$$= \big\langle x, \mathit{Preorder}(l) \oplus \mathit{Preorder}(r) \oplus a\big\rangle = \mathit{Preorder}\,\langle l \mid x \mid r\rangle \oplus a.$$

Note that the second application of IH is with $\mathit{Preorder}(r) \oplus a$ in place of $a$.

## 7.2 Binary Search Trees

**7.2.1 Introduction.** In this section we will study *binary search trees* which are useful for representing finite sets of natural numbers. The time required to search for an element of a binary search tree $t$ takes the number of steps proportional to the depth of the tree $t$. If the tree is reasonably balanced then the time is order $\lg|t|$. The same holds for the basic operations over binary search trees such as insertion and deletion.

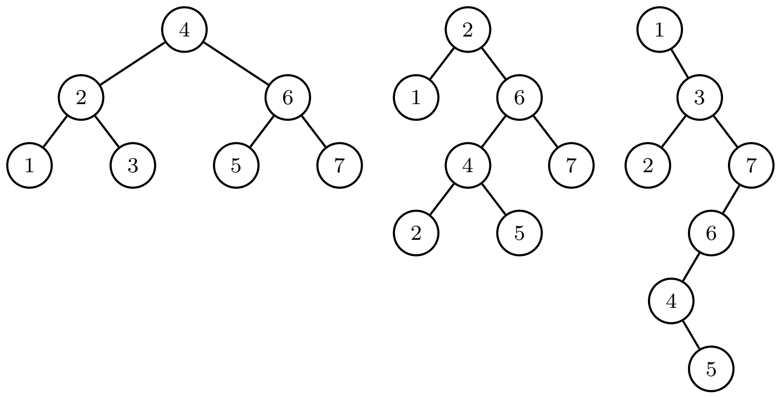

**Fig. 7.3** Examples of three different binary search trees representing the same finite set of natural numbers $\{1,2,3,4,5,6,7\}$

A binary search tree $t$ is a binary tree satisfying the following *search condition*:

for every non-empty subtree of $t$, its root label is strictly greater than the labels of its left son and strictly less than the labels of its right son.

Figure 7.3 shows three binary search trees representing the following three finite sets of natural numbers: $\{1,2,3\}$, $\{1,2,3,4\}$ and $\{1,2,\ldots,15\}$.

**7.2.2 Auxiliary specification predicates.** The predicate $x \prec t$ holds if $x$ is a *strict lower bound* of the labels of the binary tree $t$, i.e.

$$x \prec t \leftrightarrow \forall y(y \in t \to x < y).$$

The predicate $x \succ t$ holds if $x$ is a *strict upper bound* of the labels of the binary tree $t$, i.e.

$$x \succ t \leftrightarrow \forall y(y \in t \to x > y).$$

Later, in Par. 7.2.8, we will find useful the binary predicate $t_1 \prec_\mathrm{b} t_2$ holding if the labels of the binary tree $t_1$ are strictly less than the labels of the binary tree $t_2$. The predicate has the following explicit definition:

$$t_1 \prec_\mathrm{b} t_2 \leftrightarrow \forall x_1 \forall x_2(x_1 \in t_1 \land x_2 \in t_2 \to x_1 < x_2).$$

**7.2.3 Binary search trees.** The predicate $Bst(t)$ holding of binary search trees is defined explicitly as a primitive recursive predicate by

$$Bst(t) \leftrightarrow Bt(t) \land \forall x \forall l \forall r(\langle l \mid x \mid r\rangle \trianglelefteq t \to x \succ l \land x \prec r).$$

From the results of Par. 7.1.10 we obtain that the predicate satisfies

$$\vdash_{\mathrm{PA}} Bst\,\langle\rangle$$
$$\vdash_{\mathrm{PA}} Bst\,\langle l \mid x \mid r\rangle \leftrightarrow x \succ l \land x \prec r \land Bst(l) \land Bst(r).$$

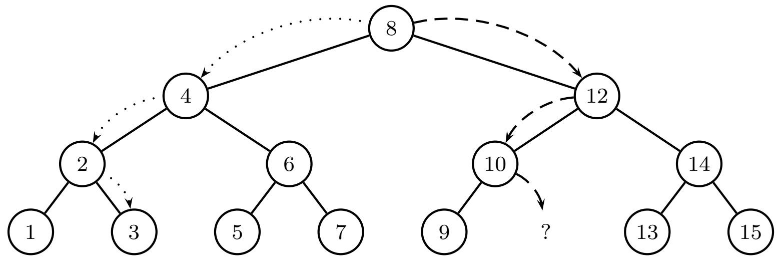

**Fig. 7.4** Testing for membership of numbers 3 (*dotted arrows*) and 11 (*dashed arrows*) in the binary search tree representing the finite set $\{1, \dots, 10\} \cup \{12, \dots, 15\}$

**7.2.4 Membership in binary search trees.** Testing for membership in binary search trees is simple (see Fig. 7.4). To determine whether a binary search tree $t$ contains a node labelled with $x$ it suffices to compare $x$ with the root label of $t$. If $x$ is smaller than the root label then it can only appear in the left subtree; if $x$ is greater then it appears in the right subtree. Otherwise they are equal and we are done. Note that the time to evaluate $x \in t$ in binary search trees is order $d(t)$.

The above reasoning can be expressed by the following property of its characteristic function:

```
Bst(t) → x ∈* t ↔ case
                   t = ⟨⟩ ⇒ 0
                   t = ⟨l | y | r⟩ ⇒
                     case
                       x < y ⇒ x ∈* l
                       x = y ⇒ 1
                       x > y ⇒ x ∈* r.
                     end
                   end
```

We can take the property as an alternative (conditional) program for computing the tree membership predicate. Its conditions of regularity

$$\vdash_{\mathrm{PA}} \; Bst(t) \land t = \langle l \mid x \mid r \rangle \land x < y \to l < t \land Bst(l)$$
$$\vdash_{\mathrm{PA}} \; Bst(t) \land t = \langle l \mid x \mid r \rangle \land x > y \to r < t \land Bst(r)$$

are trivially satisfied.

**7.2.5 Extreme values of binary search tree.** Now we consider the problem of computing extreme values of binary search trees. Because the labels of a binary search tree $t$ are sorted in increasing order the leftmost node contains the smallest label of the tree and the rightmost node contains the largest (see Fig. 7.5).

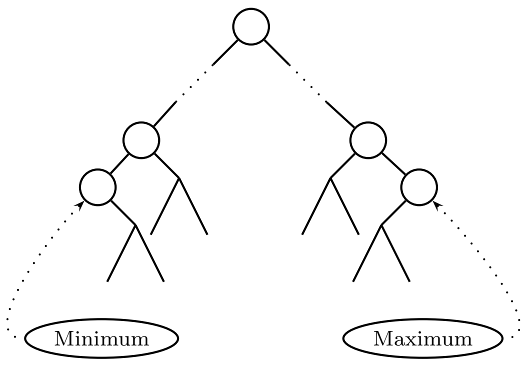

**Fig. 7.5** Extreme values of binary search trees

The problem is illustrated for the function $Max(t)$ computing the largest elements in binary search trees. The function satisfies

$$\vdash_{\mathrm{PA}} Bst(t) \wedge t \neq \langle\rangle \rightarrow Max(t) \in t \tag{1}$$
$$\vdash_{\mathrm{PA}} Bst(t) \wedge t \neq \langle\rangle \wedge x \in t \rightarrow x \leq Max(t) \tag{2}$$

and it is defined by structural recursion on binary trees as a p.r. function:

$Max\,\langle l \mid x \mid \langle\rangle\rangle = x$
$Max\,\langle l \mid x \mid r\rangle = Max(r) \leftarrow r \neq \langle\rangle.$

We intend to apply the operation $Max(t)$ only in cases when $t$ is a non-empty tree. For that we can take the following property as an alternative (conditional) program for computing the function:

```
⊢PA  $Bt(t) \wedge t \neq \langle\rangle \rightarrow Max(t) =$ let $t = \langle l \mid x \mid r\rangle$ in
                                      case
                                        $r = \langle\rangle \Rightarrow x$
                                        $r \neq \langle\rangle \Rightarrow Max(r)$
                                      end.
```

Its condition of regularity

$$\vdash_{\mathrm{PA}} Bt(t) \wedge t \neq \langle\rangle \wedge t = \langle l \mid x \mid r\rangle \wedge r \neq \langle\rangle \rightarrow r < t \wedge Bt(r) \wedge r \neq \langle\rangle$$

is trivially satisfied.

*Verification.* (1): This follows from

$$\vdash_{\mathrm{PA}} Bt(t) \wedge t \neq \langle\rangle \rightarrow Max(t) \in t$$

which can be proved by a straightforward structural induction on the binary tree $t$.

(2): By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step, when $t = \langle l \mid y \mid r\rangle$, assume $x \in \langle l \mid y \mid r\rangle$ and consider two cases. If $r = \langle\rangle$ then either $x = y$ or $x \in l$ and then $x < y$ by definition. In either case we have $x \leq y$ and the claim follows from definition since $Max\,\langle l \mid y \mid \langle\rangle\rangle = y$. If $r \neq \langle\rangle$ we continue by considering three subcases:

$$x = y \overset{(1)}{\Rightarrow} y < Max(r) = Max\,\langle l \mid y \mid r\rangle$$
$$x \in l \Rightarrow x < y \overset{(1)}{\Rightarrow} x < y < Max(r) = Max\,\langle l \mid y \mid r\rangle$$
$$x \in r \overset{\mathrm{IH}}{\Rightarrow} x \leq Max(r) = Max\,\langle l \mid y \mid r\rangle. \qquad \square$$

**7.2.6 Insertion in binary search trees.** The function $t \cup \{x\}$ takes a binary search tree $t$ and inserts $x$ into it (see Fig. 7.6). The function satisfies

$$\vdash_{\mathrm{PA}} Bst(t) \rightarrow Bst(t \cup \{x\}) \tag{1}$$
$$\vdash_{\mathrm{PA}} Bst(t) \rightarrow y \in t \cup \{x\} \leftrightarrow y \in t \vee y = x \tag{2}$$

and it is defined by structural recursion as a p.r. function:

$$\begin{aligned}
&\langle\rangle \cup \{x\} = \langle\langle\rangle \mid {}^{x} \mid \langle\rangle\rangle \\
&\langle l \mid {}^{y} \mid r\rangle \cup \{x\} = \langle l \mid {}^{y} \mid r \cup \{x\}\rangle \leftarrow y < x \\
&\langle l \mid {}^{y} \mid r\rangle \cup \{x\} = \langle l \mid {}^{y} \mid r\rangle \leftarrow y = x \\
&\langle l \mid {}^{y} \mid r\rangle \cup \{x\} = \langle l \cup \{x\} \mid {}^{y} \mid r\rangle \leftarrow y > x.
\end{aligned}$$

Note that the time to evaluate $t \cup \{x\}$ is order $d(t)$.

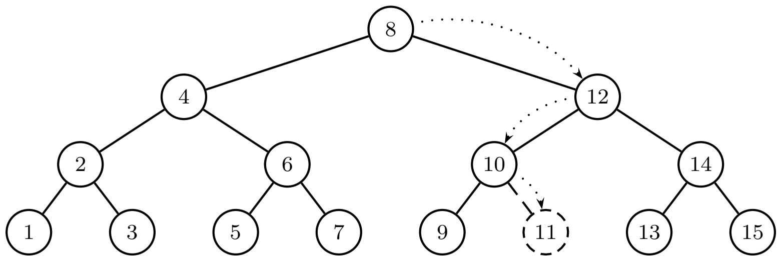

**Fig. 7.6** Insertion of the number 11 into the binary search tree representing the finite set of natural numbers $\{1, \ldots, 10\} \cup \{12, \ldots, 15\}$

*Verification.* (2): This follows from

$$\vdash_{\mathrm{PA}} Bt(t) \to y \in t \cup \{x\} \leftrightarrow y \in t \lor y = x, \qquad (\dagger_1)$$

which is proved by structural induction on the binary tree $t$. The base case is straightforward. In the induction step when $t = \langle l \mid {}^{z} \mid r\rangle$ we consider three cases. If $z < x$ then we have

$$\begin{aligned}
y \in \langle l \mid {}^{z} \mid r\rangle \cup \{x\} \Leftrightarrow y \in \langle l \mid {}^{z} \mid r \cup \{x\}\rangle \Leftrightarrow y = z \lor y \in l \lor y \in r \cup \{x\} \overset{\mathrm{IH}}{\Leftrightarrow} \\
\Leftrightarrow y = z \lor y \in l \lor y \in r \lor y = x \Leftrightarrow y \in \langle l \mid {}^{z} \mid r\rangle \lor y = x.
\end{aligned}$$

The case when $z = x$ is obvious and the case when $z > x$ is proved similarly. As a simple consequence of $(\dagger_1)$ we get

$$\vdash_{\mathrm{PA}} Bt(t) \to y \prec t \cup \{x\} \leftrightarrow y \prec t \land y < x \qquad (\dagger_2)$$

$$\vdash_{\mathrm{PA}} Bt(t) \to y \succ t \cup \{x\} \leftrightarrow y \succ t \land y > x. \qquad (\dagger_3)$$

(1): By structural induction on the binary tree $t$. The base case is straightforward. In the induction step when $t = \langle l \mid {}^{y} \mid r\rangle$ we consider three cases. If $y < x$ then we have

$$\begin{aligned}
Bst(\langle l \mid {}^{y} \mid r\rangle \cup \{x\}) \Leftrightarrow Bst\,\langle l \mid {}^{y} \mid r \cup \{x\}\rangle \Leftrightarrow \\
\Leftrightarrow y \succ l \land y \prec r \cup \{x\} \land Bst(l) \land Bst(r \cup \{x\}).
\end{aligned}$$

The last follows from IH and $(\dagger_2)$. The case when $y = x$ is trivial and the case when $y > x$ is proved similarly. □

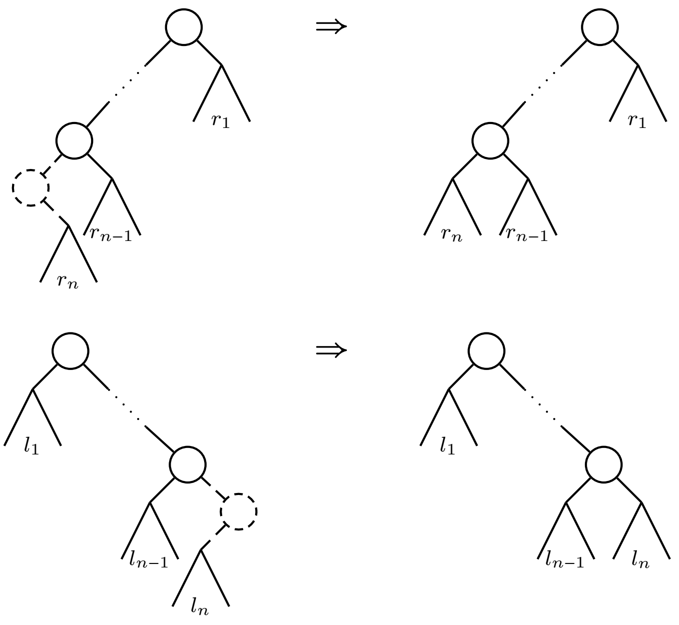

**Fig. 7.7** Deletion of extreme values in binary search trees

**7.2.7 Deletion of extreme values in binary search trees.** In this paragraph we will consider the problem of deletion of extremal values from binary search trees (see Fig. 7.7). We show here only the implementation and verification of the function deleting the gretest label in a binary search tree. We leave to the reader the implementation and verification of the function deleting the smallest label.

The function $Delmax(t)$ deleting the largest label in the binary search tree $t$ satisfies

$$\vdash_{\mathrm{PA}} Bst(t) \land t \neq \langle\rangle \to Bst\,Delmax(t) \tag{1}$$

$$\vdash_{\mathrm{PA}} Bst(t) \land t \neq \langle\rangle \to x \in Delmax(t) \leftrightarrow x \in t \land x \neq Max(t) \tag{2}$$

and it is defined by structural recursion as a p.r. function (see Fig. 7.7):

$$Delmax\,\langle l \mid x \mid \langle\rangle\rangle = l$$
$$Delmax\,\langle l \mid x \mid r\rangle = \langle l \mid x \mid Delmax(r)\rangle \leftarrow r \neq \langle\rangle.$$

We intend to apply the operation $Delmax(t)$ only in cases when $t$ is a non-empty tree. For that we can take the following property as an alternative (conditional) program for computing the function:

$$\vdash_{\mathrm{PA}} Bt(t) \land t \neq \langle\rangle \to Delmax(t) = \textbf{let } t = \langle l \mid x \mid r\rangle \textbf{ in}$$
$$\begin{aligned}&\textbf{case}\\ &\quad r = \langle\rangle \Rightarrow l\\ &\quad r \neq \langle\rangle \Rightarrow \langle l \mid x \mid Delmax(r)\rangle\\ &\textbf{end}.\end{aligned}$$

Its condition of regularity

$$\vdash_{\mathrm{PA}} Bt(t) \land t \neq \langle\rangle \land t = \langle l \mid x \mid r\rangle \land r \neq \langle\rangle \to r < t \land Bt(r) \land r \neq \langle\rangle$$

is trivially satisfied.

Note also that as a simple consequence of 7.2.5(2) and (2) we have

$$\vdash_{\mathrm{PA}} Bst(t) \land t \neq \langle\rangle \to Max(t) \succ Delmax(t). \tag{3}$$

*Verification.* (2) By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid y \mid r\rangle$ we consider two cases. If $r = \langle\rangle$ then we get the following by noting that $y \notin l$:

$$\begin{aligned}x \in Delmax\,\langle l \mid y \mid \langle\rangle\rangle \Leftrightarrow x \in l \Leftrightarrow (x = y \lor x \in l) \land x \neq y \Leftrightarrow&\\ \Leftrightarrow x \in \langle l \mid y \mid \langle\rangle\rangle \land x \neq Max\,\langle l \mid y \mid \langle\rangle\rangle.&\end{aligned}$$

If $r \neq \langle\rangle$ then we have

$$\begin{aligned}&x \in Delmax\,\langle l \mid y \mid r\rangle \Leftrightarrow x \in \langle l \mid y \mid Delmax(r)\rangle \Leftrightarrow\\ &x = y \lor x \in l \lor x \in Delmax(r) \overset{\text{IH}}{\Leftrightarrow} x = y \lor x \in l \lor x \in r \land x \neq Max(r) \overset{(*)}{\Leftrightarrow}\\ &(x = y \lor x \in l \lor x \in r) \land x \neq Max(r) \Leftrightarrow x \in \langle l \mid y \mid r\rangle \land x \neq Max\,\langle l \mid y \mid r\rangle.\end{aligned}$$

The step marked by $(*)$ follows from $Max(r) \neq y$ and $Max(r) \notin l$ by 7.2.5(1).

As a simple consequence of (2) we get

$$\vdash_{\mathrm{PA}} Bst(t) \land t \neq \langle\rangle \land x \prec t \to x \prec Delmax(t)\ . \tag{$\dagger_1$}$$

(1): By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ we consider two cases. If $r = \langle\rangle$ then the claim follows directly from the definition. If $r \neq \langle\rangle$ then we have

$$\begin{aligned}&Bst\,Delmax\,\langle l \mid x \mid r\rangle \Leftrightarrow Bst\,\langle l \mid x \mid Delmax(r)\rangle \Leftrightarrow\\ &\qquad \Leftrightarrow x \succ l \land x \prec Delmax(r) \land Bst(l) \land Bst\,Delmax(r).\end{aligned}$$

The last follows from IH and $(\dagger_1)$. □

**7.2.8 Deletion in binary search trees.** Deletion of labels from binary search trees is much harder than insertion. We wish to define the function $t \setminus \{x\}$ deleting a number $x$ from a binary search tree $t$ with the following specification:

$$\vdash_{\mathrm{PA}} \; Bst(t) \rightarrow Bst(t \setminus \{x\}) \tag{1}$$
$$\vdash_{\mathrm{PA}} \; Bst(t) \rightarrow y \in t \setminus \{x\} \leftrightarrow y \in t \wedge y \neq x. \tag{2}$$

The key problem in finding the implementation of the deletion function is: how to define $t \setminus \{x\}$ when $x$ is the root label of $t$, i.e. when $t = \langle l \mid {}^{x} \mid r \rangle$ for some $l$ and $r$. Figure 7.8 gives two answers to the problem.

In the first solution the right son $r$ is appended as a new subtree at the bottom right to the left son $l$. Note that it may happen that the depth of the resulting tree is greater than the depth of $t$.

In the second solution we take the largest label of the left son $l$ as a new root label with the left son obtained from $l$ by deleting its maximal element and with $r$ as its right son. The result is a tree which depth does not exceed the depth of the original tree.

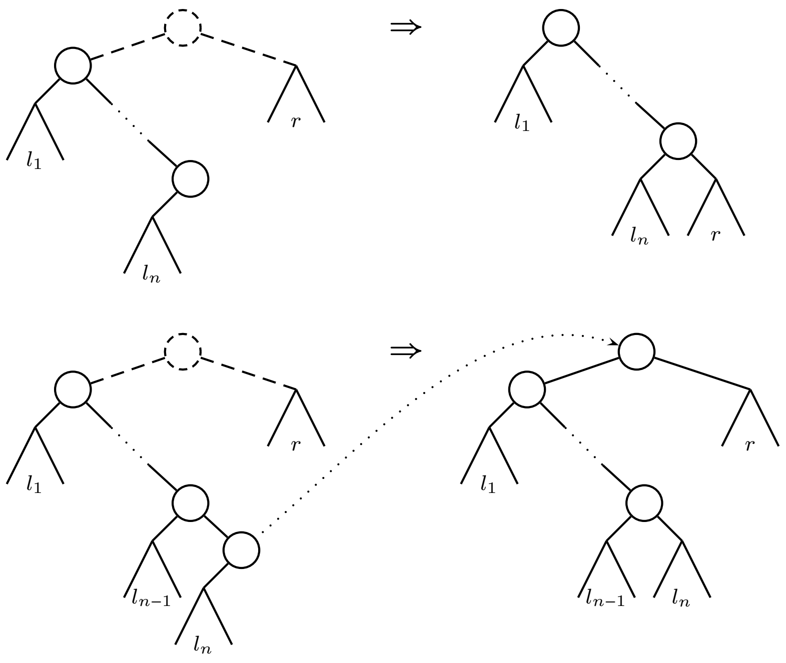

**Fig. 7.8** Deletion operation in binary search trees

We therefore take the second method as a basis of our implementation of the deletion function. We define $t \setminus \{x\}$ by structural recursion on $t$ as a primitive recursive function:

$t_1 \sqcup t_2 = t_2 \leftarrow t_1 = \langle\rangle$
$t_1 \sqcup t_2 = \langle Delmax(t_1) \mid {}^{Max(t_1)} \mid t_2 \rangle \leftarrow t_1 \neq \langle\rangle.$

$$
\begin{aligned}
&\langle\rangle \smallsetminus \{x\} = \langle\rangle\\
&\langle l \mid {}^{y} \mid r\rangle \smallsetminus \{x\} = \langle l \mid {}^{y} \mid r \smallsetminus \{x\}\rangle \leftarrow y < x\\
&\langle l \mid {}^{y} \mid r\rangle \smallsetminus \{x\} = l \sqcup r \leftarrow y = x\\
&\langle l \mid {}^{y} \mid r\rangle \smallsetminus \{x\} = \langle l \smallsetminus \{x\} \mid {}^{y} \mid r\rangle \leftarrow y > x,
\end{aligned}
$$

where the auxiliary primitive recursive function $t_1 \sqcup t_2$ joining two trees as shown in Fig. 7.8 satisfies

$$\vdash_{\text{PA}} \mathit{Bst}(t_1) \land \mathit{Bst}(t_2) \land t_1 <_{\text{b}} t_2 \to \mathit{Bst}(t_1 \sqcup t_2) \tag{3}$$

$$\vdash_{\text{PA}} \mathit{Bst}(t_1) \land \mathit{Bst}(t_2) \to x \in t_1 \sqcup t_2 \leftrightarrow x \in t_1 \lor x \in t_2. \tag{4}$$

*Verification.* Property (4) is proved by considering two cases. The case when $t_1 = \langle\rangle$ is trivial and in the case when $t_1 \neq \langle\rangle$ we have

$$
\begin{aligned}
&x \in t_1 \sqcup t_2 \Leftrightarrow x \in \langle \mathit{Delmax}(t_1) \mid {}^{\mathit{Max}(t_1)} \mid t_2\rangle \Leftrightarrow\\
&x = \mathit{Max}(t_1) \lor x \in \mathit{Delmax}(t_1) \lor x \in t_2 \overset{7.2.7(2)}{\Leftrightarrow}\\
&x = \mathit{Max}(t_1) \lor x \in t_1 \land x \neq \mathit{Max}(t_1) \lor x \in t_2 \overset{(*)}{\Leftrightarrow} x \in t_1 \lor x \in t_2.
\end{aligned}
$$

The step marked by $(*)$ follows from $\mathit{Max}(t_1) \in t_1$ which holds by 7.2.5(1). In the proof of the property (3) we consider two cases. The case when $t_1 = \langle\rangle$ is trivial and in the case when $t_1 \neq \langle\rangle$ we have

$$
\begin{aligned}
&\mathit{Bst}(t_1 \sqcup t_2) \Leftrightarrow \mathit{Bst}\,\langle \mathit{Delmax}(t_1) \mid {}^{\mathit{Max}(t_1)} \mid t_2\rangle \Leftrightarrow\\
&\mathit{Max}(t_1) \succ \mathit{Delmax}(t_1) \land \mathit{Max}(t_1) \prec t_2 \land \mathit{Bst}\,\mathit{Delmax}(t_1) \land \mathit{Bst}(t_2).
\end{aligned}
$$

The last follows from 7.2.7(3), 7.2.5(1), and 7.2.7(1).

Property (2) is proved by structural induction on $t$. The base case is straightforward. In the induction step when $t = \langle l \mid {}^{z} \mid r\rangle$ we consider three cases. If $z < x$ then we have

$$
\begin{aligned}
&y \in \langle l \mid {}^{z} \mid r\rangle \smallsetminus \{x\} \Leftrightarrow y \in \langle l \mid {}^{z} \mid r \smallsetminus \{x\}\rangle \Leftrightarrow y = z \lor y \in l \lor y \in r \smallsetminus \{x\} \overset{\text{IH}}{\Leftrightarrow}\\
&y = z \lor y \in l \lor y \in r \land y \neq x \overset{(*)}{\Leftrightarrow} (y = z \lor y \in l \lor y \in r) \land y \neq x \Leftrightarrow\\
&y \in \langle l \mid {}^{z} \mid r\rangle \land y \neq x.
\end{aligned}
$$

The step marked by $(*)$ follows by a simple case analysis on $y = x$ and $y \neq x$ by noting that we have $x \notin l$. The case when $z = x$ follows from (4) by similar arguments. The case when $z > x$ is left to the reader.

As a simple consequence of (2) we get

$$\vdash_{\text{PA}} \mathit{Bst}(t) \land y < x \to y \prec t \smallsetminus \{x\} \leftrightarrow y \prec t \tag{5}$$

$$\vdash_{\text{PA}} \mathit{Bst}(t) \land y > x \to y \succ t \smallsetminus \{x\} \leftrightarrow y \succ t. \tag{6}$$

Property (1) is proved by structural induction on $t$. The base case is straightforward. In the induction step when $t = \langle l \mid {}^{y} \mid r\rangle$ we consider three cases. If $y < x$ then we have

$$Bst(\langle l \mid y \mid r\rangle \smallsetminus \{x\}) \Leftrightarrow Bst\,\langle l \mid y \mid r \smallsetminus \{x\}\rangle \Leftrightarrow$$
$$y \succ l \land y \prec r \smallsetminus \{x\} \land Bst(l) \land Bst(r \smallsetminus \{x\}).$$

The last follows from IH and (5). If $y = x$ then the claim follows from (3). The case when $y > x$ is proved similarly. □

## 7.3 Braun Trees and Flexible Arrays

**7.3.1 Introduction.** As the second illustration of the use of binary trees, we will consider the problem of efficient implementations of flexible arrays. Arrays are implementations of sequences where the two operations, indexing and updating, are efficient. Flexible arrays are arrays with efficient implementation of the operations accessing first and last elements. Flexible arrays are very suitable for representing heaps and priority queues.

In this section we will consider one implementation of flexible arrays - Braun trees [5, 34]. For a Braun tree the following holds for every its subtree:

the size of its left subtree is the same as the size of its right subtree or one node larger.

Consequently Braun trees are optimal in size, that is the depth of a Braun tree is logarithmic in its size. As we will see below the complexity of the most operations operating on Braun trees are proportional to their depths, the cost of each operation is logarithmic to the tree-size.

This section is organized as follows. We begin with an alternative definition of Braun trees which is based on dyadic indexing scheme and show that definitions coincide. Then we prove that the size of Braun is optimal and then we give an $\mathcal{O}(\lg^2(n))$ algorithm to compute the size of Braun trees. Finally we describe basic operations on Braun trees such as indexing, updating, inserting and deleting an element on one of the sides of a tree.

**7.3.2 Dyadic indexing scheme.** We start by describing the indexing scheme for binary trees. In our scheme, the index of a node is a description of the path leading from the root to that particular node. On the other hand each such path can be easily represented by a dyadic word which is over two elements alphabet $\Sigma = \{1, 2\}$ as it is shown in Fig. 7.9. Here the symbol 1 stands for the way to the left and 2 for the way to the right in trees.

For instance, consider the path leading from the root to the node labelled with the number 29. The path can be identified with the dyadic string 2221. Similarly, the string 2122 represents the path to the node labelled with the number 26. Note also that the empty string represents the trivial path, from the root to the root itself.

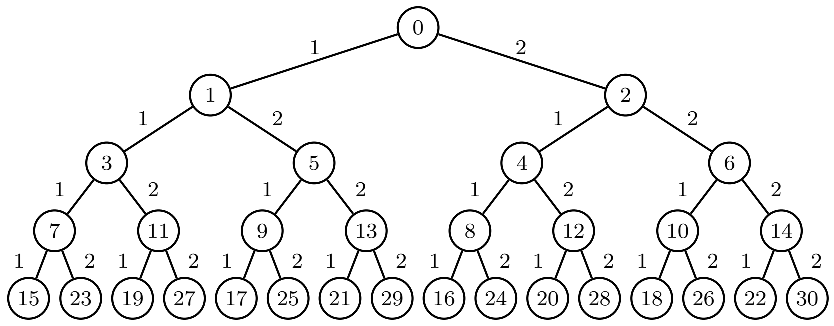

**Fig. 7.9** Dyadic indexing scheme for binary trees

Recall that there is a simple method of coding of the words over the two-symbol alphabet $\Sigma$ into natural numbers. Arithmetization is based on dyadic representation of numbers as described in Par. 1.1.5; each number has a unique representation as a dyadic numeral which are terms built up from 0 by applications of dyadic successors $x\mathbf{1} = 2x + 1$ and $x\mathbf{2} = 2x + 2$.

In our example, the code of the string 2221 is the number

$$\begin{aligned} 0\mathbf{2221} &= 2 \times (2 \times (2 \times (2 \times 0 + 2) + 2) + 2) + 1 = \\ &= 2 \times 2^3 + 2 \times 2^2 + 2 \times 2^1 + 1 \times 2^0 = 29. \end{aligned}$$

Similarly, the code of the string 2122 is the number

$$\begin{aligned} 0\mathbf{2122} &= 2 \times (2 \times (2 \times (2 \times 0 + 2) + 2) + 1) + 2 = \\ &= 2 \times 2^3 + 1 \times 2^2 + 2 \times 2^1 + 2 \times 2^0 = 26. \end{aligned}$$

A careful inspection reveals that the binary tree shown in Fig. 7.9 consists of nodes labelled by their indices. Note also that the root node is indexed by the number 0, the nodes in the left subtree are indexed by odd numbers, and the nodes in the right subtree are indexed by positive even numbers.

As we have mentioned above our indexing scheme is based on dyadic representation of numbers. Consequently, dyadic recursion and dyadic induction will play prominent role in the implementation and verification of the basic operations such as indexing and updating.

**7.3.3 Valid indices.** The predicate $\mathit{Index}(i, t)$ holds if the $i$ is a valid index of the binary tree $t$. The predicate is defined by dyadic recursion on $i$ with substitution in parameter:

$$\begin{aligned} &\mathit{Index}(0, \langle l \mid {}^{x} \mid r\rangle) \\ &\mathit{Index}(i\mathbf{1}, \langle l \mid {}^{x} \mid r\rangle) \leftarrow \mathit{Index}(i, l) \\ &\mathit{Index}(i\mathbf{2}, \langle l \mid {}^{x} \mid r\rangle) \leftarrow \mathit{Index}(i, r). \end{aligned}$$

Note that the empty tree does not have valid indices.

Valid indices of a binary tree cannot be arbitrary large numbers since we have the following simple bound for them:

$$\vdash_{\mathrm{PA}} \; Bt(t) \wedge Index(i,t) \to i + 2 \leq 2^{d(t)}. \tag{1}$$

Note also that since a binary tree $t$ has exactly $|t|$ nodes there are only $|t|$ valid indices for that tree. Consequently each selection of $|t| + 1$ numbers contains at least one number which is not a valid index of $t$. This cannot be stated in PA directly, not without the formalization of finite sets within PA, but we can state

$$\vdash_{\mathrm{PA}} \; Bt(t) \to \exists i\big(i \leq |t| \wedge \neg Index(i,t)\big). \tag{2}$$

We will find this claim very useful latter when dealing with alternative definition of Braun trees.

*Proof.* (1): By structural induction on the binary tree $t$ as $\forall i(1)$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ take any $i$ such that $Index(i, \langle l \mid x \mid r\rangle)$ and consider three cases. The case when $i = 0$ is obvious. The case when $i = j\mathbf{2}$ for some $j$ follows from

$$j\mathbf{2} + 2 = 2j + 2 + 2 = 2(j+2) \overset{\mathrm{IH}}{\leq} 2 \times 2^{d(l)} \leq 2 \times 2^{\max(d(l),d(r))} = 2^{d\langle l|x|r\rangle}.$$

Note that the induction hypothesis is applied with $j$ in place of $i$. The last case when $i = j\mathbf{1}$ for some $j$ has a similar proof.

(2): By structural induction on the binary tree $t$. In the base case it suffices to take $i := 0$. In the induction step when $t = \langle l \mid x \mid r\rangle$ we consider two subcases. If $|l| \leq |r|$ then by IH there is a number $j \leq |l|$ such that $\neg Index(j,l)$. We have $\neg Index(j\mathbf{1}, \langle l \mid x \mid r\rangle)$ and

$$j\mathbf{1} = 2j + 1 \leq 2\,|l| + 1 \leq |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.$$

It suffices to take $i := j\mathbf{1}$. Suppose now $|l| > |r|$, ie $|r| + 1 \leq |l|$. By IH there is a number $j \leq |r|$ such that $\neg Index(j,r)$. We have $\neg Index(j\mathbf{2}, \langle l \mid x \mid r\rangle)$ and

$$j\mathbf{2} = 2j + 2 \leq 2\,|r| + 2 = |r| + 1 + |r| + 1 \leq |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.$$

Now it suffices to take $i := j\mathbf{2}$. □

**7.3.4 Braun trees.** A binary tree is called a Braun tree if its valid indices are exactly the numbers strictly lesser than the size of the tree. The predicate $Brt(t)$ holding of Braun trees is explicitly defined by

$$Brt(t) \leftrightarrow Bt(t) \wedge \forall i(Index(i,t) \leftrightarrow i < |t|).$$

Note that by 7.1.7(1) and 7.3.3(1) the variable $i$ in the universal quantifier can be bounded by the number $2^{d(t)}$. Thus the predicate *Brt* is primitive recursive.

**7.3.5 Basic properties of Braun trees.** Braun trees satisfy the following conditions:

$$\vdash_{\mathrm{PA}} \mathit{Brt}\,\langle\rangle \tag{1}$$

$$\vdash_{\mathrm{PA}} \mathit{Brt}\,\langle l \mid x \mid r\rangle \leftrightarrow (|l| = |r| \lor |l| = |r| + 1) \land \mathit{Brt}(l) \land \mathit{Brt}(r). \tag{2}$$

In other words, for any non-empty Braun tree, its left subtree is either exactly the same size as its right subtree, or one node larger.

*Proof.* Property (1) is straightforward. In the proof of the property (2) we consider two cases. If either $|l| = |r|$ or $|l| = |r| + 1$ then we have

$$\begin{aligned}
&\mathit{Brt}\,\langle l \mid x \mid r\rangle \Leftrightarrow \mathit{Bt}\,\langle l \mid x \mid r\rangle \land \forall i(\mathit{Index}(i, \langle l \mid x \mid r\rangle) \leftrightarrow i < |\langle l \mid x \mid r\rangle|) \Leftrightarrow \\
&\mathit{Bt}(l) \land \mathit{Bt}(r) \land \forall i(\mathit{Index}(i, \langle l \mid x \mid r\rangle) \leftrightarrow i < |l| + |r| + 1) \overset{(*)}{\Leftrightarrow} \\
&\ \mathit{Bt}(l) \land \forall j(\mathit{Index}(j\mathbf{1}, \langle l \mid x \mid r\rangle) \leftrightarrow j\mathbf{1} < |l| + |r| + 1) \land \\
&\qquad\quad \land \mathit{Bt}(r) \land \forall j(\mathit{Index}(j\mathbf{2}, \langle l \mid x \mid r\rangle) \leftrightarrow j\mathbf{2} < |l| + |r| + 1) \Leftrightarrow \\
&\ \mathit{Bt}(l) \land \forall j(\mathit{Index}(j, l) \leftrightarrow 2j < |l| + |r|) \land \\
&\qquad\quad \land \mathit{Bt}(r) \land \forall j(\mathit{Index}(j, r) \leftrightarrow 2j + 1 < |l| + |r|) \Leftrightarrow \\
&\mathit{Bt}(l) \land \forall j(\mathit{Index}(j, l) \leftrightarrow j < |l|) \land \mathit{Bt}(r) \land \forall j(\mathit{Index}(j, r) \leftrightarrow j < |r|) \Leftrightarrow \\
&\mathit{Brt}(l) \land \mathit{Brt}(r).
\end{aligned}$$

The step marked by $(*)$ is by dyadic case analysis on $i$. The case when neither $|l| = |r|$ nor $|l| = |r| + 1$ is a straightforward consequence of the assertion

$$\vdash_{\mathrm{PA}} \mathit{Brt}\,\langle l \mid x \mid r\rangle \rightarrow (|l| = |r| \lor |l| = |r| + 1),$$

which follows from 7.3.3(2). □

**7.3.6 Alternative definition of Braun trees.** The predicate $\mathit{Brt}(t)$ holds iff $t$ is a binary tree satisfying the following *(size) balanced condition*:

for every non-empty subtree of $t$ the size of its left subtree is at most one more than the size of its right subtree.

Braun trees satisfy

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \leftrightarrow \mathit{Bt}(t) \land \forall x \forall l \forall r(\langle l \mid x \mid r\rangle \trianglelefteq t \rightarrow |l| = |r| \lor |l| = |r| + 1).$$

The property follows from 7.3.5(1)(2) and from the results of Par. 7.1.10.

**7.3.7 Braun trees are optimal.** The depth of a Braun tree is always minimum; that is, we have

$$\vdash_{\mathrm{PA}} Brt(t) \to 2^{d(t)} \div 2 \leq |t| < 2^{d(t)}. \tag{1}$$

Consequently the depth of Braun trees is logarithmic in their size.

*Proof.* The sharp inequality holds for arbitrary binary trees; this has been already proved in 7.1.7(1). The non-sharp inequality, i.e.

$$\vdash_{\mathrm{PA}} Brt(t) \to 2^{d(t)} \div 2 \leq |t|\,,$$

is proved by structural induction on the binary tree $t$ as follows. The base case is trivial. In the induction step take any non-empty Braun tree of the form $\langle l \mid x \mid r\rangle$. By 7.3.5(2) we know that $l$ and $r$ are Braun trees satisfying

$$|l| = |r| \lor |l| = |r| + 1.$$

We wish to prove

$$2^{d\langle l|x|r\rangle} \div 2 \leq |\langle l \mid x \mid r\rangle|\,.$$

First note that we have

$$2^{d\langle l|x|r\rangle} \div 2 = 2^{\max(d(l),d(r))+1} \div 2 = 2^{\max(d(l),d(r))} = \max(2^{d(l)}, 2^{d(r)}).$$

Now we consider three cases.

Suppose that both subtrees $l$ and $r$ are non-empty. Then $d(l) \neq 0 \neq d(r)$ and thus by IH we have $2^{d(l)} \leq 2\,|l|$ and $2^{d(r)} \leq 2\,|r|$. We obtain

$$\max(2^{d(l)}, 2^{d(r)}) \leq \max(2\,|l|\,, 2\,|r|) = 2\max(|l|\,, |r|) = 2\,|l|\,.$$

We consider two subcases. If $|l| = |r|$ then the desired property follows from

$$2\,|l| = |l| + |r| < |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.$$

Otherwise $|l| = |r| + 1$ and we have

$$2\,|l| = |l| + |l| = |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.$$

This concludes the proof for the case $l \neq \langle\rangle \neq r$.

If one of the intermediate subtrees of $\langle l \mid x \mid r\rangle$ is empty then by 7.3.5(2) it must the case that the right subtree $r$ is empty and that the left subtree $l$ is either empty or singleton. In either case it is easy to see that the desired inequality holds. □

**7.3.8 Quick computation of the size of Braun trees.** The recurrent property 7.3.5(2) of Braun trees allows us to quickly calculate their size. Consider a non-empty Braun tree of the form $\langle l \mid x \mid r\rangle$. By 7.3.5(2) we have

$$|l| = |r| \lor |l| = |r| + 1.$$

So if the size of the right tree is $n$ then the size of its left tree is either $n$ or $n + 1$. From definition of Braun trees we know that

$$\begin{aligned} |l| = |r| &\to \neg Index(|r|, l) \\ |l| = |r| + 1 &\to Index(|r|, l) \end{aligned}$$

and thus

$$\begin{aligned} |l| = |r| &\leftrightarrow \neg Index(|r|, l) \\ |l| = |r| + 1 &\leftrightarrow Index(|r|, l). \end{aligned}$$

So either $n$ is a proper index of the tree $l$ and then $|l| = n + 1$ or not and then $|l| = n$.

Consequently, the size of Braun trees can be implemented by the following conditional algorithm:

```
$\vdash_{PA}$ $Brt(t) \to |t| =$ **case**
                 $t = \langle\rangle \Rightarrow 0$
                 $t = \langle l \mid x \mid r\rangle \Rightarrow$ **let** $|r| = n$ **in**
                   **case**
                     $Index(n, l) \Rightarrow 2n + 2$
                     $\neg Index(n, l) \Rightarrow 2n + 1$
                   **end**
               **end**.
```

Since the complexity of the call $Index(n, t)$ is linear in the depth of $t$, the above algorithm has time complexity $\mathcal{O}\big(d(t)^2\big)$.

**7.3.9 Remark.** In the following paragraphs we will study operations over Braun trees involving one element (first or last). The complexity of these operations is proportional to the depth, i.e. logarithmic in the size. We will tacitly use the following simple properties of Braun trees:

$$\begin{aligned} &\vdash_{PA} Brt\,\langle l \mid x \mid r\rangle \to i\mathbf{1} < |\langle l \mid x \mid r\rangle| \leftrightarrow i < |l| \\ &\vdash_{PA} Brt\,\langle l \mid x \mid r\rangle \to i\mathbf{2} < |\langle l \mid x \mid r\rangle| \leftrightarrow i < |r| \end{aligned}$$

**7.3.10 Indexing operation.** We start with the definition of the indexing operation since this function will play crucial role in specification of the remaining operations. The application $t\,[i]$ yields the label indexed by $i$ in the Braun tree $t$. The function is defined by dyadic recursion on $i$ with substitution in parameter as a primitive recursive function:

$$\begin{aligned} &\langle l \mid x \mid r\rangle\,[0] = x \\ &\langle l \mid x \mid r\rangle\,[i\mathbf{1}] = l\,[i] \\ &\langle l \mid x \mid r\rangle\,[i\mathbf{2}] = r\,[i]. \end{aligned}$$

Note the missing clause for the case when the first argument is the empty tree; the result is 0 by default.

We usually intend to apply the operation $t\,[i]$ only in cases when $t$ is a non-empty Braun tree and $i$ is its valid index, i.e. $i < |t|$. For that we can take the following property as an alternative (conditional) program for computing the indexing function:

$$\begin{aligned}\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \to t\,[i] = \mathbf{case}\ & \\ & i = 0 \Rightarrow \mathbf{let}\ t = \langle l \mid x \mid r\rangle\ \mathbf{in}\ x \\ & i = j\mathbf{1} \Rightarrow \mathbf{let}\ t = \langle l \mid x \mid r\rangle\ \mathbf{in}\ l\,[j] \\ & i = j\mathbf{2} \Rightarrow \mathbf{let}\ t = \langle l \mid x \mid r\rangle\ \mathbf{in}\ r\,[j] \\ \mathbf{end}.&\end{aligned}$$

Its conditions of regularity

$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \land i = j\mathbf{1} \land t = \langle l \mid x \mid r\rangle \to l < t \land Brt(l) \land j < |l|$$
$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \land i = j\mathbf{2} \land t = \langle l \mid x \mid r\rangle \to r < t \land Brt(r) \land j < |r|$$

are trivially satisfied.

**7.3.11 Updating operation.** The next is the ternary function $t\,[i := x]$ which modifies the Braun tree $t$ at the index $i$ to obtain a new value $x$. The function can be easily specified with the help of the indexing function by

$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \to Brt(t\,[i := x]) \tag{1}$$
$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \to |t\,[i := x]| = |t| \tag{2}$$
$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \to t\,[i := x]\,[i] = x \tag{3}$$
$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \land j < |t| \land j \neq i \to t\,[i := x]\,[j] = t\,[j]\,. \tag{4}$$

Note that the properties do not specify the result of the application of the updating operation for the empty tree; from the inequality $i < |t|$ we get immediately that the tree $t$ must be non-empty.

The updating operation $t\,[i := x]$ is defined by dyadic recursion on $i$ with substitution in parameter $t$ as a p.r. function:

$$\begin{aligned}&\langle l \mid y \mid r\rangle\,[0 := x] = \langle l \mid x \mid r\rangle \\ &\langle l \mid y \mid r\rangle\,[i\mathbf{1} := x] = \langle l\,[i := x] \mid y \mid r\rangle \\ &\langle l \mid y \mid r\rangle\,[i\mathbf{2} := x] = \langle l \mid y \mid r\,[i := x]\rangle.\end{aligned}$$

The second parameter $x$ does not change in recursion.

We usually intend to apply the operation $t\,[i := x]$ only in cases when $t$ is a non-empty Braun tree and $i$ is its valid index, i.e. $i < |t|$. For that we can take the following property as an alternative (conditional) program for computing the updating function:

$$\begin{aligned}\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \to t\,[i := x] = \mathbf{case}\ & \\ & i = 0 \Rightarrow \mathbf{let}\ t = \langle l \mid y \mid r\rangle\ \mathbf{in}\ \langle l \mid x \mid r\rangle \\ & i = j\mathbf{1} \Rightarrow \mathbf{let}\ t = \langle l \mid y \mid r\rangle\ \mathbf{in}\ \langle l\,[j := x] \mid y \mid r\rangle \\ & i = j\mathbf{2} \Rightarrow \mathbf{let}\ t = \langle l \mid y \mid r\rangle\ \mathbf{in}\ \langle l \mid y \mid r\,[j := x]\rangle \\ \mathbf{end}.\end{aligned}$$

Its conditions of regularity

$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \land i = j\mathbf{1} \land t = \langle l \mid x \mid r\rangle \to l < t \land Brt(l) \land j < |l|$$
$$\vdash_{\mathrm{PA}}\ Brt(t) \land i < |t| \land i = j\mathbf{2} \land t = \langle l \mid x \mid r\rangle \to r < t \land Brt(r) \land j < |r|$$

are trivially satisfied.

*Verification.* (2): This is proved by dyadic induction on $i$ as $\forall t(2)$. In the base case take any Braun tree $t$ such that $0 < |t|$. Then $t = \langle l \mid y \mid r\rangle$ for some $y, l, r$ and we have

$$|\langle l \mid y \mid r\rangle\,[0 := x]| = |\langle l \mid x \mid r\rangle| = |l| + |r| + 1 = |\langle l \mid y \mid r\rangle|\,.$$

In the induction step, when $i = j\mathbf{1}$ for some $j$, take any Braun tree $t$ such that $j\mathbf{1} < |t|$. Then $t = \langle l \mid y \mid r\rangle$ for some $y, l, r$ and we obtain

$$\begin{aligned}|\langle l \mid y \mid r\rangle\,[j\mathbf{1} := x]| = |\langle l\,[j := x] \mid y \mid r\rangle| = |l\,[j := x]| + |r| + 1 \overset{\mathrm{IH}}{=} \\ = |l| + |r| + 1 = |\langle l \mid y \mid r\rangle|\,.\end{aligned}$$

Note that the induction hypothesis is applied with $l$ in place of $t$. The second induction step when $i = j\mathbf{2}$ for some $j$ has a similar proof.

(1): By dyadic induction on $i$ as $\forall t(1)$. We show here only the induction step when $i = j\mathbf{1}$ for some $j$ and $t$ is a Braun tree of the form $t = \langle l \mid y \mid r\rangle$ such that $j\mathbf{1} < \langle l \mid y \mid r\rangle$. We then have

$$\begin{aligned}& Brt(\langle l \mid y \mid r\rangle\,[j\mathbf{1} := x]) \Leftrightarrow Brt\,\langle l\,[j := x] \mid y \mid r\rangle \Leftrightarrow \\ & (|l\,[j := x]| = |r| \lor |l\,[j := x]| = |r| + 1) \land Brt(l\,[j := x]) \land Brt(r) \overset{(2)}{\Leftrightarrow} \\ & (|l| = |r| \lor |l| = |r| + 1) \land Brt(l\,[j := y]) \land Brt(r) \overset{\mathrm{IH}}{\Leftrightarrow} \\ & (|l| = |r| \lor |l| = |r| + 1) \land Brt(l) \land Brt(r).\end{aligned}$$

And the last follows from the assumption that $\langle l \mid y \mid r\rangle$ is a Braun tree. Note also that the induction hypothesis is applied with $l$ in place of $t$.

(3): By dyadic induction on $i$ as $\forall t(3)$. As before we show here only the induction step when $i = j\mathbf{1}$ and $t = \langle l \mid y \mid r\rangle$ is a Braun tree such that $j\mathbf{1} < \langle l \mid y \mid r\rangle$. We then have

$$\langle l \mid y \mid r\rangle\,[j\mathbf{1} := x]\,[j\mathbf{1}] = \langle l\,[j := x] \mid y \mid r\rangle\,[j\mathbf{1}] = l\,[j := x]\,[j] = x.$$

Note that the induction hypothesis is applied with $l$ in place of $t$.

Finally, the last property is proved by dyadic induction on $i$ as $\forall t \forall j(4)$. The proof is similar to those above and thus left to the reader. $\square$

**7.3.12 Creating a new Braun tree.** So far we have been talking about functions operating on existing Braun trees. Now we turn to the problem of creating a Braun tree. The simplest operation is the function $\mathit{New}(n)$ creating a Braun tree of the size $n$:

$$\vdash_{\mathrm{PA}} \mathit{Brt}\,\mathit{New}(n) \tag{1}$$
$$\vdash_{\mathrm{PA}} |\mathit{New}(n)| = n. \tag{2}$$

Note that we do not care about the elements of the resulting Braun tree.

The operation is defined by course of values recursion as a p.r. function:

$$\begin{aligned}
&\mathit{New}(0) = \langle\rangle\\
&\mathit{New}(n\mathbf{1}) = \langle \mathit{New}(n) \mid 0 \mid \mathit{New}(n)\rangle\\
&\mathit{New}(n\mathbf{2}) = \langle \mathit{New}(n+1) \mid 0 \mid \mathit{New}(n)\rangle.
\end{aligned}$$

Note that this is not the definition by dyadic recursion; the recursion in the third clause goes from $n\mathbf{2}$ to $n+1$ but not to $n$ as it is required in dyadic recursion. Note also that in our implementation the result is a tree labelled by 0.

*Verification.* Both properties are proved by complete induction on $n$. We will show the proof only for the last subcase of the induction step of (2). We have

$$\begin{aligned}
|\mathit{New}(n\mathbf{2})| = \big|\langle \mathit{New}(n+1) \mid 0 \mid \mathit{New}(n)\rangle\big| = |\mathit{New}(n+1)| + |\mathit{New}(n)| + 1 \overset{2\times\mathrm{IH}}{=}\\
= n+1+n+1 = n\mathbf{2}.
\end{aligned}$$

Note that IH has been used twice; first with the number $n+1 < n\mathbf{2}$ and secondly with $n < n\mathbf{2}$. $\square$

**7.3.13 Insertion of a new first label.** The function $\mathit{Insfirst}(t,x)$ increases by one the size of the Braun tree $t$ by inserting a new element $x$ into its first position. The function satisfies

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \to \mathit{Brt}\,\mathit{Insfirst}(t,x) \tag{1}$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \to |\mathit{Insfirst}(t,x)| = |t| + 1 \tag{2}$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \to \mathit{Insfirst}(t,x)\,[0] = x \tag{3}$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land i < |t| \to \mathit{Insfirst}(t,x)\,[i+1] = t\,[i] \tag{4}$$

and it is defined by structural recursion on the binary tree $t$ with substitution in parameter as a p.r. function (see Fig. 7.10):

$$\begin{aligned}
&\mathit{Insfirst}(\langle\rangle, x) = \langle \langle\rangle \mid x \mid \langle\rangle\rangle\\
&\mathit{Insfirst}(\langle l \mid y \mid r\rangle, x) = \langle \mathit{Insfirst}(r,y) \mid x \mid l\rangle.
\end{aligned}$$

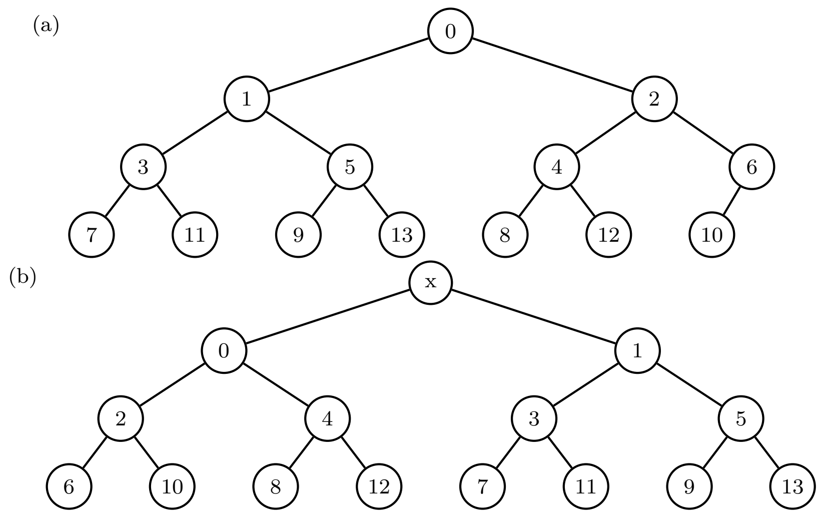

**Fig. 7.10** Insertion of a new first label $x$ into a 14-elements Braun tree: **(a)** before insertion of $x$, **(b)** after insertion of $x$.

*Verification.* (2): It follows from

$$\vdash_{\mathrm{PA}} Bt(t) \rightarrow |Insfirst(t, x)| = |t| + 1 \qquad (\dagger_1)$$

which is proved by structural induction on the binary tree $t$ as $\forall x(\dagger_1)$. The base case is obvious. In the induction step when $t = \langle l \mid y \mid r \rangle$ take any $x$ and we have

$$|Insfirst(\langle l \mid y \mid r \rangle, x)| = |\langle Insfirst(r, y) \mid x \mid l \rangle| = |Insfirst(r, y)| + |l| + 1 \overset{\mathrm{IH}}{=}$$
$$= |r| + 1 + |l| + 1 = |\langle l \mid y \mid r \rangle| + 1.$$

Note that the induction hypothesis is applied with $y$ in place of $x$.

(1): By structural induction on the binary tree $t$ as $\forall x(1)$. The base case is straightforward. In the induction step when $t = \langle l \mid y \mid r \rangle$ take any $x$ and we have

$$Brt\, Insfirst(\langle l \mid y \mid r \rangle, x) \Leftrightarrow Brt\, \langle Insfirst(r, y) \mid x \mid l \rangle \Leftrightarrow$$
$$(|Insfirst(r, y)| = |l| \lor |Insfirst(r, y)| = |l| + 1) \land$$
$$\land\, Brt\, Insfirst(r, y) \land Brt(l) \overset{(\dagger_1)}{\Leftrightarrow}$$
$$(|r| + 1 = |l| \lor |r| + 1 = |l| + 1) \land Brt\, Insfirst(r, y) \land Brt(l) \overset{\mathrm{IH}}{\Leftrightarrow}$$
$$(|l| = |r| \lor |l| = |r| + 1) \land Brt(r) \land Brt(l).$$

The last follows from the assumption that $\langle l \mid y \mid r\rangle$ is a Braun tree. Note also that the induction hypothesis is applied with $y$ in place of $x$.

(3): Obvious.

(4): By structural induction on the binary tree $t$ as $\forall i \forall x(4)$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid y \mid r\rangle$ take any $i, x$ such that $i < |\langle l \mid y \mid r\rangle|$ and consider three cases. If $i = 0$ then

$$\begin{aligned} \mathit{Insfirst}(\langle l \mid y \mid r\rangle, x)\,[0+1] = \langle \mathit{Insfirst}(r,y) \mid x \mid l\rangle\,[1] = \mathit{Insfirst}(r,y)\,[0] \overset{(3)}{=} \\ = y = \langle l \mid y \mid r\rangle\,[0]\,. \end{aligned}$$

If $i = j\mathbf{1}$ for some $j$ then we have

$$\mathit{Insfirst}(\langle l \mid y \mid r\rangle, x)\,[j\mathbf{1}+1] = \langle \mathit{Insfirst}(r,y) \mid x \mid l\rangle\,[j\mathbf{2}] = l\,[j] = \langle l \mid y \mid r\rangle\,[j\mathbf{1}]\,.$$

Finally, if $i = j\mathbf{2}$ for some $j$ then we obtain

$$\begin{aligned} \mathit{Insfirst}(\langle l \mid y \mid r\rangle, x)\,[j\mathbf{2}+1] = \langle \mathit{Insfirst}(r,y) \mid x \mid l\rangle\,[(j+1)\mathbf{1}] = \\ = \mathit{Insfirst}(r,y)\,[j+1] \overset{\mathrm{IH}}{=} r\,[j] = \langle l \mid y \mid r\rangle\,[j\mathbf{2}]\,. \end{aligned}$$

Note that in this last case the induction hypothesis is applied with $j$ and $y$ in place of $i$ and $x$, respectively. □

**7.3.14 Deletion of the first label.** The function $\mathit{Delfirst}(t)$ removes the first label from a non-empty Braun tree:

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \to \mathit{Brt}\,\mathit{Delfirst}(t) \tag{1}$$

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \to |\mathit{Delfirst}(t)| + 1 = |t| \tag{2}$$

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \land i + 1 < |t| \to \mathit{Delfirst}(t)\,[i] = t\,[i+1] \tag{3}$$

The operation is defined by structural recursion on binary trees as a p.r. function (see Fig. 7.11):

$$\begin{aligned} &\mathit{Delfirst}\,\langle\langle\rangle \mid x \mid r\rangle = \langle\rangle \\ &\mathit{Delfirst}\,\langle l \mid x \mid r\rangle = \langle r \mid l\,[0] \mid \mathit{Delfirst}(l)\rangle \leftarrow l \neq \langle\rangle. \end{aligned}$$

We usually intend to apply the operation $\mathit{Delfirst}(t)$ only in cases when $t$ is a non-empty binary tree. For that we can take the following property as an alternative (conditional) program for computing the deletion function:

```
⊢PA $Bt(t) \land t \neq \langle\rangle \to Delfirst(t) =$ let $t = \langle l \mid x \mid r\rangle$ in
                                        case
                                          $l = \langle\rangle \Rightarrow \langle\rangle$
                                          $l \neq \langle\rangle \Rightarrow \langle r \mid l\,[0] \mid Delfirst(l)\rangle$
                                        end.
```

Its condition of regularity

$$\vdash_{\mathrm{PA}} Bt(t) \land t \neq \langle\rangle \land t = \langle l \mid x \mid r\rangle \land l \neq \langle\rangle \to l < t \land Bt(l) \land l \neq \langle\rangle$$

is trivially satisfied.

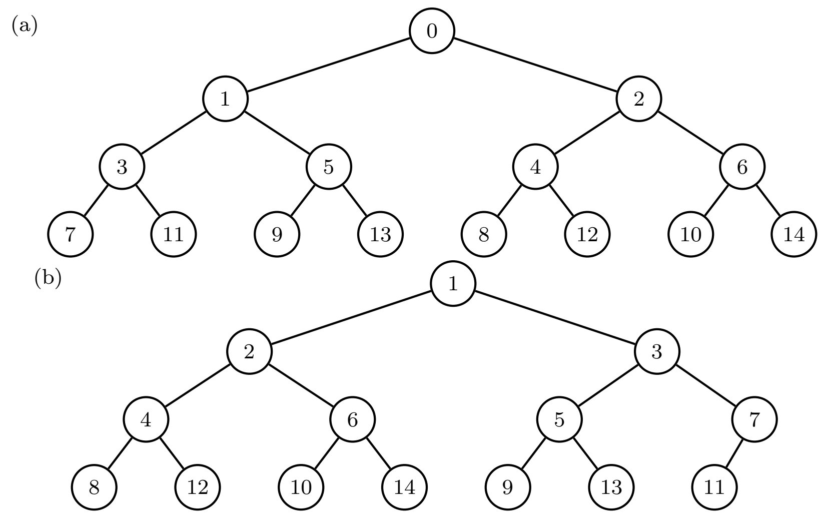

**Fig. 7.11** Deletion of the first label from a 15-elements Braun tree: **(a)** before deleting, **(b)** after deleting.

*Verification.* (2): By structural induction on on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ we consider two cases. If $l = \langle\rangle$ then it must be $r = \langle\rangle$ and we have

$$\left|\mathit{Delfirst}\left\langle\langle\rangle \mid x \mid \langle\rangle\right\rangle\right| + 1 = |\langle\rangle| + 1 = 1 = |\langle\rangle| + |\langle\rangle| + 1 = \left|\left\langle\langle\rangle \mid x \mid \langle\rangle\right\rangle\right|.$$

Otherwise $l \neq \langle\rangle$ and we obtain

$$\begin{aligned}|\mathit{Delfirst}\,\langle l \mid x \mid r\rangle| + 1 &= \left|\langle r \mid l\,[0] \mid \mathit{Delfirst}(l)\rangle\right| + 1 = \\ &= |r| + |\mathit{Delfirst}(l)| + 1 + 1 \overset{\mathrm{IH}}{=} |r| + |l| + 1 = |\langle l \mid x \mid r\rangle|.\end{aligned}$$

(1): By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ we consider two cases. If $l = \langle\rangle$ then $r = \langle\rangle$ and the claim holds trivially. Otherwise $l \neq \langle\rangle$ and we obtain

$$\mathit{Brt}\ \mathit{Delfirst}\,\langle l \mid x \mid r\rangle \Leftrightarrow \mathit{Brt}\,\langle r \mid l\,[0] \mid \mathit{Delfirst}(l)\rangle \Leftrightarrow$$

$$(|r| = |\mathit{Delfirst}(l)| \lor |r| = |\mathit{Delfirst}(l)| + 1) \land \mathit{Brt}(r) \land \mathit{Brt}\ \mathit{Delfirst}(l) \overset{(2)}{\Leftrightarrow}$$

$$(|r| + 1 = |l| \lor |r| = |l|) \land Brt(r) \land Brt\, Delfirst(l) \overset{\text{IH}}{\Leftrightarrow}$$
$$(|l| = |r| \lor |l| = |r| + 1) \land Brt(r) \land Brt(l).$$

And the last follows from the assumption that $\langle l \mid {}^{x} \mid r\rangle$ is a Braun tree.

(3): By structural induction on the binary tree $t$ as $\forall i(3)$ In the base case there is nothing to prove. In the induction step when $t = \langle l \mid {}^{x} \mid r\rangle$ take any $i$ such that $i + 1 < |\langle l \mid {}^{x} \mid r\rangle|$ and consider two cases. If $l = \langle\rangle$ then $r = \langle\rangle$ and we have a contradiction. So suppose that $l \neq \langle\rangle$. We consider three subcases. If $i = 0$ then

$$(Delfirst\, \langle l \mid {}^{x} \mid r\rangle)\,[0] = \langle r \mid {}^{l\,[0]} \mid Delfirst(l)\rangle\,[0] = l\,[0] = \langle l \mid {}^{x} \mid r\rangle\,[0+1]\,.$$

If $i = j\mathbf{1}$ for some $j$ then we have

$$\begin{aligned}(Delfirst\, \langle l \mid {}^{x} \mid r\rangle)\,[j\mathbf{1}] = \langle r \mid {}^{l\,[0]} \mid Delfirst(l)\rangle\,[j\mathbf{1}] = r\,[j] = \\ = \langle l \mid {}^{x} \mid r\rangle\,[j\mathbf{2}] = \langle l \mid {}^{x} \mid r\rangle\,[j\mathbf{1}+1]\,.\end{aligned}$$

If $i = j\mathbf{2}$ for some $j$ then $j + 1 < |l|$ and we get

$$\begin{aligned}(Delfirst\, \langle l \mid {}^{x} \mid r\rangle)\,[j\mathbf{2}] = \langle r \mid {}^{l\,[0]} \mid Delfirst(l)\rangle\,[j\mathbf{2}] = Delfirst(l)\,[j] \overset{\text{IH}}{=} \\ = l\,[j+1] = \langle l \mid {}^{x} \mid r\rangle\,[(j+1)\mathbf{1}] = \langle l \mid {}^{x} \mid r\rangle\,[j\mathbf{2}+1]\,.\end{aligned}$$

Note that the induction hypothesis is applied with $j$ in place of $i$. □

**7.3.15 Insertion of a new last label.** Unlike the previous two operations which manipulate with first elements of Braun trees accessing the last element requires to know the size of the tree. Consider for instance the problem of inserting an element $x$ at the end of a Braun tree $t$. For that we need to know the exact size of the tree since the number $|t|$ is the index of the position of the new last element $x$.

The ternary function $Inslast(n, t, x)$ which inserts $x$ at the last position of the tree $t$ has the following specification:

$$\vdash_{\mathrm{PA}} Brt(t) \rightarrow Brt\, Inslast(|t|, t, x) \tag{1}$$
$$\vdash_{\mathrm{PA}} Brt(t) \rightarrow |Inslast(|t|, t, x)| = |t| + 1 \tag{2}$$
$$\vdash_{\mathrm{PA}} Brt(t) \land i < |t| \rightarrow Inslast(|t|, t, x)\,[i] = t\,[i] \tag{3}$$
$$\vdash_{\mathrm{PA}} Brt(t) \rightarrow Inslast(|t|, t, x)\,[|t|] = x \tag{4}$$

and it is defined by dyadic recursion on $n$ with substitution in the parameter $t$ as a p.r. function:

$$\begin{aligned}&Inslast(0, t, x) = \langle\langle\rangle \mid {}^{x} \mid \langle\rangle\rangle \\ &Inslast(n\mathbf{1}, \langle l \mid {}^{y} \mid r\rangle, x) = \langle Inslast(n, l, x) \mid {}^{y} \mid r\rangle \\ &Inslast(n\mathbf{2}, \langle l \mid {}^{y} \mid r\rangle, x) = \langle l \mid {}^{y} \mid Inslast(n, r, x)\rangle.\end{aligned}$$

Note that the parameter $x$ does not change in recursion. Not also that the insertion operation is specified only non-empty Braun trees.

We intend to apply the operation $\mathit{Inslast}(n, t, x)$ only in cases when $t$ is a Braun tree and $n$ is the index of the new last position, i.e. $n = |t|$. For that we can take the following property as an alternative (conditional) program for computing the insertion function:

$$\begin{aligned}
\vdash_{\mathrm{PA}}\ & \mathit{Brt}(t) \land n = |t| \rightarrow \\
& \mathit{Inslast}(n, t, x) = \mathbf{case} \\
&\qquad n = 0 \Rightarrow \langle\langle\rangle \mid x \mid \langle\rangle\rangle \\
&\qquad n = m\mathbf{1} \Rightarrow \mathbf{let}\ t = \langle l \mid y \mid r\rangle\ \mathbf{in}\ \langle \mathit{Inslast}(m, l, x) \mid y \mid r\rangle \\
&\qquad n = m\mathbf{2} \Rightarrow \mathbf{let}\ t = \langle l \mid y \mid r\rangle\ \mathbf{in}\ \langle l \mid y \mid \mathit{Inslast}(m, r, x)\rangle \\
&\quad \mathbf{end}.
\end{aligned}$$

Its conditions of regularity

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land n = |t| \land n = m\mathbf{1} \land t = \langle l \mid y \mid r\rangle \rightarrow l < t \land \mathit{Brt}(l) \land m = |l|$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land n = |t| \land n = m\mathbf{2} \land t = \langle l \mid y \mid r\rangle \rightarrow r < t \land \mathit{Brt}(r) \land m = |r|$$

are trivially satisfied.

*Verification.* (2): By structural induction on the binary tree $t$. The base is obvious. In the induction step when $t = \langle l \mid y \mid r\rangle$ we consider two cases. If $|l| = |r|$ then $|\langle l \mid y \mid r\rangle| = 2\,|l| + 1$ and thus

$$\begin{aligned}
&|\mathit{Inslast}(|\langle l \mid y \mid r\rangle|, \langle l \mid y \mid r\rangle, x)| = |\mathit{Inslast}(|l|\mathbf{1}, \langle l \mid y \mid r\rangle, x)| = \\
&\quad = |\langle \mathit{Inslast}(|l|, l, x) \mid y \mid r\rangle| = |\mathit{Inslast}(|l|, l, x)| + |r| + 1 \overset{\mathrm{IH}}{=} \\
&\quad = |l| + 1 + |r| + 1 = |\langle l \mid y \mid r\rangle| + 1.
\end{aligned}$$

Otherwise we have $|l| = |r| + 1$ and therefore $|\langle l \mid y \mid r\rangle| = 2\,|r| + 2$; the proof proceeds similarly.

(1): By structural induction on the binary tree $t$. The base is obvious. In the induction step when $t = \langle l \mid y \mid r\rangle$ we consider two cases. If $|l| = |r|$ then by the same arguments as above we obtain

$$\begin{aligned}
&\mathit{Brt}\,\mathit{Inslast}(|\langle l \mid y \mid r\rangle|, \langle l \mid y \mid r\rangle, x) \Leftrightarrow \mathit{Brt}\,\langle \mathit{Inslast}(|l|, l, x) \mid y \mid r\rangle \Leftrightarrow \\
&(|\mathit{Inslast}(|l|, l, x)| = |r| \lor |\mathit{Inslast}(|l|, l, x)| = |r| + 1) \land \\
&\quad \land \mathit{Brt}\,\mathit{Inslast}(|l|, l, x) \land \mathit{Brt}(r) \overset{(2)}{\Leftrightarrow} \\
&(|l| + 1 = |r| \lor |l| + 1 = |r| + 1) \land \mathit{Brt}\,\mathit{Inslast}(|l|, l, x) \land \mathit{Brt}(r) \overset{\mathrm{IH}}{\Leftrightarrow} \\
&(|l| = |r| \lor |l| = |r| + 1) \land \mathit{Brt}(l) \land \mathit{Brt}(r).
\end{aligned}$$

The last follows from the assumption that $\langle l \mid y \mid r\rangle$ is a Braun tree. The case $|l| = |r| + 1$ is proved similarly.

(3): By structural induction on the binary tree $t$ as $\forall i(3)$. The base is obvious. In the induction step when $t = \langle l \mid y \mid r\rangle$ take any $i$ and consider two cases. If $|l| = |r|$ (and similarly when $|l| = |r|+1$) then we consider the following three subcases. If $i = j\mathbf{1}$ then we have

$$\mathit{Inslast}(|\langle l \mid y \mid r\rangle|, \langle l \mid y \mid r\rangle, x)\,[j\mathbf{1}] = \langle \mathit{Inslast}(|l|, l, x) \mid y \mid r\rangle\,[j\mathbf{1}] =$$
$$= \mathit{Inslast}(|l|, l, x)\,[j] \overset{\mathrm{IH}}{=} l\,[j] = \langle l \mid y \mid r\rangle\,[j\mathbf{1}]\,.$$

Other subcases when $i = 0$ are $i = j\mathbf{2}$ are straightforward.

The last property (4) is proved similarly. □

**7.3.16 Deletion of the last label.** The final operation is removal of the last element from Braun trees. This is realized with the binary function $\mathit{Dellast}(n,t)$ which deletes the last label of the non-empty Braun tree $t$. Here the number $n$ is the index of the last element in the tree $t$, i.e. we have $n = |t| \dot{-} 1$. The function satisfies

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \to \mathit{Brt}\,\mathit{Dellast}(|t| \dot{-} 1, t) \tag{1}$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \to |\mathit{Dellast}(|t| \dot{-} 1, t)| + 1 = |t| \tag{2}$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \land i + 1 < |t| \to \mathit{Dellast}(|t| \dot{-} 1, t)\,[i] = t\,[i] \tag{3}$$

and it is defined by dyadic recursion on $n$ with substitution in parameter as a p.r. function:

$$\mathit{Dellast}(0,t) = \langle\rangle$$
$$\mathit{Dellast}(n\mathbf{1}, \langle l \mid x \mid r\rangle) = \langle \mathit{Dellast}(n,l) \mid x \mid r\rangle$$
$$\mathit{Dellast}(n\mathbf{2}, \langle l \mid x \mid r\rangle) = \langle l \mid x \mid \mathit{Dellast}(n,r)\rangle.$$

We intend to apply the operation $\mathit{Dellast}(n,t)$ only in cases when $t$ is a non-empty Braun tree and $n$ is the index of its last position, i.e. $n = |t| \dot{-} 1$. For that we can take the following property as an alternative (conditional) program for computing the deletion function:

```
$\vdash_{\mathrm{PA}}$ $\mathit{Brt}(t) \land t \neq \langle\rangle \land n = |t| \dot{-} 1 \to$
    $\mathit{Dellast}(n,t) =$ **case**
                 $n = 0 \Rightarrow \langle\rangle$
                 $n = m\mathbf{1} \Rightarrow$ **let** $t = \langle l \mid x \mid r\rangle$ **in** $\langle \mathit{Dellast}(n,l) \mid x \mid r\rangle$
                 $n = m\mathbf{2} \Rightarrow$ **let** $t = \langle l \mid x \mid r\rangle$ **in** $\langle l \mid x \mid \mathit{Dellast}(n,r)\rangle$
               **end**.
```

Its conditions of regularity

$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \land n = |t| \dot{-} 1 \land n = m\mathbf{1} \land t = \langle l \mid x \mid r\rangle \to$$
$$l < t \land \mathit{Brt}(l) \land l \neq \langle\rangle \land m = |l| \dot{-} 1$$
$$\vdash_{\mathrm{PA}} \mathit{Brt}(t) \land t \neq \langle\rangle \land n = |t| \dot{-} 1 \land n = m\mathbf{2} \land t = \langle l \mid x \mid r\rangle \to$$
$$r < t \land \mathit{Brt}(r) \land r \neq \langle\rangle \land m = |r| \dot{-} 1$$

are trivially satisfied.

*Verification.* (2): By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ is a non-empty Braun tree we consider two cases. If $l = \langle\rangle$ then $r = \langle\rangle$, i.e. $t$ is a singleton tree$\langle\langle\rangle \mid x \mid \langle\rangle\rangle$. We then have

$$\begin{aligned}|\mathit{Dellast}(|\langle\langle\rangle \mid x \mid \langle\rangle\rangle| \dot{-} 1, \langle\langle\rangle \mid x \mid \langle\rangle\rangle)| + 1 = |\mathit{Dellast}(0, \langle\langle\rangle \mid x \mid \langle\rangle\rangle)| + 1 = \\ = |\langle\rangle| + 1 = 1 = |\langle\langle\rangle \mid x \mid \langle\rangle\rangle|\,.\end{aligned}$$

So suppose $l \neq \langle\rangle$. Now we consider two subcases. If $|l| = |r| + 1$ then

$$|\langle l \mid x \mid r\rangle| \dot{-} 1 = |l| + |r| = 2\,|l| \dot{-} 1 = 2(|l| \dot{-} 1) + 1 = (|l| \dot{-} 1)\mathbf{1} \qquad (\dagger_1)$$

by noting that $|l| \neq 0$.

$$\begin{aligned}&|\mathit{Dellast}(|\langle l \mid x \mid r\rangle| \dot{-} 1, \langle l \mid x \mid r\rangle)| + 1 \overset{(\dagger_1)}{=} \\ &\quad = |\mathit{Dellast}((|l| \dot{-} 1)\mathbf{1}, \langle l \mid x \mid r\rangle)| + 1 = |\langle \mathit{Dellast}(|l| \dot{-} 1, l) \mid x \mid r\rangle| + 1 = \\ &\quad = |\mathit{Dellast}(|l| \dot{-} 1, l)| + 1 + |r| + 1 \overset{\text{IH}}{=} |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.\end{aligned}$$

If $|l| = |r|$ then

$$|\langle l \mid x \mid r\rangle| \dot{-} 1 = |l| + |r| = 2\,|r| = 2(|r| \dot{-} 1) + 2 = (|r| \dot{-} 1)\mathbf{2} \qquad (\dagger_2)$$

since $|r| \neq 0$. We then have

$$\begin{aligned}&|\mathit{Dellast}(|\langle l \mid x \mid r\rangle| \dot{-} 1, \langle l \mid x \mid r\rangle)| + 1 \overset{(\dagger_2)}{=} \\ &\quad = |\mathit{Dellast}((|r| \dot{-} 1)\mathbf{2}, \langle l \mid x \mid r\rangle)| + 1 = |\langle l \mid x \mid \mathit{Dellast}(|r| \dot{-} 1, r)\rangle| + 1 = \\ &\quad = |l| + |\mathit{Dellast}(|r| \dot{-} 1, r)| + 1 + 1 \overset{\text{IH}}{=} |l| + |r| + 1 = |\langle l \mid x \mid r\rangle|\,.\end{aligned}$$

This concludes the proof of the induction step.

(1): By structural induction on the binary tree $t$. In the base case there is nothing to prove. In the induction step when $t = \langle l \mid x \mid r\rangle$ is a non-empty Braun tree we consider two cases. If $l = \langle\rangle$ then $r = \langle\rangle$, i.e. $t$ is a singleton tree$\langle\langle\rangle \mid x \mid \langle\rangle\rangle$. Then

$$\mathit{Dellast}(|\langle\langle\rangle \mid x \mid \langle\rangle\rangle| \dot{-} 1, \langle\langle\rangle \mid x \mid \langle\rangle\rangle) = \mathit{Dellast}(0, \langle\langle\rangle \mid x \mid \langle\rangle\rangle) = \langle\rangle$$

is clearly a Braun tree. So suppose $l \neq \langle\rangle$. Now we consider two subcases. If $|l| = |r| + 1$ then

$$\begin{aligned}
&\mathit{Brt}\,\mathit{Dellast}(|\langle l \mid x \mid r\rangle| \dot{-} 1, \langle l \mid x \mid r\rangle) \overset{(\dagger_1)}{\Leftrightarrow} \mathit{Brt}\,\mathit{Dellast}((|l| \dot{-} 1)\mathbf{1}, \langle l \mid x \mid r\rangle) \Leftrightarrow \\
&\Leftrightarrow \mathit{Brt}\,\langle \mathit{Dellast}(|l| \dot{-} 1, l) \mid x \mid r\rangle \Leftrightarrow \\
&\Leftrightarrow (|\mathit{Dellast}(|l| \dot{-} 1, l)| = |r| \lor |\mathit{Dellast}(|l| \dot{-} 1, l)| = |r| + 1) \land \\
&\qquad\qquad \mathit{Brt}\,\mathit{Dellast}(|l| \dot{-} 1, l) \land \mathit{Brt}(r) \Leftrightarrow \\
&\Leftrightarrow (|\mathit{Dellast}(|l| \dot{-} 1, l)| + 1 = |r| + 1 \lor |\mathit{Dellast}(|l| \dot{-} 1, l)| + 1 = |r| + 2) \land \\
&\qquad\qquad \mathit{Brt}\,\mathit{Dellast}(|l| \dot{-} 1, l) \land \mathit{Brt}(r) \overset{(2)}{\Leftrightarrow} \\
&\Leftrightarrow (|l| = |r| + 1 \lor |l| = |r| + 2) \land \mathit{Brt}\,\mathit{Dellast}(|l| \dot{-} 1, l) \land \mathit{Brt}(r) \Leftrightarrow \\
&\Leftrightarrow |l| = |r| + 1 \land \mathit{Brt}\,\mathit{Dellast}(|l| \dot{-} 1, l) \land \mathit{Brt}(r).
\end{aligned}$$

The last follows from the assumption that $\langle l \mid x \mid r\rangle$ is a Braun tree and from IH applied to the left subtree $l$. The subcase when $|l| = |r|$ is proved similarly.

(3): By structural induction on the binary tree $t$ as $\forall i(3)$. The proof is similar to that above and thus left to the reader. □

## 7.4 Symbolic Expressions

**7.4.1 Introduction.** 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 Sect. 7.1. We will use in the following paragraphs another union type to arithmetize a certain class of expressions.

Suppose that we wish to operate symbolically on numeric terms which are formed from variables $x_i$, constants $n$ by the numeric operators + (addition) and × (multiplication). Functional programming languages use the following union type to specify the domain of numeric terms:

$$\mathit{Term} = \mathit{Var}(\mathrm{N}) \mid \mathit{Const}(\mathrm{N}) \mid \mathit{Add}(\mathit{Term}, \mathit{Term}) \mid \mathit{Mult}(\mathit{Term}, \mathit{Term}).$$

A value of type *Term* is therefore either a variable $\mathit{Var}(i)$, or a constant $\mathit{Const}(n)$, or an addition $\mathit{Add}(t_1, t_2)$, or a multiplication $\mathit{Mult}(t_1, t_2)$, where $i$ and $n$ are of type N and $t_1$ and $t_2$ are values of type *Term*. The functions $\mathit{Var}(i)$, $\mathit{Const}(n)$, $\mathit{Add}(t_1, t_2)$ and $\mathit{Mult}(t_1, t_2)$ are called *constructors*.

**7.4.2 Constructors of numeric terms.** Arithmetization of numeric expressions is done with the help of the following four pair constructors with pairwise different tags (see Par. 5.2.10 for details):

$$\begin{aligned}
\mathrm{x}_i^\bullet &= \langle 0, i\rangle && \text{(variables)}\\
n^\bullet &= \langle 1, n\rangle && \text{(constants)}\\
t_1 +^\bullet t_2 &= \langle 2, t_1, t_2\rangle && \text{(addition)}\\
t_1 \times^\bullet t_2 &= \langle 3, t_1, t_2\rangle. && \text{(multiplication)}
\end{aligned}$$

From the properties of the pairing function we obtain the constructors are pairwise disjoint and that the constructors are injective mappings, e.g.

$$\begin{aligned}
&\vdash_{\mathrm{PA}} \ \mathrm{x}_i^\bullet \neq t_1 +^\bullet t_2\\
&\vdash_{\mathrm{PA}} \ t_1 \times^\bullet t_2 = t_1' \times^\bullet t_2' \to t_1 = t_1' \land t_2 = t_2'.
\end{aligned}$$

Similar properties hold also for the other constructors.

The pattern matching style of definitions of functions operating over the codes of numeric terms is obtained with the conditionals of the form

```
case
  $t = \mathrm{x}_i^\bullet \Rightarrow_i \beta_1[i]$
  $t = n^\bullet \Rightarrow_n \beta_2[n]$
  $t = t_1 +^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_3[t_1, t_2]$
  $t = t_1 \times^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_4[t_1, t_2]$
  otherwise $\Rightarrow \beta_5$.
end
```

This is called *discrimination on the constructors of numeric terms.* Recall that such conditionals are instances of pair constructors discrimination terms discussed in Par. 5.2.10.

The above conditional is evaluated as follows. Consider, for instance, its third variant $t = t_1 +^\bullet t_2$. The expression

$$\mathit{Tuple}_*(3, t) \land_* [t]_1^3 =_* 2$$

is its characteristic term as we have

$$\vdash_{\mathrm{PA}} \ \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \leftrightarrow \mathit{Tuple}(3, t) \land [t]_1^3 = 2.$$

Note also that

$$\vdash_{\mathrm{PA}} \ t = t_1 +^\bullet t_2 \to t_1 = [t]_2^3 \land t_2 = [t]_3^3$$

and therefore, the terms $[t]_2^3$ and $[t]_3^3$ are the witnessing terms for the output variables $t_1, t_2$ of this variant. Similarly for the other variants.

**7.4.3 Arithmetization of numeric terms.** We wish to assign to every numeric $\tau$ a unique number $\ulcorner\tau\urcorner$, called *the code of* $\tau$. The mapping is defined inductively on the structure of numeric terms:

$$\begin{aligned}
\ulcorner x_i \urcorner &= \mathrm{x}_i^\bullet \\
\ulcorner n \urcorner &= n^\bullet \\
\ulcorner \tau_1 + \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner +^\bullet \ulcorner \tau_1 \urcorner \\
\ulcorner \tau_1 \times \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner \times^\bullet \ulcorner \tau_1 \urcorner .
\end{aligned}$$

We can now encode, for instance, the term $4 \times x_5 + x_7$ by the number

$$4^\bullet \times^\bullet \mathrm{x}_5^\bullet +^\bullet \mathrm{x}_7^\bullet = 103\,635\,707\,473\,048\,605\,704.$$

Discrimination on the constructors of numeric terms is used in the definition of the p.r. predicate $\mathit{Term}(t)$ holding of the codes of numeric terms. The predicate is defined by course of values recursion as follows:

$$\begin{aligned}
&\mathit{Term}(\mathrm{x}_i^\bullet)\\
&\mathit{Term}(n^\bullet)\\
&\mathit{Term}(t_1 +^\bullet t_2) \leftarrow \mathit{Term}(t_1) \wedge \mathit{Term}(t_2)\\
&\mathit{Term}(t_1 \times^\bullet t_2) \leftarrow \mathit{Term}(t_1) \wedge \mathit{Term}(t_2).
\end{aligned}$$

In the sequel we identify numeric terms with their codes and from now on we will say *the numeric term $t$* instead of *the code $t$ of a numeric term.*

**7.4.4 Case analysis on numeric terms.** From the definition of the predicate *Term* we get directly the following property

$$\vdash_{\mathrm{PA}} \mathit{Term}(t) \to \exists i\, t = \mathrm{x}_i^\bullet \vee \exists n\, t = n^\bullet \vee \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \vee \exists t_1 \exists t_2\, t = t_1 \times^\bullet t_2.$$

This is called the principle of *structural case analysis on the constructors of the numeric term $t$.*

We can use the above principle of structural case analysis in order to establish the admissibility of a certain kind of conditional discriminations on the constructors of numeric terms. These are of the form

```
$\mathit{Term}(t) \to$ case
                 $t = \mathrm{x}_i^\bullet \Rightarrow_i \beta_1[i]$
                 $t = n^\bullet \Rightarrow_n \beta_2[n]$
                 $t = t_1 +^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_3[t_1,t_2]$
                 $t = t_1 \times^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_4[t_1,t_2]$.
               end
```

Because of the precondition $\mathit{Term}(t)$, we have to evaluate only four alternatives instead of five. Moreover, characteristic terms of each alternative can be selected much simpler than those in Par. 7.4.2. For instance, we have

$$\mathit{Term}(t) \to \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \leftrightarrow [t]_1^3 = 2$$

and thus we can use the expression $[t]_1^3 =_* 2$ as the characteristic term of the third variant of the above conditional. Compare with the characteristic term $\mathit{Tuple}_*(3,t) \wedge_* [t]_1^3 =_* 2$ of the same variant from Par. 7.4.2.

**7.4.5 Structural induction on numeric terms.** The principle of structural induction over numeric terms can be informally stated as follows. To prove by structural induction that a property $\varphi[t]$ holds for every numeric term $t$ it suffices to prove:

*Base cases:* the property holds for every variable $\mathrm{x}_i^\bullet$ and constant $n^\bullet$.
*Induction steps:* if the property holds for the terms $t_1, t_2$ then it holds also for the terms $t_1 +^\bullet t_2$ and $t_1 \times^\bullet t_2$.

This is expressed formally in PA by

$$\vdash_{\mathrm{PA}} \forall i\, \varphi[\mathrm{x}_i^\bullet] \land \forall n\, \varphi[n^\bullet] \land \forall t_1 \forall t_2 (\varphi[t_1] \land \varphi[t_2] \to \varphi[t_1 +^\bullet t_2]) \land \\ \forall t_1 \forall t_2 (\varphi[t_1] \land \varphi[t_2] \to \varphi[t_1 \times^\bullet t_2]) \to \mathit{Term}(t) \to \varphi[t]$$

The theorem is called the principle of *structural induction on the numeric term t for* $\varphi[t]$.

*Proof.* The principle of structural induction for numeric terms is derived in PA as follows. Under the assumptions corresponding to the base cases and induction steps of the structural induction take any numeric term $t$ and prove that $\varphi[t]$ holds by complete induction on $t$. We consider the following four cases according to Par. 7.4.4. The cases when $t = \mathrm{x}_i^\bullet$ or $t = n^\bullet$ are trivial. In the case when $t = t_1 +^\bullet t_2$ for some $t_1, t_2$ we have $\varphi[t_1]$ and $\varphi[t_2]$ by IH since $t_1 < t_1 +^\bullet t_2$ and $t_2 < t_1 +^\bullet t_2$. From the assumption we get $\varphi[t_1 +^\bullet t_2]$. The case when $t = t_1 \times^\bullet t_2$ for some $t_1, t_2$ is similar. □

**7.4.6 Structural recursion on numeric terms.** Structural induction over numeric terms is used to prove properties of functions defined by the scheme of *structural recursion on numeric terms*. In its simplest form, the operator of structural recursion over numeric terms introduces a function $f$ from functions $g_1$, $g_2$, $g_3$ and $g_4$ satisfying

$$\begin{aligned} f(t,y) = \mathbf{case}\; & \\ & t = \mathrm{x}_i^\bullet \Rightarrow g_1(i,y) \\ & t = n^\bullet \Rightarrow g_2(n,y) \\ & t = t_1 +^\bullet t_2 \Rightarrow g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & t = t_1 \times^\bullet t_2 \Rightarrow g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & \mathbf{otherwise} \Rightarrow 0 \\ \mathbf{end}. & \end{aligned}$$

Note that this is a recursive definition regular in the first argument with discrimination on the constructors of numeric terms (output variables of each variant are omitted).

The following identities form the clausal form of the above definition

$$\begin{aligned} & f(\mathrm{x}_i^\bullet, y) = g_1(i,y) \\ & f(n^\bullet, y) = g_2(n,y) \\ & f(t_1 +^\bullet t_2, y) = g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & f(t_1 \times^\bullet t_2, y) = g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \end{aligned}$$

Note here that this is a typical example where we wish to use the default clauses – in this case

$$f(t,y) = 0 \leftarrow \neg\exists i\, t = \mathrm{x}_i^\bullet \wedge \neg\exists n\, t = n^\bullet \wedge \neg\exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \wedge \neg\exists t_1 \exists t_2\, t = t_1 \times^\bullet t_2$$

in order not to clutter the definition. We do not care what value is yielded by the application $f(t,y)$ if $t$ is not the code of a numeric term.

The above definition for the function $f$ can be easily rewritten to a conditional program for the same function as we have

$$\begin{aligned} \vdash_{\mathrm{PA}}\ \mathit{Term}(t) \rightarrow f(t,y) = \mathbf{case}\ & \\ & t = \mathrm{x}_i^\bullet \Rightarrow g_1(i,y) \\ & t = n^\bullet \Rightarrow g_2(n,y) \\ & t = t_1 +^\bullet t_2 \Rightarrow g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & t = t_1 \times^\bullet t_2 \Rightarrow g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & \mathbf{end} \end{aligned}$$

Its conditions of regularity, e.g. for the variant $t = t_1 +^\bullet t_2$

$$\vdash_{\mathrm{PA}}\ \mathit{Term}(t) \wedge t = t_1 +^\bullet t_2 \rightarrow t_1 < t \wedge \mathit{Term}(t_1)$$
$$\vdash_{\mathrm{PA}}\ \mathit{Term}(t) \wedge t = t_1 +^\bullet t_2 \rightarrow t_2 < t \wedge \mathit{Term}(t_2),$$

are trivially satisfied.

Similar schemes, when we allow terms with arbitrary number of parameters on the right-hand side of the above identities, substitution in parameters, or even nested recursive applications, will be also called definitions by structural recursion on numeric terms.

**7.4.7 Size of numeric terms.** The function $|t|$ yields the *size* of the numeric term $t$, i.e. the number of operations including variables needed to construct the term $t$. The function is defined by parameterless structural recursion on the numeric term $t$ as a p.r. function:

$$\begin{aligned} &|\mathrm{x}_i^\bullet| = 1 \\ &|n^\bullet| = 1 \\ &|t_1 +^\bullet t_2| = |t_1| + |t_2| + 1 \\ &|t_1 \times^\bullet t_2| = |t_1| + |t_2| + 1. \end{aligned}$$

**7.4.8 Denotation of numeric terms.** We now define the binary *denotation (valuation)* function $[\![t]\!]_v$ which takes the code $t$ of a numeric term $\tau$ and the assignment $v$ which is a list assigning the value $v[i]$ to the variable $x_i$ and yields the value of the term $\tau$. The function $[\![t]\!]_v$ is defined by structural recursion on the numeric term $t$ as a p.r. function:

$$\begin{aligned} &[\![\mathrm{x}_i^\bullet]\!]_v = v[i] \\ &[\![n^\bullet]\!]_v = n \\ &[\![t_1 +^\bullet t_2]\!]_v = [\![t_1]\!]_v + [\![t_2]\!]_v \\ &[\![t_1 \times^\bullet t_2]\!]_v = [\![t_1]\!]_v \times [\![t_2]\!]_v. \end{aligned}$$

For instance, if $v = \langle 10, 11, 12, 13, 0\rangle$ then

$$\big[\!\big[(\mathrm{x}_1^\bullet +^\bullet 2^\bullet) \times^\bullet \mathrm{x}_3^\bullet\big]\!\big]_v = [\![\mathrm{x}_1^\bullet +^\bullet 2^\bullet]\!]_v \times [\![\mathrm{x}_3^\bullet]\!]_v = \big([\![\mathrm{x}_1^\bullet]\!]_v + [\![2^\bullet]\!]_v\big) \times [\![\mathrm{x}_3^\bullet]\!]_v = \\ = \big(v[1] + 2\big) \times v[3] = (11 + 2) \times 13 = 169.$$

**7.4.9 The compiler and postfix machine.** In this example we give the proof of correctness of a simple compiler for numeric terms. A term is compiled into a program of a postfix machine and then the program is executed.

The instructions are defined with the help of four pair constructors:

$$\begin{aligned} LOAD(i) &= \langle 0, i\rangle \\ PUSH(n) &= \langle 1, n\rangle \\ ADD &= \langle 2, 0\rangle \\ MULT &= \langle 3, 0\rangle. \end{aligned}$$

A program of the machine is just a list of instructions.

Numeric terms are compiled into programs with the help of $Cmp(t)$. The compilation function is defined by structural recursion on numeric terms as a p.r. function:

$$\begin{aligned} &Cmp(\mathrm{x}_i^\bullet) = \langle LOAD(i), 0\rangle \\ &Cmp(n^\bullet) = \langle PUSH(n), 0\rangle \\ &Cmp(t_1 +^\bullet t_2) = Cmp(t_1) \oplus Cmp(t_2) \oplus \langle ADD, 0\rangle \\ &Cmp(t_1 \times^\bullet t_2) = Cmp(t_1) \oplus Cmp(t_2) \oplus \langle MULT, 0\rangle. \end{aligned}$$

For instance, the following is the compiled program

$$\langle LOAD(1), PUSH(2), ADD, LOAD(3), MULT, 0\rangle$$

for (the code of) the numeric term $(x_1 + 2) \times x_3$.

The operation of the postfix machine itself is described by the ternary function $Run(p, v, s)$, where $p$ is a program, $v$ is an assignment (environment), and $s$ is a list of values (I/O stack). The function $Run(p, v, s)$ is defined by recursion on the list $p$ with substitution in the parameter $s$ as a p.r. function:

$$\begin{aligned} &Run(0, v, \langle t, s\rangle) = t \\ &Run(\langle LOAD(i), p\rangle, v, s) = Run(p, v, \langle v[i], s\rangle) \\ &Run(\langle PUSH(n), p\rangle, v, s) = Run(p, v, \langle n, s\rangle) \\ &Run(\langle ADD, p\rangle, v, \big\langle t_2, t_1, s\big\rangle) = Run(p, v, \langle t_1 + t_2, s\rangle) \\ &Run(\langle MULT, p\rangle, v, \langle t_2, t_1, s\rangle) = Run(p, v, \langle t_1 \times t_2, s\rangle). \end{aligned}$$

Note that the other parameter $v$ does not change in recursion.

Correctness of the compiler is expressed by the following formula:

$$\vdash_{\mathrm{PA}} \; Term(t) \to Run(Cmp(t), v, 0) = [\![t]\!]_v \,. \tag{1}$$

In order to prove it we need the following auxiliary claim:

$$\vdash_{\mathrm{PA}} \; \mathit{Term}(t) \to \forall p \forall s \big( \mathit{Run}(\mathit{Cmp}(t) \oplus p, v, s) = \mathit{Run}(p, v, \langle [\![ t ]\!]_v, s \rangle) \big). \tag{2}$$

This is proved by structural induction on the numeric term $t$. So take any numbers $p, s$ and continue by case analysis on the numeric term $t$. If $t = \mathrm{x}_i^\bullet$ for some $i$ then we have

$$\begin{aligned} \mathit{Run}(\mathit{Cmp}(\mathrm{x}_i^\bullet) \oplus p, v, s) &= \mathit{Run}(\langle \mathit{LOAD}(i), p \rangle, v, s) = \mathit{Run}(p, v, \langle v[i], s \rangle) = \\ &= \mathit{Run}(p, v, \langle [\![ \mathrm{x}_i^\bullet ]\!]_v, s \rangle). \end{aligned}$$

If $t = t_1 +^\bullet t_2$ for some $t_1, t_2$ then we obtain

$$\begin{aligned} &\mathit{Run}(\mathit{Cmp}(t_1 +^\bullet t_2) \oplus p, v, s) = \\ &\quad = \mathit{Run}\big(\langle \mathit{Cmp}(t_1) \oplus \mathit{Cmp}(t_2) \oplus \langle \mathit{ADD}, p \rangle \rangle, v, s\big) \overset{\mathrm{IH}}{=} \\ &\quad = \mathit{Run}\big(\langle \mathit{Cmp}(t_2) \oplus \langle \mathit{ADD}, p \rangle \rangle, v, \langle [\![ t_1 ]\!]_v, s \rangle\big) \overset{\mathrm{IH}}{=} \\ &\quad = \mathit{Run}(\langle \mathit{ADD}, p \rangle, v, \langle [\![ t_2 ]\!]_v, [\![ t_1 ]\!]_v, s \rangle) = \\ &\quad = \mathit{Run}(p, v, \langle [\![ t_1 ]\!]_v + [\![ t_2 ]\!]_v, s \rangle) = \mathit{Run}(p, v, \langle [\![ t_1 +^\bullet t_2 ]\!]_v, s \rangle). \end{aligned}$$

Note that the first induction hypothesis is applied with $\mathit{Cmp}(t_2) \oplus \langle \mathit{ADD}, p \rangle$ in place of $p$ while $s$ is unchanged; and that the second induction hypothesis is applied with $\langle \mathit{ADD}, p \rangle$ and $\langle [\![ t_1 ]\!]_v, s \rangle$ in place of $p$ and $s$, respectively. The remaining cases are proved similarly.

We are now in position to prove (1). Take any term $t$ and we have

$$\mathit{Run}(\mathit{Cmp}(t), v, 0) \overset{(2)}{=} \mathit{Run}(0, v, \langle [\![ t ]\!]_v, 0 \rangle) = [\![ t ]\!]_v .$$

**7.4.10 Rearranging terms into expressions with left associated addition.** In this paragraph we give an example of a program which goes beyond structural recursion. Consider the problem of rearranging numeric terms so that the additions which they contain are associated to left (see [49]). For instance, the term $(x_1 + x_2) + (x_3 + (x_4 + x_5))$ is transformed to an equivalent term $(((x_1 + x_2) + x_3) + x_4) + x_5$ with left associated addition.

More formally, let $\mathit{Lassoc}(t)$ be a predicate holding of terms with left associated addition. The predicate is defined by course of values recursion as primitive recursive by

$$\begin{aligned} &\mathit{Lassoc}(t) \leftarrow \neg \exists t_1, t_2 \, t = t_1 +^\bullet t_2 \\ &\mathit{Lassoc}(t_1 +^\bullet t_2) \leftarrow \neg \exists t_3, t_4 \, t_2 = t_3 +^\bullet t_4 \wedge \mathit{Lassoc}(t_1). \end{aligned}$$

We are looking for a p.r. function $f(t)$ satisfying

$$\vdash_{\mathrm{PA}} \; \mathit{Term}(t) \to \mathit{Term}\, f(t) \tag{1}$$

$$\vdash_{\mathrm{PA}} \; \mathit{Term}(t) \to \mathit{Lassoc}\, f(t) \tag{2}$$

$$\vdash_{\mathrm{PA}} \; \mathit{Term}(t) \to |f(t)| = |t| \tag{3}$$

$$\vdash_{\mathrm{PA}} \; \mathit{Term}(t) \to [\![ f(t) ]\!]_v = [\![ t ]\!]_v . \tag{4}$$

The desired function is defined by

$$\begin{aligned}&f(t) = t \leftarrow \neg\exists t_1, t_2\, t = t_1 +^{\bullet} t_2\\&f(t_1 +^{\bullet} t_2) = f(t_1) +^{\bullet} t_2 \leftarrow \neg\exists t_3, t_4\, t_2 = t_3 +^{\bullet} t_4\\&f\big(t_1 +^{\bullet}(t_2 +^{\bullet} t_3)\big) = f(t_1 +^{\bullet} t_2 +^{\bullet} t_3).\end{aligned}$$

Is this a correct definition? The first two clauses are structurally recursive, but this does not hold for the third, in which the recursion goes from $t_1 +^{\bullet}(t_2 +^{\bullet} t_3)$ to $t_1 +^{\bullet} t_2 +^{\bullet} t_3$. We claim that the above definition is the definition with measure $m(t)$:

$$\begin{aligned}&m(t) = 1 \leftarrow \neg\exists t_1, t_2\, t = t_1 +^{\bullet} t_2\\&m(t_1 +^{\bullet} t_2) = m(t_1) + 2m(t_2) + 1.\end{aligned}$$

Indeed, the regularity condition for the third clause follows from:

$$\begin{aligned}m(t_1 +^{\bullet} t_2 +^{\bullet} t_3) &= m(t_1) + 2m(t_2) + 2m(t_3) + 2 <\\&< m(t_1) + 2m(t_2) + 4m(t_3) + 3 = m(t_1 +^{\bullet}(t_2 +^{\bullet} t_3)).\end{aligned}$$

Properties (1)-(4) can be proved straightforwardly by the corresponding induction principle.

## 7.5 Universal Function

**7.5.1 Introduction.** In this last section we will consider the problem of defining a universal function for primitive recursive functions. Recall that the class of primitive recursive functions is the smallest class of functions containing the initial functions $Z(x) = 0$, $S(x) = x + 1$, $I_i^n(\vec{x}) = x_i$, and it is closed under composition

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

and primitive recursion

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

In order to find a universal function for p.r. functions we need to assign indices to primitive recursive functions. This is done by arithmetization of primitive recursive function symbols.

**7.5.2 Primitive recursive function symbols.** For every $n \geq 1$, the class $\mathrm{PR}^n$ of $n$-ary *primitive recursive function symbols* (*PR-function symbols* for short) is defined inductively as follows:

- $Z \in \mathrm{PR}^1$, $S \in \mathrm{PR}^1$ and $I_i^n \in \mathrm{PR}^n$ for $1 \leq i \leq n$,

- if $h \in \mathrm{PR}^m$ and $g_1, \dots, g_m \in \mathrm{PR}^n$ then $Comp^n_m(h, g_1, \dots, g_m) \in \mathrm{PR}^n$,
- if $g \in \mathrm{PR}^n$ and $h \in \mathrm{PR}^{n+2}$ then $Rec_{n+1}(g, h) \in \mathrm{PR}^{n+1}$.

We set $\mathrm{PR} = \bigcup_{n \geq 1} \mathrm{PR}^n$.

We interpret $n$-ary PR-function symbols by $n$-ary functions. The interpretation $f^{\mathcal{N}}$ of a PR-function symbol $f$ is defined by induction on the structure of PR-function symbols as follows:

- $Z^{\mathcal{N}}$ is the zero function $Z(x) = 0$,
- $S^{\mathcal{N}}$ is the successor function $S(x) = x + 1$,
- $(I^n_i)^{\mathcal{N}}$ is the identity function $I^n_i(\vec{x}) = x_i$,
- $\big(Comp^n_m(h, g_1, \dots, g_m)\big)^{\mathcal{N}}$ is the $n$-ary function defined by composition:

$$\big(Comp^n_m(h, g_1, \dots, g_m)\big)^{\mathcal{N}}(\vec{x}) = h^{\mathcal{N}}\big(g_1^{\mathcal{N}}(\vec{x}), \dots, g_m^{\mathcal{N}}(\vec{x})\big),$$

- $\big(Rec_n(g, h)\big)^{\mathcal{N}}$ is the $n$-ary function defined by primitive recursion:

$$\begin{aligned} \big(Rec_n(g, h)\big)^{\mathcal{N}}(0, \vec{y}) &= g^{\mathcal{N}}(\vec{y}) \\ \big(Rec_n(g, h)\big)^{\mathcal{N}}(x + 1, \vec{y}) &= h^{\mathcal{N}}\big(x, \big(Rec_n(g, h)\big)^{\mathcal{N}}(x, \vec{y}), \vec{y}\big). \end{aligned}$$

In the sequel we will often drop the superscript in $f^{\mathcal{N}}$ and write shortly $f$ instead of $f^{\mathcal{N}}$.

It is easy to see that primitive recursive functions are exactly those functions which are denoted by PR-function symbols. In other words, the class of primitive recursive functions is just the set $\bigcup_{n \geq 1}\{f^{\mathcal{N}} \mid f \in \mathrm{PR}^n\}$.

**7.5.3 Arithmetization of primitive recursive function symbols.** Now we consider the problem of coding of PR-function symbols into N. The symbols are arithmetized with the help of the following pair constructors:

$$\begin{aligned} \boldsymbol{Z} &= \langle 0, 0\rangle && \text{(zero)} \\ \boldsymbol{S} &= \langle 1, 0\rangle && \text{(successor)} \\ \boldsymbol{I}^n_i &= \langle 2, n, i\rangle && \text{(identities)} \\ \boldsymbol{\langle} g, gs \boldsymbol{\rangle} &= \langle 3, g, gs\rangle && \text{(contraction)} \\ \boldsymbol{Comp}^n_m(h, gs) &= \langle 4, n, m, h, gs\rangle && \text{(composition)} \\ \boldsymbol{Rec}_n(g, h) &= \langle 5, n, g, h\rangle. && \text{(primitive recursion)} \end{aligned}$$

The arities of the constructors are as shown in their definitions. We postulate that the binary constructor $\boldsymbol{\langle} g, gs \boldsymbol{\rangle}$ groups to the right and has the same precedence as the pairing function $\langle x, y\rangle$.

The assignment of the code $\ulcorner f \urcorner$ to the PR-function symbol $f$ is defined inductively on the structure of PR-function symbols:

$$\begin{aligned}
\ulcorner Z \urcorner &= \boldsymbol{Z} \\
\ulcorner S \urcorner &= \boldsymbol{S} \\
\ulcorner I_i^n \urcorner &= \boldsymbol{I}_i^n \\
\ulcorner Comp_m^n(h, g_1, \dots, g_m) \urcorner &= \boldsymbol{Comp}_m^n\big(\ulcorner h \urcorner, \boldsymbol{\langle} \ulcorner g_1 \urcorner, \dots, \ulcorner g_m \urcorner \boldsymbol{\rangle}\big) \\
\ulcorner Rec_n(g, h) \urcorner &= \boldsymbol{Rec}_n(\ulcorner g \urcorner, \ulcorner h \urcorner).
\end{aligned}$$

Note that the binary operator $\boldsymbol{\langle} g, gs \boldsymbol{\rangle}$ plays a similar role as the pairing function $\langle x, y \rangle$ does for $n$-tuples of natural numbers. Its sole purpose is to represent the $m$-tuple $\ulcorner g_1 \urcorner, \dots, \ulcorner g_m \urcorner$ of the codes of PR-function symbols by its *contraction* which is the number of the form $\boldsymbol{\langle} \ulcorner g_1 \urcorner, \dots, \ulcorner g_m \urcorner \boldsymbol{\rangle}$.

**7.5.4 Interpreter of primitive recursive functions.** In this paragraph we give a definition of a binary function $e \bullet x$ which effectively realizes the interpretation of PR-function symbols. The application $\ulcorner f \urcorner \bullet \langle x_1, \dots, x_n \rangle$ takes the code of an $n$-ary PR-function symbol $f$ and the contraction of an $n$-tuple $x_1, \dots, x_n$ of numbers, and yields the number $f(x_1, \dots, x_n)$ as the result, i.e.

$$\ulcorner f \urcorner \bullet \langle x_1, \dots, x_n \rangle = f(x_1, \dots, x_n).$$

To improve readability we will write $e_1 \bullet e_2 \bullet x$ instead of $e_1 \bullet (e_2 \bullet x)$, that is we let the operator associates right.

The interpreter $e \bullet x$ of primitive recursive functions is defined by

$$\begin{aligned}
&\boldsymbol{Z} \bullet x = 0 \\
&\boldsymbol{S} \bullet x = x + 1 \\
&\boldsymbol{I}_i^n \bullet x = [x]_i^n \\
&\boldsymbol{\langle} g, gs \boldsymbol{\rangle} \bullet x = \langle g \bullet x, gs \bullet x \rangle \\
&\boldsymbol{Comp}_m^n(h, gs) \bullet x = h \bullet gs \bullet x \\
&\boldsymbol{Rec}_n(g, h) \bullet \langle 0, y \rangle = g \bullet y \\
&\boldsymbol{Rec}_n(g, h) \bullet \big\langle x + 1, y \big\rangle = h \bullet \big\langle x, \boldsymbol{Rec}_n(g, h) \bullet \langle x, y \rangle, y \big\rangle.
\end{aligned}$$

This is an example of regular recursive definition which is into the lexicographical order $(x_1, y_1) <_{\text{lex}} (x_2, y_2)$ of natural numbers. This is because the first argument of each recursive application except the one in the last recursive clause goes down. In the recursive application of the last recursive clause the first argument $\boldsymbol{Rec}_n(g, h)$ stays the same and the second argument goes down since $\langle x, y \rangle < \langle x + 1, y \rangle$. We have therefore

$$\big(\boldsymbol{Rec}_n(g, h), \langle x, y \rangle\big) <_{\text{lex}} \big(\boldsymbol{Rec}_n(g, h), \langle x + 1, y \rangle\big).$$

By the results of Sect. 5.3, we can see that the interpreter is effectively computable; the closed form of the above definition constitutes a program for the reduction model discussed in Sect. 5.3. As we will see later the function is not primitive recursive.

**7.5.5 Enumeration functions.** The $(n{+}1)$-ary function $\theta$ is said to be an *enumeration function* for the class of $n$-ary functions $\mathcal{F}$ if we have the following for every $n$-ary function $f$:

> $f \in \mathcal{F}$ iff there is a number $e$ such that
> $$f(x_1, \ldots, x_n) = \theta(e, x_1, \ldots, x_n)$$
> holds for all numbers $x_1, \ldots, x_n$.

The function $\theta$ is often called the *universal function* for the class $\mathcal{F}$.

In the next theorem we will prove for every $n \geq 1$ that the $(n{+}1)$-ary function $U_n$ explicitly defined by

$$U_n(e, x_1, \ldots, x_n) = e \bullet \langle x_1, \ldots, x_n \rangle$$

is the enumeration function for the class of $n$-ary primitive recursive functions. Since the function $e \bullet x$ is effectively computable, so is $U_n$. Note also that $U_2$ and $e{\bullet}x$ are the same functions. In the sequel we will often abbreviate $U_1(e, x)$ to $U(e, x)$.

**7.5.6 Enumeration theorem.** *For every $n \geq 1$, the $U_n$ is an effectively computable function enumerating the class of $n$-ary primitive recursive functions.*

*Proof.* We wish to prove that the following holds for every $n$-ary function $f$:

> the function $f$ is primitive recursive iff there is a number $e$ such that
> $$f(x_1, \ldots, x_n) = U_n(e, x_1, \ldots, x_n)$$
> for every $x_1, \ldots, x_n$.

For the proof of the ($\Rightarrow$)-part of the claim it suffices to show that for every $n$-ary PR-function symbol $f$ and every $x_1, \ldots, x_n$ we have

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

This is proved by induction on the structure of PR-function symbols. So take any $n$-ary PR-function symbol $f$, any $n$-tuple $\vec{x}$ of numbers, and continue by the case analysis of $f$. The cases when $f \equiv Z$ or $f \equiv S$ are straightforward. Now, if $f \equiv \mathit{Comp}^n_m(h, g_1, \ldots, g_m)$ then we have

$$\begin{aligned}
&\ulcorner \mathit{Comp}^n_m(h, g_1, \ldots, g_m) \urcorner \bullet \langle \vec{x} \rangle = \boldsymbol{Comp}^n_m(\ulcorner h \urcorner, \boldsymbol{\langle} \ulcorner g_1 \urcorner, \ldots, \ulcorner g_m \urcorner \boldsymbol{\rangle}) \bullet \langle \vec{x} \rangle = \\
&\quad = \ulcorner h \urcorner \bullet \boldsymbol{\langle} \ulcorner g_1 \urcorner, \ldots, \ulcorner g_m \urcorner \boldsymbol{\rangle} \bullet \langle \vec{x} \rangle = \ulcorner h \urcorner \bullet \big\langle \ulcorner g_1 \urcorner \bullet \langle \vec{x} \rangle, \ldots, \ulcorner g_m \urcorner \bullet \langle \vec{x} \rangle \big\rangle \overset{\text{IH}}{=} \\
&\quad = h\big(g_1(\vec{x}), \ldots, g_m(\vec{x})\big) = \mathit{Comp}^n_m(h, g_1, \ldots, g_m)(\vec{x}).
\end{aligned}$$

Finally, if $f \equiv \mathit{Rec}_n(g, h)$ then $\vec{x} \equiv z, \vec{y}$ for some $z$ and a non-empty $\vec{y}$. The desired property

$$\ulcorner Rec_n(g,h)\urcorner \bullet \langle z,\vec{y}\rangle = Rec_n(g,h)(z,\vec{y})$$

is proved by (inner) induction on $z$. In the base case we have

$$\begin{aligned}\ulcorner Rec_n(g,h)\urcorner \bullet \langle 0,\vec{y}\rangle &= \boldsymbol{Rec}_n(\ulcorner g\urcorner,\ulcorner h\urcorner)\bullet\langle 0,\vec{y}\rangle = \ulcorner g\urcorner\bullet\langle\vec{y}\rangle \overset{\text{outer IH}}{=}\\ &= g(\vec{y}) = Rec_n(g,h)(0,\vec{y}).\end{aligned}$$

In the induction step we have

$$\begin{aligned}&\ulcorner Rec_n(g,h)\urcorner \bullet \langle z+1,\vec{y}\rangle = \boldsymbol{Rec}_n(\ulcorner g\urcorner,\ulcorner h\urcorner)\bullet\langle z+1,\vec{y}\rangle =\\ &\quad= \ulcorner h\urcorner \bullet \big\langle z, \boldsymbol{Rec}_n(\ulcorner g\urcorner,\ulcorner h\urcorner)\bullet\langle z,\vec{y}\rangle,\vec{y}\big\rangle \overset{\text{outer IH}}{=}\\ &\quad= h\big(z,\boldsymbol{Rec}_n(\ulcorner g\urcorner,\ulcorner h\urcorner)\bullet\langle z,\vec{y}\rangle,\vec{y}\big) = h\big(z,\ulcorner Rec_n(g,h)\urcorner\bullet\langle z,\vec{y}\rangle,\vec{y}\big) \overset{\text{inner IH}}{=}\\ &\quad= h\big(z,Rec_n(g,h)(z,\vec{y}),\vec{y}\big) = Rec_n(g,h)(z+1,\vec{y}).\end{aligned}$$

This finishes the proof for the case when $f \equiv Rec_n(g,h)$.

In the proof of ($\Leftarrow$)-part of the claim it suffices to show that the unary functions $\phi_e$ explicitly defined by

$$\phi_e(x) = e \bullet x$$

are primitive recursive functions. This is proved by complete induction on $e$. So take any $e$ and continue by case analysis on $e$. If $e = \boldsymbol{Z}$, $e = \boldsymbol{S}$ or $e = \boldsymbol{I}_i^n$ then the following explicit definitions listed in that order

$$\begin{aligned}\phi_e(x) &= 0\\ \phi_e(x) &= x+1\\ \phi_e(x) &= [x]_i^n\end{aligned}$$

are derivations of $\phi_e$ as a primitive recursive function. If $e = \boldsymbol{Comp}_m^n(e_1,e_2)$ for some $e_1$ and $e_2$ then the functions $\phi_{e_1}$ and $\phi_{e_2}$ are primitive recursive by IH and we derive $\phi_e$ as a primitive recursive function by composition:

$$\phi_e(x) = \phi_{e_1}\,\phi_{e_2}(x).$$

If $e = \boldsymbol{Rec}_n(e_1,e_2)$ for some $e_1$ and $e_2$ then the functions $\phi_{e_1}$ and $\phi_{e_2}$ are primitive recursive by IH and we derive $\phi_e$ as a primitive recursive function by the following course of values recursive definition:

$$\begin{aligned}\phi_e(0) &= 0\\ \phi_e\langle 0,y\rangle &= \phi_{e_1}(y)\\ \phi_e\langle x+1,y\rangle &= \phi_{e_2}\big\langle x,\phi_e\langle x,y\rangle,y\big\rangle.\end{aligned}$$

If neither of the above cases applies then we derive $\phi_e$ as a primitive recursive function by explicit definition:

$$\phi_e(x) = 0. \qquad \square$$

**7.5.7 Enumeration functions are not primitive recursive.** We already know that the enumeration functions $U_n$ are effectively computable. In this paragraph none of the enumeration functions is primitive recursive. We prove this fact for the case when $n = 1$ and left the proof the general result to the reader.

The standard proof uses a diagonal argument. Suppose by contradiction that the enumeration function $U(e, x)$ is primitive recursive. Then also the explicitly defined function $f$:

$$f(x) = U(x, x) + 1 \tag{1}$$

is a primitive recursive function. By the Enumeration Theorem there is a number $e$ such that for every number $x$ we have

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

We obtain contradiction from

$$f(e) \overset{(1)}{=} U(e, e) + 1 \overset{(2)}{=} f(e) + 1.$$

**7.5.8 Primitive recursive indices.** We say that a number $e$ is a *primitive recursive index* of the $n$-ary function $f$ if

$$f(x_1, \dots, x_n) = U_n(e, x_1, \dots, x_n)$$

for all numbers $x_1, \dots, x_n$. This can be expressed equivalently by

$$f(x_1, \dots, x_n) = e \bullet \langle x_1, \dots, x_n \rangle.$$

Primitive index is said to be *well-formed* if it is a code of some PR-function symbol. As a simple corollary of the Enumeration Theorem we can see that a function is primitive recursive iff it has a primitive recursive index.

For every $n \geq 1$ by $\phi_e^{(n)}$ we denote the $n$-ary primitive recursive function with the primitive recursive index $e$. Note that we then have

$$\phi_e^{(n)}(x_1, \dots, x_n) = U_n(e, x_1, \dots, x_n).$$

and

$$\phi_e^{(n)}(x_1, \dots, x_n) = e \bullet \langle x_1, \dots, x_n \rangle.$$

By the Enumeration Theorem, an $n$-ary function $f$ is primitive recursive iff $f = \phi_e^{(n)}$ for some $e$. In the sequel we will often abbreviate $\phi_e^{(1)}(\vec{x})$ to $\phi_e(\vec{x})$.

**7.5.9 Indices of initial primitive recursive functions.** We clearly have

$$\begin{aligned}&\vdash_{\mathrm{PA}} \boldsymbol{Z} \bullet x = 0\\ &\vdash_{\mathrm{PA}} \boldsymbol{S} \bullet x = x + 1\\ &\vdash_{\mathrm{PA}} \boldsymbol{I}^n_i \bullet \langle x_1, \ldots, x_n\rangle = x_i\end{aligned}$$

and therefore the numbers $\boldsymbol{Z}$, $\boldsymbol{S}$ and $\boldsymbol{I}^n_i$ are p.r. indices of the initial p.r. functions $Z(x) = 0$, $S(x) = x + 1$ and $I^n_i(\vec{x}) = x_i$, respectively.

**7.5.10 Indices of constant functions.** There is a unary p.r. function $\boldsymbol{C}_m$ which yields p.r. indices of unary constant functions $C_m(x) = m$, i.e. we have

$$\phi_{\boldsymbol{C}_m}(x) = m$$

for every $x$. The property can be easily expressed in the language PA by

$$\vdash_{\mathrm{PA}} \boldsymbol{C}_m \bullet x = m \tag{1}$$

Note that we have

$$\begin{aligned}&\vdash_{\mathrm{PA}} C_0(x) = 0\\ &\vdash_{\mathrm{PA}} C_{m+1}(x) = S\,C_m(x)\end{aligned}$$

and thus the function $\boldsymbol{C}_m$ has the following primitive recursive definition:

$$\begin{aligned}\boldsymbol{C}_0 &= \boldsymbol{Z}\\ \boldsymbol{C}_{m+1} &= \boldsymbol{Comp}^1_1(\boldsymbol{S}, \boldsymbol{C}_m).\end{aligned}$$

There is a binary primitive recursive function $\boldsymbol{C}^n_m$ which yields p.r. indices of $n$-ary constant functions $C^n_m(\vec{x}) = m$, i.e. we have

$$\phi_{\boldsymbol{C}^n_m}(x_1, \ldots, x_n) = m.$$

This is expressed in the language PA by

$$\vdash_{\mathrm{PA}} \boldsymbol{C}^n_m \bullet \langle x_1, \ldots, x_n\rangle = m \tag{2}$$

Note that we have

$$\vdash_{\mathrm{PA}} C^n_m(x_1, \ldots, x_n) = C_m\, I^n_1(x_1, \ldots, x_n)$$

and thus the function $\boldsymbol{C}^n_m$ has the following primitive recursive definition:

$$\boldsymbol{C}^n_m = \boldsymbol{Comp}^n_1(\boldsymbol{C}_m, \boldsymbol{I}^n_1).$$

*Verification.* (1): By induction on $m$. The base case is trivial and the induction step follows from

$$\boldsymbol{C}_{m+1} \bullet x = \boldsymbol{Comp}^1_1(\boldsymbol{S}, \boldsymbol{C}_m) \bullet x = \boldsymbol{S} \bullet \boldsymbol{C}_m \bullet x \overset{\text{IH}}{=} \boldsymbol{S} \bullet m = m + 1.$$

(2): It follows from

$$\begin{aligned} \boldsymbol{C}^n_m \bullet \langle x_1, \dots, x_n\rangle &= \boldsymbol{Comp}^n_1(\boldsymbol{C}_m, \boldsymbol{I}^n_1) \bullet \langle x_1, \dots, x_n\rangle = \\ &= \boldsymbol{C}_m \bullet \boldsymbol{I}^n_1 \bullet \langle x_1, \dots, x_n\rangle = \boldsymbol{C}_m \bullet x_1 \overset{(1)}{=} m. \end{aligned}$$ □

**7.5.11 Explicit definitions.** Now we consider the problem of finding p.r. indices of functions defined by explicit definitions:

$$f(x_1, \dots, x_n) = \tau[x_1, \dots, x_n] \tag{1}$$

We suppose here that the term $\tau$ is built up from from variables and constants by applications of p.r. function symbols.

The primitive recursive index $\ulcorner\lambda\vec{x}.\tau\urcorner$ of the function $f$ defined by (1) is constructed inductively on the structure of the term $\tau$ as follows:

$$\begin{aligned} \ulcorner\lambda\vec{x}.x_i\urcorner &= \boldsymbol{I}^n_i \\ \ulcorner\lambda\vec{x}.m\urcorner &= \boldsymbol{C}^n_m \\ \ulcorner\lambda\vec{x}.g(\tau_1, \dots, \tau_m)\urcorner &= \boldsymbol{Comp}^n_m\big(\ulcorner g\urcorner, \boldsymbol{\langle}\ulcorner\lambda\vec{x}.\tau_1\urcorner\boldsymbol{,} \dots\boldsymbol{,} \ulcorner\lambda\vec{x}.\tau_m\urcorner\boldsymbol{\rangle}\big). \end{aligned}$$

We have

$$\vdash_{\text{PA}} \ulcorner\lambda\vec{x}.\tau\urcorner \bullet \langle x_1, \dots, x_n\rangle = \tau[x_1, \dots, x_n]. \tag{2}$$

*Proof.* Property (2) is proved by (meta-)induction on the structure of the term $\tau$. If $\tau \equiv x_i$ then we have

$$\ulcorner\lambda\vec{x}.x_i\urcorner \bullet \langle x_1, \dots, x_n\rangle = \boldsymbol{I}^n_i \bullet \langle x_1, \dots, x_n\rangle = x_i.$$

Similarly, if $\tau \equiv m$ then we have

$$\ulcorner\lambda\vec{x}.m\urcorner \bullet \langle x_1, \dots, x_n\rangle = \boldsymbol{C}^n_m \bullet \langle x_1, \dots, x_n\rangle \overset{7.5.10(2)}{=} m.$$

Finally, if $\tau \equiv g(\tau_1, \dots, \tau_m)$ then we have

$$\begin{aligned} &\ulcorner\lambda\vec{x}.g(\tau_1, \dots, \tau_m)\urcorner \bullet \langle x_1, \dots, x_n\rangle = \\ &\quad= \boldsymbol{Comp}^n_m\big(\ulcorner g\urcorner, \boldsymbol{\langle}\ulcorner\lambda\vec{x}.\tau_1\urcorner, \dots\boldsymbol{,} \ulcorner\lambda\vec{x}.\tau_m\urcorner\boldsymbol{\rangle}\big) \bullet \langle x_1, \dots, x_n\rangle = \\ &\quad= \ulcorner g\urcorner \bullet \boldsymbol{\langle}\ulcorner\lambda\vec{x}.\tau_1\urcorner\boldsymbol{,} \dots\boldsymbol{,} \ulcorner\lambda\vec{x}.\tau_m\urcorner\boldsymbol{\rangle} \bullet \langle x_1, \dots, x_n\rangle = \\ &\quad= \ulcorner g\urcorner \bullet \big\langle\ulcorner\lambda\vec{x}.\tau_1\urcorner \bullet \langle x_1, \dots, x_n\rangle, \dots, \ulcorner\lambda\vec{x}.\tau_m\urcorner \bullet \langle x_1, \dots, x_n\rangle\big\rangle \overset{\text{IH}}{=} \\ &\quad= \ulcorner g\urcorner \bullet \langle\tau_1[x_1, \dots, x_n], \dots, \tau_m[x_1, \dots, x_n]\rangle = \\ &\quad= g(\tau_1, \dots, \tau_m)[x_1, \dots, x_n]. \end{aligned}$$ □

**7.5.12 Example.** Addition is defined by primitive recursion (cf. Par. 1.2.11)

$$0 + y = I(y)$$
$$(x+1) + y = h(x, x+y, y)$$

from the identity function $I(y) = y$ and the ternary function $h(x,a,y) = S(a)$. By Par. 7.5.11, $\ulcorner\lambda x_1x_2x_3.S(x_2)\urcorner$ is a p.r. index of $h$ and therefore the number

$$\boldsymbol{Rec}_2(\boldsymbol{I}^1_1, \ulcorner\lambda x_1x_2x_3.S(x_2)\urcorner)$$

is a p.r. index of the addition. Primitive recursive indices of some other p.r. functions (e.g. multiplication) are obtained similarly (cf. Sect. 1.2).

**7.5.13 Parametric function.** The binary *parametric* function $e/x$ takes a p.r. index $e$ of a binary p.r. $f$ and a number $x$ and yields a p.r. index of the unary p.r. function $g$ such that $g(y) = f(x,y)$, i.e. we have

$$\phi_{e/x}(y) = \phi^{(2)}_e(x,y).$$

This is expressed in PA by

$$\vdash_{\text{PA}} (e/x) \bullet y = e \bullet \langle x, y\rangle. \tag{1}$$

The parametric function is defined explicitly as a p.r. function by

$$e/x = \boldsymbol{Comp}^1_2\big(e, \boldsymbol{\langle} \boldsymbol{C}_x, \boldsymbol{I}^1_1 \boldsymbol{\rangle}\big).$$

*Verification.* Property (1) follows from

$$(e/x) \bullet y = \boldsymbol{Comp}^1_2\big(e, \boldsymbol{\langle} \boldsymbol{C}_x, \boldsymbol{I}^1_1 \boldsymbol{\rangle}\big) \bullet y = e \bullet \boldsymbol{\langle} \boldsymbol{C}_x, \boldsymbol{I}^1_1 \boldsymbol{\rangle} \bullet y =$$
$$= e \bullet \langle \boldsymbol{C}_x \bullet y, \boldsymbol{I}^1_1 \bullet y\rangle \overset{7.5.10(1)}{=} e \bullet \langle x, y\rangle. \qquad \square$$

**7.5.14 S-m-n theorem.** For every $m, n \geq 1$, there exists an $(m+1)$-ary primitive recursive function $\mathrm{s}^m_n(e, \vec{x})$ such that

$$\phi^{(n)}_{\mathrm{s}^m_n(e,\vec{x})}(\vec{y}) = \phi^{(m+n)}_e(\vec{x}, \vec{y}).$$

This is expressed in PA by

$$\vdash_{\text{PA}} \mathrm{s}^m_n(e, x_1, \dots, x_m) \bullet \langle y_1, \dots, y_n\rangle = e \bullet \langle x_1, \dots, x_m, y_1, \dots, y_n\rangle. \tag{1}$$

The function $\mathrm{s}^m_n(e, \vec{x})$ is defined explicitly as p.r. function by

$$\mathrm{s}^m_n(e, x_1, \dots, x_m) = \boldsymbol{Comp}^n_{m+n}\big(e, \boldsymbol{\langle} \boldsymbol{C}^n_{x_1}, \dots, \boldsymbol{C}^n_{x_m}, \boldsymbol{I}^n_1, \dots, \boldsymbol{I}^n_n \boldsymbol{\rangle}\big).$$

*Verification.* Property (1) is proved as follows ($\vec{y} \equiv y_1, \dots, y_m$):

$$\begin{aligned}
&\mathrm{s}^m_n(e,x_1,\dots,x_m)\bullet\langle y_1,\dots,y_n\rangle = \\
&\quad= \boldsymbol{Comp}^n_{m+n}\big(e,\boldsymbol{\langle} \boldsymbol{C}^n_{x_1},\dots,\boldsymbol{C}^n_{x_m},\boldsymbol{I}^n_1,\dots,\boldsymbol{I}^n_n\boldsymbol{\rangle}\big)\bullet\langle y_1,\dots,y_n\rangle = \\
&\quad= e\bullet\boldsymbol{\langle} \boldsymbol{C}^n_{x_1},\dots,\boldsymbol{C}^n_{x_m},\boldsymbol{I}^n_1,\dots,\boldsymbol{I}^n_n\boldsymbol{\rangle}\bullet\langle y_1,\dots,y_n\rangle = \\
&\quad= e\bullet\big\langle \boldsymbol{C}^n_{x_1}\bullet\langle\vec{y}\rangle,\dots,\boldsymbol{C}^n_{x_m}\bullet\langle\vec{y}\rangle,\boldsymbol{I}^n_1\bullet\langle\vec{y}\rangle,\dots,\boldsymbol{I}^n_n\bullet\langle\vec{y}\rangle\big\rangle \overset{7.5.10(2)}{=} \\
&\quad= e\bullet\langle x_1,\dots,x_m,y_1,\dots,y_n\rangle. \qquad \square
\end{aligned}$$

**7.5.15 Self-reproducing machine.** We conclude this section by solving the following question. Does exist a primitive recursive function which produces its own description? More precisely, we wish to find a unary recursive function $\phi_e(x)$ which yields its own well-formed index $e$ for every input $x$, i.e. we would like to have

$$\phi_e(x) = e,$$

or equivalently

$$\vdash_{\mathrm{PA}} e\bullet x = e. \tag{1}$$

For that consider the following unary p.r. function defined explicitly by

$$g(y) = \boldsymbol{Comp}^1_1(y, \boldsymbol{C}_y). \tag{2}$$

Let $\ulcorner g\urcorner$ be one of its well-formed p.r. indices. Then the number

$$e = g(\ulcorner g\urcorner) \tag{3}$$

is a well-formed index satisfying (1):

$$\begin{aligned}
&e\bullet x \overset{(3)}{=} g(\ulcorner g\urcorner)\bullet x \overset{(2)}{=} \boldsymbol{Comp}^1_1(\ulcorner g\urcorner, \boldsymbol{C}_{\ulcorner g\urcorner})\bullet x = \ulcorner g\urcorner\bullet \boldsymbol{C}_{\ulcorner g\urcorner}\bullet x \overset{7.5.10(1)}{=} \\
&\quad= \ulcorner g\urcorner\bullet\ulcorner g\urcorner \overset{\text{index}}{=} g(\ulcorner g\urcorner) \overset{(3)}{=} e.
\end{aligned}$$

# Chapter 8
# Conclusion

This thesis described logical principles behind the declarative programming language CL which comes with its own proof system for proving properties of defined functions and predicates. We have hopefully demonstrated that the seemingly weak formal theory Peano Arithmetic is sufficient to introduce a usable programming language and its verification system. We hope that we have convinced the reader that we can code the basic data structures needed in the computer programming into natural numbers with the level of comfort comparable to that in other declarative programming languages.

This should be contrasted with other specification-verification systems which formalization may be as strong as ZF (or even stronger), or based in category theory. Many of them are based on highly non-trivial theory of Scott's domains. As a consequence, such systems are almost impossible to teach at the introductory levels of undergraduate studies. The semantic intuition of such systems, so obvious to the students of CL, is practically non-existent.

We plan to apply the theoretical results of this work into the design of a successor to the language CL. The current system has a fixed syntax of programming constructs and permits only terminating programs as they are definitions as well. We hope that the future version of CL will allow non-termination for efficiency reason and programming with flexible user-defined discrimination constructs but still within the framework of PA.

We plan to extend our research also to the area of *modular programming*. Current version of CL implement modularity with the help of non-conservative extensions of PA in the form of theories. Pavol Voda [56, 55] have already been contemplating the ways of expressing modularity within the framework of second order arithmetic. One of the most suitable systems is $\mathrm{ACA}_0$, a subsystem of second order arithmetic with arithmetical comprehension axiom scheme. This is one of the weakest extensions of PA into second order because it is conservative over PA. This and similar systems are intensively studied in that part of mathematical logic which is called *Reverse mathematics* [46, 45].

# List of Figures

# List of Symbols

Location of each entry refers to the paragraph containing the symbol. Notations introduced in *Prerequisites and Notation* are not indexed here.

$Nodoubles(x)$ . . . 6.2.6
$Minl(x)$ . . . 6.2.7
$\#_a(x)$ . . . 6.2.8
$x \sim y$ . . . 6.2.9
$x \sqsupset y$ . . . 6.3.2
$Suffixes(x)$ . . . 6.3.2
$Mape(a, x)$ . . . 6.3.3
$y \sqsubset x$ . . . 6.3.4
$Prefixes(x)$ . . . 6.3.4
$y \subset x$ . . . 6.3.5
$Segments(x)$ . . . 6.3.5
$y \approx x[\downarrow a]$ . . . 6.3.6
$Interleave(a, x)$ . . . 6.3.6
$Mapi(a, x)$ . . . 6.3.7
$Perms(x)$ . . . 6.3.8
$a \preceq x$ . . . 6.4.2
$Ord(x)$ . . . 6.4.3
$Insert(a, x)$ . . . 6.4.5
$Isort(x)$ . . . 6.4.6
$Split(x)$ . . . 6.4.8
$Merge(x, y)$ . . . 6.4.9
$Msort(x)$ . . . 6.4.10
$\langle\rangle$ . . . 7.1.2
$\langle l \mid x \mid r\rangle$ . . . 7.1.2
$\ulcorner T \urcorner$ . . . 7.1.3
$Bt(t)$ . . . 7.1.3
$d(t)$ . . . 7.1.7
$|t|$ . . . 7.1.7
$x \in t$ . . . 7.1.8
$t_1 \trianglelefteq t_2$ . . . 7.1.9
$Preorder(t)$ . . . 7.1.11
$x \prec t$ . . . 7.2.2
$x \succ t$ . . . 7.2.2
$Bst(t)$ . . . 7.2.3
$Max(t)$ . . . 7.2.5
$t \cup \{x\}$ . . . 7.2.6
$Delmax(t)$ . . . 7.2.7
$t_1 \sqcup t_2$ . . . 7.2.8
$t \setminus \{x\}$ . . . 7.2.8
$Index(i, t)$ . . . 7.3.3
$Brt(t)$ . . . 7.3.4
$t[i]$ . . . 7.3.10
$t[i := x]$ . . . 7.3.11
$New(n)$ . . . 7.3.12
$Insfirst(t, x)$ . . . 7.3.13
$Delfirst(t)$ . . . 7.3.14
$Inslast(n, t, x)$ . . . 7.3.15
$Dellast(n, t)$ . . . 7.3.16
$\mathrm{x}_i^\bullet$ . . . 7.4.2
$n^\bullet$ . . . 7.4.2
$t_1 +^\bullet t_2$ . . . 7.4.2
$t_1 \times^\bullet t_2$ . . . 7.4.2
$\ulcorner \tau \urcorner$ . . . 7.4.3
$Term(t)$ . . . 7.4.3
$|t|$ . . . 7.4.7
$[\![t]\!]_v$ . . . 7.4.8
$LOAD(i)$ . . . 7.4.9
$PUSH(n)$ . . . 7.4.9
$ADD$ . . . 7.4.9
$MULT$ . . . 7.4.9
$Cmp(t)$ . . . 7.4.9
$Run(p, v, s)$ . . . 7.4.9
$Lassoc(t)$ . . . 7.4.10
$\mathrm{PR}^n$ . . . 7.5.2
$Comp_m^n(h, g_1, \ldots, g_m)$ . . . 7.5.2
$Rec_{n+1}(g, h)$ . . . 7.5.2
$\mathrm{PR}$ . . . 7.5.2
$f^{\mathcal{N}}$ . . . 7.5.2
$\ulcorner f \urcorner$ . . . 7.5.3
$\boldsymbol{Z}$ . . . 7.5.3
$\boldsymbol{S}$ . . . 7.5.3
$\boldsymbol{I}_i^n$ . . . 7.5.3
$\boldsymbol{\langle} g, gs \boldsymbol{\rangle}$ . . . 7.5.3
$\boldsymbol{Comp}_m^n(h, gs)$ . . . 7.5.3
$\boldsymbol{Rec}_n(g, h)$ . . . 7.5.3
$e \bullet x$ . . . 7.5.4
$U_n(e, \vec{x})$ . . . 7.5.5
$U(e, \vec{x})$ . . . 7.5.5
$\phi_e^{(n)}(x_1, \ldots, x_n)$ . . . 7.5.8
$\phi_e(\vec{x})$ . . . 7.5.8
$\boldsymbol{C}_m$ . . . 7.5.10
$\boldsymbol{C}_m^n$ . . . 7.5.10
$\ulcorner \lambda\vec{x}.\tau \urcorner$ . . . 7.5.11
$e/x$ . . . 7.5.13
$\mathrm{s}_n^m(e, \vec{x})$ . . . 7.5.14

# Bibliography

1. Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 170–176. Springer, 2000.
2. J. Barwise. An introduction to first-order logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North-Holland, 1977.
3. Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.
4. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.
5. W. Braun and M. Rem. A logarithmic implementation of flexible arrays, 1983. Memorandum MR83/4. Eindhoven University of Technology.
6. R. M. Burstall, D. B. MacQueen, and D. T. Sanella. Hope: an experimental applicative language. In *Proceedings of the ACM Lisp Conference*, 1980.
7. R.J. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986.
8. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at http://www.csl.sri.com/wift-tutorial.html.
9. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems.* Springer, New York, 2005.
10. J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In M. K. Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, 1979. Reprinted in *Software Specification Techniques*, N. Gehani and A. McGettrick, editors, Addison-Wesley, 1985, pages 391-420.
11. Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* Foundations of Computing Series. The MIT Press, Cambridge, MA, 1996.
12. M. J. C. Gordon and T. F. Melhalm, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.
13. David A. Greve, Matt Kaufmann, Panagiotis Manolios, J. Strother Moore, Sandip Ray, José-Luis Ruiz-Reina, R. O. B. Sumners, Daron Vroon, and Matthew Wilding. Efficient execution in an automated reasoning environment. *J. Funct. Program.*, 18(1):15–46, 2008.
14. The RAISE Language Group. *The RAISE Specification Language.* Prentice-Hall, 1992.

15. J. V. Guttag and J.J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science. Springer Verlag, 1993.
16. P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic.* Springer Verlag, 1993.
17. G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial. Technical Report 178, INRIA, July 1995.
18. C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1990.
19. M. Kaufmann, P. Manolios, and J.S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.
20. Matt Kaufmann and J Strother Moore. A precise description of the acl2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1998.
21. S. C. Kleene. *Introduction to Metamathematics.* Wolters-Noordhoff and North-Holland, 1952.
22. J. Komara and P. J. Voda. Syntactic reduction of predicate tableaux to propositional tableaux. In P. Baumgartner, R. Haehnle, and J. Posegga, editors, *Proceedings of TABLEAUX '95*, number 918 in LNAI, pages 231–246. Springer Verlag, 1995.
23. J. Komara and P. J. Voda. On quasitautologies. In D. Galmiche, editor, *Proceedings of TABLEAUX '97*, number 1227 in LNAI, pages 231–245. Springer Verlag, 1997.
24. J. Komara and P. J. Voda. Computer programming as mathematics in a programming language and proof system CL (tutorial). In H. de Swart, editor, *Proceedings of TABLEAUX '98*, number 1397 in LNAI, pages 42–43. Springer Verlag, 1998.
25. J. Komara and P. J. Voda. Theorems of Péter and Parsons in computer programming. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Proceedings of CSL'98*, number 1584 in LNCS, pages 204–223. Springer Verlag, 1999.
26. J. Komara and P. J. Voda. Extraction of efficient programs in $I\Sigma_1$-arithmetic. Technical report, Institute of Informatics, Faculty of Mathematics and Physics, Comenius University, Bratislava, 2000.
27. J. Komara and P. J. Voda. Lecture Notes in Theory of Computability, December 2001. Unpublished manuscript.
28. J. Komara and P. J. Voda. Metamathematics of Computer Programming, May 2001. Unpublished manuscript.
29. Krstic and Matthews. Inductive invariants for nested recursion. In *IWHOLTP: 16th International Workshop on Higher Order Logic Theorem Proving and Its Applications.* LNCS, 2003.
30. Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.
31. Thomas F. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *TPHOLs*, pages 350–357. IEEE Computer Society, 1991.
32. P.M. Melliar-Smith and J. Rushby. The Enhanced HDM system for specification and verification. In *Proc. VerkShop III*, pages 41–43, February 1985. Published as ACM Software Engineering Notes, Vol. 10, No. 4, Aug. 85.
33. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.
34. Chris Okasaki. Three algorithms on braun trees. *J. Funct. Program.*, 7(6):661–666, 1997.
35. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
36. Sam Owre and Natarajan Shankar. A brief overview of PVS. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 22–27. Springer, 2008.

37. Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reasoning*, 5(3):363–397, 1989.
38. R. Péter. Konstruktion nichtrekursiver Funktionen. *Mathematische Annalen*, 111:42–60, 1935.
39. R. Péter. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annalen*, 110:612–632, 1935.
40. R. Péter. Über die mehrfache Rekursion. *Mathematische Annalen*, 113:489–527, 1937.
41. R. Péter. *Recursive Functions*. Academic Press, 1967.
42. Programming Language and Proof Assistant CL (Clausal Language). Available at http://ii.fmph.uniba.sk/cl/.
43. H. E. Rose. *Subrecursion: Functions and Hierarchies*. Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.
44. J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.
45. S. G. Simpson, editor. *Reverse Mathematics 2001*, volume 21 of *Lecture Notes in Logic*. Association for Symbolic Logic, 2005. X + 401 pages.
46. Stephen G. Simpson. *Subsystems of second order arithmetic*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1999.
47. Konrad Slind. Another look at nested recursion. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 498–518. Springer, 2000.
48. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.
49. Simon Thompson. Proof for functional programming, 1998.
50. D. A. Turner. SASL language manual. Technical report, St. Andrews University, December 1976.
51. D. A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Marinus J. Plasmeijer, editors, *FPLE*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.
52. D. A. Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.
53. P. J. Voda. Subrecursion as a basis for a feasible programming language. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL'94*, number 933 in LNCS, pages 324–338. Springer Verlag, 1995.
54. P. J. Voda. Theory of Recursive Functions & Computability: from Computer Programmer's View, 2000.
55. P. J. Voda. Grundlagenstreit in the theory of programming, February 2003.
56. P. J. Voda. What can we gain by integrating a language processor with a theorem prover?, February 2003.

# Index

Location of each entry refers to the page containing the topic. Topics introduced in *Prerequisites and Notation* are not indexed here.