Paul J. Voda

Meta-Mathematics of Computer Programming

(Preliminary Draft)

Institute of Informatics, Comenius University, Bratislava

May 16, 2001

Table of Contents

Preface VII

1. Why the Theory of Programming in Peano Arithmetic? .. 1
1.1 Effectively Computable Functions 1
1.2 Imperative vs. Declarative Programming 4
1.3 Arguments in Favor of Natural Numbers 9
1.4 Bootstrapping of PA 22
1.5 Clausal Extensions of PA 33
1.6 Limits of Provably Recursive Definitions in PA 46
1.7 Computation of Clausal Definitions 51
1.8 Data Types 53
1.9 Intensional Functionals 68
1.10 Second-Order Issues of Modularity 87
1.11 Issues Open to Further Research 87

Part I. First-Order Logic and Peano Arithmetic

2. First-Order Languages 91
2.1 Language of First-Order Logic 91

3. Propositional Logic 95
3.1 Tautologies 95
3.2 Propositional Tableaux 97
3.3 Admissible Expansion Rules 103
3.4 Tautological Consequence 112
3.5 Tableaux with Axioms 115

4. Identity Logic 121
4.1 Some Syntactic Concepts 121
4.2 Quasitautological Consequence 121
4.3 Identity Tableaux 127

5. Quantification Logic 133
5.1 Some Syntactic Concepts 133
5.2 Logical Consequence 136
5.3 Quantification Tableaux 144

6. First-order Theories 157
6.1 Theorems of Predicate Calculus 157
6.2 Extensions of Theories 162
6.3 Extensions by Explicitly Defined Predicates 164
6.4 Skolem Extensions 168
6.5 Extensions by Contextually Defined Functions 178
6.6 Extensions by Definitions 180

7. Peano Arithmetic 185
7.1 Basic Theorems in PA 185
7.2 Extensions of PA 190
7.3 Introduction of Basic Predicates into PA 192
7.4 Introduction of Basic Functions into PA 196
7.5 The Lattice of Divisibility 200

8. Recursive Bootstrapping of PA 209
8.1 Exponentiation Function 209
8.2 Primitive Recursion 223
8.3 Suitable Pairing Function 230
8.4 Course of Values Recursion with Measure 250

9. Proof Theory of PA 267

Part II. Computer Programming

10. Clausal Language 271
10.1 Clausal Definitions 271
10.2 Regular Clausal Definitions 281
10.3 The Strength of Clausal Definitions 283

11. Computation of Clausal Programs 285
11.1 Computation over Monadic Numerals 285
11.2 Computation over Binary Numerals 286
11.3 Computation over Pair Numerals 287
11.4 Computation over Mixed Numerals 288

12. Data Types 291
12.1 Pascal-Style Typing 291
12.2 ML-Style Typing 293

13. Functional Programming 295

14. Modular Programming 297
14.1 Extraction of Programs from Proofs 297

References 298

Preface

A Historical Parallel.

We have seriously considered to name this book *Principia Programmationis* in allusion to the celebrated *Principia Mathematica* by Russell and Whitehead, which was published in 1910 in a gallant attempt to reduce mathematics to logic. Mathematics had stormily developed and matured during the nineteenth century and there was a real need to lay it on firm foundations. Especially after the Russell's paradox has shattered the monumental efforts in this direction by G. Frege and seriously dented the credibility of the naive set theory of G. Cantor.

Logicians have agreed only much later that the attempt was futile because with its principle of infinity mathematics relies on extra-logical facts, i.e. facts which do not hold in all possible worlds. The Zermelo-Fraenkel set theory is nowadays accepted by most mathematicians as the foundations of mathematics.

With the benefit of this hindsight we do not attempt in this book to reduce computer programming to logic. We reduce it to the most basic of formal theories: to the formalization of arithmetic known as *Peano Arithmetic* (PA for short). We think that after a half-century of tumultuous development of computer programming languages, and after many presentations of formal theories of programming (some of which go even beyond the powers of ZF) we have succeeded in the explication of mosts aspects of modern computer programming languages in such a simple formal theory (at least as its semantics is concerned). We view our choice of PA as a kind of Occam's razor. No programming language can be without arithmetic and we believe that nothing more is needed.

We have been encouraged in this belief by the successful use of our implementation of our computer programming language CL (*C*lausal *L*anguage) in teaching of mathematical logic and theory of programming for the last four years at the Comenius University of Bratislava. We have approximately three hundred students per year taking four courses based on CL. CL is not only a programming language but also a formal system for the specification and proofs of properties of its programs.

Outline of the Text.

There is one more parallel between Principia Mathematica and the present text. Both, in their detailed development of their respective subjects (which cannot be avoided for the first time), can be perceived as tedious. For instance, the development of Peano Arithmetic in Chapters 7 and 8 is given in detail but the development is necessary for the formal justification of general recursive definitions which are considered de rigueur in modern computer programming.

We have included as extended introduction a long Chapter 1 entitled 'Why the Theory of Programming in Peano Arithmetic?' in order to shield the reader interested mainly in the general ideas of our philosophy from the rather technical development of the subject . We argue in the chapter in favor of the explication of computer programming in arithmetic. This means that an implemented computer programming language based on these concepts should use a formal system of arithmetic. Peano arithmetic is then the most natural choice. The introductory chapter covers the entire text in sufficient detail for the reader to understand our ideas and it avoids the formal apparatus of theorems and proofs although we mostly present the necessary definitions with some technical details. If the reader is well-versed in mathematical logic and/or the theory of computer programming he might be even satisfied with the argument contained in the chapter. He will perhaps go to the following chapters only for a detail or two.

The explication of computer programming in the arithmetic is formally done in the Part II of this text. We have included the Part I, entitled *First-order Logic and Peano Arithmetic*, mainly as a prerequisite for the understanding of Part II. In this part we discuss first-order theories in general and Peano arithmetic in detail. The part is included not only to make this text self-contained, but also to investigate the proof system of CL which are the signed tableaux of Smullyan. Since we extensively use the proof system for concrete proofs (as opposed to merely investigating it meta-theoretically), we present the tableaux in the positive sense as proving logical consequence rather than in the usual negative sense of deriving contradictions. In the chapters dealing with Peano arithmetic we then investigate strong schemas of extensions of PA by definitions which support the kinds of flexible recursive definitions needed in the practice of computer programming.

The Part II is entitled *Computer Programming* and it deals within the framework of PA with

- the definition of the clausal language,
- discusses in the meta-theory the computation of clausal programs,
- introduces data types in order to optimize the execution of clausal programs on computers,
- shows how to obtain the flexibility of high-order programming with functionals in the first-order Peano arithmetic,

– discusses the questions of modularization and abstract specification of programs in the language of second-order Peano arithmetic but within a framework which is conservative over PA and thus does not add any extra power (except expressivity) to it.

Targeted Audience.

We address this book to all readers interested in the theory of computer programming. We expect the reader to know at least the basics of mathematical logic and of the theory of programming languages. Of special interest to us are the experts in either mathematical logic or in the theory of computer programming who are interested in the issues of the other area.

Logicians. We address this text to the logicians interested in Computer Science, specifically in the theory of programming. Logicians, obviously, will not have any problems with the material covered in Part I, but they might appreciate that the meta-theory is completely finitary (we even give a new finitary proof of the conservativity of Skolem axioms; something not normally done in the introductory texts). Logicians are also familiar with at least two mathematically impeccably explicated programming languages, namely the language of Primitive recursive functions and of Primitive recursive functionals of finite types (Gödel's T). They also know how to compute such functions **effectively**. They might, however, be not aware of the complex set of requirements which the designers of computer programming languages have to meet.

Of course, all logicians know that a computer programming language must be able to execute the programs **efficiently**. There are additional, no less important, pragmatical requirements of readability, simplicity, easy programmability, extensibility, and modularity. These are necessary because computer programs are large and they constantly undergo change. A good computer programming language should have versatile data structures within a typed environment. It must have provisions for modular design where whole blocks of code can be replaced with a new one without affecting the rest of the program. Our logician readers might be interested in how do we propose to meet this formidable set of practical criteria within the constraints of the relatively terse language of PA and of the relative weakness of its axioms.

Computer Scientists. The second, and probably the most critical, type of audience are Computer Scientists, especially those interested in the theory of programming and mathematical logic. During the last fifty years an enormous number of computer programming languages emerged (and most of them quickly submerged) so our computer scientist reader is probably extremely sceptical of a yet another programming language. Moreover, a choice of a programming language is like a choice of a dress to wear. Everyone has a

strong set of preferences and adheres to a certain style. Just like fashion, programming languages are judged by value judgments with an apparent absence of objective criteria. Not all programming languages are based on a formal theory. This is not too unexpected as such languages are direct descendants of programs for Turing machines and the latter are basically kitchen recipes (do first this, then that, and repeat until done).

A Theory of Computer Programming.

Current programming languages based on formal theories widely differ in the strength of the theory. The theory may be as strong or even stronger as ZF. Examples are the languages based on the HOL (High-order logic) framework, or languages of the OBJ variety which are based in the category theory. A wide group of languages is based on the Scott-Ershov theory of domains of partial continuous functionals. Example is the well-known functional programming language Haskell.

We try to explicate computer programming in the style and strength in many respects similar to the Computational Logic of Boyer and Moore. The main difference is that we use as the basis a formalization of arithmetic, permit unrestricted quantification, and include the data types and functionals. Everyone is familiar with arithmetic (if not with PA) from high school. Not only is the semantic basis of the clausal language simple, but the programs in the language are just formulas of the language of Peano arithmetic. The definitions have no reserved words whatsoever.

We know that many computer programmers, discouraged by the complexities of formal theories, wholly concentrate on the writing of programs without pausing to ask what they do and what kind of objects they manipulate with. Situation is even worse because more often than not a programming language is designed only for its elegant syntax and neat features. Only after the syntactical issues of its computation (so called *operational* semantics) have been fixed, the language designer starts to worry about the semantical issues of what the programs are (so called *denotational* semantics).

This practice of syntax before semantics runs totally opposite to the typical situation in mathematics where, as J. Shoenfield in the introduction to his *Mathematical logic* so convincingly argues:

> We may now describe what a mathematician does as follows. He presents us with certain basic concepts and certain axioms about these concepts. He then explains these concepts to us until we understand them sufficiently well to see that the axioms are true. He then proceeds to define derived concepts and to prove theorems about the basic and derived concepts. The entire edifice which he constructs, consisting of basic concepts, derived concepts, axioms, and theorems, is called an *axiom system.*

This is exactly how we propose to build a framework for computer programming languages except that the edifice which we construct is a framework for programming languages. Our basic concept is that programs are computable functions. By the Thesis of Church-Turing the computable functions are Turing computable functions over natural numbers and so we decide that our programs are to be definitions of functions and predicates over natural numbers. We do not have to build the theory of natural numbers from the scratch because we can use its well-known formalization: Peano arithmetic. We still have to introduce the derived concepts as used in computer programming (data structures, types, etc.). And we also have to bootstrap PA to the extent that it admits a very flexible kind of extensions by clausal definitions of functions and predicates. Clausal definitions serve a dual purpose. They describe *extensionally* the properties of the defined functions and predicates and at the same time they serve *intensionally* as rules for the computation of these objects.

Our experience with CL, which is an implementation of these ideas, shows that our students have no problems with understanding that they are writing programs operating over natural numbers. They also know that our coding of data structures into numbers in the style of LISP is perfectly natural. Finally, they can see that we can execute CL programs with the efficiency of Pascal and C operating over symbolic data structures.

Our Contribution. The main contribution of this book is an integrated attempt to explicate many of the issues of modern computer programming within a single framework of Peano Arithmetic. Within this overall goal there are some new perspectives which are our own:

1. the design of the clausal language with extensible syntax yet totally within the language of Peano arithmetic (no reserved words),
2. a new view of abstraction types (abstract data types) which goes around the sorts and congruences needed in the usual algebraic approach,
3. the development of a fragment of second-order Peano arithmetic which is conservative over first-order PA and offers a flexible calculus of extensions (modularization),
4. calculus for the extraction of efficient programs from proofs as as a kind of programming by proving.
5. the characterization of a fragment of *bland* intensional functionals (as opposed to *curried* functionals) which can be introduced into PA with the typing comfort not available for the codes of primitive recursive (curried) functionals. For the latter one needs a non-conservative extension of PA with the function V (see below).

In the Part I dealing with logic we have contributed the following:

1. a presentation of the Henkin reduction of quantification logic to tautologies in three natural steps via quasi-tautological consequence,

2. a positive calculus of Smullyan's signed tableaux which gives them the flavor of natural deduction,
3. a simple finitary proof of conservativity of Skolem axioms,
4. a simple new Δ_0-definition of the graph $2^x = y$ of the exponentiation function,
5. a simple Δ_1-definable function V which is not provably recursive in PA. The function V is closely connected to the Wainer-Schwichtenberg fast growing hierarchies of functions.

Acknowledgments. The author would like to express his gratitude to the enormous contribution to this text by his colleague Ján Komara. The clausal language was designed as our joint project and we have also implemented it in the form of CL. Ján has actually implemented the theorem prover which comprises about 85% of approximately 100K lines of the CL's source code which is in the author's declarative language Trilogy II. Ján has also prepared excellent lecture notes for the second year undergraduate course *Program specification and verification* [15] which we teach in CL. In the course we deal with approximately 200 hundred CL programs and the students are expected to formally prove in CL over 500 hundred theorems describing properties of programs. The author strongly recommends that readers interested in more details of programming and proving with the clausal language download Ján's lecture notes.

We have worked jointly with Ján on the presentation of first-order logic and the development of positive semantic tableaux (see also [11, 12]), on the finitary proof that Skolem axioms are conservative, we have investigated the use of Parson's rule in proving properties of programs [13], and the extraction of efficient clausal programs from proofs [14].

The author would also like to thank his colleague Ján Šefránek for the unswerving support during the last eight years we have worked together in the same institute. Ján is a fine logician and computer scientist (AI) who has always believed in our goals of explicating computer programming in PA and was always willing to listen to our then semi-coherent ideas. He had been tremendously supportive especially in bad times when we were designing the courses based on our research at the same time as doing the research. We had to redesign the courses quite a few times and a few of our Computer Science faculty have explicitly disapproved of our insistence that the students should be exposed to the technology on the cutting edge of research.

1. Why the Theory of Programming in Peano Arithmetic?

This chapter is intended as extended introduction. We informally present in it the entire material of this text. We argue here in favor of declarative programming over the domain of natural numbers within the most fundamental of formal theories: Peano arithmetic. Our argument is not in favor of a single language. We argue for the use of PA as a framework for the theory of computer programming. Although we occasionally discuss our language CL it should be understood only as an example of a possible implementation of the ideas.

The style of presentation in this chapter is aimed at logicians and computer scientists. We expect the reader to be acquainted with the language and basic methods of the first-order logic as well as with the basic issues of the theory of computer programming languages. Undergraduate students will probably not be able to follow the exposition unless they have had a first reading exposure to the predicate calculus and the theory of programming languages.

An important note: The extended introduction temporarily serves also as a detailed description of the yet not written chapters and sections. This should explain a more detailed overview of these parts here. Eventually, when the missing chapters and sections of this text will be supplied, some of the explanatory material will be removed. The level of presentation of parts not yet written is usually higher and more technical than the level of finished parts because the description is meant for expert logicians and computer scientists who might be refereeing this text.

1.1 Effectively Computable Functions

1.1.1 Formal systems. Both mathematical logic and theory of computer programming study formal systems. Formal systems deal with concretely presented syntactic objects. Syntactic objects are terms, formulas, and proofs in logic and programs in computer programming.

Syntactic objects are usually assigned denotations (meaning) in some mathematical domain. However, the main reason for the study of formal systems in the logical discipline called *meta-mathematics* is to study the constructive processes connected with axioms and rules of inference. Semantic

questions of denotation are usually non-constructive whereas the manipulation of syntactic objects can be done by effectively computable processes.

1.1.2 Effectively computable functions. Mathematical logic was developed in order to give firm foundations to mathematics. As it turned out, the identification and characterization of *effectively computable* functions was an important subgoal of the foundational goal. Such functions can be evaluated by a mechanical procedure. Alan Turing gave in 1936 a completely satisfactory explication of computable functions by means of what is now called Turing machines. He used the term *computer* to designate the agent performing the mechanical procedure. Turing meant a human computer, but as it turned out some ten years later chiefly by efforts of John von Neumann and Turing himself, computers could be constructed as mechanical or electronic devices.

The theory of computer programming studies the effectively computable functions but due to the practical limitations of computer hardware it has to deal mostly with its rather small subclass of *efficiently* (feasibly) computable functions.

1.1.3 The domain of effectively computable functions. Although one can develop the theory of computable functions over integers (both negative and positive numbers), over a subset of real numbers, over word domains, or over many suitably structured domains; logicians know that it suffices to study the domain $\mathbb{N}$ of *natural numbers*: 0, 1, 2,.... This is primarily because the greatest success of the nineteen century mathematics was the discovery that all numeric domains can be defined from the domain of natural numbers. The second reason is that K. Gödel in the proof of his Incompleteness theorem convincingly demonstrated that symbolic (syntactic) objects can be coded into numbers. The process of coding is called *arithmetization* and the codes are called *Gödel numbers*.

Whether we can do the same reduction of domains of computer programming languages to $\mathbb{N}$ is by no means obvious. We will argue in Sect. 1.3 in favor of the reduction, but many computer scientists think that special domains are necessary in order to attain efficiency.

1.1.4 Notational basis for computable functions. We know from elementary school that addition and multiplication can be effectively computed when their operands are concretely presented, say, as decimal numerals. The choice of notation basis is not important when we are interested only in the effectivity and so logicians prefer the *monadic* notation where a natural number n is presented as the syntactic object, called the *monadic numeral*:

$$(0\overbrace{'\ldots'}^{n}) .$$

When we want also efficiency then the base should be larger than one and the most natural one for electronic computers is binary. *Binary numerals* are

terms constructed from the constant 0 by applications of two unary functions, called *binary successors*, which write in the postfix notation: $x\mathbf{0} = 2{\cdot}x + 0$ and $x\mathbf{1} = 2{\cdot}x + 1$. The reader will note that, for instance, we have

$$5 = 2{\cdot}(2{\cdot}(2{\cdot}0 + 1) + 0) + 1 = 0\mathbf{101} \ .$$

Leading zeroes cause notational ambiguity because $1 = 0\mathbf{1} = 00\mathbf{1} = 000\mathbf{1}\ldots$ and so we require that binary numerals of a form $\tau\mathbf{0}$ have the term $\tau \not\equiv 0$. We designate by $\underline{n}_b$ the binary numeral denoting the number n. This term yielding function satisfies:

$$\begin{aligned} \underline{0}_b &\equiv 0 \\ \underline{n\mathbf{0}}_b &\equiv \underline{n}_b\mathbf{0} \qquad \text{if } n > 0 \\ \underline{n\mathbf{1}}_b &\equiv \underline{n}_b\mathbf{1} \ . \end{aligned}$$

Here and below we designate by $\equiv$ the identity of two syntactic objects treated as finite sequences of symbols.

The reader will note that for any number x we have

$$x = 0 \vee \exists z(x = z\mathbf{0} \wedge z > 0) \vee \exists z\, x = z\mathbf{1}$$

with the numbers z uniquely determined, and with the three conditions pairwise exclusive.

1.1.5 Closure of operations over natural numbers. Addition and multiplication are *closed* over natural numbers. This means that both $x + y$ and $x{\cdot}y$ are natural numbers when x and y are. The remaining two basic arithmetical operations go outside of natural numbers, for instance $3 - 5$ and $\frac{3}{5}$. We must use instead *modified subtraction* $x \dot{-} y$ satisfying:

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

and *integer division* and *remainder* functions $x \div y$ and $x \mod y$ satisfying:

$$\begin{gathered} y > 0 \to x = (x \div y){\cdot}y + x \bmod y \wedge x \bmod y < y \\ x \div 0 = x \bmod 0 = 0 \ . \end{gathered}$$

All five basic arithmetic functions over $\mathbb{N}$ can be efficiently computed by elementary school algorithms.

Unless we explicitly say otherwise all functions and predicates will be over natural numbers. Also the variables and quantifiers will range over natural numbers.

1.2 Imperative vs. Declarative Programming

The style of programming derived from operations of Turing machines where programs are recipes for the manipulation of computer's store is called in computer science *imperative* programming. The style of programming where programs are definitions of mathematical objects such as functions or predicates is called *declarative* programming. The difference between the two styles is best explained with two presentations of the greatest common divisor function.

1.2.1 Greatest common divisor. The binary relation x *divides* y, in symbols $x \mid y$, can be defined in the language of arithmetic by

$$x \mid y \leftrightarrow \exists z\, x{\cdot}z = y \ .$$

A number z is the *greatest common divisor of* x *and* y, in symbols $\gcd(x,y) = z$, iff either $x = y = 0$ and $z = 0$ or if $x + y > 0$ and z divides both x and y and z is the largest of such numbers:

$$x + y > 0 \rightarrow z \mid x \wedge z \mid y \wedge \forall d(d \mid x \wedge d \mid y \rightarrow d \leq z) \ . \tag{1}$$

We have $1 \mid x$ and $1 \mid y$ and so when $x + y > 0$ we can effectively find the greatest common divisor by a brute force search which starts from x and goes downwards towards 1.

The ancient mathematician Euclides discovered a faster algorithm computing the function gcd which relies on the following property of divisibility:

$$y > 0 \rightarrow (z \mid x \wedge z \mid y \leftrightarrow z \mid y \wedge z \mid x \bmod y). \tag{2}$$

1.2.2 The algorithm of Euclides imperatively. The algorithm of Euclides can be expressed by an imperative program in the style of Pascal as:

$\{x = x_0 \wedge y = y_0\}$
while $y > 0$ *do* (1)
  $a := x \bmod y;\ x := y;\ y := a;$
$\{\gcd(x_0, y_0) = x\}$.

The reader will note that the text enclosed within the pair of braces $\{\}$ is not a part of the program and should be viewed as a remark.

The algorithm proceeds by operation on three registers x, y, and a located on the tape of a Turing machine or in the memory of a computer. It is basically a kitchen recipe which says that the contents of the three registers should be repeatedly swapped in the given order as long as the register y contains a positive number. When the algorithm is started with the registers x and y containing the numbers x_0 and y_0 respectively and with arbitrary contents of the contents of register a then when, and if, the swapping terminates the register x will contain the number $\gcd(x_0, y_0)$.

For arguments $x_0 = 12$ and $y_0 = 21$ we need five tests of the register y which happen in the following sequence of memory snapshots with the contents of registers given immediately above the single solid lines:

x	y	a
12	21	–
12	21	12
21	21	12
21	12	12
21	12	9
12	12	9
12	9	9
12	9	3
9	9	3
9	3	3
9	3	0
3	3	0
3	0	0

1.2.3 The algorithm of Euclides declaratively. We will discuss in Part II of this text how *Peano arithmetic*, which is a well-known logical theory formalizing the arithmetic, permits an introduction of the binary function symbol gcd by a *clausal definition*:

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

The reader will note that the symbol $\leftarrow$, which is customarily used in clausal definitions, is just a converse implication. The two clauses should be understood as axioms implicitly defining the function gcd. The definition is legal because a *measure* of the two arguments x and y goes down in recursion. The measure is the second argument y and since, the PA proves $y > 0 \rightarrow x \bmod y < y$, the second argument $x \bmod y$ of the recursive application $\gcd(y, x \bmod y)$ in the clause (2) decreases.

Incidentally, this is exactly the reason why the imperative program in Par. 1.2.2 always terminates. We, namely, go into the *while* loop only when $y > 0$. The register y will contain $x \bmod y$ just before the next iteration where x and y are the values at the moment of the entry to the *while* loop.

The declarative computation of $\gcd(12, 21)$ can be visualized as a sequence of syntactic operations, called *reductions*, which are initially applied to the term $\gcd(12, 21)$ and which apply the two clauses as rewriting rules in an attempt to simplify the term to a decimal numeral:

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

The reader will note that the computation sequence is structurally equivalent to the underlined memory snapshots for the imperative program where the two arguments of gcd play the role of the registers x and y.

1.2.4 Proving the correctness of the two gcd **programs.** We will now briefly look into the question of how to formally ensure that the imperative and declarative programs for the greatest common divisor work correctly. The function gcd is uniquely determined by satisfying its specification from Par. 1.2.1 in the following form:

$$\gcd(x,y) \mid x \wedge \gcd(x,y) \mid y \tag{1}$$

$$x + y > 0 \wedge d \mid x \wedge d \mid y \rightarrow d \leq \gcd(x,y) \tag{2}$$

$$\gcd(0,0) = 0\ . \tag{3}$$

The simplest way of proving the correctness of the imperative program is to reason in a system of formal arithmetic, say PA, where we have already demonstrated the conditions (1) through (3). From the conditions we prove the recurrences 1.2.3(1)(2). We then annotate the imperative program with comments as follows:

$\{x = x_0 \wedge y = y_0 \wedge \gcd(x,y) = \gcd(x_0,y_0)\}$
while $y > 0$ *do*
  $a := x \bmod y;\ x := y;$
  $y := a;\ \{\gcd(x,y) = \gcd(x_0,y_0)\}$
$\{y = 0 \wedge x \overset{1.2.3(1)}{=} \gcd(x,0) = \gcd(x,y) = \gcd(x_0,y_0)\}$.

We can then use the calculus of pre and post conditions of A. Hoare and show that the formulas given at the annotated points can be proved. The formula $\gcd(x,y) = \gcd(x_0,y_0)$ is an *invariant* of the *while* loop because it holds whenever one tests $y > 0$.

In order to prove that the invariant holds at the end of the body of the loop one has to prove in Hoare's calculus:

$\{x = x_1 \wedge y = y_1 \wedge \gcd(x_1,y_1) = \gcd(x_0,y_0)\}$
$a := x \bmod y;$
$\{a = x_1 \bmod y_1 \wedge y = y_1 \wedge \gcd(x_1,y_1) = \gcd(x_0,y_0)\}$
$x := y;$
$\{a = x_1 \bmod y_1 \wedge x = y_1 \wedge \gcd(x_1,y_1) = \gcd(x_0,y_0)\}$
$y := a;$
$\{y = x_1 \bmod y_1 \wedge x = y_1\}$
$\{\gcd(x,y) = \gcd(y_1, x_1 \bmod y_1) \overset{1.2.3(2)}{=} \gcd(x_1,y_1) = \gcd(x_0,y_0)\}$

On the other hand, in order to prove the correctness of the declarative definition of the function gcd by clauses 1.2.3(1)(2) we just have to demonstrate its specification formulas (1) through (3).

We prove $\forall x(1)$ by complete induction on y. We take any x and consider two cases. If $y = 0$ then, since $\gcd(x, 0) = x$ by 1.2.3(1), we trivially have $\gcd(x, 0) \mid x$ and $\gcd(x, 0) \mid 0$. If $y > 0$ then we have $\gcd(x, y) = \gcd(y, x \bmod y)$ by 1.2.3(2) and, since $x \bmod y < y$, we obtain $\gcd(y, x \bmod y) \mid y$ and $\gcd(y, x \bmod y) \mid x \bmod y$ by IH. We then use 1.2.1(2) to get $\gcd(y, x \bmod y) \mid x$.

We prove $\forall x(2)$ by complete induction on y. We take any x, assume $x + y > 0$, $d \mid x$, $d \mid y$ and consider two cases again. If $y = 0$ then, since we must have $x > 0$, we get $d \leq x = \gcd(x, 0)$. If $y > 0$ then, since $d \mid x \bmod y$ by 1.2.1(2) and $y < x \bmod y$, we get

$$d \overset{\text{IH}}{\leq} \gcd(y, x \bmod y) = \gcd(x, y) \ .$$

Property (3) trivially follows from 1.2.3(1).

1.2.5 Imperative versus declarative programming I. If we do not care about the correctness of our programs and are concerned only with their efficiency then we should probably program imperatively because such programming is with few restrictions. If we are willing to live with some restrictions then the declarative development of highly symbolic programs can be faster and cheaper even without proofs of correctness (see Par. 1.2.7).

If the correctness of our programs is important then the proofs are much easier for declarative programs as was shown in the preceding paragraph. In both cases we had to use a formal theory axiomatizing the domain of our objects. For the example of the greatest common divisor the domain was $\mathbb{N}$ and so the the formal theory will probably be PA. In both cases we had to introduce the function gcd into the theory and prove some of its properties.

For the proof of the declarative program we never left PA because the clausal definition of the new function was in the language of PA and its clauses were axioms. The proof proceeded by simple induction.

For the proof of the imperative program we used two languages: the imperative programming language and the language of PA. We also used two formal systems: the calculus of Hoare and PA. The correctness proof was much longer, we had to introduce auxiliary variables, and at the end we have proven only the *partial* correctness of the program. We still have to do an inductive proof that the program always terminates.

The only argument in favor of imperative programming is efficiency. In the given example both programs have comparable efficiency. In the following paragraph we show the same with another example.

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

$$\mathrm{fib}(0) = 1 \tag{1}$$
$$\mathrm{fib}(1) = 1 \tag{2}$$
$$\mathrm{fib}(n + 2) = \mathrm{fib}(n + 1) + \mathrm{fib}(n). \tag{3}$$

The definition is by straightforward recursion decreasing the argument in the relation $<$. We can directly use the recurrences for computation. For instance, we can compute $\mathrm{fib}(4) = 3$ as follows:

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

The only problem is that the computation sequence is too long. In order to compute the number $\mathrm{fib}(n+2)$ one needs to use the recurrence (3) exactly $\mathrm{fib}(n+2)$ times. The Fibonacci function grows as fast as the exponential function and to compute the function in this way is simply too wasteful.

The following Pascal-like program computes $\mathrm{fib}(n)$ into the variable b:

$a := 1;\ b := 0;$
while $n > 0$ *do*
$\quad n := n \dot{-} 1;\ c := a;\ a := a + b;\ b := c;$

The reader will note that the 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 one. The argument is fallacious as one can define an auxiliary ternary function $f(n, a, b)$ with two *accumulators* a and b by *primitive recursion* decreasing the first argument:

$f(0, a, b) = b$
$f(n+1, a, b) = f(n, a+b, a).$

By straightforward induction on n we can prove:

$$\forall k\, f(n, \mathrm{fib}(k+1), \mathrm{fib}(k)) = \mathrm{fib}(n+k)$$

and so we can explicitly define:

$$\mathrm{fib}(n) = f(n, 1, 1).$$

The number of reductions of f 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 exactly as the *while*-loop in the above Pascal-like program.

1.2.7 Imperative versus declarative programming II. Imperative programs can be better compiled than declarative ones when large data structures are modified in them. Imperative programs can do updates directly in the memory. This is not possible in declarative programs without some restrictions. We will discuss this point in more detail in subsection 1.11 and we only note here that our implementation of CL is written in the declarative programming language Trilogy 2 designed and implemented in 1991 by the

author. Trilogy 2 has declarative provisions for in situ updates and thus the CL compiler executes with comparable efficiency as if it were written in an imperative language.

Trilogy programs, by being declarative, are on a much higher level of abstraction than the commands for the modification of storage typical for imperative programming. It is our estimate that the cost of implementation and maintenance of the Trilogy 2 compiler for CL was about a quarter of what the cost would have been had we implemented CL in an imperative language.

1.3 Arguments in Favor of Natural Numbers

We have argued in the preceding section in favor of declarative over imperative programming. With the decision that our programs should have denotations as functions and predicates we must still decide on the domain of interpretation. We know from Church-Turing thesis that natural numbers suffice as the domain of computable functions. In this section we extend the argument to programming languages and we outline a natural development of data structures needed in computer programming within the domain $\mathbb{N}$. This kind of development is called *arithmetization*.

Arithmetization of Word Domains.

1.3.1 Word domains. Turing machines operate over word domains which consist of finite sequences of symbols from an alphabet. More precisely, an *alphabet* Σ is given by a finite set of *symbols* $\{a_1, a_2, \cdots, a_n\}$. A *word* over an alphabet Σ is a finite sequence of symbols of Σ. The *empty* word is the empty sequence denoted by $\emptyset$. The *length* of a word is the length of its sequence. The *domain of words* over Σ, denoted by Σ^*, is the set of all words over Σ.

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

$$\#|^{x_1}\#|^{x_2}\ldots\#|^{x_n}\# \ .$$

1.3.2 Arithmetization of word domains. Word domains are by no means more general than natural numbers. We can turn functions over the word domain Σ^* given by the p-element alphabet $\Sigma = \{a_1, a_2, \cdots, a_p\}$ into functions over natural numbers by means of *p-adic representation* of natural numbers. For that purpose we define p functions, called *p-adic successor* functions, S_1, S_2, ..., S_p as follows:

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

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

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

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

The reader will note that the monadic representation of natural numbers introduced in Par. 1.1.4 is a special case of p-adic representation with $p = 1$. The monadic successor function $S_1(x) = 1{\cdot}x{+}1 = x'$ is the successor function.

1.3.3 Dyadic representation of $\mathbb{N}$. A special case of p-adic representation with $p = 2$ is the *dyadic* representation. The advantage of the dyadic representation over binary (see Par. 1.1.4) is that there is no restriction on leading digits. The dyadic successor functions S_1 and S_2 are written in the postfix notation: $x\mathbf{1} = 2{\cdot}x + 1$ and $x\mathbf{2} = 2{\cdot}x + 2$. *Dyadic numerals* are the least class of terms containing the constant 0 and with every term τ also the terms $\tau\mathbf{1}$ and $\tau\mathbf{2}$.

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

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

The corresponding dyadic numerals are:

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

The process of going from operations over certain domain to the operations over the codes of elements of the domain in $\mathbb{N}$ is called the *arithmetization* of the domain.

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

$$|0|_d = 0$$
$$|x\mathbf{1}|_d = |x|_d + 1$$
$$|x\mathbf{2}|_d = |x|_d + 1 \ .$$

Clearly, the numbers x such that

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

and no others have the dyadic size n.

1.3.5 Dyadic concatenation. The basic operation over words is *concatenation*. For instance, the words **211** and **122** over the alphabet $\{\mathbf{1},\mathbf{2}\}$ are concatenated into the word **211122**.

The arithmetization of concatenation over the alphabet $\{\mathbf{1},\mathbf{2}\}$ is the binary function $x \star y$, called *dyadic concatenation*. It 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 function has the following clausal definition:

$$x \star 0 = x \tag{1}$$
$$x \star y\mathbf{1} = (x \star y)\mathbf{1} \tag{2}$$
$$x \star y\mathbf{2} = (x \star y)\mathbf{2} \ . \tag{3}$$

The dyadic word **211** is coded by the number $0\mathbf{211} = 11$ and the word **122** by $0\mathbf{122} = 10$. We obtain the code of the concatenated word **211122** by using the clauses of the dyadic concatenation function in the computation as follows:

$$0\mathbf{211} \star 0\mathbf{122} \overset{(3)}{=} (0\mathbf{211} \star 0\mathbf{12})\mathbf{2} \overset{(3)}{=} (0\mathbf{211} \star 0\mathbf{1})\mathbf{22} \overset{(2)}{=} (0\mathbf{211} \star 0)\mathbf{122} \overset{(1)}{=} 0\mathbf{211122} \ .$$

The reader will note that although the code of the concatenated words is $0\mathbf{211122} = 88$, the computation can proceeds by a simple rewriting without ever having to convert into the decimal notation.

The arithmetization of word domains is so natural that we will identify words with their codes.

Arithmetization of Finite Sequences.

1.3.6 Symbolic domains. Computer programming, in addition to the standard numerical types, involves a large number of *data structures* such as n-tuples, multidimensional arrays (vectors and matrices), lists, stacks, tables,

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

1.3.7 Pairing function. We obtain the coding convenience of LISP in the domain of natural numbers by arithmetizing the domain of symbolic expressions with the help of a suitable binary *pairing* function (x, y) satisfying:.

$$(x, y) = (v, w) \to x = v \wedge y = w \tag{1}$$

$$v < (v, w) \wedge w < (v, w) \tag{2}$$

$$x = 0 \vee \exists v \exists w\, x = (v, w)\ . \tag{3}$$

The property (1) is called the *pairing property* and it ensures that for every number n in the range of the pairing function, i.e. such that $n = (x, y)$ for some x and y, the numbers x and y, called the *first* and *second projections of* n respectively, are uniquely determined. The pairing property says that the pairing function is an injection. From the property (2) we get $0 \leq x < (x, y)$. This means that 0 is not in the range of the pairing function and so it has no projections, i.e. $0 \neq (x, y)$. Thus the number 0 is an *atom* and plays the role of the atom *nil* of LISP. The property (3) asserts that the pairing function is onto the set $\mathbb{N} \setminus \{0\}$, i.e. that 0 is the only atom.

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

1.3.9 Pair size. The length of the pair numeral τ denoting the number x is $5{\cdot}n{+}1$ where n is the number of pairing operations used in the construction of the term τ. The arithmetization of the length function is the *pair size* function $|x|_p$ yielding the number of pairing operations needed for the construction of the pair numeral denoting x. The function is defined by the following clausal definition:

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

1.3.10 Cantor's pairing function. Without fixing the pairing function (x, y) we do not know the pair representation of any natural number except

0 and $1 = (0,0)$. Our next step is to determine the function. We first note that the standard *diagonal* function J of Cantor (see [4]) offset by one (to account for the atom 0), i.e. the function

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

satisfies the properties 1.3.7(1) through 1.3.7(3). The initial segment of J is tabulated in Fig. 1.1. The subscripts of values for $J(x,y)$ give the pair size $|J(x,y)|_p$.

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

Fig. 1.1. Pairing function $J(x,y)$

Unfortunately, the pairing function J is not suitable for our purposes because it cannot be used for the development of small functional computational classes such as polynomial time, polynomial space, linear space etc. This was demonstrated in [26]. The reason why J is not suitable can be seen when we enumerate the natural numbers in the J-pair representation. The numbers with the same pair size are not grouped together as it is the case with the binary size of numbers enumerated in the binary notation. We can see from Fig. 1.1 that in the middle of the group of consecutive numbers 4 through 10 we have two numbers with the pair size 4, namely $8 = J(J(0,0),J(0,J(0,0)))$ and $9 = J(J(0,J(0,0)),J(0,0))$, while the remaining numbers have the pair size three.

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

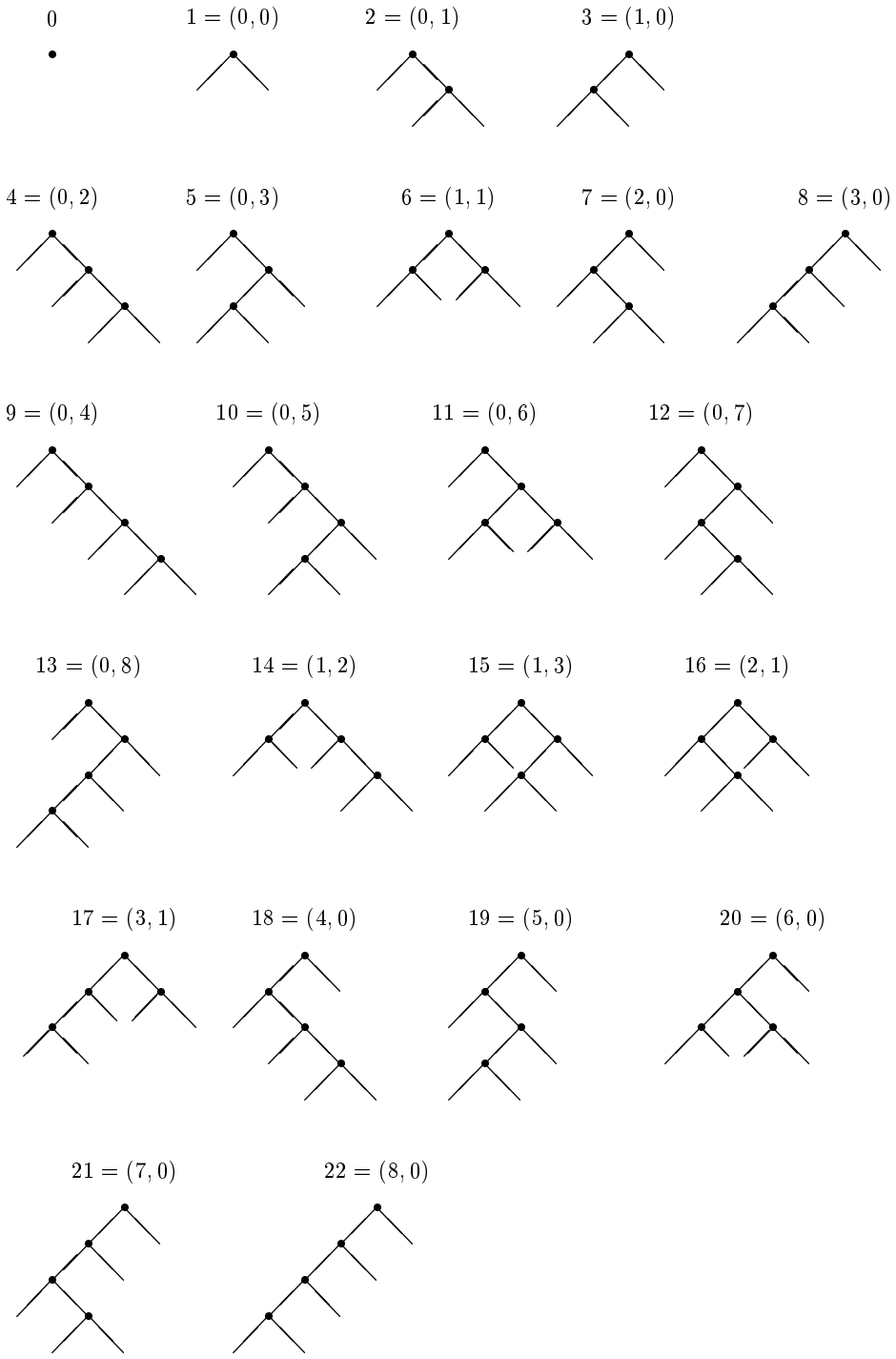


Fig. 1.2. Enumeration of binary trees

if its left subtree is listed before that of t_2, or if the left subtrees are identical, the right subtree of t_1 is listed before that of t_2. An initial segment of the enumeration is given in Fig. 1.2.

The enumeration of binary trees uniquely fixes the pairing function (x, y). Namely, for two numbers x and y we take the x-th and y-th binary trees t_1 and t_2 (counting from zero). The position of the binary tree $\langle t_1, t_2 \rangle$ is the value of (x, y). Fig. 1.3 lists the initial segment of values of (x, y) in a tabular form. The subscripts give the pair size $|(x, y)|_p$. The function will be formally introduced into PA in Sect. 8.3 where we also prove the properties 1.3.7(1) through 1.3.7(3) as theorems of PA.

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

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

Fig. 1.3. Pairing function (x, y)

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

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

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

1.3.13 Projection functions. The pairing property 1.3.7(3) asserts that every non-zero number x has a form (v, w) for some numbers v and w which are uniquely determined by 1.3.7(1). The numbers are accessed by two unary projection functions. The *first projection* function H (head) satisfies the following identities:

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

The *second projection* function T (tail) satisfies the following identities:

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

It should be clear that we have

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

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

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

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

From the above we get

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

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

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

and thus we have

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

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

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

From this we get

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

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

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

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

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

1.3.15 Arithmetization of finite sequences over $\mathbb{N}$. There is a simple way of arithmetizing finite sequences over natural numbers. We assign the code 0 to the empty sequence $\emptyset$. The non-empty sequence $x_1 \; x_2 \; \ldots \; x_n$ is coded by the number $(x_1, x_2, \ldots, x_n, 0)$ (see Fig. 1.4). The reader will note that the assignment of codes is one to one, every finite sequence of natural numbers is coded by exactly one natural number, and vice versa, every natural number is the code of exactly one finite sequence of natural numbers. Codes of finite sequences are called *lists* in computer science and this is how we will be calling them from now on.

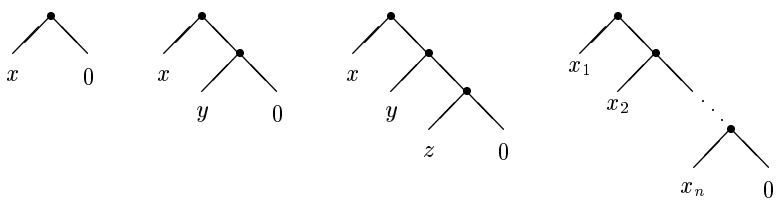


Fig. 1.4. Lists

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

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

1.3.17 Concatenation of lists. The binary *list concatenation* function $x \oplus y$ has the following clausal definition:

$$0 \oplus y = y \tag{1}$$
$$(v, w) \oplus y = v, w \oplus y \quad . \tag{2}$$

As an example we show here the computation of $(1, 2, 3, 0) \oplus (4, 5, 0)$. The computation uses the clauses as rewriting rules and there is no need to convert into decimal (or binary notation) during the computation:

$$(1, 2, 3, 0) \oplus (4, 5, 0) \stackrel{(2)}{=} 1, (2, 3, 0) \oplus (4, 5, 0) \stackrel{(2)}{=} 1, 2, (3, 0) \oplus (4, 5, 0) \stackrel{(2)}{=}$$
$$1, 2, 3, 0 \oplus (4, 5, 0) \stackrel{(1)}{=} 1, 2, 3, 4, 5, 0 \, .$$

1.3.18 List indexing. The binary *list indexing* function $(x)_i$ yields the i-th element (counting from 0) of the list x. For instance, $(0, 1, 2, 3, 0)_2 = 2$. The indexing function is defined by the following clausal definition:

$(0)_x = 0$
$(v, w)_0 = v$
$(v, w)_{i+1} = (w)_i$

with recursion decreasing in the second argument.

1.3.19 List membership predicate $x \; \varepsilon \; a$. The binary *list membership* predicate $x \; \varepsilon \; a$ satisfying the specifications:

$$x \; \varepsilon \; a \leftrightarrow \exists i (i < L(a) \wedge (a)_i = x)$$

can be defined by the following clausal definition:

$x \; \varepsilon \; y, a \leftarrow x = y$
$x \; \varepsilon \; y, a \leftarrow x \neq y \wedge x \; \varepsilon \; a$.

The negative clause $x \not\varepsilon 0$ is implied by default.

Arithmetization of Trees.

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

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

This asserts that a value of type Bt is either the *leaf* E or a *node* of the form $Nd(x, t_1, t_2)$ where t_1 and t_2 are values of type Bt and $x \in \mathbb{N}$ is the *label* of the node. The constant E and the function Nd are called *constructors*.

We arithmetize the values of the type Bt with the help of two constructor functions $E = 0, 0$ and $Nd(x, t_1, t_2) = 1, x, t_1, t_2$. Note that this guarantees that we have $E \neq Nd(x, t_1, t_2)$. The predicate $Bt(t)$ holding of codes of labeled binary trees t has a clausal definition corresponding to the union type definition:

$$Bt(E)$$
$$Bt\, Nd(x, t_1, t_2) \leftarrow Bt(t_1) \wedge Bt(t_2) \quad .$$

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

1.3.21 Flattening of binary trees. As an example of an operation over binary trees we consider the function $Flat(t)$ taking a binary tree t and yielding the list of all labels in t read off t from left to right (inorder). The function has the following clausal definition:

$$Flat(E) = 0$$
$$Flat\, Nd(x, t_1, t_2) = Flat(t_1) \oplus (x, Flat(t_2)) \quad .$$

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

$$f(E, a) = a$$
$$f(Nd(x, t_1, t_2), a) = f(t_1, x, f(t_2, a)) \quad .$$

We can now explicitly define a fast flatten by:

$$Flat(t) = f(t, 0) \quad .$$

The correctness of this definition on labelled trees follows from the following property:

$$Bt(t) \rightarrow \forall a\, f(t, a) = Flat(t) \oplus a$$

proved by complete induction on t.

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

$$Incr(t) \leftrightarrow \forall i \forall j (i < j < L(t) \rightarrow (a)_i < (a)_j)$$
$$Bst(t) \leftrightarrow Incr\, Flat(t) \quad .$$

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

$$Bst(t) \to (x \in_b t \leftrightarrow \exists i\, x\, \varepsilon\, Flat(t)) \ .$$

The predicate has the following clausal definition:

$x \in_b Nd(y, t_1, t_2) \leftarrow x < y \wedge x \in_b t_1$
$x \in_b Nd(y, t_1, t_2) \leftarrow x = y$
$x \in_b Nd(y, t_1, t_2) \leftarrow x > y \wedge x \in_b t_2$.

Arithmetization of Symbolic Expressions

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

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

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

$Cns(x) = 0, x$
$Var(x) = 1, x$
$Add(x, y) = 2, x, y$
$Mul(x, y) = 3, x, y$.

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

$$Add(Mul(Cns(356), Var(3)), Var(5)) =$$
$$2, (3, (0, 356), 1, 3), 1, 5 = 3\,442\,660\,716\,284 \ .$$

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

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

$Expr\ Cns(c)$
$Expr\ Var(i)$
$Expr\ Add(a, b) \leftarrow Expr(a) \wedge Expr(b)$
$Expr\ Mul(a, b) \leftarrow Expr(a) \wedge Expr(b)$.

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

$$Val(Cns(c), v) = c$$
$$Val(Var(i), v) = (v)_i$$
$$Val(Add(a, b), v) = Val(a, v) + Val(b, v)$$
$$Val(Mul(a, b), v) = Val(a, v) \cdot Val(b, v) \ .$$

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

Arithmetization of Numeric Domains

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

$$\mathbb{Z}(+(n))$$
$$\mathbb{Z}(-(n+1)) \ .$$

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

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

holds, has the following explicit clausal definition:

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

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

1.4 Bootstrapping of PA

The reader accepting our arguments in preceding sections probably agrees that our programs are to be definitions of computable functions and predicates over the domain $\mathbb{N}$ of natural numbers. Definitions are concrete objects given in a formal system of arithmetic and the defined function and predicate symbols posses denotations in the domain $\mathbb{N}$. Formal systems are necessary because neither humans nor computers can work directly with the abstract domain of natural numbers. The most natural candidate for the formal system is the Peano arithmetic.

In this section we discuss the definitions of functions and predicates in PA from the *extensional* point of view where we are prima facie not interested whether they are computable. Non-computable functions and predicates play quite important role in the theory of programming because they are often used with advantage for the specification and proofs of properties of computable functions.

1.4.1 Peano arithmetic. Peano arithmetic (PA) is formulated as a first-order logical theory in the language consisting of the constant 0, the unary function symbol x', and of two binary function symbols $x + y$ and $x{\cdot}y$. The intended interpretation of PA is in the *standard model* $\mathcal{N}$ with the domain of natural numbers $\mathbb{N}$ and with the interpretation of its symbols in the above order as the number 0, the *successor* function $S(x) = x+1$, the addition, and the multiplication functions.

The axioms of Peano arithmetic are

$$\begin{gathered} 0 \neq x' \\ x' = y' \to x = y \\ 0 + y = y \\ x' + y = (x + y)' \\ 0{\cdot}y = y \\ x'{\cdot}y = x{\cdot}y + y \end{gathered}$$

plus the infinite number of *induction axioms* of the form

$$\phi[0] \wedge \forall x(\phi[x] \to \phi[x']) \to \phi[x] \tag{1}$$

for every formula $\phi[x]$ of the language. It is easy to see that all axioms are satisfied in the standard model $\mathcal{N}$.

We denote by $\underline{n}_m$ the monadic numeral (see Par. 1.1.4) for n. Precisely, $\underline{n}_m$ is defined as a meta-theoretical function yielding terms of PA to satisfy:

$$\begin{aligned} \underline{0}_m &\equiv 0 \\ \underline{n+1}_m &\equiv (\underline{n}_m)' \ . \end{aligned}$$

1.4.2 Extensions of Peano arithmetic. Computer programming requires constant addition of definitions of new functions and predicates. Formal tools in logic for this are *extensions* of theories, in our case extensions of PA. This happens by the addition of a new function or predicate symbol to the language of the current extension of PA together with new axiom(s) defining the symbol.

We do not wish the extended theory to be *inconsistent* in the sense that it can prove theorems not satisfiable in the standard model, for instance $0 = 1$. Once $0 = 1$ is a theorem, the theory can prove any formula. The unextended theory PA cannot prove any formulas not satisfied in $\mathcal{N}$ and so it is consistent.

In order to maintain the consistency we restrict our extensions to *conservative* ones. Conservative extensions do not prove any new theorems in the language before the extension, i.e. formulas which do not contain the new symbol. Since $0 = 1$ is such a formula the extended theory is consistent. We actually go even further and restrict our extensions to *extensions by definitions* which are a special case of conservative extensions where the extended theory cannot prove any properties not expressible already in the original language of PA. New function and predicate symbols are thus only a notational convenience which make our theorems and definitions more readable. We gain on expressivity but not on power.

When we say PA in the following we always mean the current extension of Peano arithmetic. Similarly, the term *standard model of PA* refers to the standard model of PA from Par. 1.4.1 expanded by interpretations of all function and predicate symbols in the current extension of PA.

1.4.3 Extensions of PA by explicitly defined predicates. We can, for instance, extend PA with the binary comparison predicate $x < y$ by adding it to the language of the theory and then add a new axiom fully describing the predicate:

$$x < y \leftrightarrow \exists z\, x + z' = y \ .$$

We can then use the axiom in proving properties of the new symbol, for instance the transitivity:

$$x < y \wedge y < z \rightarrow x < z \ .$$

The extension does not add any strength to PA because we can always eliminate all occurrences of $<$ from a formula by replacing applications $\tau_1 < \tau_2$ with formulas $\exists z\, \tau_1 + z' = \tau_2$.

We can then further extend PA with the binary predicate symbol $\leq$ with the defining axiom:

$$x \leq y \leftrightarrow x < y \vee x = y \ .$$

In general, for any formula $\phi[x_1, \ldots, x_n]$ of the current extension T of PA we can form its extension S by *explicit definition of the predicate* R by adding a new n-ary predicate symbol R together with its *defining axiom*:

$$R(x_1, \ldots, x_n) \leftrightarrow \phi[x_1, \ldots, x_n] \ . \tag{1}$$

The new symbol can be always eliminated from a formula ψ of S by a *translation* similar to the one describes above whereby we obtain a formula $\psi^\star$ of T. The extended theory S proves that the two formulas are equivalent: $\psi \leftrightarrow \psi^\star$. The conservativity of S over T follows from the fact that S proves ψ iff T proves $\psi^\star$.

We can even use the new predicate symbol R in induction formulas ψ of the form 1.4.1(1) where ϕ is arbitrary formula of the language of S. This is because the translation $\psi^\star$ is an induction formula provable in the theory T. S, by being an extension of T, proves $\psi^\star$ and, since it also proves $\psi \leftrightarrow \psi^*$, the theory S proves the induction formula ψ.

The reader will note that we have already used a definition by explicit definition of a predicate in Par. 1.2.1 where we have defined

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

1.4.4 Extensions of PA by contextually defined functions. We can extend PA by explicit definitions of functions similarly to extensions by explicit definitions of predicates. For instance, we can introduce the *square* function x^2 by explicit definition: $x^2 = x \cdot x$. In the general case we can take a term $\tau[x_1, \ldots, x_n]$ in the language of the current extension of PA and introduce a new n-ary function symbol f by explicit definition with the defining axiom:

$$f(x_1, \ldots, x_n) = \tau[x_1, \ldots, x_n] \ . \tag{1}$$

While certainly useful, this kind of extensions does not fully utilize the power of logical notation to the same extent as extensions by explicit definitions of predicates. Formulas ϕ in extensions 1.4.3(1) can contain arbitrary propositional connectives and quantifiers. The simpler syntax of terms τ in extensions (1) has not the same expressivity.

We can use formulas in explicit definitions of function symbols where we define a new function symbol in the context of a formula where instead of the function f we introduce its *graph* $f(x_1, \ldots, x_n) = y$ as an $(n+1)$-ary relation. For instance, the modified subtraction function $x \dot{-} y$ can be introduced by an explicit contextual definition as:

$$x \dot{-} y = z \leftrightarrow x \geq y \wedge x + z = y \vee x < y \wedge z = 0 \ .$$

The point is that no matter what the numbers x and y are, exactly one of $x \geq y$ and $x < y$ holds. Moreover, in the former case there is a unique z s.t. $x + z = y$.

In general, for any formula $\phi[x_1, \ldots, x_n, y]$ of the current extension T of PA for which T proves the *existence* and *uniqueness* conditions:

$$\exists y \phi[x_1, \ldots, x_n, y] \tag{2}$$
$$\phi[x_1, \ldots, x_n, y_1] \wedge \phi[x_1, \ldots, x_n, y_2] \rightarrow y_1 = y_2 \tag{3}$$

we can form the extension S by *contextual definition of the function* f by adding a new n-ary function symbol f together with its *defining axiom*:

$$f(x_1, \ldots, x_n) = y \leftrightarrow \phi[x_1, \ldots, x_n, y] \ . \tag{4}$$

The new symbol can be always eliminated from a formula ψ of S by a translation $\psi^\star$. The main idea is that for every formula $\psi[y]$ s.t. $\psi[f(\tau_1, \ldots, \tau_n)]$ is a formula of S the theory S proves

$$\psi[f(\tau_1, \ldots, \tau_n)] \leftrightarrow \exists y(f(\tau_1, \ldots, \tau_n) = y \wedge \psi[y])$$

Since (4) is an axiom of S, the theory then proves:

$$\psi[f(\tau_1, \ldots, \tau_n)] \leftrightarrow \exists y(\phi[\tau_1, \ldots, \tau_n, y] \wedge \psi[y])$$

where the formula on the right has at least one application of f less then the formula on the right. Continuing in this way, we can eliminate all applications of f from the formula ψ whereby we obtain a translation $\psi^\star$, which is formula of T such that T proves $\psi \leftrightarrow \psi^*$. That S is conservative over T follows from the highly non-trivial fact that S proves ϕ iff T proves $\phi^\star$.

We can use the new function symbol f in induction formulas because they are theorems of S for the same reason as was the case with the explicit definition of predicates.

1.4.5 Introduction of integer division into PA. As an example of extension by contextually defined functions we show how to introduce the integer division function $x \div y$ into PA by contextual definition:

$$x \div y = q \leftrightarrow y > 0 \wedge \exists r(x = q{\cdot}y + r \wedge r < y) \vee y = 0 \wedge q = 0 \ .$$

This can be done because PA proves the existence condition:

$$\exists q(y > 0 \wedge \exists r(x = q{\cdot}y + r \wedge r < y) \vee y = 0 \wedge q = 0)$$

as it directly follows from $y > 0 \rightarrow \exists q \exists r(x = q{\cdot}y + r \wedge r < y)$. The uniqueness condition follows easily from the uniqueness of quotients and remainders:

$$q_1{\cdot}y + r_1 = q_2{\cdot}y + r_2 \wedge r_1 < y \wedge r_2 < y \rightarrow q_1 = q_2 \wedge r_1 = r_2 \ .$$

1.4.6 The case analysis function D. Another example of extension by a contextually defined function is the introduction into PA of the ternary *case discrimination function* D satisfying:

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

Computer scientists can visualize the function as

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

The case discrimination function is introduced into PA as follows:

$$D(x, y, z) = v \leftrightarrow x > 0 \wedge v = y \vee x = 0 \wedge v = z \ .$$

The existence and uniqueness conditions are obviously provable.

1.4.7 Extensions of PA by explicitly defined functions. Extensions of PA by explicit definitions 1.4.4(1) are special case of extensions by contextual definitions:

$$f(x_1, \ldots, x_n) = y \leftrightarrow \tau[x_1, \ldots, x_n] = y \ .$$

For instance, the remainder function is is explicitly introduced into PA by:

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

1.4.8 Characteristic functions of predicates. The *characteristic function* of an n-ary predicate P is the n-ary function f such that:

$$f(\vec{x}) = \begin{cases} 1 & \text{if } P(\vec{x}) \\ 0 & \text{otherwise.} \end{cases}$$

We obviously have

$$f(\vec{x}) > 1 \leftrightarrow P(\vec{x}) \ .$$

We usually designate the characteristic function of a predicate P by P_*.

1.4.9 Characteristic functions of $=$ and $<$. The binary characteristic functions $=_*$ and $<_*$ of the identity and less than predicates can be introduced into PA by contextual definitions:

$$(x =_* y) = z \leftrightarrow x = y \wedge z = 1 \vee x \neq y \wedge z = 0$$
$$(x <_* y) = z \leftrightarrow x < y \wedge z = 1 \vee x \geq y \wedge z = 0$$

whose existence and uniqueness conditions hold trivially.

1.4.10 Extensions of PA by definitions. Let T be an extension of PA and S_1 an extension of T either by explicit definition of a predicate P or by contextual definition of a function f.

We say that a theory S whose language is the same as S_1 is an *extension by definition* of T if S and S_1 are *elementarily equivalent* i.e. if both theories prove exactly the same theorems.

The theory S is thus conservative over T, permits to translate away the introduced symbols, and proves all induction formulas containing the newly introduced symbol. An *extension by definitions* is a finite sequence of extensions with every sequence an extension by definition.

We will sketch out below how to formulate a few increasingly more powerful forms of extensions of PA in the sense of expressive power and comfort but which will still be extensions by definitions. Comfortable and easy to use forms of extensions of PA are needed in order to obtain the kind of programming style computer programmers are used to. The last schema will the schema of extensions by clausal definitions. Clausal definitions, besides being very readable and comfortable to use, have yet another important property because they are flexible enough to control the computation of defined functions and predicates (see Sect. 1.7).

1.4.11 Extensions of PA by minimalization. Every extension of PA which contains the predicate $<$ and which proves the existence condition:

$$\exists y \phi[\vec{x}, y]$$

can be extended *by minimalization* with a new function symbol f and with the defining axiom:

$$\phi[\vec{x}, f(\vec{x})] \wedge \forall y (y < f(\vec{x}) \rightarrow \neg \phi[\vec{x}, y]) \ . \tag{1}$$

We will use a more suggestive notation as an abbreviation for (1):

$$f(\vec{x}) = \mu_y[\phi[\vec{x}, y]] \ . \tag{2}$$

The idea is that the function f defined by this definition yields for every $\vec{x}$ the minimal y such that $\phi[\vec{x}, y]$ holds.

Extension by minimalization is an extension by definition because (1) follows from the extension by contextual definition:

$$f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y] \wedge \forall z (z < y \rightarrow \neg \phi[\vec{x}, z]) \ .$$

As an example we introduce into PA the *whole part of the square root* function:

$$[\sqrt{x}] = \mu_y[x < (y+1)^2] \tag{3}$$

which satisfies the specification:

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

The extension is legal because PA proves the existence condition $\exists y \, x < (y+1)^2$.

1.4.12 Extensions of PA by primitive recursion. We wish to extend PA with a new function symbol f satisfying the defining axioms:

$$f(0,\vec{y}) = \tau_1[\vec{y}] \tag{1}$$
$$f(x',\vec{y}) = \tau_2[x,\vec{y},f(x,\vec{y})] \tag{2}$$

where $\tau_1[\vec{y}$ and $\tau_2[x,\vec{y},z]$ are terms in the language of PA before the extension containing at most the indicated variables. This is an extension by *primitive recursion* and one has to work hard before one can demonstrate that it is an extension by definition.

The difficulty with this kind of extension lies in the finding of a formula $\phi[x,\vec{y},z]$ in the language of PA before the extension so f can be introduced by contextual definition:

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

The formula must encode in its bound variable s the *course of values* sequence $f(0,\vec{y})$, $f(1,\vec{y})$, ..., $f(x{-}1,\vec{y})$, $f(x,\vec{y})$ needed in the computation of $f(x,\vec{y})$. The elements of the sequence s can be encoded as digits in the base 2^k:

digit position:	$2^{k\cdot x}$	$2^{k\cdot(x-1)}$	$\cdots$	$2^{k\cdot 1}$	$2^{k\cdot 0}$
$s =$	$f(x,\vec{y})$	$f(x{-}1,\vec{y})$	$\cdots$	$f(1,\vec{y})$	$f(0,\vec{y})$

.

The number k must be sufficiently large so we have $f(i,\vec{y}) < 2^k$ for all i s.t. $0 \le i \le x$. The formula ϕ is the one on the right-hand-side of the following contextual definition:

$$f(x,\vec{y}) = z \leftrightarrow \exists k \exists s((s)_0^{[k]} = \tau_1[\vec{y}] \wedge (s)_x^{[k]} = z \wedge \forall i(i < x \rightarrow (s)_{i+1}^{[k]} = \tau_2[i,\vec{y},(s)_i^{[k]}]))$$

where the ternary *bounded* indexing function $(s)_i^{[k]}$ is introduced into PA by explicit definition:

$$(s)_i^{[k]} = s \div 2^{k\cdot i} \bmod 2^k\ .$$

We have sketched above how to introduce into PA the function f by primitive recursion. The careful reader has noted that we can do the introduction only in such extensions of PA which contain the exponentiation function 2^x.

The introduction into PA of the function 2^x is even harder than primitive recursion. We somehow have to encode in the language of PA its course of values sequence for the recursive clauses

$2^0 = 1$
$2^{x\mathbf{0}} = (2^x)^2 \leftarrow x > 0$
$2^{x\mathbf{1}} = 2{\cdot}(2^x)^2$

by extremely limited means with basically just the addition, multiplication, and the predicate of divisibility. We devote the entire section 8.1 to this task.

The extension by primitive recursion does not mean not only the addition of the two defining axioms (1) and (2) for f. We still need that the induction axioms for formulas containing the new symbol f are theorems of the extended theory. It can be shown that for that it suffices to add a single axiom of induction for the formula $\phi[x, \vec{y}, f(x, \vec{y})]$ where ϕ is as above.

1.4.13 Extensions of PA by course of values recursion with measure. The next schema of recursion is extremely general. We wish to extend the current extension of PA, call it T, to the theory S by *course of values recursion with measure* μ with a new n-ary function symbol f and with the axiom:

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

Here $\tau[f; \vec{x}]$ is an arbitrary term of the language S and the *measure* term $\mu[\vec{x}]$ is of the language T.

Without any restrictions on recursive applications of f in τ we cannot always expand the standard model of T with an interpretation of f so S satisfies $f(\vec{x}) = \tau[f; \vec{x}]$. However, we can always surround every recursive application with a *guard* guaranteeing the decrease of recursive arguments in the measure μ. This means that every recursive application $f(\vec{\rho})$ in τ is replaced by the term:

$$D((\mu[\vec{\rho}]<_*\mu[\vec{x}]), f(\vec{\rho}), 0)$$

whereby we obtain the term on the right-hand-side of (1) which is expressed by the quasi-lambda notation.

We assume that the language of T contains the pairing function which is introduced into PA by quite a complicated series of extensions whose description takes the whole of Sect. 8.3. The important thing is that the extensions involve nothing more complicated than primitive recursion.

We now outline how to find a formula $\phi[\vec{x}; y]$ of T such that when T is extended by definition to S_1 with the axiom

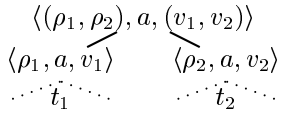
$$f(\vec{x}) = y \leftrightarrow \phi[\vec{x}; y] \tag{2}$$

the theories S and S_1 are elementarily equivalent, i.e. they prove the same theorems. In order to achieve this we have to guarantee that S proves the induction axioms for formulas applying the new symbol f. This can be done with a single induction axiom for the formula $\phi[\vec{x}, f(\vec{x})]$. This, and (1) are thus the new axioms of S.

We will introduce the function f with the help of a predicate arithmetizing the computation of f in a *computation* tree. Without loss of generality, because we can always work with contractions, we assume that f is unary and that the term τ is built up from the variable x and from the numerals $\underline{n}_m$ by pairing (τ_1, τ_2) and by applications of unary functions $g_1, \ldots, g_k$. Computation trees are binary and they have triples $\langle \rho, a, v \rangle$ as labels in their non-leaf

nodes. The node with such a label records the computation of a subterm ρ of τ with the value a assigned to the variable x and with v recording the value (denotation) of ρ in this assignment. The sons of the node record the necessary subcomputations.

The form of the term ρ determines the shape of the sons t_1 and t_2 as follows. If $\rho \equiv \rho_1, \rho_2$ then the computation tree is:

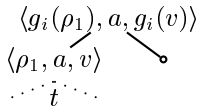


where the value v_1 of ρ_1 is computed in the left son and the value v_2 of ρ_2 in the right son. The value of (ρ_1, ρ_2) is then the pair (v_1, v_2).

If $\rho \equiv x$ or $\rho \equiv \underline{n}_m$ then the respective computation trees are:

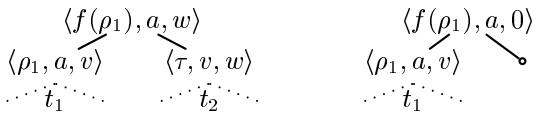
where there are no subcomputations because the values of x and $\underline{n}_m$ can be determined directly as a and n respectively.

If $\rho \equiv g_i(\rho_1)$ then the computation tree is



where we record in the left son the computation of the argument ρ_1 into the value v and then the value of $g_i(\rho_1)$ is then $g_i(v)$. There is not need to record any computation in the right son.

Finally, if $\rho \equiv f(\rho_1)$ then there are two possible computation trees:



In both cases the value v of the argument ρ_1 is computed in the left son. The two cases are determined by the outcome of the test $\mu[v] < \mu[a]$. If the measure decreases then the computation tree is shown above on the left. This is when the identity $f(x) = \tau[f; x]$ is used as the computation rule in the form $f(\underline{v}_m) \mapsto \tau[f; \underline{v}_m]$. The value w of the term τ is computed in the right son and the value of the recursive application $f(\underline{v}_m)$ is determined as w. If the outcome of the test is negative then the computation tree is shown above on the right where the value of $f(\underline{v}_m)$ is 0. The reader will note that the computation of $f(\rho_1)$ thus evaluates the recursive guard.

It should be obvious that we can construct a computation tree for any subterm ρ of τ and any assignment a of the value of the variable x because the terms in the labels of the tree are always smaller except in the right sons of recursive applications but then the measure decreases and the initial measure $\mu[a]$ can decrease only finitely many times.

We arithmetize this computation with the help of a predicate $Ct(t)$ holding iff t codes a computation tree for a subterm ρ of τ. The informal reasoning above on the existence computation trees is arithmetized in such a way that T proves:

$$\exists v \exists t_1 \exists t_2 \, Ct((\ulcorner \tau \urcorner, x, v), t_1, t_2)$$

and that the value v and the subtrees t_1 and t_2 are unique. Here $\ulcorner \tau \urcorner$ stands for a term of T denoting the code of the term τ. We can now define f by the contextual definition:

$$f(x) = y \leftrightarrow \exists t_1 \exists t_2 \, Ct((\ulcorner \tau \urcorner, x, y), t_1, t_2) \ .$$

The right-hand-side formula, after translating away the introduced auxiliary functions and predicates (such as Ct), becomes the formula ϕ in the language of T from (2).

The auxiliary predicate Ct must have a definition which uses at most primitive recursion. We can explicitly define it from two auxiliary predicates as follows:

$$Ct(t) \leftrightarrow \forall u(u \trianglelefteq t \wedge u > 0 \rightarrow Nd(u))$$

The predicate $u \trianglelefteq t$ holds whenever u is (the code of) a subtree of (the tree coded by) t and the predicate $Nd(u)$ has a simple explicit definition codifying the local properties of nodes of computation trees which are expressed by the six tree diagrams given above. Both auxiliary predicates use the pairing function for coding. The subtree predicate has a definition by *course of values recursion*:

$$u \trianglelefteq t \leftarrow u = t \vee \exists n \exists t_1 \exists t_2 (t = n, t_1, t_2 \wedge (u \trianglelefteq t_1 \vee u \trianglelefteq t_2))$$

which is a simple case of the general schema (1), i.e.

$$u \trianglelefteq_* t = D(u =_* t, 1, D(u, D(u \trianglelefteq_* H \, T(t), 1, u \trianglelefteq_* T \, T(t)))) \ , \tag{3}$$

that it and it can be easily reduced to an instance of primitive recursion. The reader will note that the guards around the two recursive applications are superfluous because the recursion goes down in the second argument.

1.4.14 Fragments of Peano Arithmetic. The strength of functions introduced into PA can be measured by the quantifier complexity of inductive axioms needed to prove the existence conditions for their introduction.

The simplest quantifiers are the *bounded* ones:

$$\exists x(x \leq \tau \wedge \phi[x])$$
$$\exists x(x \leq \tau \rightarrow \phi[x])$$

where the term τ does not contain the variable x. Bounded quantifiers can be replaced by functions which successively try the finitely many values $x = 0, 1, \ldots, \tau$ and test the formula $\phi[x]$ for each of them. The functions are primitive recursive in the functions needed to implement the characteristic function of ϕ_*.

Actually, every formula of the language of PA (before any extensions except with the predicate $\leq$) is provably equivalent in PA to one of the formulas:

$$\exists x_1 \forall x_2 \ldots Q x_n \phi \tag{1}$$
$$\forall x_1 \exists x_2 \ldots Q x_n \phi \tag{2}$$

where the (unbounded) quantifiers alternate and Q is either $\exists$ or $\forall$ depending on the arity of n. The formula ϕ contains at most bounded quantifiers and propositional connectives. The two kinds of formulas are designated as Σ_n and Π_n respectively.

The induction axioms of PA permit induction formulas of arbitrary quantifier complexity. The *fragments* $I\Sigma_n$ are the subtheories of PA with the induction schemas restricted to Σ_n-formulas. It can be shown that the fragment $I\Sigma_n$ proves all induction axioms for Π_n-formulas.

Every recursive function has a contextual definition by a Σ_1-formula $\phi[\vec{x}, y]$:

$$f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y] \tag{3}$$

although PA ($I\Sigma_n$) is not always strong enough to prove its existence condition $\exists y \phi[\vec{x}, y]$. Thus not all recursive functions can be introduced into PA ($I\Sigma_n$). Those which can, are called the *provably recursive* functions of PA ($I\Sigma_n$). The provably recursive functions of $I\Sigma_1$ are exactly the primitive recursive functions whereas those of PA are $\prec\epsilon_0$-recursive functions (see [8]).

A function f is Σ_k*-definable* if the formula ϕ in (3) is a Σ_k-formula. If its existence condition can be proved in the theory T (PA or a fragment $I\Sigma_n$) then f is Σ_k*-definable in* T. Provably recursive functions of T are thus Σ_1-definable in T.

Every Σ_k-definable function is also Π_k-definable because every such definition (3) can be reformulated as

$$f(\vec{x}) = y \leftrightarrow \forall z(\phi[\vec{x}, z] \rightarrow y = z)$$

where the formula on the right-hand-side is equivalent to a Π_k-formula.

A predicate P is Σ_k*-definable (*Π_k*-definable)* if it can be explicitly defined as

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

with $\phi[\vec{x}]$ a Σ_k-formula (Π_k-formula). Recursive predicates are characteristic predicates of recursive functions and so both recursive functions and predicates have Σ_1 and Π_1 definitions. Such functions and predicates are called Δ_1-definable. In general, Δ_k-*definable* functions and predicates have both Σ_k and Π_k definitions.

If we define a function or a predicate as provably recursive in $I\Sigma_1$ then we will usually not mention this explicitly and we will just say that the function or predicate is *introduced into PA*. For any other extension we mention either the definability of the formula and/or the strength of the theory.

1.5 Clausal Extensions of PA

The schema of definitions by course of values recursion with measure 1.4.13(1) is extremely general because it does not restrict in any way the recursion in the term τ. Nevertheless, it is not a good one to use in computer programming for two reasons.

The first reason are the explicit guards on recursive applications whose evaluation can be time-consuming. We can eliminate the guards if we restrict ourselves to special forms of defining terms τ for which PA can prove that the measure of recursive applications decreases. Definitions with such terms are called *regular*.

The second reason is the low readability of definitions by the schema. The reader should inspect the definition 1.4.13(3) of the subtree predicate $\trianglelefteq$ via its characteristic function and he should reformulate the same definition in LISP, Pascal, or C. The problem with such definitions is the use of *destructor* functions (in this case the projection functions H and T) instead of the modern *pattern matching* notation as it is known from functional programming languages. The difference in readability between the two style is striking as can be seen from the following clausal definition of the same predicate:

$$u \trianglelefteq t \leftarrow u = t$$
$$u \trianglelefteq n, t_1, t_2 \leftarrow u \neq n, t_1, t_2 \wedge (u \trianglelefteq t_1 \vee u \trianglelefteq t_2) \ .$$

In the design of our clausal definitions we have substantially expanded the concept of pattern matching by going to patterns in formulas rather than staying with patterns in terms. The difference in expressivity and readability is comparable to the difference between explicit and contextual definitions of functions (see Par. 1.4.4). The schema of clausal definitions is probably best explained with a concrete example.

1.5.1 Square root function revisited. Suppose that we wish to extend PA with a better program for the function $[\sqrt{z}]$ satisfying:

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

than the one by minimalization in 1.4.11(3). This was a 'naive' program exponentially slower than it should have been because it executed a brute force search in time $\mathcal{O}(z)$. The following clausal definition introduces the same function in a computationally optimal form which runs in time $\mathcal{O}(\log(z))$:

$$
\begin{array}{ll}
[\sqrt{0}] = 0 & \\
[\sqrt{z}] = 1 & \leftarrow z < 4 \wedge z > 0 \\
[\sqrt{4{\cdot}x + i}] = s\mathbf{0} & \leftarrow 4{\cdot}x + i \geq 4 \wedge i < 4 \wedge [\sqrt{x}] = s \wedge 4{\cdot}x + i < (s\mathbf{1})^2 \\
[\sqrt{4{\cdot}x + i}] = s\mathbf{1} & \leftarrow 4{\cdot}x + i \geq 4 \wedge i < 4 \wedge [\sqrt{x}] = s \wedge 4{\cdot}x + i \geq (s\mathbf{1})^2 \ .
\end{array}
$$

The clauses are just formulas in the language of PA even though their implications are customarily written in the converse form. The clauses should be completely self-explanatory and the reader should have no difficulties whatsoever to understand the properties of the defined function.

$$
\begin{array}{l}
[\sqrt{z}] = y \leftrightarrow \textbf{case} \\
\qquad z < 4 \rightarrow \textbf{case} \\
\qquad\qquad z = 0 \rightarrow 0 = y \\
\qquad\qquad z > 0 \rightarrow 1 = y \\
\qquad z \geq 4 \rightarrow \textbf{let} \\
\qquad\qquad z = 4{\cdot}x + i \wedge i < 4 \rightarrow_{x,i} \\
\qquad\qquad\ \ \textbf{let} \\
\qquad\qquad\quad [\sqrt{x}] = s \rightarrow_s \ \textbf{case} \\
\qquad\qquad\qquad\qquad 4{\cdot}x + i < (s\mathbf{1})^2 \rightarrow s\mathbf{0} = y \\
\qquad\qquad\qquad\qquad 4{\cdot}x + i \geq (s\mathbf{1})^2 \rightarrow s\mathbf{1} = y
\end{array}
$$

Fig. 1.5. The definition of $[\sqrt{z}]$ with case formulas.

1.5.2 Generalized case formulas. The only problem with the clausal definition of the function $[\sqrt{z}]$ in Par. 1.5.1 is to recognize that the clauses constitute a definition. We will show that the clauses are an extension of PA by definition. Toward that end we write the clauses in a form with *case formulas* given in Fig. 1.5 which resemble contextual definitions (albeit recursive). The reader should think of case formulas as conjucts of their alternatives some of which have universally quantified *local* variables. For instance, he should visualize the following three alternative case formula:

$$
\begin{array}{l}
\textbf{case} \\
\quad \phi_1 \rightarrow \psi_1 \\
\quad \phi_2[x,y] \rightarrow_{x,y} \psi_2[x,y] \\
\quad \phi_3[x] \rightarrow_x \psi_3[x]
\end{array}
$$

as standing for the formula

$$(\phi_1 \rightarrow \psi_1) \wedge \forall x \forall y(\phi_2[x,y] \rightarrow \psi_2[x,y]) \wedge \forall x(\phi_3[x] \rightarrow \psi_3[x]) \ .$$

The assumptions in the alternatives of a case formula should be *complete*, pairwise *exclusive*, and the local variables should be uniquely determined. In the above three alternative case this means that we have:

$$\begin{gathered}\phi_1 \vee \exists x! \exists y! \, \phi_2[x,y] \vee \exists x! \, \phi_3[x] \\ \phi_2[x,y] \vee \phi_3[x] \rightarrow \neg\phi_1 \\ \phi_2[x,y] \rightarrow \neg\exists x \phi_3[x] \ .\end{gathered}$$

Under those conditions the case formula is equivalent to the following disjunctive formula:

$$\phi_1 \wedge \psi_1 \vee \exists x \exists y (\phi_2[x,y] \wedge \psi_2[x,y]) \vee \exists x (\phi_3[x] \wedge \psi_3[x]) \ .$$

1.5.3 Let formulas. Some of the case formulas have only one alternative and their sole purpose is to introduce local variables; the reader should visualize them as a generalization of *let* constructs as known from functional programming languages. For that reason we write **let** instead of **case** as, for instance:

let
$z = 4{\cdot}x + i \wedge i < 4 \rightarrow_{x,i} \phi[x,i]$

which splits the argument z by *pattern matching* into the unique values x and i satisfying the shown relation. The local variables x and i can be then referred to in the *body* $\phi[x,i]$ of the alternative.

1.5.4 Unfolding of the contextual definition of $[\sqrt{z}]$. The contextual definition from Fig. 1.5 is transformed to the four clauses for $[\sqrt{z}]$ by *unfolding*. Unfolding, in this case, means the splitting of the alternatives presented in the disjunctive form where we rely on the fact that $\phi \leftarrow \psi_1 \vee \psi_2$ and

$$(\phi \leftarrow \psi_1) \wedge (\phi \leftarrow \psi_2)$$

are propositionally equivalent and that $\phi \leftarrow \exists \vec{x} \psi$ and $\forall \vec{x}(\phi \leftarrow \psi)$ are logically equivalent provided the variables $\vec{x}$ do not occur in ϕ. We then split the conjuncted clauses, drop the outermost universal quantifiers, and use properties of identity. For instance, the first and the third clauses for $[\sqrt{z}]$ are in the following form just before the properties of identity are applied to them:

$$\begin{aligned}&[\sqrt{z}] = y \leftarrow z < 4 \wedge z = 0 \wedge 0 = y \\ &[\sqrt{z}] = y \leftarrow z \geq 4 \wedge z = 4{\cdot}x + i \wedge i < 4 \wedge \\ &\quad [\sqrt{x}] = s \wedge 4{\cdot}x + i < (s\mathbf{1})^2 \wedge s\mathbf{0} = y \ .\end{aligned}$$

1.5.5 Definition of $[\sqrt{z}]$ by generalized course of values definition with measure. We will now transform the contextual recursive formula for $[\sqrt{z}]$ in Fig. 1.5 into a *generalized* course of values definition with measure. The definition is given in Fig. 1.6 and contains *case terms* instead of case

formulas. Both definitions are satisfied by the same function and so they are equivalent. Case terms are significantly less readable than the case formulas and require explanation given in Paragraphs 1.5.6 through 1.5.9. We will define the functions applied in the case terms in Fig. 1.6 in Par. 1.5.16.

$$
\begin{array}{l}
[\sqrt{z}] = \mathbf{case} \\
\qquad \overline{sgn}(z{<_*}4) = 0 \to \mathbf{case} \\
\qquad\qquad\qquad\qquad \overline{sgn}(z{=_*}0) = 0 \to 0 \\
\qquad\qquad\qquad\qquad \overline{sgn}(z{=_*}0) = 1 \to 1 \\
\qquad \overline{sgn}(z{<_*}4) = 1 \to \mathbf{let} \\
\qquad\qquad\qquad\qquad 0, qr(z) = 0, x, i \to_{x,i} \\
\qquad\qquad\qquad\qquad\quad \mathbf{let} \\
\qquad\qquad\qquad\qquad\qquad 0, [\sqrt{x}] = 0, s \to_s \ \mathbf{case} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \overline{sgn}(4{\cdot}x + i{<_*}(s\mathbf{1})^2) = 0 \to s\mathbf{0} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \overline{sgn}(4{\cdot}x + i{<_*}(s\mathbf{1})^2) = 1 \to s\mathbf{1}
\end{array}
$$

Fig. 1.6. The definition of $[\sqrt{z}]$ with case terms.

$$
\begin{array}{l}
\mathbf{case} \\
\quad \tau = 0 \to \alpha_0 \\
\quad\ \vdots \\
\quad \tau = \underline{m-1}_m \to \alpha_{m-1} \\
\quad \tau = \underline{m}_m, x_1, \ldots, x_{n_m} \to_{x_1,\ldots,x_{n_m}} \alpha_m[x_1, \ldots, x_{n_m}] \\
\quad\ \vdots \\
\quad \tau = \underline{k-1}_m, x_1, \ldots, x_{n_{k-1}} \to_{x_1,\ldots,x_{n_{k-1}}} \alpha_{k-1}[x_1, \ldots, x_{n_{k-1}}]
\end{array}
$$

Fig. 1.7. The general form of a case term.

1.5.6 Case terms. The general form of case terms is given in Fig. 1.7 where $0 \leq m < k$ and all $n_m, \ldots n_{k-1}$ are positive numbers. For $m \leq j < k$ we abbreviate the local variables $x_1, \ldots x_{n_j}$ to $\vec{x}_j$. The local variables $\vec{x}_j$ may occur in the terms $\alpha_1[\vec{x}_j]$ but not in the term τ. Although all local variables in $\vec{x}_j$ must be pairwise different, the sets $\vec{x}_{j_1}$ and $\vec{x}_{j_2}$ may share variables whenever $j_1 \neq j_2$. The terms τ and α_j may contain additional *non-local* variables as parameters.

Before the case term from Fig. 1.7 is admitted as legal it must satisfy the following *completeness* condition:

$$\tau = 0 \vee \ldots \vee \tau = \underline{m-1}_m \vee \exists \vec{x}_i\, \tau = \underline{m}_m, \vec{x}_m \vee \ldots \vee \exists \vec{x}_{k-1}\, \tau = \underline{k-1}_m, \vec{x}_{k-1}\ . \tag{1}$$

We will need more general case terms guarded by a condition which is a formula of PA. The case term from Fig. 1.7 is legal under a *guard* ϕ if PA

proves the completeness condition (1) under the assumption ϕ. Note that an *absolute* case term, i.e. a case term without guard, is guarded by any condition.

When the guard of a case term is satisfied then the *discriminator* term τ denotes a *(disjoint) union value determined by* m, k, n_m, ..., n_{k-1}. The *tag* of the union value τ is the denotation of τ if $\tau < \underline{m}_m$ and the denotation of $H(\tau)$ if $\tau \geq \underline{m}_m$.

The reader will note that the alternatives in (1) are pairwise disjoint and, for instance, for $j_1 < m$ and $m \leq j_2 < k$ we have

$$\tau = \underline{j_2}_m, \vec{x}_{j_2} \rightarrow \tau \neq \underline{j_1}_m$$

because $j_1 < j_2 < j_2, \vec{x}_{j_2}$ holds.

The tag j of τ determines the meaning (denotation) of the case term as the meaning of the term α_j if $j < m$ and the meaning of $\alpha_j[\vec{x}_j]$ when $m \leq j < k$ with the assignments to the local variables $\vec{x}_j$ uniquely determined from the union value $\tau = \underline{j}_m, \vec{x}_j$. We abbreviate the case term from Fig. 1.7 to

$$case_{m,k}(\tau, \alpha_0, \ldots, \alpha_m[\vec{x}_m], \ldots) \ . \tag{2}$$

The generalized term (2) is *well-formed under a guard* ϕ if ϕ is a guard for the case term, for every $j < m$ the generalized term α_j is well-formed under the guard $\phi \wedge \tau = \underline{j}_m$, and for every $m \leq j < k$ the generalized term $\alpha_j[\vec{y}_j]$ is well-formed under the guard $\phi \wedge \tau = \underline{j}_m, \vec{y}_j$.

1.5.7 Let terms. *Let* terms are case terms with $m = 0$ and $k = 1$, i.e. one alternative case terms with local variables. For instance

let
$0, [\sqrt{x}] = 0, s \rightarrow_s \alpha[s]$.

Note that the tag 0 is superfluous and it is included in order to maintain the uniform form of case terms. Also note that the sole purpose of let terms is to introduce local variables.

$$\begin{array}{l}\phi \rightarrow \alpha = y \leftrightarrow \textbf{case} \\ \qquad\qquad \phi_0 \rightarrow \alpha_0 = y \\ \qquad\qquad \vdots \\ \qquad\qquad \phi_{m-1} \rightarrow \alpha_{m-1} = y \\ \qquad\qquad \phi_m[x_1, \ldots, x_{n_m}] \rightarrow_{x_1, \ldots, x_{n_m}} \alpha_m[x_1, \ldots, x_{n_m}] = y \\ \qquad\qquad \vdots \\ \qquad\qquad \phi_{k-1}[x_1, \ldots, x_{n_{k-1}}] \rightarrow_{x_1, \ldots, x_{n_{k-1}}} \alpha_{k-1}[x_1, \ldots, x_{n_{k-1}}] = y\end{array}$$

Fig. 1.8. The correspondence between the case terms and formulas.

1.5.8 Annotation of case terms. The connection of case terms to case formulas is established by the *annotation* of a case term with *assumption* formulas. The assumption formulas for the case term in Fig. 1.7 guarded by ϕ are $\phi_0, \ldots, \phi_{m-1}, \phi_m[\vec{y}_m], \ldots, \phi_{k-1}[\vec{y}_{k-1}]$ with the condition that we have

$$\phi \to \tau = \underline{j}_m \leftrightarrow \phi_j \tag{1}$$

whenever $j < m$ and

$$\phi \to \tau = \underline{j}_m, \vec{y}_j \leftrightarrow \phi_j[\vec{y}_j] \tag{2}$$

whenever $m \leq j < k$. The reader will note that this guarantees the completeness and pairwise exclusivity of the assumption formulas as well as the uniqueness of local variables. This annotated case term is abbreviated to

$$case_{m,k}(\tau, (\phi_0 \to \alpha_0), \ldots, (\phi_m \to_{\vec{y}_m} \alpha_m), \ldots) \; . \tag{3}$$

We can think of all case terms as being annotated because the unannotated term in Fig. 1.7 can be brought into the annotated form (3) by taking as its assumption formulas the corresponding identities on the left-hand-sides of (1) and (2). If we abbreviate the term (3) by α then the connection to the corresponding case formula is given in Fig. 1.8.

$$\begin{array}{l} D(\tau =_* \underline{0}_m, \alpha_0^\star, \\ \quad \ldots \\ \quad D(\tau =_* \underline{m-1}_m, \alpha_{m-1}^\star, \\ \qquad D(H(\tau) =_* \underline{m}_m, \alpha_m^\star[HT^1(\tau), \ldots, HT^{n_m-1}(\tau), T^{n_m}(\tau)], \\ \qquad\quad \ldots \\ \qquad\quad D(H(\tau) =_* \underline{k-2}_m, \alpha_{k-2}^\star[HT^1(\tau), \ldots, HT^{n_{k-2}-1}(\tau), T^{n_{k-2}}(\tau)], \\ \qquad\qquad\qquad\qquad \alpha_{k-1}^\star[HT^1(\tau), \ldots, HT^{n_{k-1}-1}(\tau), T^{n_{k-1}}(\tau)]) \ldots)) \ldots) \end{array}$$

Fig. 1.9. The term of PA which is the translation of the generalized term in Fig. 1.7.

1.5.9 The meaning of case terms. We have informally presented the intended meaning of case terms in Par. 1.5.6. Case terms may bind local variables and so they are instances of *variable binding operators*. The lambda terms $\lambda x.\tau$ are another well-known examples of such operators. Because of technical complications of bound variables we do not introduce the case terms formally into PA and treat them as concretely presented syntactic objects only on the level of meta-theory. We use α, β, as syntactic variables to range over *generalized* terms which are built up from the object terms of the language of PA by the case term constructs.

To every generalized term α we associate as its *translation* an object term $\alpha^\star$ in such a way that the denotations of $\alpha^\star$ in the models of PA are the

same as the intended denotations of the generalized terms. The translation is defined by recursion on the structure of generalized terms in such a way that the translation of the generalized term 1.5.6(2) (or 1.5.8(3)) is the object term given in Fig. 1.9.

1.5.10 Generalized course of values definitions with measure. We call

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

a *generalized* course of values definition with measure μ if its translation $f(\vec{x}) = \alpha^*[[f]^{\mu}_{\vec{x}}; \vec{x}]$ is a definition of f by course of values with measure μ. The function f is said to be defined by the generalized definition.

1.5.11 Regular generalized definitions. We can drop the guards around recursive applications of formulas in generalized definitions 1.5.10(1) if we restrict the recursive applications in α to *regular* applications. The idea is that for every recursive application $f(\vec{\rho})$ occurring in α PA proves

$$\psi_1 \wedge \ldots \wedge \psi_k \rightarrow \mu[\vec{\rho}] < \mu[\vec{x}]$$

where $\psi_1, \ldots, \psi_k$ are the annotation formulas *governing* the recursive occurrence in α. This means that if present the generalized definition (1) in an equivalent contextual form with a case formula ϕ:

$$f(\vec{x}) = y \leftrightarrow \phi[f; \vec{x}]$$

then the governing formulas are read off from the assumptions on case formulas enclosing the recursive application. For instance, for the single recursive application $[\sqrt{x}]$ in the definition in Fig. 1.5 we have

$$z \geq 4 \wedge z = 4{\cdot}x + i \wedge i < 4 \rightarrow x < z$$

and so the measure of the definition is the argument z itself.

Generalized definitions 1.5.10(1) with regular recursive applications are called *regular* definitions. For regular generalized definitions we not only have the extensional property that

$$\alpha^\star[[f]^{\mu}_{\vec{x}}; \vec{x}] = \alpha^\star[f; \vec{x}]$$

holds for all $\vec{x}$ but we also have a stronger intensional property that the recursive applications $f(\vec{\rho})$ can be *strictly* evaluated whenever their governing formulas hold. Strict evaluation (cf. Par. 1.7.3) means that the arguments $\vec{\rho}$ are evaluated to $\vec{\rho}_1$ before the rewrite rule $f(\vec{\rho}_1) \blacktriangleright \alpha[f; \vec{\rho}_1]$ is applied.

1.5.12 Extensions of PA by clausal definitions. Let us designate the current extension of PA by T. We now describe a *clausal* extension of T to S with a new function symbol f. The *clausal definition* of a function f is obtained by the unfolding of a regular generalized definition:

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

with α a well-formed generalized term under the guard $\top$. The translated formula $\alpha^\star$ must be in the language of S and there must be an extension by definition of the theory T to S_1 with a contextual definition $f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$ for some formula ϕ of the language of T. The formula ϕ is obtained from the course of values definition with measure $f(\vec{x}) = \alpha^\star[[f]^\mu_{\vec{x}}; \vec{x}]$

We obtain the clauses for f by transforming the generalized definition (1) into an equivalent contextual form with generalized case formulas

$$f(\vec{x}) = y \leftrightarrow \psi[f; \vec{x}]$$

and then by unfolding. The unfolded clauses are added as new axioms of S together with the single induction formula for $\phi[\vec{x}; f(\vec{x})]$. The theory S is equivalent to the theory S_1 and so it proves all induction formulas containing the new function symbol f. Moreover, because S_1 is an extension of T by definition, so is S.

For instance, the clausal definition of the function $[\sqrt{z}]$ given in Par. 1.5.1 is obtained from the regular generalized definition in Fig. 1.6 which is transformed into a form with generalized case formulas in Fig. 1.5. The clauses for $[\sqrt{z}]$ are unfolded as described in Par. 1.5.4.

1.5.13 $\mathcal{D}$-case terms. In this, and the next two paragraphs, we present three kinds of case terms which are *basic*.

The $\mathcal{D}$-case terms are absolute and come in two forms differing only in annotation:

$$case_{2,2}(\overline{sgn}(x), (x > 0 \to \alpha_0), (x = 0 \to \alpha_1)) \tag{1}$$
$$case_{2,2}(\overline{sgn}(P_*(\vec{\tau})), (P(\vec{\tau}) \to \alpha_0), (\neg P(\vec{\tau}) \to \alpha_1)) \ . \tag{2}$$

The unary function $\overline{sgn}$ has the following explicit definition:

$$\overline{sgn} = 1 \dot{-} x \ .$$

The reader will note that we have $\overline{sgn}(x+1) = 0$ and $\overline{sgn}(0) = 1$ and so PA proves

$$case_{2,2}(\overline{sgn}(0), \alpha_0, \alpha_1)^\star = case_{2,2}(1, \alpha_0, \alpha_1)^\star = \alpha_1^\star = D(0, \alpha_0^\star, \star_1)$$

and

$$case_{2,2}(\overline{sgn}(x+1), \alpha_0, \alpha_1)^\star = case_{2,2}(0, \alpha_0, \alpha_1)^\star = \alpha_0^\star = D(x+1, \alpha_0^\star, \alpha_1^\star) \ .$$

This means that $\mathcal{D}$-case terms are extensionally applications of the case discrimination (**if-then-else**) function D. However, there is a crucial intensional difference in the way the two are computed. Applications of the function D are computed strictly, i.e. arguments before the application D, while the $\mathcal{D}$-case terms, as all case terms, are computed non-strictly (see Par. 1.7.4).

For the first form of $\mathcal{D}$-case terms we have $\overline{sgn}(x) = 0 \leftrightarrow x > 0$ and $\overline{sgn}(x) = 1 \leftrightarrow x = 0$ and thus the conditions on the assumption formulas are satisfied. The second form of $\mathcal{D}$-case terms is extensionally just an instantiation of the first form with $P_*(\vec{\tau})$ where P_* is the characteristic function of the predicate P. The annotation with assumption formulas is legal because we have $\overline{sgn}\, P_*(\vec{\tau}) = 0 \leftrightarrow P(\vec{\tau})$ and $\overline{sgn}\, P_*(\vec{\tau}) = 1 \leftrightarrow \neg P(\vec{\tau})$.

1.5.14 Binary case terms. *Binary* case terms are absolute and have the annotated form

$$case_{0,2}(b(\tau), (\tau = z\mathbf{0} \to_z \alpha_0[z]), (\tau = z\mathbf{0} \to_z \alpha_1[z])) \tag{1}$$

where the unary function b is introduced into PA by a contextual definition:

$$\begin{aligned} b(x) = y \leftrightarrow {} & \exists z(x = z\mathbf{0} \wedge y = 0, z) \vee \\ & \exists z(x = z\mathbf{1} \wedge y = 1, z) \end{aligned}$$

whose existence and uniqueness conditions are easily provable.

The function $b(x)$ yields the union values $0, z$ or $1, z$ depending on the parity of the number x. The reader will note that the conditions on the assumption formulas which are $b(x) = 0, z \leftrightarrow x = z\mathbf{0}$ and $b(x) = 1, z \leftrightarrow x = z\mathbf{1}$ are satisfied.

1.5.15 Cartesian case terms. *Cartesian* case (let) terms have the annotated form

$$case_{0,1}(c(\tau), (\tau = v, w \to_{v,w} \alpha_0[v, w])) \tag{1}$$

where the unary function c is introduced into PA by a contextual definition:

$$c(x) = y \leftrightarrow x = 0 \wedge y = 0 \vee x > 0 \wedge y = 0, x\ .$$

The completeness condition (cf 1.5.6(1)) for Cartesian case terms is

$$\exists v \exists w\, c(\tau) = 0, v, w$$

and it is implied by any guard at least as strong as $\tau > 0$. Under such guard the condition on the assumption formula is satisfied because we have $c(\tau) = 0, v, w \leftrightarrow \tau = v, w$.

1.5.16 Derived case terms. *Derived* case terms have the functions applied in their discriminator terms definable by generalized definitions with basic case terms.

For instance, the two **let** terms used in the generalized definition in Fig. 1.6 are derived terms. The second of them is a **let** term as known from functional programming languages. The general form of such terms is:

$$case_{0,1}((0,\tau),(\tau = x \rightarrow_x \alpha_0[x]))\ .$$

The first **let** term of Fig. 1.6 applies the function qr yielding the pair consisting of the quotient and remainder after the division of z by 4. The term has the form:

$$case_{0,1}((0,qr(\tau)),(\tau = 4{\cdot}q + r \wedge r < 4 \rightarrow_{q,r} \alpha_0[q,r]))$$

with the condition on its assumption formula satisfied because we have:

$$0, qr(\tau) = 0, q, r \leftrightarrow \tau = 4{\cdot}q + r \wedge r < 4\ .$$

This **let** term is derived because the function $qr(z)$ can be defined with two nested binary case terms:

$$\begin{array}{lll} qr(z) = \textbf{case} & & \\ \quad b(z) = 0, u \rightarrow_u & \textbf{case} & \\ & \quad b(u) = 0, q \rightarrow_q q, 0 & \\ & \quad b(u) = 1, q \rightarrow_q q, 2 & \\ \quad b(z) = 1, u \rightarrow_u & \textbf{case} & \\ & \quad b(u) = 0, q \rightarrow_q q, 1 & \\ & \quad b(u) = 1, q \rightarrow_q q, 3\ . & \end{array}$$

The unfolded clausal definition of qr just before the identity optimizations is:

$$\begin{array}{l} qr(z) = q, 0 \leftarrow z = u\mathbf{0} \wedge u = q\mathbf{0} \\ qr(z) = q, 2 \leftarrow z = u\mathbf{0} \wedge u = q\mathbf{1} \\ qr(z) = q, 1 \leftarrow z = u\mathbf{1} \wedge u = q\mathbf{0} \\ qr(z) = q, 3 \leftarrow z = u\mathbf{1} \wedge u = q\mathbf{1} \end{array}$$

which becomes after the identity optimizations:

$$\begin{array}{l} qr(q\mathbf{00}) = q, 0 \\ qr(q\mathbf{10}) = q, 2 \\ qr(q\mathbf{01}) = q, 1 \\ qr(o\mathbf{11}) = q, 3\ . \end{array}$$

1.5.17 Pair case terms. *Pair* case terms are another example of derived terms and they have the annotated form

$$case_{1,2}(p(\tau),(\tau = 0 \rightarrow \alpha_0),(\tau = v, w \rightarrow_{v,w} \alpha_1[v,w]))$$

where the unary function p is introduced into PA by explicit clausal definition:

$p(0) = 0$
$p(v, w) = 1, v, w$.

The function $p(x)$ yields the union values 0 or $1, v, w$. The conditions on the assumption formulas are satisfied because we have $p(x) = 0 \leftrightarrow x = 0$ and $p(x) = 1, v, w \leftrightarrow x = v, w$.

The clausal definition of p is unfolded from the generalized definition with basic terms:

$$
\begin{array}{l}
p(x) = \textbf{case} \\
\qquad \overline{sgn}(x) = 0 \to \textbf{let} \\
\qquad\qquad\qquad c(x) = 0, v, w \to_{v,w} 1, v, w \\
\qquad \overline{sgn}(x) = 1 \to 0 \ .
\end{array}
$$

The reader will note that the let term in the definition is well-formed because its Cartesian case term has its guard satisfied because $\overline{sgn}(x) = 0 \leftrightarrow x > 0$.

A pair case term is, for instance, used in the generalized definition

$$
\begin{array}{l}
x \oplus y = \textbf{case} \\
\qquad p(x) = 0 \to y \\
\qquad p(x) = 1, v, w \to_{v,w} v, w \oplus y
\end{array}
$$

from which the clausal definition of the list concatenation function is unfolded:

$0 \oplus y$
$(v, w) \oplus y = v, w \oplus y$.

1.5.18 Defaults in clausal definitions. From a given clausal definition of a function f we can always remove the clauses $f(\vec{\rho}) = 0 \leftarrow \ldots$ and imply them by *default*.

For instance, the list indexing function from Par. 1.3.18 can be defined without its default clause as:

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

The clausal definition of the denotation function *Val* for expressions was given in Par. 1.3.25 without the default clauses. The full definition is

$Val(Cns(c), v) = c$
$Val(a, v) = 0 \leftarrow \neg\exists c\, a = Cns(c)$
$Val(Var(i), v) = (v)_i$
$Val(a) = 0 \leftarrow \neg\exists i\, a = Var(i)$
$Val(Add(a, b), v) = Val(a, v) + Val(b, v)$
$Val(c) = 0 \leftarrow \neg\exists a \exists b\, c = Add(a, b)$
$Val(Mul(a, b), v) = Val(a, v) \cdot Val(b, v)$
$Val(c) = 0 \leftarrow \neg\exists a \exists b\, c = Mul(a, b)$.

This definition has a cluttered look and it should demonstrate the rationale behind the default clauses.

1.5.19 Clausal definitions of predicates. Clausal definitions of predicates are obtained by systematic modifications of clausal definitions of their characteristic functions. We explain the modifications with an example of the clausal definition of list membership predicate $x\ \varepsilon\ a$ given in Par. 1.3.19. The clausal definition of its characteristic function $x\varepsilon_* a$ can be given with the default clauses included as follows:

$$
\begin{aligned}
&x\varepsilon_* 0 = 0\\
&x\varepsilon_*(y,a) = 1 \leftarrow x = y\\
&x\varepsilon_*(y,a) = 1 \leftarrow x \neq y \wedge x\varepsilon_* a = 1\\
&x\varepsilon_*(y,a) = 0 \leftarrow x \neq y \wedge x\varepsilon_* a = 0\ .
\end{aligned}
$$

Since we have $P_*(\vec{x}) = 1 \leftrightarrow P(\vec{x})$ and $P_*(\vec{x}) = 0 \leftrightarrow \neg P(\vec{x})$, we can write the definition as:

$$
\begin{aligned}
&x\not\varepsilon 0\\
&x\ \varepsilon\ y, a \leftarrow x = y\\
&x\ \varepsilon\ y, a \leftarrow x \neq y \wedge x\ \varepsilon\ a\\
&x\not\varepsilon y, a \leftarrow x \neq y \wedge x\not\varepsilon a\ .
\end{aligned}
$$

When we omit the first and last clauses by default we obtain the definition in Par. 1.3.19.

1.5.20 A digression on the role of notation in mathematics and in computer programming. We have been always puzzled by the question why, in the age of dazzling computer generated graphics, the theoreticians of computer programming can live with an ancient typewriter style notation for their programs. A glance in most books on functional programming languages reveals programs like:

```
filter []                 = []
filter (y:ys) | p y       = y : filter ys
filter (y:ys) | otherwise = filter ys  .
```

whereas the same definition in our mathematical notation looks as follows:

$$
\begin{aligned}
&\mathit{filter}(0) = 0\\
&\mathit{filter}(v,w) = v, \mathit{filter}(w) \leftarrow P(v)\\
&\mathit{filter}(v,w) = \mathit{filter}(w) \quad \leftarrow \neg P(v)\ .
\end{aligned}
$$

The reader will note that the test $P(v)$ in our clauses unfolds from a $\mathcal{D}$-case term and so the repeated evaluation of $P(v)$ is avoided. The same is achieved with the `otherwise` caluse in the first program.

The typewriter style is even more puzzling because the readability of programs is one of the most basic dictums of a good programming language design. The situation is similar in the existing automated theorem provers, or rather intelligent proof checkers. We typically see in them formulas like:

```
forall x ( Sqrt(x)^2 <= x and x < ( Sqrt(x) + 1 )^2 )
```

although mathematicians, who in general do not put so much emphasis on the readability as computer scientists, would not dream of presenting the formula in a form different than:

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

Part of the reason for this strange way of presentation lies in the traditionally understood role the syntax analysis should play in the design and presentation of computer programs. Language designers seem to insist on the 'what you see is what you get' approach to the writing of programs. This is because otherwise, they claim, the parsing of programs with fancy mathematical notation would be difficult. True, but the same computer scientists use, apparently without any thoughts, a mark-up language (most often Latex) when writing their papers. They know that they have to enter their formulas in a slightly less readable marked up form and for the price of this minor inconvenience the formulas will be presented in an esthetically pleasing form.

The reader has certainly noticed that we consistently use mathematical notation for function and predicate applications, logical connectives, and quantifiers. CL, our current implementation of the clausal language, has somewhat limited Latex-like possibilities of presenting formulas as

$$\exists y(x \oplus y \prec [\sqrt{x}] \wedge x \neq y) \ .$$

In order to simplify the syntax analysis, the above (somewhat nonsensical) formula is typed in as

```
\e y ( Prec(App(x,y),Sqrt(x)) & x != y) .
```

The cultural gap between what is the standard in the supposedly ultramodern computer science and in the old-fashioned mathematics is wide. Referees of our papers which we submit to computer science conferences almost invariably chide us for the funny looking syntax of our programs without apparently realizing that we do not use anything special, just the language of Peano arithmetic.

The clauses of the present CL system are still syntactically analyzed and so we cannot use fancy assumption formulas in our pattern matching presentation of arguments. We plan to change this dramatically with the prepared new version of CL. We plan to limit the syntax analysis to the parsing of simple terms and formulas. The case terms will be prompted in a top-down fashion. This, together with the enormous flexibility of extensible syntax of our case terms, will permit the use of arbitrary assumption formulas in the annotations of case terms. The reader will note that the 'syntax-less' syntax of our clausal definitions, and the almost unlimited power of extensibility of case terms, will be possible only because the constraints on case terms will have to be proved in the proof system before a new kind of a case term will be admitted.

The existing version of CL has a builtin set of pattern matching expressions by far surpassing the ones permitted in current functional languages such as Haskell. Patterns in Haskell are expressions (terms) whereas already the current version CL permits also pattern formulas such as $z = 4{\cdot}x + i \wedge i < 4$. The dramatic difference between the powers of expressibility of patterns as terms and patterns as formulas can be compared to the similarly dramatic difference between the power of explicit definitions $f(\vec{x}) = \tau$ in PA which are limited to applications in τ of previously introduced function symbols and the power of contextual definitions $f(\vec{x}) = y \leftrightarrow \phi$ where one has at disposal in the formula ϕ the full logical apparatus of propositional connectives and quantifiers.

We are convinced that the radical step of abandoning in the planned version of CL of the syntax analysis in favor of top-down prompting, together with the full extensibility of case terms, will unleash dramatically new ways of presentation of programs. The reader will note that this will happen fully within the language of Peano arithmetic without the addition of any reserved words or fancy syntax of current programming languages.

1.6 Limits of Provably Recursive Definitions in PA

1.6.1 Ordinal numbers less than ϵ_0. We wish to give a definition of a function V which is Δ_1-definable and so it is effectively computable (recursive) but not provably recursive in PA. The function V will grow so enormously fast that it will be practically uncomputable for all but the smallest arguments. The function will be defined with a measure into the initial segment of ordinal numbers less than the first *epsilon* number ϵ_0. The number satisfies the identity $\epsilon_0 = \omega^{\epsilon_0}$.

It is well-known that every ordinal number $\prec \epsilon_0$ can be uniquely denoted in the *Cantor's normal form* by a term

$$\omega^{\alpha_1} + \omega^{\alpha_2} \ldots + \omega^{\alpha_n} + 0 \tag{1}$$

where $n \geq 0$, $\alpha_1 \succeq \alpha_2 \succeq \ldots \succeq \alpha_n$, and the terms α_i are constructed similarly.

Let $\alpha \prec \epsilon_0$ be an ordinal whose Cantor's normal form is (1) and $\beta \prec \epsilon_0$ an ordinal whose Cantor's normal form is

$$\omega^{\beta_1} + \omega^{\beta_2} \ldots + \omega^{\beta_m} + 0 \; .$$

The *natural sum* $\alpha \sharp \beta$ of the two ordinals is the ordinal less than ϵ_0 whose Cantor's normal form is

$$\omega^{\gamma_1} + \omega^{\gamma_2} \ldots + \omega^{\gamma_{n+m}} + 0$$

where each γ_i is some α_j or β_j and all α_j and β_j are some γ_k. It should be clear that we have $\alpha \sharp \beta = \beta \sharp \alpha \succeq \alpha, \beta$.

It is not hard to see that every ordinal number less than ϵ_0 is denoted (but not uniquely) by a term built up from 0 by the binary operation $\alpha \sharp \omega^\beta$. We abbreviate $(\alpha \sharp \omega^\beta) \sharp \omega^\gamma$ to $\alpha \sharp \omega^\beta \sharp \omega^\gamma$. The advantage of this representation is that we do not have to order the exponents of ω in non-increasing order and, for instance, the terms $\alpha \sharp \omega^\beta \sharp \omega^\gamma$ and $\alpha \sharp \omega^\gamma \sharp \omega^\beta$ denote the same ordinals.

1.6.2 Coding of ordinals $\prec \epsilon_0$ in $\mathbb{N}$. We encode the ordinals less than ϵ_0 into natural numbers by encoding the ordinal number 0 by $0 \in \mathbb{N}$ and the ordinal number $\alpha \sharp \omega^\beta$ by the natural number b, a where a and b encode α and β respectively. It should be clear that every natural number is a code of an ordinal. Note that we have $b, c, a \neq c, b, a$ whenever $b \neq c$ but both numbers code the same ordinal: $\alpha \sharp \omega^\beta \sharp \omega^\gamma$.

We 'overload' the ordinal function $\alpha \sharp \omega^\beta$ and explicitly introduce it as a binary function over $\mathbb{N}$:

$$a \sharp \omega^b = b, a \ .$$

In order to distinguish the two functions we will use variables α, β, γ, ... to range over ordinals and the variables a, b, c, ... to range over the corresponding natural numbers coding ordinals.

The ternary function $a \sharp \omega^b \cdot k$ over $\mathbb{N}$ yielding the code of the ordinal

$$\alpha \sharp \overbrace{w^\beta \sharp \ldots \sharp w^\beta}^{k}$$

is introduced into PA by primitive recursion:

$$a \sharp \omega^b \cdot 0 = a$$
$$a \sharp \omega^b \cdot (k{+}1) = a \sharp \omega^b \cdot k \sharp \omega^b \ .$$

The binary function ω_k^i over $\mathbb{N}$ yielding the code of the ordinal designated by the same symbol is introduced into PA by primitive recursion:

$$\omega_0^i = 0 \sharp \omega^0 \cdot i$$
$$\omega_{k+1}^i = 0 \sharp \omega^{\omega_k^i} \ .$$

We also overload the ordinal relation $\prec$ and use it as a binary relation $a \prec b$ over $\mathbb{N}$ holding of codes if $\alpha \prec \beta$ holds. The relation is provably recursive in $I\Sigma_1$, and although transitive and irreflexive, it is not an order because the law of trichotomy does not hold for it (this is because of the non-unique coding of ordinals). We will need a special case of $\prec$ as a ternary relation $a \prec \omega_k^i$ which is introduced into PA by course of values recursion in a:

$$0 \prec \omega_k^i \leftarrow k > 0 \vee i > 0$$
$$a \sharp \omega^0 \prec \omega_0^{i+1} \leftarrow a \prec \omega_0^i$$
$$a \sharp \omega^b \prec \omega_{k+1}^i \leftarrow a \prec \omega_{k+1}^i \wedge b \prec \omega_k^i \ .$$

1.6.3 Arithmetization of fundamental sequences for limit ordinals. We call the code a of an ordinal less than ϵ_0 a *right successor* code if $a = b \sharp \omega^0$ for some b. A non-zero code a which is not a right successor is a *right limit* code. The code a is a right limit iff $a = c, b = b \sharp \omega^c$ for some b and $c > 0$.

We arithmetize the *fundamental sequences* $(\alpha)[k]$ for limit ordinals α (see [19]) by primitive recursion:

$$(a \sharp \omega^{b \sharp \omega^0})[k] = a \sharp \omega^b \cdot (k+1)$$
$$(a \sharp \omega^{b \sharp \omega^c})[k] = a \sharp \omega^{(b \sharp \omega^c)[k]} \leftarrow c \neq 0 \ .$$

$I\Sigma_1$ proves

$$a \neq 0 \rightarrow (a)[k] \prec a \ .$$

1.6.4 A function growing faster than functions provably recursive in PA. Consider the following clauses for the unary function symbol V:

$$V(0 \sharp \omega^0 \cdot k) = 0 \sharp \omega^0 \cdot k$$
$$V(a \sharp \omega^{b \sharp \omega^0} \sharp \omega^0 \cdot k) = V((a \sharp \omega^{b \sharp \omega^0})[k] \sharp \omega^0 \cdot 1)$$
$$V(a \sharp \omega^{b \sharp \omega^c} \sharp \omega^0 \cdot k) = V((a \sharp \omega^{b \sharp \omega^c})[k] \sharp \omega^0 \cdot k) \leftarrow c \neq 0 \ .$$

The clauses for V are well-discriminated because every number x can be uniquely written as $x = d \oplus e$ where $\forall y (y \, \varepsilon \, d \rightarrow y = 0)$ and

$$e = 0 \vee \exists a \exists b \exists c \, e = (c, b), a \ .$$

Note that then d codes a finite ordinal, i.e. $d = 0 \sharp \omega^0 \cdot n$ for $n = L(d)$, and e is either 0 or it codes a transfinite ordinal $e = a \sharp \omega^{b \sharp \omega^c}$. Thus the first clause, where $e = 0$, is discriminated from the remaining two where $e \neq 0$. The last two clauses are discriminated on whether $b \sharp \omega^c$ is a right successor ($c = 0$) or a right limit ($c \neq 0$). We have

$$a \sharp \omega^{b \sharp \omega^c} \sharp \omega^0 \cdot k \succ (a \sharp \omega^{b \sharp \omega^c})[k] \sharp \omega^0 \cdot \max(1, k)$$

and so the recursive argument in the last two clauses decreases in the relation $\prec$. This relation is not an order over $\mathbb{N}$ so it cannot be a well-order. However, the relation is well-founded and so the recursion has to stop after finitely many steps.

The problem is that PA is not strong enough to prove (in the form of a schema of well-founded induction on $\prec$) that the relation $\prec$ is well-founded. This means that the function V, although Δ_1-definable, i.e. recursive, is not provably recursive and so it cannot be introduced into PA by extensions by definitions.

On the other hand, for every $m > 0$, $n > 0$ and any i the fragment $I\Sigma_{m+n-1}$ proves that the restriction of $\prec$ to codes $\prec w_n^i$, i.e. the explicitly defined relation

$$a \prec_{n,i} b \leftrightarrow a \prec b \wedge b \prec w_{\underline{n}_m}^{i_m} \ ,$$

is well-founded. This means that $I\Sigma_n$ proves a schema of well-founded induction on $\prec_{n,i}$ for Σ_1-formulas.

1.6.5 Extended Ackermann-Péter function. We have chosen the above form of definition of the function V for two reasons. The first one is that the function has a close connection to an extension into transfinite codes of the well-known Ackermann-Péter function from which we can define the so called fast growing hierarchy of functions. The second reason is a close connection to intensional primitive recursive functionals (see Par. 1.9.27).

We define a binary function A over $\mathbb{N}$ by explicit definition:

$$A(a,n) = L\,V(0\sharp\omega^a\sharp\omega^0{\cdot}n)\ .$$

The definition must be done in the standard model of PA because the function A cannot be introduced into PA. The clauses for V imply (in the model) the following recurrences for the function A:

$$\begin{aligned} A(0,n) &= n+1 \\ A(a\sharp\omega^0,0) &= A(a,1) \\ A(a\sharp\omega^0,n+1) &= A(a,A(a\sharp\omega^0,n)) \\ A(a\sharp\omega^b,n) &= A((a\sharp\omega^b)[n],n) \leftarrow b\neq 0\ . \end{aligned}$$

The binary Ackermann-Péter function which grows faster than all primitive recursive functions and satisfies:

$$Ack(0,n) = n+1 \tag{1}$$

$$Ack(m+1,0) = Ack(m,1) \tag{2}$$

$$Ack(m+1,n+1) = Ack(m,Ack(m+1,n)) \tag{3}$$

can be now explicitly defined as $Ack(m,n) = A(0\sharp\omega^0{\cdot}m,n)$. The reader will note that we have $0\sharp\omega^0{\cdot}m \prec \omega_1^1$ and so the codes of ordinals used in the computation of Ack do not even start to exploit the incredible rate of growth of the function A applied to the codes of larger ordinals.

1.6.6 Fast growing hierarchy. The extended Ackermann-Péter function can be also used to define a fast growing hierarchy of functions F_α similar to the ones studied by Wainer [27] and Schwichtenberg [20] (see also Rose [19]). To that end we assign to every ordinal $\alpha \prec \epsilon_0$ as its *canonical* code a the code obtained from the Cantor's normal form for α by replacing $+$ by natural sums $\sharp$. We then explicitly define:

$$F_\alpha(x) = A(a,x)$$

and obtain the following properties of functions F_α:

$$\begin{aligned} F_0(x) &= x+1 \\ F_{\alpha+1}(x) &= F_\alpha^{x+1}(1) \\ F_\alpha(x) &= F_{(\alpha)[x]}(x) \qquad \alpha \text{ is a limit ordinal.} \end{aligned}$$

We have chosen the hierarchy functions in such a way that for finite ordinals m we have $F_m(n) = Ack(m,n)$.

1.6.7 Provably recursive restrictions of V. It is well-known that the function F_{ϵ_0} satisfying:

$$F_{\epsilon_0}(x) = F_{(\epsilon_0)[x]}(x) = F_{\omega^1_x}(x) = A(\omega^1_x, x) = \\ LV(0\sharp\omega^{\omega^1_x}\sharp\omega^0{\cdot}x) = LV(\omega^1_{x+1}\sharp\omega^0{\cdot}x)$$

is not provably recursive in PA. This is another reason why the function V cannot be provably recursive.

Let us now investigate restrictions of V which are provably recursive in PA. The unary function $V_e(a)$ is introduced into PA by explicit clausal definition:

$V_e(0\sharp\omega^0{\cdot}n) = 0\sharp\omega^0{\cdot}n$
$V_e(a\sharp\omega^{b\sharp\omega^0}\sharp\omega^0{\cdot}n) = a\sharp\omega^b{\cdot}(n{+}1)\sharp\omega^0{\cdot}1$
$V_e(a\sharp\omega^{b\sharp\omega^c}\sharp\omega^0{\cdot}n) = a\sharp\omega^{(b\sharp\omega^c)[n]}\sharp\omega^0{\cdot}n \leftarrow c \neq 0$.

and its binary iteration $V^n_e(p)$ is introduced by primitive recursion:

$V^0_e(p) = p$
$V^{n+1}_e(p) = V_e\,V^n_e(p)$.

The standard model of PA then satisfies:

$$\exists n\exists m\, V^n_e(a) = 0\sharp\omega^0{\cdot}m \tag{1}$$

which is the condition of regularity for an expansion of the standard model of PA by a function V defined by minimalization:

$$V(a) = T\,\mu_p[V^{H(p)}_e(a) = 0\sharp\omega^0{\cdot}T(p)] \tag{2}$$

The clauses for V are then provable in PA extended by the axiom (2). This means that Peano arithmetic is not strong enough to prove the existence condition (1) because otherwise V would be provably recursive.

On the other hand, for $k \geq 1$ and any i the fragment $I\Sigma_k$ proves

$$a \prec \omega^{\underline{i}_m}_{\underline{k}_m} \rightarrow \exists n\exists m\, V^n_e(a) = 0\sharp\omega^0{\cdot}m \tag{3}$$

and so the function $V_{k,i}$ satisfying

$V_{k,i}(d) = 0\sharp\omega^0{\cdot}n \leftarrow d \prec \omega^{\underline{i}_m}_{\underline{k}_m} \wedge d = 0\sharp\omega^0{\cdot}n$
$V_{k,i}(d) = V_{k,i}(a\sharp\omega^b{\cdot}(n{+}1)\sharp\omega^0{\cdot}1) \leftarrow d \prec \omega^{\underline{i}_m}_{\underline{k}_m} \wedge d = a\sharp\omega^{b\sharp\omega^0}\sharp\omega^0{\cdot}n$
$V_{k,i}(d) = V_{k,i}(a\sharp\omega^{(b\sharp\omega^c)[n]}\sharp\omega^0{\cdot}n) \leftarrow d \prec \omega^{\underline{i}_m}_{\underline{k}_m} \wedge d = a\sharp\omega^{b\sharp\omega^c}\sharp\omega^0{\cdot}n \wedge c \neq 0$

is provably recursive in $I\Sigma_k$ and the standard model of PA satisfies

$$a \prec \omega^{\underline{i}_m}_{\underline{k}_m} \rightarrow V_{k,i}(a) = V(a)\ .$$

For any $k \geq 1$ and any ordinal $\alpha \prec \omega^1_k$ with the canonical code a we have $\alpha \prec \omega^i_{k-1}$ for some $i > 0$ and so $\omega^\alpha \prec \omega^i_k$. Thus $0\sharp\omega^a\sharp\omega^0{\cdot}x \prec \omega^{\underline{i}_m}_{\underline{k}_m}$ and, since

$$F_\alpha(x) = A(a,x) = LV(0\sharp\omega^a\sharp\omega^0{\cdot}x) = LV_{k,i}(0\sharp\omega^a\sharp\omega^0{\cdot}x)\ ,$$

the function F_α is provably recursive in $I\Sigma_k$. Hence all functions F_α where $\alpha \prec \epsilon_0$ are provably recursive in PA.

1.7 Computation of Clausal Definitions

We have introduced clausal definitions with the help of generalized terms in such a way that to every clausal definition there is a regular generalized course of values definition with measure. We will now show how to use the generalized definitions for effective (and efficient) computation. Annotations on case terms play no rule in their computation.

Computation, like provability, has to do with the shape of terms rather than with their denotations and so it is an intensional affair concerned with the syntactical form of definitions of functions rather than with the functions themselves.

1.7.1 Mixed numerals. We will compute over the class of *mixed numerals* which are the least set of terms containing the constant 0, with every terms ρ_1, ρ_2 also the terms $\rho_1\mathbf{1}$ and (ρ_1,ρ_2), and with every term $\rho \not\equiv 0$ also the term $\rho\mathbf{0}$. Binary and pair numerals are thus proper subsets of mixed numerals.

In contrast to monadic, binary, or pair numerals, the mixed numerals do not represent the natural numbers uniquely. For instance, the number three is denoted by four mixed numerals which are all different as terms:

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

1.7.2 Mixed definitions of functions. We will compute functions defined by *mixed definitions* which are certain regular generalized course of values definitions with measure. The generalized definitions use $\mathcal{D}$, binary, Cartesian, and case terms derived from these. The object terms in the definitions are built up from the constant 0 and variables by applications of binary successors, pairing, and of previously defined functions. Decimal numerals are abbreviations for the corresponding binary numerals and predicates are computed from definitions of their characteristic functions.

We distinguish two classes of mixed definitions. Mixed definitions in the *narrow* sense, or *narrow* mixed definitions, are such that the measure terms in recursive generalized definitions apply only functions which are also definable by narrow mixed definitions. Unlimited mixed definitions allow functions in the measures to be introduced into PA by any extension by definition, say by a contextual definition.

The difference in the two classes is substantial. It is easy to see that the narrow mixed definitions define only primitive recursive functions while arbitrary mixed definitions define all provably recursive functions of PA, i.e. the $\prec\epsilon_0$-recursive functions.

1.7.3 Reductions. We compute the well-formed generalized terms α by *reductions*. The terms α must be *closed*, i.e. without free (non-local) variables. Reductions proceed by locating in α, which is not yet a mixed numeral, the leftmost and innermost generalized term, called the *redex*, and by rewriting it with a corresponding generalized term, called *contractum*. The rewriting is repeated until the generalized term being reduced becomes a mixed numeral. For the full list of redex-contracta pairs see Par. 1.7.4. We write $\alpha \blacktriangleright \alpha_1$ if α_1 is obtained from α by a finite number (possibly zero) of reductions.

Reductions are simpler to understand than the description of computation by compilation (translation) into a well understood code whether machine or abstract. Experienced reader can easily visualize the translation from the description of reductions. In order to make the description of reductions as simple as possible we use substitution of closed terms for free variables. Substitution will be replaced in the translated code by environments which will be accessed for the values of variables (for the idea of environments see Par. 1.9.13).

It can be shown that every well-formed closed generalized term α reduces after finitely many reductions to a mixed numeral ρ, i.e. $\alpha \blacktriangleright \rho$. Mixed numerals are *irreducible* because they do not contain any redexes. Although α and ρ are possibly different as terms, they have the same denotation, i.e. $\alpha = \rho$ is satisfied in the standard model (extended with generalized terms), or alternatively PA proves $\alpha^* = \rho$.

1.7.4 Redexes and their contracta. In the following we list all redex contracta pairs using the notation $\blacktriangleright$. We denote by α closed generalized terms (which in special case are terms of PA), by τ closed terms of (the current extension of) PA, and ρ by the mixed numerals. The reader will note that for all redex contracta pairs $\alpha_1 \blacktriangleright \alpha_2$ the terms α_1 and α_2 denote the same natural number, i.e. PA proves $\alpha_1^\star = \alpha_2^\star$. The reduction

$$\rho\mathbf{00} \blacktriangleright \rho\mathbf{0}$$

removes one leading zero. Reductions

$$\begin{aligned} \overline{sgn}(0) &\blacktriangleright 0\mathbf{1} \\ \overline{sgn}(\rho\mathbf{1}) &\blacktriangleright 0 \\ \overline{sgn}(\rho\mathbf{0}) &\blacktriangleright 0 \qquad \text{if } \rho \not\equiv 0 \\ \overline{sgn}(\rho_1, \rho_2) &\blacktriangleright 0 \end{aligned}$$

simplify the discriminators of $\mathcal{D}$-case terms. Reductions

$$
\begin{aligned}
b(0) &\blacktriangleright 0, 0 \\
b(\rho\mathbf{0}) &\blacktriangleright 0, \rho \qquad \text{if } \rho \neq 0 \\
b(\rho\mathbf{1}) &\blacktriangleright 0\mathbf{0}, \rho \\
b(\rho_1, \rho_2) &\blacktriangleright 0, \rho \qquad \text{where } \rho\mathbf{0} = \rho_1, \rho_2 \qquad (1) \\
b(\rho_1, \rho_2) &\blacktriangleright 0\mathbf{1}, \rho \qquad \text{where } \rho\mathbf{1} = \rho_1, \rho_2 \qquad (2)
\end{aligned}
$$

simplify the discriminators of binary case terms. Reductions

$$c(\rho) \blacktriangleright 0, \rho_1, \rho_2 \qquad \text{where } \rho = \rho_1, \rho_2 \qquad (3)$$

simplify the discriminators of Cartesian case terms. The reader will note that in any such then we must have $\rho \not\equiv 0$. The reduction

$$f(\vec{\rho}) \blacktriangleright \alpha[f, \vec{\rho}]$$

opens a mixed definition $f(\vec{x}) = \alpha[f; \vec{x}]$.

The final set of reductions involves case terms (both basic and derived)

$$case_{m,k}(\rho, \alpha_0, \ldots, \alpha_m[\vec{y}_m], \ldots) \blacktriangleright \alpha \qquad (4)$$

where if $\rho = \underline{j}_b$ with $j < m$ then $\alpha \equiv \alpha_j$ and if $\rho = \underline{j}_b, \vec{\rho}_j$ then $\alpha \equiv \alpha_j[\vec{\rho}_j]$. The reader will note that the well-formedness restrictions on case terms guarantee that the term ρ denotes a union value determined by m, k, n_m, $\ldots$, n_{k-1}.

Reductions (1), (2) involve and reductions (3), (4) may involve conversions between representations of mixed numerals. The no conversion condition for the reduction (3) is $\rho \equiv \rho_1, \rho_2$ and for (4) the condition is $\rho \equiv \underline{j}_b$ with $j < m$ and $\rho \equiv \underline{j}_b, \vec{\rho}_j$ with $m \leq j < k$. Conversions are effectively computable in linear space and polynomial time (see [26]), but nevertheless they are time consuming.

1.8 Data Types

Conversions in the marked reductions of Par. 1.7.4 are quite time consuming and we would like to avoid them. This can be done by means of syntactic restrictions on the form of mixed definitions by means of *typing*. As an additional bonus, the well-typed definitions of functions can be efficiently compiled into machine code (or to *C* for that matter).

Types are intuitively certain sets of data values. Since our values are natural numbers, the types are certain subsets of $\mathbb{N}$, and hence can be defined by unary *type* predicates. With a proof system integrated with a programming language we can define arbitrary type predicates and provide proofs that our definitions of functions yield values of certain types when provided with arguments of some types. We, however, feel that the full power of the proof system should be reserved to the proofs of theorems about our functions rather than for the typing.

We think that the type predicates should form a rather weak class so the typing can be performed in a decidable manner. Moreover, as mentioned above, the typing should prevent needless conversion of representation of data values and should enable efficient compilation. For that the types should reflect the memory representation of data structures and it seems to us that the Pascal-style typing is a natural choice. Pascal-style type predicates can be then naturally extended to *polymorphic* type predicates in the style of ML.

Our basic philosophy of typing differs from the standard theory of types in programming languages (see for instance [17]) in that the latter explicates the types by many sorted theories. This sharply contrasts with our approach where the definitions of programs are typeless (all functions are over $\mathbb{N}$). We understand the typing only as an add on feature which enables efficient compilation and can catch many trivial errors. Our definitions are computed independently of, and unaffected by, the polymorphic types assigned to them.

There are two aspects to typing. The extensional aspect deals with the specification of the class of type predicates. The intensional aspect deals with a decidable typing calculus for deriving assertions about the types of terms, generalized terms, and definitions. Well-typed closed generalized terms then reduce without any conversion of mixed numerals. Well-typed definitions of functions and predicates with the polymorphic types instantiated, can be efficiently translated into machine code.

1.8.1 Pascal-style type predicates. Pascal-style type predicates can be defined by ordinary clausal definitions of predicates whose syntax is severely restricted. We will by convention choose the predicate symbols to be postfix operators whose applications are written as $x : T$ and read as (the value of) x is of type T. The reader should view the combination $: T$ as the symbol of a type predicate rather than to view : as a binary infix operator taking a value and a type. In the following we will use S, T as meta-variables ranging over the symbols for type predicates.

Pascal-style type predicates can be classified as *primitive*, *Cartesian*, *list*, and *union* type predicates.

1.8.2 Primitive type predicates. The type of natural numbers is given by the predicate N with the following explicit clausal definition:

$x : N$.

The type N is the type of natural numbers. Values of type N can be represented in computer memory as the bignums of LISP, i.e. as numbers in the unlimited precision.

In a concrete implementation of CL we can choose more kinds of primitive type predicates. Unsigned and signed limited precision numbers represented in memory by, say, 32-bit words come to mind.

1.8.3 Cartesian type predicates. Suppose that $T_1, \ldots T_n$ have been introduced into PA as type predicates. The unary type predicate T explicitly defined by the clausal definition

$$x_1, \ldots, x_n : T \leftarrow x : T_1 \wedge \ldots \wedge x : T_n$$

is the type predicate of *Cartesian product* of $T_1, \ldots, T_n$.

For instance, the predicate

$$x, y, z : N_3 \leftarrow x : N \wedge x : N \wedge x : N$$

is the type of triples of natural numbers. The reader will note that the same predicate has a simpler definition

$$x, y, z : N_3$$

which, however, does not have the prescribed syntactic form.

1.8.4 List type predicates. Suppose that T hat been introduced into PA as a type predicate. The predicate

$$0 : \mathit{List}(T)$$
$$x, y : \mathit{List}(T) \leftarrow x : T \wedge y : \mathit{List}(T) \ .$$

is the type predicate of *lists* with elements from T. The reader should read the combination : $\mathit{List}(T)$ as a 'structured' predicate symbol rather than as an operator taking types and yielding types.

Note that in the absence of polymorphism, the Pascal-style typing requires that, for instance, the predicates $\mathit{List}(N)$ and $\mathit{List}(N_3)$ should be introduced as two different predicates. The reader will also note that we have $\forall x\, x : \mathit{List}(N)$.

The memory representation of values of type $\mathit{List}(T)$ can be a 32-bit pointer which is either *nil* for empty lists or a pointer to a Cartesian pair $T \times \mathit{List}(T)$.

1.8.5 Union type predicates. *Union* type predicates hold of union values. Rather than discussing the general form of union types we illustrate them with an example. Consider the following clausal definition:

$$0 : \mathit{Bt}(T)$$
$$1, n, l, r : \mathit{Bt}(T) \leftarrow n : T \wedge l : \mathit{Bt}(T) \wedge r : \mathit{Bt}(T)$$

where the type predicate T has been previously introduced into PA. A more familiar definition of the same predicate is with the help of two constructor functions $E = 0$ and $\mathit{Nd}(n, l, r) = 1, n, l, r$ as

$$E : \mathit{Bt}(T)$$
$$\mathit{Nd}(n, l, r) : \mathit{Bt}(T) \leftarrow n : T \wedge l : \mathit{Bt}(T) \wedge r : \mathit{Bt}(T) \quad .$$

Values of type $Bt(T)$ are union values determined by 1, 2, and 3 (the arity of the value with the tag 1) where the values $1, n, l, r$ have the components appropriately typed.

Values of this type may be represented by pointers. The value E by *nil* and the values with the tag 1 by a pointer to a Cartesian quadruple $N \times N \times Bt(T) \times Bt(T)$ with the first component being the tag 1.

The reader will note that we have

$$x : Bt(T) \rightarrow x : Bt(N)$$

for any union type predicate $Bt(T)$.

1.8.6 ML-style type predicates. *Polymorphic* typing in the style of the programming language ML corrects the greatest shortcoming of Pascal-style typing where the symbol T in the application $x : List(T)$ is a part of the predicate symbol rather than a second argument to the binary predicate $x : List(t)$ where the *type parameter* t ranges over all Pascal-style typing predicates. We cannot achieve this without going to the second-order arithmetic because in PA both arguments x and t must be natural numbers. Hence, the argument t does not range over type predicates but over the codes of type predicates. The polymorphic list predicate is then introduced into PA as a binary predicate by the following clausal definition:

$0 : List(t)$
$x, y : List(t) \leftarrow x \mathbf{:} t \wedge y : List(t)$

which applies a binary *universal* typing predicate $x \mathbf{:} t$. The reader will note the bold colon symbol signifying that this is a binary infix predicate rather than a part of the predicate symbol.

The universal predicate, which will be discussed in Par. 1.8.9, is such that PA proves

$$x \mathbf{:} {}'T(t_1, \ldots, t_n) \leftrightarrow x : T(t_1, \ldots, t_n) \tag{1}$$

for every type predicate T with n type parameters, i.e. an $(n+1)$-ary type predicate $x : T(t_1, \ldots, t_n)$, introduced into PA. Here ${}'T(\tau_1, \ldots, \tau_n)$ is an abbreviation for a certain term of PA which denotes the code of $x : T(\tau_1, \ldots, \tau_n)$. When T has no paramaters, i.e. when $x : T$ is a unary type predicate, then (1) should be read as

$$x \mathbf{:} {}'T \leftrightarrow x : T \ .$$

This style of polymorphism which permits type parameters in the form of variables ranging over (codes of) Pascal-style types but no new types is called *predicative polymorphism* (see, for instance, [17]).

With the polymorphic list type predicate one now takes arbitrary type predicate T and writes $x : List({}'T)$ instead of having to introduce the type predicate $: List(T)$ separately for each T.

We introduce a convention of writing the *type applications* $'T(\tau_1, \ldots, \tau_n)$ for $n \geq 0$ in an abbreviated form $T(\tau_1, \ldots, \tau_n)$. We can thus write $x : List(T)$ instead of $x : List('T)$. The reader will note that by (1) the last is equivalent to $x : 'List('T)$ which can be abbreviated to $x : List(T)$.

Polymorphic type of binary trees is defined by the type predicate with one type parameter:

$E : Bt(t)$
$Nd(n, l, r) : Bt(t) \leftarrow n : t \wedge l : Bt(t) \wedge r : Bt(t)$.

We can now, for instance, assert $x : Bt\ List(N_3)$ which is an abbreviation for $x : Bt('List('N_3))$ or equivalently $x : Bt\ List(N_3)$ which is an abbreviation for $x : 'Bt('List('N_3))$.

1.8.7 Polymorphic vector type predicate. Vectors (arrays) are extensionally lists of given length. The importance of vectors lies in their intensional properties of having good memory representations. It would seem that we need two kinds of vector types. The ternary polymorphic type predicate $x : Vect_1(n, t)$ holding of *fixed* vectors is introduced into PA by primitive recursion:

$0 : Vect_1(0, t)$
$v, w : Vect_1(n + 1, t) \leftarrow v : t \wedge w : Vect_1(n, t)$.

For every positive number n and every type predicate T fixed vectors $x : Vect_1(\underline{n}_b, 'T)$ are represented in memory just like the Pascal arrays of type **array**$[0..(n-1)]$ **of** T.

The binary polymorphic type predicate $x : Vect(t)$ holding of *flexible* vectors is introduced explicitly into PA by:

$n, x : Vect(t) \leftarrow x : Vect_1(n, t)$.

For every type predicate T the flexible vectors $x : Vect('T)$ are represented in computer memory as pointers to *dependent* Cartesian pairs of type $N \times Vect_1(n, 'T)$ where the value of the first component is n. The first component thus holds the length of the vector.

As it turns out, the practice of programming in our programming language Trilogy II, in which we have implemented the system CL, shows that fixed vectors are not used at all. For this reason we will work below with the flexible vectors only.

1.8.8 Operations on vectors. Although vectors are extensionally just lists, it is of advantage to select some operations on them as basic ones and enforce by means of typing that the vectors are efficiently operated upon through these operations (see Par. 1.8.17). The restrictions will guarantee efficient compilation of vector operations into machine code provided the problems of in-place modification (see subsection 1.11) have been solved.

There are three operations on vectors: creation of new vectors $New(n, v)$, indexing of vectors $a[i]$, and modification of vectors $a[i := v]$. The operations are introduced into PA with the help of auxiliary operations performing the same tasks with fixed vectors whose clausal definitions are:

$New_1(0, v) = 0$
$New_1(n + 1, v) = v, New_1(n, v)$
$(v, a)[0]_1 = v$
$(v, a)[i + 1]_1 = a[i]_1$
$(v, a)[0 := w]_1 = w, a$
$(v, a)[i + 1 := w]_1 = v, a[i := w]_1$.

The operations for (flexible) vectors are then introduced into PA by explicit clausal definitions:

$New(n, v) = n + 1, New_1(n + 1, v)$
$(n, a)[i] = a[i]_1 \leftarrow i < n$
$(n, a)[i] = a[0]_1 \leftarrow i \geq n$
$(n, a)[i := v] = n, a[i := v]_1 \leftarrow i < n$
$(n, a)[i := v] = n, a \leftarrow i \geq n$.

The reader will note that in the application $Nev(n, v)$ the number n is not the length of the vector but rather its highest index. This means that the result is a vector with at least one component. This arrangment is important for the smooth typing of vector operations where the indexing operation yields an element of its argument vector (rather than the default 0 which is difficult to type properly) even if the index is out of bounds.

1.8.9 The universal typing predicate. We can introduce into PA by a detour through some auxiliary predicates a binary primitive recursive predicate $x : c$ which acts as the universal typing predicate and is such that PA proves

$$x : \ulcorner T \urcorner \circ (t_1, \ldots, t_n, 0) \leftrightarrow x : T(t_1, \ldots, t_n) \tag{1}$$

for every type predicate T with n type parameters introduced into PA. Here $\ulcorner T \urcorner$ is a certain term of PA denoting the code of T obtained by the arithmetization (see below) of the definition of T. The infix constructor function $\circ$ codes the type application of T to its parameters. When T is without parameters then (1) should be read as

$$x : \ulcorner T \urcorner \circ 0 \leftrightarrow x : T \ .$$

The notation $'T(\rho_1, \ldots, \rho_n)$ discussed in Par. 1.8.6 stands for the term $\ulcorner T \urcorner \circ (\rho_1, \ldots, \rho_n, 0)$ and so (1) can be written as:

$$x : 'T(\rho_1, \ldots, \rho_n) \leftrightarrow x : T(t_1, \ldots, t_n) \ .$$

The arithmetization of type predicates calls for the following constructors which are explicitly introduced into PA as follows: $\boldsymbol{N} = 0$; $\boldsymbol{trec} = 1$; $c \circ p = 2, c, p$; $\boldsymbol{vect}(t) = 3, t$; $t_1 \times t_2 = 4, t_1, t_2$; $t_1 \times^* t_2 = 5, t_1, t_2$; $\boldsymbol{Un}(m, s) = 6, m, s$; and $\boldsymbol{par}(n) = 7, n$.

PA proves the following properties of the universal typing predicate:

$$x : t \to t = \boldsymbol{N} \vee \exists s\, t = \boldsymbol{vect}(s) \vee \exists s_1 \exists s_2\, t = s_1 \times s_2 \vee$$
$$\exists s_1 \exists s_2\, t = s_1 \times^* s_2 \vee \exists m \exists s\, t = \boldsymbol{Un}(m, s) \vee \exists c \exists p\, t = c \circ p$$
$$x : \boldsymbol{N}$$
$$x : \boldsymbol{vect}(t) \leftrightarrow \exists n \exists a (x = n + 1, a \wedge L(a) = n + 1 \wedge \forall i (i \leq n \to (a)_i : t))$$
$$x : t_1 \times t_2 \leftrightarrow \exists v \exists w (x = v, w \wedge v : t_1 \wedge w : t_2)$$
$$x : t_1 \times^* t_2 \leftrightarrow x = 0 \vee x : t_1 \times t_2$$
$$x : c \circ p \leftrightarrow x : sb(c, c, p)$$
$$x : \boldsymbol{Un}(m, s) \leftrightarrow x < m \vee \exists i \exists y (i < L(s) \wedge x = m + i, y \wedge y : (s)_i) \ .$$

The constant $\boldsymbol{N}$ denotes the code of the definition of the type predicate N (see Par. 1.8.2), i.e. $\ulcorner N \urcorner \equiv \boldsymbol{N}$, and so

$$'N \equiv \ulcorner N \urcorner \circ 0 \equiv \boldsymbol{N} \circ 0 \ .$$

The term $\boldsymbol{vect}\, \boldsymbol{par}(0)$ denotes the code of the definition of the one-parameter type predicate *Vect* (see Par. 1.8.7), i.e. $\ulcorner Vect \urcorner \equiv \boldsymbol{vect}\, \boldsymbol{par}(0)$, and so

$$'Vect(\rho) \equiv \ulcorner Vect \urcorner \circ (\rho, 0) \equiv \boldsymbol{vect}\, \boldsymbol{par}(0) \circ (\rho, 0) \ .$$

For every t_1 and t_2 the term $t_1 \times t_2$ denotes the code of the Cartesian product of t_1 and t_2. We abbreviate $t_1 \times (t_2 \times t_3)$ to $t_1 \times t_2 \times t_3$. Thus, for instance, the code $\ulcorner N_3 \urcorner$ of the type predicate N_3 (see Par. 1.8.3) is $'N \times 'N \times 'N$, and so

$$'N_3 \equiv \ulcorner N_3 \urcorner \circ 0 \equiv$$
$$(\boldsymbol{N} \circ 0 \times \boldsymbol{N} \circ 0 \times \boldsymbol{N} \circ 0) \circ 0 \ .$$

For every t_1 and t_2 the term $t_1 \times^* t_2$ codes the 'optional' Cartesian product of t_1, t_2 which holds of 0 or of Cartesian products $t_1 \times t_2$.

For instance, the code $\ulcorner List \urcorner$ of the polymorphic list type predicate $List(t)$ (see Par. 1.8.6) is as in the following:

$$'List(\rho) \equiv \ulcorner List \urcorner \circ (\rho, 0) \equiv (\boldsymbol{par}(0) \times^* \boldsymbol{trec}) \circ (\rho, 0) \ .$$

The constructor $\boldsymbol{par}$ codes the type parameters in the codes of type predicates with type parameters such that the i-th parameter is coded by $\boldsymbol{par}(\underline{i-1}_b)$. The constructor $\boldsymbol{trec}$ codes recursive applications of type predicates in their definitions. The purpose of both constructors can be seen from the following:

$$
\begin{aligned}
x : {}'List(t) &\Leftrightarrow x : (\boldsymbol{par}(0)\times^*\boldsymbol{trec})\circ(t,0) \Leftrightarrow \\
&x : sb(\boldsymbol{par}(0)\times^*\boldsymbol{trec}, \boldsymbol{par}(0)\times^*\boldsymbol{trec}, t, 0) \Leftrightarrow \\
&x : t\times^*(\boldsymbol{par}(0)\times^*\boldsymbol{trec})\circ(t,0) \Leftrightarrow \\
&x = 0 \vee x : t\times(\boldsymbol{par}(0)\times^*\boldsymbol{trec})\circ(t,0) \Leftrightarrow \\
&x = 0 \vee \exists v \exists w (x = v, w \wedge v : t \wedge w : (\boldsymbol{par}(0)\times^*\boldsymbol{trec})\circ(t,0) \Leftrightarrow \\
&x = 0 \vee \exists v \exists w (x = v, w \wedge v : t \wedge w : {}'List(t)) \Leftrightarrow \\
&x = 0 \vee x : t\times{}'List(t) \Leftrightarrow x : t\times^*{}'List(t)\ .
\end{aligned}
$$

The three-place *substitution* function $sb(c,d,p)$ yields the code obtained from the code c by substituting in it for every occurrence of parameter $\boldsymbol{par}(i)$ the value $(p)_i$ and for every occurrence of $\boldsymbol{trec}$ the value $d\circ p$. The substitution works as just explained only if for a constructor $c\circ p$ all occurrences of the constructor $\boldsymbol{trec}$ within c are enclosed by at least one occurrence of $\boldsymbol{vect}$, $\times$, $\times^*$, or $\boldsymbol{Un}$. For instance, in the above type derivation for $x : {}'List(t)$ we had $(\boldsymbol{par}(0)\times^*\boldsymbol{trec})\circ(t,0)$. If the condition on $\boldsymbol{trec}$ in c is not satisfied then $sb(c,c,p) = 0$. The condition on $\boldsymbol{trec}$ guarantees the termination of recursive openings of $\circ$. Clausal definitions of recursive type predicates must be regular in the first argument and so the condition on $\boldsymbol{trec}$ is always satisfied for their codes.

The substitution $sb(c,d,p)$ is *shallow* in the sense that it does not enter the codes $e\circ q$ embedded in c. This means that we have no non-local parameters and recursion in the codes and as a consequence the predicate $x : c$ is primitive recursive.

The constructor $\boldsymbol{Un}$ codes union type predicates. The idea is that we have $x : \boldsymbol{Un}(m, \rho_m, \ldots, \rho_{k-1}, 0)$ iff x is a union value determined by m, k, n_m, $\ldots$, n_{k-1} (see Par. 1.5.6). Moreover, for every j s.t. $m \leq j < k$ the number n_j is the 'arity' of the code ρ_j, i.e. the code has a form $\sigma_1\times\ldots\times\sigma_{n_j}$, and we have $x = \underline{j}_b, y_1, \ldots, y_{n_j}$ with $y_1 : \sigma_1, \ldots, y_{n_j} : \sigma_{n_j}$. The reader will note that the last is equivalent to $x = \underline{j}_b, y$ and $y : \rho_j$.

For instance, the code $\ulcorner Bt \urcorner$ of the polymorphic type predicate of labelled binary trees $Bt(t)$ (see Par. 1.8.6) is as in the following:

$$
\begin{aligned}
{}'Bt(\rho) \equiv {} & \ulcorner Bt \urcorner\circ(\rho,0) \equiv \\
& \boldsymbol{Un}(\underline{1}_b, (\boldsymbol{par}(0)\times\boldsymbol{trec}\times\boldsymbol{trec}), 0)\circ(\rho,0)\ .
\end{aligned}
$$

1.8.10 Type terms. We will present below a calculus for deriving assertions about well-typed generalized terms and definitions. For that we need a flexible language whose terms ρ denote codes of type predicates. The terms are called *type* terms and they are used in applications $x : \rho$. Type terms are constructed from variables (say t_1, t_2, $\ldots$) ranging over type codes (i.e. over $\mathbb{N}$) by constructors $\rho_1\times\rho_2$, $\rho_1\times^*\rho_2$,

$$
\boldsymbol{Un}(\underline{m}_b, \rho_m, \ldots, \rho_{k-1}, 0)\ ,
$$

and by type applications of the form $'T(\rho_1, \ldots, \rho_n)$ where T is a type predicate with n-parameters previously introduced into PA. The reader will recall that the last type application abbreviates the term $\ulcorner T \urcorner \circ (\rho_1, \ldots, \rho_n, 0)$.

For every type predicate T with n parameters ($n \geq 0$) introduced into PA except N and *Vect* we take its code $\ulcorner T \urcorner$, express it via constructors as a term of PA, shallowly replace in it every subterm $\boldsymbol{par}(\underline{i}_b)$ by the variable t_{i+1}, every subterm $\boldsymbol{trec}$ by the type term $'T(t_1, \ldots, t_n)$ We obtain thereby a type term $\rho[t_1, \ldots, t_n]$ such that PA proves

$$x : 'T(t_1, \ldots, t_n) \leftrightarrow x : \rho[t_1, \ldots, t_n] \tag{1}$$

For instance, for the type predicates mentioned in Par. 1.8.9 PA proves

$$\begin{aligned} x : N_3 &\leftrightarrow x : N \times N \times N \\ x : \mathit{List}(t) &\leftrightarrow x : t \times^* \mathit{List}(t) \\ x : \mathit{Bt}(t) &\leftrightarrow x : \boldsymbol{Un}(1, (t \times \mathit{Bt}(t) \times \mathit{Bt}(t)), 0) \end{aligned}$$

where we have consistently used the abbreviations for type applications (see Par. 1.8.6).

1.8.11 The typing calculus. We wish to type a mixed definition of the n-ary function symbol f in the polymorphic style of ML (see [7]) as

$$f :: \rho_1, \ldots, \rho_n \mapsto \rho \tag{1}$$

where ρ, ρ_1, ..., ρ_n are type terms. This extensionally means

$$x_1 : \rho_1 \wedge \ldots x_n : \rho_n \rightarrow f(x_1, \ldots, x_n) : \rho \ , \tag{2}$$

but intensionally the typing (1) is a much stronger assertion because it will be derived in a considerably weaker calculus, indeed a decidable one, than a formal proof system of the first-order theory PA.

Consider for instance, the explicit clausal definition:

$f(x) = 3, x$

for which PA trivially proves

$$x : N \rightarrow f(x) : N$$

but we will not be able to derive $f :: N \mapsto N$ in the type calculus.

The derivation of (1) in the weak type calculus will guarantee for all mixed numerals τ_1, ..., τ_n for which the typing calculus derives $\tau_1 : \rho_1$, ..., $\tau_n : \rho_n$ the computation by reduction of applications $f(\tau_1, \ldots, \tau_n)$ without any mixed numeral conversions. A mixed numeral τ such that the typing calculus derives $\tau : \rho$ is called a *canonical* numeral of type ρ. Another advantage of the derivation of (1) in the typing calculus with closed type terms (without

free type variables) is that the definition of f can be efficiently compiled into machine code or into the programming language C using the Pascal-like representation of its data types.

The typing calculus will be deriving either assertions of the form (1) or *type sequents* of the form

$$\Delta \Rightarrow \alpha : \rho \tag{3}$$

where α is a generalized term and ρ a type term. Δ is a sequence of finitely many (possibly empty) assumptions (guards). The assumptions are of the form $x : \rho$ or $f :: \rho_1, \ldots, \rho_n \mapsto \rho$. The order of assumptions in Δ is irrelevant and the reader can treat Δ as a finite set or, alternatively, he can enrich the type calculus by *structural* rules permitting interchange, duplication, and the removal of duplications in the assumption sequence Δ. We say that the typing calculus derives $\alpha : \rho$ if it derives the sequent $\emptyset \Rightarrow \alpha : \rho$.

Derivations in the typing calculus are trees with roots of the form (1) or (3). The derivation trees are constructed in the usual way by means of *rules of inference* which are of the form $\dfrac{\phi_1 \ \ldots \ \phi_n}{\phi}$ where the *conclusion* ϕ is derived from the *premises* $\phi_1, \ldots \phi_n$. We do not exclude $n = 0$ and such rules are *axioms*.

The typing calculus is extensible in the sense that with every extension by definition of PA the typing calculus is extended with new rules of inference involving the newly introduced symbols.

We leave it to the reader to check that whenever the typing calculus derives the sequent $\Delta \Rightarrow \alpha : \rho$ then PA proves $\Delta^\star \to \alpha^* : \rho$ where $\Delta^\star$ stands for the conjunction of translated assumptions. Here an assumption $x : \rho$ is translated into itself and an assumption $f :: \rho_1, \ldots, \rho_n \mapsto \rho$ into

$$\forall x_1 \ldots \forall x_n (x_1 : \rho_1 \wedge \ldots x_n : \rho_n \to f(x_1, \ldots, x_n) : \rho) \ .$$

Similarly, when (1) is derived in the typing calculus then PA proves (2). These two facts prove the soundness of the typing calculus. The completeness of the calculus is not desirable because of its intensional benefits.

In the following paragraphs we present the inference rules of the typing calculus categorized into groups. We use the possibly subscripted metavariables α to range over generalized terms (including PA terms), τ to range over PA terms, and ρ, σ to range over type terms.

1.8.12 Inference rules using assumptions. Only two kinds of inference rules use assumptions (guards) in sequents. The first kind are axioms which type object variables and the second kind are rules with n premises typing applications of n-ary function symbols:

$$\frac{}{\Delta, x : \rho \Rightarrow x : \rho}$$

$$\frac{\Delta, f :: \rho_1, \ldots, \rho_n \mapsto \rho \Rightarrow \tau_1 : \rho_1 \quad \ldots \quad \Delta, f :: \rho_1, \ldots, \rho_n \mapsto \rho \Rightarrow \tau_n : \rho_n}{\Delta, f :: \rho_1, \ldots, \rho_n \mapsto \rho \Rightarrow f(\tau_1, \ldots, \tau_n) : \rho}$$

1.8.13 Axioms typing the constant 0. The constant 0 can be typed in three ways. This kind of multiple typing is often called *ad hoc polymorphism*. The first two kinds are given by the following axioms and the third kind is discussed in Par. 1.8.16:

$$\frac{}{\Delta \Rightarrow 0 : {}'N} \qquad \frac{}{\Delta \Rightarrow 0 : \rho_1 \times^* \rho_2}$$

1.8.14 Axioms typing the binary successor functions. The binary successor functions $\cdot\mathbf{0}$ and $\cdot\mathbf{1}$ have the following typing:

$$\frac{}{\cdot\mathbf{0} :: {}'N \mapsto {}'N} \qquad \frac{}{\cdot\mathbf{1} :: {}'N \mapsto {}'N}$$

The reader will note that this, the typing of 0 as $'N$, and the typing of function applications (see Par. 1.8.12) permits to derive the sequents $\Delta \Rightarrow \tau : {}'N$ where τ are binary numerals. The assumptions about the types of binary successor functions can be eliminated from Δ by the rules discussed in Par. 1.8.21. This enables the derivation of $\tau : {}'N$ for all binary numerals. As it happens, these are the only mixed numerals which can be typed as N. Compare this with a trivial proof of $\tau : {}'N$ in PA for arbitrary terms τ.

1.8.15 Inference rules typing applications of the pairing function. Pairs can be typed in three ways. The first two kinds type pairs as Cartesian products:

$$\frac{\Delta \Rightarrow \tau_1 : \rho_1 \quad \Delta \Rightarrow \tau_2 : \rho_2}{\Delta \Rightarrow \tau_1, \tau_2 : \rho_1 \times \rho_2} \qquad \frac{\Delta \Rightarrow \tau_1, \tau_2 : \rho_1 \times \rho_2}{\Delta \Rightarrow \tau_1, \tau_2 : \rho_1 \times^* \rho_2}$$

and the third kind as union values (see Par. 1.8.16).

1.8.16 Inference rules typing union values. Union values are typed by two kinds of inference rules:

$$\frac{}{\Delta \Rightarrow \underline{j}_b : \boldsymbol{Un}(\underline{m}_b, \rho_m, \ldots, \rho_{k-1}, 0)} \quad \text{where } j < m$$

$$\frac{\Delta \Rightarrow \tau : \rho_j}{\Delta \Rightarrow \underline{j}_b, \tau : \boldsymbol{Un}(\underline{m}_b, \rho_m, \ldots, \rho_{k-1}, 0)} \quad \text{where } m \leq j < k$$

The reader will note that the binary numerals $\underline{j}_b$ can be typed either as $'N$ (see Par. 1.8.14) or as union values. Binary numerals are included as tags also in union value pairs typed by the inference rules of the second kind. Pairs can be thus typed either as (optional) Cartesian products or as union values.

1.8.17 Axioms typing operations on vectors. The three operations on vectors introduced in Par. 1.8.8 have the following polymorphic typing:

$$\overline{New :: {'N}, t{\mapsto}{'Vect(t)}} \qquad \overline{\cdot[\cdot] :: {'Vect(t)}, {'N}{\mapsto}t}$$

$$\overline{\cdot[\cdot := \cdot] :: {'Vect(t)}, {'N}, t{\mapsto}{'Vect(t)}}$$

These are the only inference rules for the typing of vectors. Thus every well-typed function definition with the vector type must be built up from the three operations and/or auxiliary functions applying the operations.

1.8.18 Axioms typing the discriminators of basic case terms. The discriminators of case terms (both basic and derived) yield union-values. For the typing of case terms we wish the union values to be typed by union types. This is achieved for the discriminators of the three basic case terms by the following typing axioms

$$\overline{sgn} :: {'N}{\mapsto}{'Bool} \qquad \overline{sgn} :: {'Bool}{\mapsto}{'Bool}$$
$$b :: {'N}{\mapsto}\boldsymbol{Un}(0, {'N}, {'N}, 0) \qquad c :: t_1{\times}t_2{\mapsto}\boldsymbol{Un}(0, (t_1{\times}t_2), 0)$$

where the union type predicate *Bool* is introduced into PA as follows:

$Bool(0)$
$Bool(1)$.

It should be clear that we have

$$x : {'Bool} \leftrightarrow x : \boldsymbol{Un}(\underline{2}_b, 0)$$

but the reader should not associate the value 0 with truth and 1 with falsehood because the characteristic functions of predicates yield 1 for truth and 0 for falsehood. The type *Bool* should be understood simply as holding of 0 or 1.

The function $\overline{sgn}$ is typed in two ways in a form of ad hoc polymorphism. The first typing is intended to be used with the first form 1.5.13(1) of $\mathcal{D}$-terms which test for zero. The second typing is intended for the second form 1.5.13(2) where the arguments of $\overline{sgn}$ are results of characteristic functions of predicates. The reader will recall that we treat the clausal definitions of characteristic functions of predicates as defining the predicates themselves and we wish the functions to be typed as $\ldots{\mapsto}{'Bool}$.

1.8.19 Inference rules typing case terms. The idea of typing of a case term

$$case_{m,k}(\tau, \alpha_0, \ldots, \alpha_{m-1}, \alpha_m[y_1, \ldots, y_{n_m}], \ldots, \alpha_{k-1}[y_1, \ldots, y_{n_{k-1}}])$$

under assumptions Δ by a type term ρ is to type its discriminator by a union type

$$\boldsymbol{Un}(\underline{m}_b, (\sigma_1^{(m)} \times \ldots \times \sigma_{n_m}^{(m)}), \ldots (\sigma_1^{(k-1)} \times \ldots \times \sigma_{n_{k-1}}^{(k-1)}), 0) \qquad (1)$$

and all of its terms α_i by ρ. Note that the assumption formulas in case terms do not play any role. We can type only such guarded case terms whose guards are implied by the assumptions Δ. However, the implication is not to be proved in PA but rather in the typing calculus with its limited means. The sole purpose of guards is to act as assumptions for the proof of the completeness condition 1.5.6(1). Note that the property is satisfied under assumptions Δ by the typing under the same assumptions of the disciminator term τ by the union type (1). These considerations lead to the following inference rule:

$$\frac{\begin{array}{l}\Delta \Rightarrow \tau : \boldsymbol{Un}(\underline{m}_b, (\sigma_1^{(m)} \times \ldots \times \sigma_{n_m}^{(m)}), \ldots (\sigma_1^{(k-1)} \times \ldots \times \sigma_{n_{k-1}}^{(k-1)}), 0) \\ \Delta \Rightarrow \alpha_0 : \rho \\ \quad \vdots \\ \Delta \Rightarrow \alpha_{m-1} : \rho \\ \Delta, y_1 : \sigma_1^{(m)}, \ldots, y_{n_m} : \sigma_{n_m}^{(m)} \Rightarrow \alpha_m : \rho \\ \quad \vdots \\ \Delta, y_1 : \sigma_1^{(k-1)}, \ldots, y_{n_{k-1}} : \sigma_{n_{k-1}}^{(k-1)} \Rightarrow \alpha_{k-1} : \rho\end{array}}{\Delta \Rightarrow case_{m,k}(\tau, \alpha_0, \ldots, \alpha_{m-1}, \alpha_m[y_1, \ldots, y_{n_m}], \ldots, \alpha_{k-1}[y_1, \ldots, y_{n_{k-1}}]) : \rho}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \Rightarrow 0 : \boldsymbol{Un}(\underline{2}_b, 0)}}{\emptyset \Rightarrow 0 : Bool}}{\emptyset \Rightarrow 0 : Bool} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \Rightarrow 1 : \boldsymbol{Un}(\underline{2}_b, 0)}}{\emptyset \Rightarrow 1 : Bool} \quad \dfrac{\dfrac{}{\emptyset \Rightarrow 0 : Bool \times^* List(Bool)}}{\emptyset \Rightarrow 0 : List(Bool)}}{\emptyset \Rightarrow 1, 0 : Bool \times List(Bool)}}{\emptyset \Rightarrow 1, 0 : Bool \times^* List(Bool)}}{\emptyset \Rightarrow 1, 0 : List(Bool)}}{\emptyset \Rightarrow 0, 1, 0 : Bool \times List(Bool)}}{\emptyset \Rightarrow 0, 1, 0 : Bool \times^* List(Bool)}}{\emptyset \Rightarrow 0, 1, 0 : List(Bool)}$$

Fig. 1.10. Example of a derivation in the typing calculus.

1.8.20 Inference rules introducing defined type predicates. Every type predicate T other than N or *Vect* can be presented by a typed term in the form 1.8.10(1). For such predicates we have the following typing rules:

$$\frac{\Delta \Rightarrow \alpha : \rho[\rho_1, \ldots, \rho_n]}{\Delta \Rightarrow \alpha : {}'T(\rho_1, \ldots, \rho_n)}$$

As an example of such rules we show in Fig. 1.10 the derivation of $0, 1, 0 :$ *List*(*Bool*) where the reader will recall that 1 stands for the binary numeral $\underline{1}_b$.

1.8.21 Inference rules eliminating assumptions on function types. For the typing of function applications (see Par. 1.8.12) we need assumptions of the form 1.8.11(1). The assumptions can be eliminated by the following inference rules:

$$\frac{\Delta, f :: \rho_1, \ldots, \rho_n \mapsto \rho \Rightarrow \alpha : \rho \qquad f :: \sigma_1, \ldots, \sigma_n \mapsto \sigma}{\Delta \Rightarrow \alpha : \rho}$$

where the assumption $f :: \rho_1, \ldots, \rho_n \mapsto \rho$ is an instance of the polymorphic typing $f :: \sigma_1, \ldots, \sigma_n \mapsto \sigma$ in the second premise. The instance is obtained by a possible simultaneous substitution of suitable type terms for some (or all) type variables occurring in the type terms σ, σ_1, ..., σ_n.

1.8.22 Inference rules typing mixed definitions of functions. We can derive the typing of a form 1.8.11(1) for a mixed definition $f(x_1, \ldots, x_n) = \alpha[f; x_1, \ldots, x_n]$ with $n \geq 1$ by typing the body $\alpha[f; x_1, \ldots, x_n]$ under assumptions about the types of its free variables and about the type of f (this is needed for possible recursive applications). This is captured by the following inference rules:

$$\frac{x_1 : \rho_1, \ldots, x_n : \rho_n, f :: \rho_1, \ldots, \rho_n \mapsto \rho \Rightarrow \alpha : \rho}{f :: \rho_1, \ldots, \rho_n \mapsto \rho}$$

If f is a constant, i.e. a nullary function, and it has an explicit mixed definition $f = \alpha$ then we type it by the following inference rules

$$\frac{\emptyset \Rightarrow \alpha : \rho}{f :: \rho} .$$

1.8.23 Type inference. Type inference is the name for the general problem of deriving a type assertion about an untyped or partially typed program. Clausal definitions are untyped. Even when a type definition is well-typed, i.e. typeable, it can happen that it can be typed in an ad hoc polymorphism by more than one unrelated types. For instance

$f(x) = x, 0, 0$

can be typed as $f :: t \mapsto t \times N \times N$, $f :: t_1 \times^* t_2 \to List(t_1 \times^* t_2)$, or $f :: N \mapsto List(N)$. There are many more typings when one considers union types. It is probably the case that any typeable definition can be typed by a finite number of *principal*, i.e. most general typings, such that any typing is an instance of one of the typings.

What is important is that for any sequent $\Delta \Rightarrow \alpha : \rho$ we can decide the formula

$$\exists \vec{t}(\Delta^\star \to \alpha^\star : \rho)$$

by finding the type witnesses for $\vec{t}$. This is done by adaptation of the unification algorithm for polymorphic typing of Milner [16]. We can thus decide whether function definitions are well-typed.

1.8.24 Computation of well-typed mixed terms. All closed generalized terms used in mixed definitions reduce to mixed numerals. We claim that the well-typed ones reduce without taking the reductions marked in Par. 1.7.4. We note first of all, that all reductions preserve the types and so all terms in a chain of reductions are well-typed.

Reductions 1.7.4(1) or (2) are taken when the discriminator of a binary case term is $b(\rho_1, \rho_2)$ with ρ_1 and ρ_2 mixed numerals. Since the case term is well-typed, we have $b :: N \mapsto \boldsymbol{Un}(0, N, N, 0)$, the argument of b must be typed as N. However, all mixed numerals of type N are binary numerals and so the situation cannot happen.

Reductions 1.7.4(3) are taken when reducing the discriminator of a Cartesian case term $c(\rho)$ with ρ a mixed numeral. Since the only typing for c is $c :: t_1 \times t_2 \mapsto \boldsymbol{Un}(0, (t_1 \times t_2), 0)$, we must have $\rho : t_1 \times t_2$ for some t_1 and t_2. Thus $\rho \equiv \rho_1, \rho_2$ and there is no conversion.

Reductions 1.7.4(4) are taken when the mixed term discriminator ρ of a case term denotes a union value. Since the case term is well-typed, ρ must be of an union type 1.8.19(1). But the mixed numerals of this type are of the form either $\underline{j}_b$ or $\underline{j}_b, \vec{\rho}_j$ and so the reduction takes place without conversion.

If a mixed definition of f is typed as $f :: \sigma_1, \ldots, \sigma_n \mapsto \sigma$ then its application $f(\rho_1, \ldots, \rho_n)$ to mixed numerals s.t. $\rho_1 : \sigma_1$, $\ldots$, $\rho_n : \sigma_n$ is typed as ρ and the application reduces to a mixed numeral without conversions.

1.8.25 Compilation of well-typed mixed definitions. If a mixed definition of a function f is non-polymorphically typed, i.e. typed as $f :: \sigma_1, \ldots, \sigma_n \mapsto \sigma$ without variables in the type terms σ, σ_1, $\ldots$, σ_n, then the types are Pascal-like. Since all applications of auxiliary functions needed by the definition of f must go through inference rules in Par. 1.8.21, also the types of the auxiliary functions are Pascal-like. There is no conversion in the computation of any well-typed application of f and so the definition of f and of its auxiliary functions can be compiled into machine code (or C) with the standard Pascal-like representation of all types used. True, the auxiliary functions can be polymorphically typed and applied with different instantiations

but there is only a finite number of such. Thus the auxiliary functions can be compiled in multiple copies with different Pascal-like types. An objection that this is memory consuming is not a serious one because the machine (or C) code is compact, memories of modern computers are large, and the additional storage is nothing compared with the storage needed by the graphic, music, or video files.

1.9 Intensional Functionals

It is often said that LISP is a programming language based on the lambda calculus of functionals, i.e. functions operating on functions. It seems to us quite surprising that this characterization was not seriously questioned before because the domain of LISP are S-expressions. and so there can be no *extensional* functionals in LISP. Functionals of LISP are only coded into S-expressions in the form of *intensional* functionals. This is done by a universal partial function *Apply*. Since S-expressions can be embedded into natural numbers by the pairing function, it suffices to analyze the situation in the domain $\mathbb{N}$.

We would like to note that although intensional functionals have been studied in logic, for instance by Tait [23], the author does not know of an analysis done on such a large scale as ours. By that we mean that we analyze a real typing system, which is moreover a polymorphic one, and the definitions of functionals expressed in a real programming language such as the extensible language of clausal definitions. In contrast, the analysis by Tait was of a simple type system whose types have been generated from N by $\mapsto$, and of a simple language of primitive recursive functionals (Gödel's T). Moreover, our proposal for the introduction of functionals into a programming language leads to a very useful extension of mixed definitions. The proposed language is very simple to understand and use.

1.9.1 Universal function. To every Σ_1-definition of a unary partial function f we can assign its code c. The reader will note that such an f does not have a Π_1-definition unless it is total, i.e. recursive. We can then find a Σ_1-definition (but not a Π_1) of a binary partial *universal* function $U(c, x)$. The universal function is such that $U(c, x) \simeq f(x)$ holds for all x. This should be read as that either both $U(c, x)$ and $f(x)$ are defined and the results are identical, or neither is defined.

The assignment of codes can be extended to n-ary partial Σ_1-definable functions f to which we assign the same codes c as to their unary contractions $\langle f \rangle$. We then have

$$f(x_1, \ldots, x_n) \simeq \langle f \rangle (x_1, \ldots, x_n) \simeq U(c, x_1, \ldots, x_n) \ .$$

The universal function U has a code because its contraction $\langle U \rangle$ is Σ_1-definable. Applications $U(c, x)$ are in the Computability theory usually writ-

ten as $\{c\}(x)$. Functional programming languages would use the notation $c(x)$ or $c\,x$. We will write $c{\bullet}x$ in order to make visible that the application operator is a binary function over $\mathbb{N}$. The operator associates to the left and we let $c{\bullet}x{\bullet}y$ to stand for $(c{\bullet}x){\bullet}y$.

We work in PA with total functions only and so one possibility would to introduce the universal function $\bullet$ into PA as the Δ_2-definable completion of the partial function. Any definition of a partial function f with an argument c passed in the body of f to the first argument of $\bullet$ can be viewed as an intensional functional where the argument c is the code of a function rather than the function directly. Intensional functionals would thus be Δ_2-definable as completed partial functions.

We are interested in a characterization of **decidable** (actually primitive recursive) constraints on arguments under which the intensional functionals can be computed. Decidable constraints will allow us to use instead of the Δ_2-definable $\bullet$ a Δ_1-definition of its restriction to properly typed arguments.

One of the possibilities for the constraints on arguments is to use a syntactic scheme of stratification of arguments in order to prevent the self-application $c{\bullet}c$ of which we cannot decide in general whether its computation terminates or not.

1.9.2 Hereditarily recursive operations. The simplest stratification of arguments of functionals is by *function types* $s{\mapsto}t$ which are called in logic *finite* types. The ground type is N and when the types s and t have been defined then we can define the type $s{\mapsto}t$ by explicit definition:

$$c : s{\mapsto}t \leftrightarrow \forall x(x : s \rightarrow \exists y(c{\bullet}x \simeq y \wedge y : t)) \ .$$

The function type $s{\mapsto}t$ thus holds of all intensional functionals c which are defined and yield functionals of type t if applied to functionals of type s. Such functionals are known as *hereditarily recursive operations* [5, 25]. This is a perfectly feasible approach except that it does not integrate well-with our data types. The reason is that the function types are of increasing quantifier complexity. While the predicate $c : N{\mapsto}N$ can be defined by a Π_2-definition, the definition of $c : (N{\mapsto}N){\mapsto}N{\mapsto}N$ requires a Π_3-formula and the definition of

$$c : (\mathbb{N}{\mapsto}N){\mapsto}N{\mapsto}N){\mapsto}(N{\mapsto}N){\mapsto}N{\mapsto}N$$

a Π_4-formula etc. The increasing quantifier complexity of function types means that we cannot introduce into PA the predicate $x : t$ with t admitting besides data types also the function types.

In order to stay within PA we substantially weaken the function types where we intuitively let $c : s{\mapsto}t$ to hold iff c codes a mixed definition of a unary function f s.t. $f :: s{\mapsto}t$ is derivable in the typing calculus (see Par. 1.9.11).

1.9.3 Bland intensional functionals. We will investigate first a rather limited class of intensional functionals which we call *bland* as opposed to a substantially larger class of *curried* functionals which will be introduced in Par. 1.9.18.

The idea of bland functionals is simple enough. Suppose for a moment that we have managed to introduce into PA the application operator $\bullet$ and that we have suitably extended the definition of the universal typing predicate $x : t$ (see Par. 1.8.9) to function types $t = s_1 \mapsto s_2$ by treating the symbol $\mapsto$ as a new constructor function: $s \mapsto t = 8, s, t$. PA will prove

$$x : s \mapsto t \wedge y : s \to x \bullet y : t \ . \tag{1}$$

In order to be able to derive this also in the typing calculus we extend it with the axiom:

$$\frac{}{\cdot \bullet \cdot :: (s \mapsto t), s \mapsto t} \ . \tag{2}$$

We also extend the **narrow** mixed definitions by allowing in them applications of the operator $\bullet$ and call them *bland* definitions. Suppose now that we have introduced into PA by a bland definition an n-ary function symbol f. Suppose further that the definition is well-typed and that we have derived in the (extended) typing calculus $f :: s_1, \ldots, s_n \mapsto t$. The function f will have assigned a code $'f$ such that PA will prove:

$$'f : s_1 \times \ldots \times s_n \mapsto t \tag{3}$$

$$x_1 : s_1 \wedge \ldots \wedge x_n : s_n \to {'f} \bullet (x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \ . \tag{4}$$

Whenever, the *index property* (4) holds we say that the function f has an *index* $'f$. We will see below that in the case of bland functionals the sufficient condition for f to have an index is that we have a well-typed bland definition of f.

The typing property (3) is derivable in PA but not in the typing calculus. For that we add to the typing calculus new axioms:

$$\frac{}{\Delta, f :: s_1, \ldots, s_n \mapsto t \Rightarrow {'f} : s_1 \times \ldots \times s_n \mapsto t} \ . \tag{5}$$

1.9.4 Computation of well-typed bland terms. We compute bland functionals by adding to the mixed numerals as new irreducible terms also the codes $'f$ of bland functionals f with indices and close them under pairing and binary successors. We call this extended class the *bland* numerals. Note that unlike the mixed numerals, the class of bland numerals is defined relatively to the current extension of PA. The new reductions are

$${'f} \bullet (\vec{\rho}) \blacktriangleright f(\vec{\rho}) \tag{1}$$

where $\vec{\rho}$ are bland numerals and $f(\vec{\rho})$ is well-typed.

It can be shown that the reductions of all well-typed terms terminate with bland terminals with no conversions whatsoever. Note that while terms typed by function types are obtained from the indices of bland functionals by the operator $\bullet$, the only irreducible numerals of function types are the codes $'f$ of functions with bland definitions.

1.9.5 Example: The bland functional *Map*. The canonical example of use of functionals is in the functional programming the binary functional *Map* which is defined by

$$Map(f,0) = 0$$
$$Map(f,v,w) = f\bullet v, Map(f,w) \ .$$

The function is polymorphically typeable as

$$f :: s \mapsto t, List(s) \mapsto List(t)$$

and so it has an index $'Map$ such that $'Map : (s \mapsto t) \times List(s) \mapsto List(t)$ and PA proves:

$$f : s \mapsto t \wedge x : List(s) \rightarrow {}'Map\bullet(f,x) = Map(f,x) \ . \tag{1}$$

1.9.6 Comment on the introduction of bland functionals. The reader will probably agree that the idea and use of (bland) intensional functionals is simple. The description of their computation is straightforward because it takes just one additional easy to understand kind of reductions 1.9.4(1). This should be contrasted with the following outline of the hard work needed to introduce the function types and the application operator into PA and to prove the indexing properties 1.9.3(4). The hard work involves the arithmetization of both the typing calculus and of the computation process. Both are needed for the definition of the application operator $\bullet$ and for the proofs of the indexing properties. We stress here that the difficult introduction of $\bullet$ and its inefficient definition has to do only with the semantics of proving that the reduction 1.9.4(1) preserves meaning, i.e. that PA proves $'f\bullet(\vec{\rho}) = f(\vec{\rho})$. The simplicity and the efficiency of computation with bland functionals is not affected by the inefficient definition of $\bullet$.

1.9.7 Arithmetization of generalized terms. We now arithmetize the generalized terms used in bland definitions. For that we use the following constructors: $\boldsymbol{z} = 0$; $\boldsymbol{S}_0(a) = 1, a$; $\boldsymbol{S}_1(a) = 2, a$; $\boldsymbol{P}(a,b) = 3, a, b$; $a\underline{\bullet}b = 4, a, b$; $\overline{\boldsymbol{sgn}}(a) = 5, a$; $\boldsymbol{b}(a) = 6, a$; $\boldsymbol{c}(a) = 7, a$; $\boldsymbol{cs}(d,m,t) = 8, d, m, t$; $\boldsymbol{rec}(a) = 9, a$; $\boldsymbol{x}_i = 10, i$; and $\boldsymbol{cd}(a,n,m,e) = 11, a, n, m, e$. The first eight constructors code in that order applications of the nullary function 0, binary successors $x\mathbf{0}$, $x\mathbf{1}$, pairing, the application operator $\bullet$, and of the functions $\overline{sgn}$, b, and c used in the basic case terms.

Case terms

$$case_{m,k}(\tau, \alpha_0, \ldots, \alpha_{m-1}, \alpha_m[y_1, \ldots, y_{n_m}], \ldots, \alpha_{k-1}[y_1, \ldots, y_{n_{k-1}}])$$

are coded as

$$\boldsymbol{cs}(\ulcorner\tau\urcorner, m, (0, \ulcorner\alpha_0\urcorner), \ldots, (0, \ulcorner\alpha_{m-1}\urcorner), (n_m, \ulcorner\alpha_m\urcorner), \ldots, (n_{k-1}, \ulcorner\alpha_{k-1}\urcorner), 0) \ .$$

The n-tuples of arguments $\tau_1, \tau_2, \ldots, \tau_{n-1}, \tau_n$ applied to function symbols are coded by pairing

$$\boldsymbol{P}(\ulcorner\tau_1\urcorner, \boldsymbol{P}(\ulcorner\tau_2\urcorner, \ldots, \boldsymbol{P}(\ulcorner\tau_{n-1}\urcorner, \ulcorner\tau_n\urcorner)\ldots))$$

when $n \geq 2$, and as $\ulcorner\tau_1\urcorner$ when $n = 1$.

Applications $f(\vec{\tau})$ of n-ary function symbols defined by bland definitions are coded as $'f\underline{\bullet}(\ulcorner\vec{\tau}\urcorner)$ or as $\boldsymbol{rec}(\ulcorner\vec{\tau}\urcorner)$. The last when f is applied recursively within its definition. Codes $'f$ of defined functions are explained in Par. 1.9.8.

The coding of variables by $\boldsymbol{x}_i$ requires some explanation because $'f$ codes a definition of the unary contraction function $\langle f \rangle$. Moreover, the coding of local variables in case terms has to be taken into account. With unary functions a single variable and the projection functions H, T suffice. We can extract by projections both arguments and local variables from a single *environment* $(((x, y_1), \ldots), y_{p-1}), y_p$ where x collects the arguments, y_p collects the local variables of the closest enclosing case term, y_{p-1} collects the local variables of the next enclosing case term and so on. The reader will note that such a scheme makes the coding of arguments and local variables dependent on their position within a generalized term. Actually, we can dispense with the projection functions because we can extract the values from an environment x by the *dyadic indexing* function $x[i]$ with the clausal definition:

$$\begin{aligned} x[0] &= x \\ (v, w)[i\mathbf{1}] &= v[i] \\ (v, w)[i\mathbf{2}] &= w[i] \ . \end{aligned}$$

For instance, if the environment is $x = ((a, b, c), d, e), f$ then we have $x[19] = b$ and $x[13] = e$. We then code by $\boldsymbol{x}_i$ the (value of the) variable extracted from the environment x by the dyadic index i.

1.9.8 Arithmetization of bland definitions. We will now assign codes to the bland definitions $f(x_1, \ldots, x_n) = \alpha[f; x_1, \ldots, x_n]$ where $n > 0$. For that we use the four-place constructor $\boldsymbol{cd}$. If the generalized definition is explicit then we code it simply as $'f \equiv \boldsymbol{cd}(\ulcorner\alpha\urcorner, \underline{n_b}, 0, 0)$.

If the definition is recursive we have to work harder. The reader will recall that our goal is to extend the primitive recursive universal typing predicate with function types. If the above definition is typed as $f :: s_1, \ldots, s_n \mapsto t$ we wish $'f : s_1 \times \ldots \times s_n \mapsto t$ to hold. If the predicate is to remain primitive recursive we need a syntactic check of regularity of definitions. Regularity conditions are semantic ones but we enforce them by proofs within PA. The

check that the proof p proves the regularity condition for f is primitive recursive and so we could include p, say as $\boldsymbol{cd}(\ulcorner\alpha\urcorner, \underline{n}_b, \ulcorner p\urcorner, 0)$, in the code of f. This, of course, would call for the arithmetization of proofs in PA which by itself would not be that complicated, but in order to prove properties of codes we would need to derive in PA many properties of the arithmetized proof predicate. This amounts to the work needed in the formal proof of the second Incompleteness theorem of Gödel which is hardly ever done in detail.

Fortunately, we can fall back on the use of syntactic guards from course of values definitions with measure. For the above f we have a measure term μ for which the typing calculus derives

$$x_1 : s_1, \ldots, x_n : s_n \Rightarrow \mu : N \ .$$

The reader will note that this is the reason why we require that bland functionals be defined by mixed definitions in the narrow sense.

One problem still remains when the guard on a recursive application of f within α fails. We have to yield a default value, say 0. This is perfectly in order with untyped definitions but with the typed ones we wish that the typing property 1.9.3(1) of the application operator $\bullet$ holds. For that we have to yield a default value of type t. But what if t is a polymorphic type we know nothing about? For instance, $f(x) = f(x)$ is a perfectly legal course of values definition with the measure x. The defined function satisfies $f(x) = 0$ because the guard always fails. The reader can easily convince himself that our typing calculus derives among others also the polymorphic typing $f :: \mathit{List}(t) \mapsto t$. Even if we had the argument x of f at our disposal it could be the empty list 0 and we would not be able to produce a value of type t.

Fortunately, we can include into the code of f the code of an auxiliary *default* term α_1 for which the typing calculus derives:

$$x_1 : s_1, \ldots, x_n : s_n \Rightarrow \alpha_1 : t \ .$$

The sole purpose of α_1 is to yield default values when the recursion guard fails. The reader will note that with the nonsensical definition $f(x) = f(x)$ we would have problems to come up with a default term typed as $x : \mathit{List}(t) \Rightarrow \alpha_1 : t$. It can be shown that to every well-typed regular bland definition of a recursive function f there is a default term. This is because the satisfaction of regularity conditions requires that there is at least one branch in the generalized term α without recursive applications.

These considerations lead us to the assignment of code

$$'f \equiv \boldsymbol{cd}(\ulcorner\alpha\urcorner, \underline{n}_b, \ulcorner\mu\urcorner, \ulcorner\alpha_1\urcorner)$$

to the above recursive definition of f where μ is the measure term and α_1 is the default term. We have included the arity n of f in its code so we can assign the same code both to bland and curried functionals. The arity is not needed in the definition of the bland application operator $\bullet$.

If the reader is worried about such complicated coding we hasten to say that the codes of bland functionals are not needed in order to compute them. During the computation we rely just on the reduction 1.9.3(1) where the code $'f$ might be just the address of the (pseudo) machine code for f. The coding would be important only in an extremely unlikely case when one wished to prove properties of codes.

1.9.9 Arithmetization of the typing calculus. Our next task is the arithmetization of the typing calculus from Sect. 1.8. To that end we introduce into PA a unary primitive recursive predicate $\boldsymbol{Dt}(p)$ holding of codes p of derivation trees in the calculus. A node in p, which corresponds to one application of rule of inference, is coded with the help of a five-place constructor function explicitly introduced into PA as:

$$\left(\frac{p}{a[r;s]\,\mathbf{:}\,t}\right) = (a,r,s,t),p\ .$$

Here the bottom line of the function symbol represents the sequent in the conclusion of an inference rule where a is the code of a generalized term typed by the type t, r is the type coding the typing assumptions on the recursive applications in a and s is the type coding the assumptions on the variables free in a. The premises of the rule are coded by the list p of codes of derivation trees; $p = 0$ means that the coded inference rule has no premises, i.e. that it is an axiom.

Because the predicate $\boldsymbol{Dt}(p)$ has many clauses corresponding to the inference rules of the type calculus we illustrate its definition only with a few clauses. The measure of recursion in $\boldsymbol{Dt}$ is the argument p which means that the definition is by course of values recursion.

Here are a couple of clauses for $\boldsymbol{Dt}$ coding axiom rules:

$$\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{z}[r;s]\,\mathbf{:}\,\boldsymbol{N}}\right)$$
$$\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{z}[r;s]\,\mathbf{:}\,t_1\times^* t_2}\right)\ .$$

Some clauses coding inference rules with one or two premises:

$$\boldsymbol{Dt}\left(\frac{p_1,0}{\boldsymbol{S}_1(a)[r;s]\,\mathbf{:}\,\boldsymbol{N}}\right) \leftarrow \boldsymbol{Dt}(p_1)\wedge p_1 = \left(\frac{q_1}{a[r;s]\,\mathbf{:}\,\boldsymbol{N}}\right)$$
$$\boldsymbol{Dt}\left(\frac{p_1,p_2,0}{\boldsymbol{P}(a,b)[r;s]\,\mathbf{:}\,t_1\times t_2}\right) \leftarrow \boldsymbol{Dt}(p_1)\wedge p_1 = \left(\frac{q_1}{a[r;s]\,\mathbf{:}\,t_1}\right)\wedge$$
$$\boldsymbol{Dt}(p_2)\wedge p_2 = \left(\frac{q_2}{b[r;s]\,\mathbf{:}\,t_2}\right)$$
$$\boldsymbol{Dt}\left(\frac{p_1,0}{\boldsymbol{P}(a,b)[r;s]\,\mathbf{:}\,t_1\times^* t_2}\right) \leftarrow \boldsymbol{Dt}(p_1)\wedge p_1 = \left(\frac{q_1}{\boldsymbol{P}(a,b)[r;s]\,\mathbf{:}\,t_1\times t_2}\right)\ .$$

Clauses for $\boldsymbol{Dt}$ making use of assumptions in the antecedents of sequents are:

$$\boldsymbol{Dt}\left(\frac{p_1}{\boldsymbol{rec}(a)[r_1{\mapsto}r_2;s]\,\boldsymbol{:}\,t}\right) \leftarrow \boldsymbol{Dt}(p_1)\wedge p_1=\left(\frac{q_1}{a[r_1{\mapsto}r_2;s]\,\boldsymbol{:}\,r_1}\right)\wedge t=r_2$$
$$\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{x}_i[r;s]\,\boldsymbol{:}\,t}\right) \leftarrow t=s[\![i]\!]$$

where the indexing function $s[\![i]\!]$, which selects the type of the variable $\boldsymbol{x_i}$ from the assumption type s, is introduced into PA by course of values recursion:

$t[\![0]\!]=t$
$(t_1{\times}t_2)[\![i\boldsymbol{1}]\!]=t_1[\![i]\!]$
$(t_1{\times}t_2)[\![i\boldsymbol{2}]\!]=t_2[\![i]\!]$.

Clauses coding the inference rules concerned with the application operator are:

$$\boldsymbol{Dt}\left(\frac{p_1,p_2,0}{(a\underline{\bullet}b)[r;s]\,\boldsymbol{:}\,t_2}\right) \leftarrow \boldsymbol{Dt}(p_1)\wedge p_1=\left(\frac{q_1}{a[r;s]\,\boldsymbol{:}\,t_1{\mapsto}t_2}\right)\wedge \boldsymbol{Dt}(p_2)\wedge p_2=\left(\frac{q_2}{b[r;s]\,\boldsymbol{:}\,t_1}\right)$$
$$\boldsymbol{Dt}\left(\frac{p_1,p_2,p_3,0}{\boldsymbol{cd}(a,n,m,e)[r;s]\,\boldsymbol{:}\,t_1{\mapsto}t_2}\right) \leftarrow n>0\wedge \mathit{Arity}(t_1)\geq n\wedge \boldsymbol{Dt}(p_1)\wedge p_1=\left(\frac{q_1}{a[t_1{\mapsto}t_2;t_1]\,\boldsymbol{:}\,t_2}\right)\wedge \boldsymbol{Dt}(p_2)\wedge p_2=\left(\frac{q_2}{m[0;t_1]\,\boldsymbol{:}\,\boldsymbol{N}}\right)\wedge \boldsymbol{Dt}(p_3)\wedge p_3=\left(\frac{q_3}{e[0;t_1]\,\boldsymbol{:}\,t_2}\right) .$$

Here the function $\mathit{Arity}(t)$, which counts the Cartesian types in the type t, is introduced into PA by:

$\mathit{Arity}(t_1{\times}t_2)=\mathit{Arity}(t_2)+1$
$\mathit{Arity}(t)=1\leftarrow\neg\exists t_1\exists t_2\, t=t_1{\times}t_2$.

One clause for $\boldsymbol{Dt}$ opens the type definitions:

$$\boldsymbol{Dt}\left(\frac{p_1}{a[r;s]\,\boldsymbol{:}\,c{\circ}u}\right)\leftarrow \boldsymbol{Dt}(p_1)\wedge p_1=\left(\frac{q_1}{a[r;s]\,\boldsymbol{:}\,sb\,(c,c,u)}\right)$$

1.9.10 Arithmetized bland typing predicate. The four-place predicate $a[r;s]\,\underline{\boldsymbol{:}}\,t$ arithmetizing the relation of derivability in the typing calculus and holding whenever the sequent $a[r;s]\,\boldsymbol{:}\,t$ is derivable is introduced into PA by the following explicit definition:

$$a[r;s]\,\underline{\boldsymbol{:}}\,t\leftrightarrow\exists p\boldsymbol{Dt}\left(\frac{p}{a[r;s]\,\boldsymbol{:}\,t}\right) .$$

In view of the discussion in Par. 1.8.23 it should not be too surprising that PA proves that there is a primitive recursive bound on the existential quantifier in above definition and so the predicate is primitive recursive.

Nevertheless, the predicate $a[r;s]\underline{:}t$ is quite complex because it is the arithmetization of the typing calculus which has inference rules relative to the current extension of PA. The above predicate must take care of all possible extensions with bland definitions of intensional functionals and with definitions of type predicates. This happens through encoding of function and type definitions.

1.9.11 Bland function types. We are now ready to add function types to the universal typing predicate. Toward that end we introduce the constructor function $s{\mapsto}t = 8, s, t$ and replace the definition of the universal typing predicate $x : t$ (see Par. 1.8.9) with its proper extension where the following holds

$$x : s{\mapsto}t \leftrightarrow \exists a \exists n \exists m \exists e (x = \boldsymbol{cd}(a,n,m,e) \wedge x[0;0]\underline{:}s{\mapsto}t) \ . \tag{1}$$

The reader will note that if $x : s{\mapsto}t$ holds then we have $s = s_1 \times \ldots \times s_n$ and $\boldsymbol{cd}(a,n,m,e)$. Moreover, by inspecting the codes a, s_1, ..., s_n, and t we should be able to extend PA with suitable definitions of types and bland functionals in such a way that x codes a bland definition of an n-ary function f and the typing calculus derives $f :: s_1, \ldots, s_n{\mapsto}t$.

1.9.12 Type levels. An important measure of complexity of a recursive typed definition is its *type level* which counts the complexity of its function types. We introduce into PA the unary function $lev(t)$ yielding the type level of the (code of the) type t to satisfy the following:

$$\begin{gathered} lev(\boldsymbol{N}) = 0 \\ lev\, \boldsymbol{vect}(t) = lev(t) \\ lev(s \times t) = lev(s \times^* t) = \max(lev(s), lev(t)) \\ lev(c \circ p) = lev\ sb(c, \boldsymbol{N}, p) \\ lev(\boldsymbol{Un}(m,s) = \max_{t \varepsilon s} lev(t) \\ lev(s{\mapsto}t) = \max(lev(s)+1, lev(t)) \ . \end{gathered}$$

Type level of a value x s.t. $x : t$ is $lev(x)$. Note that the type level of a function definition with index $'f : s{\mapsto}t$ is thus $lev(s{\mapsto}t) > 0$. Values with type level 0 are data structures typed by types without any function types. We for instance, have $lev\ List\ Vect(N \times N) = 0$ and $lev\ List\ Vect(N \times (s{\mapsto}t)) = lev(s{\mapsto}t) > 0$. Functions over data structures with type level 0 are of type level 1 and functions taking such functions as arguments are at least of type level 2.

1.9.13 Arithmetization of computation of bland terms. We will now arithmetize the computation of generalized bland terms. We will capture the computations by a computation tree rather than by a reduction sequence. We

$$
\begin{aligned}
&\boldsymbol{Ct}\left(\frac{0}{\boldsymbol{z}[c;x]=0}\right)\\
&\boldsymbol{Ct}\left(\frac{0}{\boldsymbol{x}_i[c;x]=x[i]}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\boldsymbol{S}_0(a)[c;x]=y\boldsymbol{0}}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\boldsymbol{S}_1(a)[c;x]=y\boldsymbol{1}}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,p_2,0}{\boldsymbol{P}(a,b)[c;x]=y_1,y_2}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{b[c;x]=y_2}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\overline{\boldsymbol{sgn}}(a)[c;x]=0}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y+1}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\overline{\boldsymbol{sgn}}(a)[c;x]=1}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=0}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\boldsymbol{b}(a)[c;x]=0,y}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y\boldsymbol{0}}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\boldsymbol{b}(a)[c;x]=1,y}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y\boldsymbol{1}}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,0}{\boldsymbol{c}(a)[c;x]=0,y_1,y_2}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1,y_2}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,p_2,0}{\boldsymbol{cs}(a,m,t)[c;x]=y_2}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge y_1<m\wedge\\
&\qquad\qquad (t)_{y_1}=n,b\wedge \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{b[c;x]=y_2}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,p_2,0}{\boldsymbol{cs}(a,m,t)[c;x]=y_3}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge y_1\geq m\wedge\\
&\qquad\qquad y_1=j,y_2\wedge j\geq m\wedge j<L(t)\wedge(t)_j=n,b\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{b[c;(x,y_2)]=y_3}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,p_2,p_3,p_4,0}{\boldsymbol{rec}(a)[c;x]=y_4}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge c=\boldsymbol{cd}(b,n,m,e)\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{m[0;y_1]=y_2}\right)\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_3)\wedge p_3=\left(\frac{q_3}{m[0;x]=y_3}\right)\wedge y_2<y_3\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_4)\wedge p_4=\left(\frac{q_4}{b[c;y_1]=y_4}\right)\\
&\boldsymbol{Ct}\left(\frac{p_1,p_2,p_3,p_4,0}{\boldsymbol{rec}(a)[c;x]=y_4}\right) \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge c=\boldsymbol{cd}(b,n,m,e)\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{m[0;y_1]=y_2}\right)\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_3)\wedge p_3=\left(\frac{q_3}{m[0;x]=y_3}\right)\wedge y_2\geq y_3\wedge\\
&\qquad\qquad \boldsymbol{Ct}(p_4)\wedge p_4=\left(\frac{q_4}{e[0;x]=y_4}\right)\\
&\boldsymbol{Ct}\left(\frac{0}{\boldsymbol{cd}(a,n,m,e)[c;x]=\boldsymbol{cd}(a,n,m,e)}\right).
\end{aligned}
$$

Fig. 1.11. Part I of predicate holding of computation trees for functionals.

$$\boldsymbol{Ct}\left(\frac{p_1,p_2,p_3,0}{(a\underline{\bullet}b)[c;x]=y_3}\right) \quad \leftarrow \boldsymbol{Ct}(p_1)\wedge p_1=\left(\frac{q_1}{a[c;x]=y_1}\right)\wedge y_1=\boldsymbol{cd}(d,n,m,e)\wedge$$
$$\boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{b[c;x]=y_2}\right)\wedge$$
$$\boldsymbol{Ct}(p_3)\wedge p_3=\left(\frac{q_3}{d[y_1;y_2]=y_3}\right)\ .$$

Fig. 1.12. Part II of predicate holding of computation trees for bland functionals.

do this similarly to the arithmetization of the typing calculus by introducing into PA a five-place constructor function:

$$\left(\frac{p}{a[c;x]=y}\right)=(a,c,x,y),p\ .$$

Here the bottom line of the function symbol represents one step in the computation where the (code of) term a is evaluated into y in the environment where c codes recursive applications and x holds the values of free variables in a. The list p codes the subcomputations.

Figure 1.11 contains a part of clauses of a definition by course of values recursion of the primitive recursive predicate $\boldsymbol{Ct}$. $\boldsymbol{Ct}(p)$ holds if p encodes the computation tree for a bland term. The clauses in the figure are common to trees encoding computation of both bland and curried functionals. Figure 1.12 contains a clause for $\boldsymbol{Ct}$ pertaining to bland functionals only.

The reader will note the change of the environment in the second clause evaluating the codes of case terms in Figure 1.11. The code a of the discriminator term has been evaluated to $y_1=j,y_2$ where y_2 are the values of local variables for the j+1-st (code of) term b in the list of codes t. The parameters of the case are coded by x and the code b is evaluated in the extended environment x,y_2.

The two clauses for $\boldsymbol{rec}(a)$ in the same figure show how the code a of the recursive argument is evaluated to y_1, its measure to y_2, and the measure of arguments x to y_3. If $y_2<y_3$ then the recursion guard is satisfied and recursion takes place where the code of the recursive definition b is evaluated in the environment y_1. If $y_2\geq y_3$ then the guard fails and the default code e is evaluated instead.

Figure 1.12 contains a single clause evaluating the code $a\underline{\bullet}b$. If this is well-typed then the code a must evaluate to $y_1=\boldsymbol{cd}(d,n,m,e)$ coding a bland definition. Its arguments b are evaluated to y_2 and the evaluation of the body d of the definition takes place with the recursion environment y_1 and argument environment y_2.

1.9.14 Characterization of computation of bland terms. The Main lemma on bland functionals asserts that well-typed codes can be evaluated, i.e. PA proves

$$(r = 0 \vee r = r_1 \mapsto r_2 \wedge c : r) \wedge (s = 0 \vee x : s) \wedge a[r; s] \underline{:} t$$
$$\rightarrow \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{a[c; x] = y}\right) . \qquad (1)$$

The cases $r = 0$ and/or $s = 0$ take care of codes a without recursive invocations and/or without free occurrences of variables. By inspection of the definition of the predicate $\boldsymbol{Ct}$ one can see that the values y and p are uniquely determined.

Bland definitions of functions such that neither they nor their measures apply the application operator $\bullet$ are the mixed definitions in the narrow sense and they define exactly the primitive recursive functions. Moreover, all primitive recursive functions can be defined by well-typed bland definitions. Every unary primitive recursive function has a bland definition such that for its code c we have $c : \boldsymbol{N} \mapsto \boldsymbol{N}$. Thus $(c \underline{\bullet} \boldsymbol{x}_0)[0; x] \underline{:} \boldsymbol{N}$ and so we obtain $\boldsymbol{Ct}\left(\frac{p}{(c \underline{\bullet} \boldsymbol{x}_0)[0; x] = y}\right)$ for some y and p from the main lemma. This means that the lemma guarantees the evaluation of every unary primitive recursive function and so the lemma cannot be proved in the fragment $I\Sigma_1$. It is mildly surprising that the lemma can be proved already in the fragment $I\Sigma_2$. Moreover, when $lev(s) = 0$ then already $I\Sigma_1$ proves for every n the following special case of the main lemma:

$$x : s \wedge lev(s) = 0 \wedge \underline{n}_m[0; x] \underline{:} t \rightarrow \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{\underline{n}_m[0; x] = y}\right) .$$

1.9.15 Definition of the bland application operator $\bullet$. In preparation for the introduction into PA of the application operator for bland functionals we explicitly define a binary predicate

$$Wtappl(c, x) \leftrightarrow \exists s \exists t (c : s \mapsto t \wedge x : s) .$$

The unbounded existential quantifiers have primitive recursive bounds and so the predicate is a primitive recursive one.

We now introduce the application operator $\bullet$ by the following Σ_1 contextual definition:

$$c \bullet x = y \leftrightarrow Wtappl(c, x) \wedge \exists p \boldsymbol{Ct}\left(\frac{p}{(c \underline{\bullet} \boldsymbol{x_0})[0; x] = y}\right) \vee \neg Wtappl(c, x) \wedge y = 0$$

whose existence condition follows from the Main lemma 1.9.14(1) and the uniqueness condition holds because the values y and p are uniquely determined. Although the definition of the operator is Σ_1, it is not a primitive recursive function. This is because the Main lemma is provable only in $I\Sigma_2$.

1.9.16 Characterization of bland functionals. Functions definable by bland definitions (both typed and untyped) are clearly primitive recursive in the bland application operator $\bullet$ which is provably recursive in $I\Sigma_2$.

The fragment $I\Sigma_2$, but not $I\Sigma_1$, proves the typing property 1.9.3(1) of the application operator $\bullet$. Similarly, for every well-typed definition of $f :: s_1, \ldots, s_n \to t$, $I\Sigma_2$ proves that $'f$ is the index of f by proving the properties 1.9.3(3) and 1.9.3(4). If the type levels of types $s_1, \ldots, s_n$ are 0 then both properties are provable even in $I\Sigma_1$ and so the function f is primitive recursive.

We can also characterize bland functionals in terms of computer programming languages as those afforded by the applicative part of the programming language C enriched with polymorphism. This is because functions can be arguments and results in C but we do not have there the true lambda abstraction (see Par. 1.9.17). Thus the only functions accepted and yielded by C functionals are those defined in C.

1.9.17 Ackermann-Péter function defined by a detour through bland functionals. We introduce into PA by primitive recursion a binary bland *iteration* functional

$$F(f,0) = f(1)$$
$$F(f,m+1) = f \bullet F(f,m)$$

typed as $F :: (N \mapsto N), N \mapsto N$. The functional has thus a bland index $'F$ typed as

$$'F : (N \mapsto N) \times N \mapsto N \ . \tag{1}$$

Note that we have $F(f,m) = f^{(m+1)}(1)$ where $f^{(0)}(a) = a$ and $f^{(i+1)}(a) = f \bullet f^{(i)}(a)$. We would like to define a function

$$A(0) = 'S$$
$$A(n+1) = \lambda m.'F \bullet (A(n), m) \ .$$

where $S :: N \mapsto N$ is the successor function. We do not intend to introduce lambda terms into PA (because they are variable binding operators just as the case terms), but we can equivalently define by primitive recursion:

$$A(0) = 'S$$
$$A(n+1) = 'Cf \bullet A(n) \ .$$

provided we can find an auxiliary functional Cf which is a *curried* form of F, i.e. such that

$$Cf :: (N \mapsto N) \mapsto N \mapsto N \tag{2}$$

and $'Cf \bullet f \bullet m = 'F \bullet (f,m)$. We can then explicitly define the Ackermann-Péter function as $Ack(n,m) = A(n) \bullet m$. The function Ack is not primitive recursive and so it cannot have a bland definition with the typing $Ack :: N, N \mapsto N$. Thus the functional Cf cannot have a typed bland definition. On the other hand, the functional Cf can be introduced into PA by an explicit definition

$$Cf(f) = \boldsymbol{cd}('F\underline{\bullet}\boldsymbol{P}(f, \boldsymbol{x}_0), 1, 0, 0) \quad .$$

PA even proves

$$f : N \mapsto N \to L(f) : N \mapsto N \wedge {}'Cf \bullet f \bullet m = {}'F \bullet (f, m)$$

but Cf cannot be typed as (2). Nevertheless, the above explicit definition of Ack shows the power of the bland application operator $\bullet$ as at least as strong as the Ackermann-Péter function.

We will introduce below the *curried* functionals where the index of $'F$ will obtain the curried type $'F : (N \mapsto N) \mapsto N \mapsto N$ rather than the bland type (1). We will not need then the functional Cf because we can directly define the above A by well-typed primitive recursion:

$A(0) = 'S$
$A(n+1) = 'F \bullet A(n)$.

The Ackermann-Péter function is then defined as a typed curried functional with the same explicit definition: $Ack(n, m) = A(n) \bullet m$.

Curried types of indices give the curried functionals powers equivalent to lmabda abstraction. But, as we will see below, there is a price to be paid for the increased power of definability. For instance, the application operator $\bullet$ for curried functionals still has a Σ_1-definition but the whole of PA cannot prove its existence condition.

1.9.18 Curried intensional functionals. Curried intensional functionals differ from the the bland ones in the index properties, in function types, and in the meaning of the application operator $\bullet$.

Curried definitions are just like bland definitions, i.e. narrow mixed definitions in $\bullet$, except that for every well-typed definition $f :: s_1, \ldots, s_n \mapsto t$ the code $'f$ of f has the *curried index properties*:

$$'f : s_1 \mapsto \ldots \mapsto s_n \mapsto t \tag{1}$$

$$x_1 : s_1 \wedge \ldots \wedge x_n : s_n \to {}'f \bullet x_1 \bullet \ldots \bullet x_n = f(x_1, \ldots, x_n) \ . \tag{2}$$

This seemingly small change from the bland index type $s_1 \times \ldots \times s_n \mapsto t$ to the curried index type $s_1 \mapsto \ldots \mapsto s_n \mapsto t$ has far reaching consequences as (2) will not be provable in PA unless the type levels of $s_1, \ldots, s_n$ will be 0. Property (2) will be though satisfied in the standard model of PA.

The typing property (1) will be derivable in PA but not in the typing calculus. For that we have to replace the bland typing axioms 1.9.3(5) by

$$\frac{}{\Delta, f :: s_1, \ldots, s_n \mapsto t \Rightarrow {}'f : s_1 \mapsto \ldots \mapsto s_n \mapsto t} \ . \tag{3}$$

The reader can visualize the curried application operator $\bullet$ to be the completion of the following partial function operating on codes:

$$c\bullet x \simeq \begin{cases} c_1 & c \text{ codes } f(y_1, y_2 \ldots, y_n) \simeq \tau[f; y_1, y_2 \ldots, y_n],\ n > 1, \text{ and} \\ & c_1 \text{ codes } g(y_2, \ldots, y_n) \simeq \tau[g; x, y_2, \ldots, y_n] \\ \{c\}(x) & \text{otherwise.} \end{cases}$$

Here the partial universal function $\{c\}(x)$ discussed in Par. 1.9.1 can be visualized as defined by

$$\{c\}(x) \simeq y \leftrightarrow \exists p\, \boldsymbol{Ct}\left(\frac{p}{(c\underline{\bullet}\boldsymbol{x}_0)[0; x] = y}\right) .$$

1.9.19 Computation of well-typed curried terms. We compute bland terms into *curried* numerals which are obtained from the mixed numerals by the addition as new irreducible terms of all well-typed application terms

$$'f\bullet\rho_1 \ldots \bullet\rho_k$$

where $0 \leq k < n$, the n-ary function f has a well-typed curried definition, and $\rho_1, \ldots, \rho_k$ are curried numerals. We close the above terms under pairing and binary successors. Note that similarly as bland numerals, the class of curried numerals is defined relatively to the current extension of PA. We replace the reductions 1.9.3(1) for bland indices by the following ones:

$$'f\bullet\rho_1\bullet \ldots \bullet\rho_n \blacktriangleright f(\rho_1, \ldots, \rho_n) \tag{1}$$

where f is as above, $f(\rho_1, \ldots, \rho_n)$ is well-typed, and $\rho_1, \ldots, \rho_n$ are curried numerals.

It can be shown that the reductions of all well-typed curried terms terminate in curried terminals with no conversions whatsoever.

1.9.20 Arithmetized curried typing predicate. Going from bland to curried functionals we must change the codes of defined function applications $f(\tau_1, \ldots, \tau_n)$ from $'f\underline{\bullet}(\ulcorner\tau_1, \ldots, \tau_n\urcorner)$ to

$$'f\underline{\bullet}\ulcorner\tau_1\urcorner\bullet \ldots \underline{\bullet}\ulcorner\tau_n\urcorner .$$

We then modify the definition of the arithmetized type derivation predicate $\boldsymbol{Dt}(p)$ from Par. 1.9.9 by replacing its clause concerned with the typing of codes $\boldsymbol{cd}(a, n, m, e)$ of function definitions by the following clause taking into account the curried typing axioms 1.9.18(3):

$$\begin{aligned} \boldsymbol{Dt}\left(\frac{p_1, p_2, p_3, 0}{\boldsymbol{cd}(a, n, m, e)[r; s] : t}\right) \leftarrow\ & n > 0 \wedge \mathit{Uncurry}(n, t) = t_1 \mapsto t_2 \wedge \\ & \boldsymbol{Dt}(p_1) \wedge p_1 = \left(\frac{q_1}{a[t_1 \mapsto t_2; t_1] : t_2}\right) \wedge \\ & \boldsymbol{Dt}(p_2) \wedge p_2 = \left(\frac{q_2}{m[0; t_1] : \boldsymbol{N}}\right) \wedge \\ & \boldsymbol{Dt}(p_3) \wedge p_3 = \left(\frac{q_3}{e[0; t_1] : t_2}\right) . \end{aligned}$$

Here the binary primitive recursive function *Uncurry* is introduced into PA to satisfy:

$$Uncurry(n,t) = \begin{cases} s_1 \times \ldots \times s_n \mapsto u & \text{if } t = s_1 \mapsto \ldots \mapsto s_n \mapsto u \\ 0 & \text{otherwise.} \end{cases}$$

The typing predicate $a[r;s] \underline{:} t$ arithmetizing the derivability in the curried typing calculus is now introduced into PA as a primitive recursive predicate just as in Par. 1.9.10.

1.9.21 Curried function types. Curried function types are defined similarly as in Par. 1.9.11 except that the property 1.9.11(1) is replaced by

$$\begin{aligned} x : s \mapsto t \leftrightarrow \exists a \exists n \exists m \exists e (x = \boldsymbol{cd}(a,n,m,e) \wedge x[0;0] \underline{:} s \mapsto t) \vee \\ \exists y \exists z \exists u (x = y \underline{\bullet} z \wedge y : u \mapsto s \mapsto t \wedge z : u) \ . \end{aligned}$$

In spite of the unbounded existential quantifier on u, the predicate $x : t$ is primitive recursive.

1.9.22 Arithmetization of computation of curried terms. We modify the predicate $\boldsymbol{Ct}$ arithmetizing computations of bland functionals (see Par. 1.9.13) by replacing its $a \underline{\bullet} b$ clause given in Figure 1.12 by the following clauses:

$$\begin{aligned} \boldsymbol{Ct}\left(\frac{p_1,p_2,p_3,0}{(a \underline{\bullet} b)[c;x] = y_4}\right) \leftarrow \boldsymbol{Ct}(p_1) \wedge p_1 = \left(\frac{q_1}{a[c;x] = y_1}\right) \wedge \\ \boldsymbol{Ct}(p_2) \wedge p_2 = \left(\frac{q_2}{b[c;x] = y_2}\right) \wedge \\ Red(1,y_1,y_2) = d, y_3 \wedge d = \boldsymbol{cd}(k,n,m,e) \wedge \\ \boldsymbol{Ct}(p_3) \wedge p_3 = \left(\frac{q_3}{k[d;y_3] = y_4}\right) \\ \boldsymbol{Ct}\left(\frac{p_1,p_2,0}{(a \underline{\bullet} b)[c;x] = y_3}\right) \leftarrow \boldsymbol{Ct}(p_1) \wedge p_1 = \left(\frac{q_1}{a[c;x] = y_1}\right) \wedge \\ \boldsymbol{Ct}(p_2) \wedge p_2 = \left(\frac{q_2}{b[c;x] = y_2}\right) \wedge \\ Red(1,y_1,y_2) = 0 \wedge y_1 \underline{\bullet} y_2 = y_3 \ . \end{aligned}$$

Here the ternary function *Red* is introduced into PA by course of values recursion in the second argument:

$$\begin{aligned} &Red(i, \boldsymbol{cd}(k,n,m,e), y) = \boldsymbol{cd}(k,n,m,e), y \leftarrow n = i \\ &Red(i, \boldsymbol{cd}(k,n,m,e), y) = 0 \leftarrow n > i \\ &Red(i, a \underline{\bullet} b, y) = Red(i+1, a, b, y) \ . \end{aligned}$$

We have $Red(1,y_1,y_2) = d, y_3$ iff $d = \boldsymbol{cd}(k,n,m,e)$,

$$y_1 = d \underline{\bullet} x_1 \underline{\bullet} \ldots \underline{\bullet} x_{n-1} \ ,$$

and $y_3 = x_1, \ldots, x_{n-1}, y_2$. This means that the computation step codes the reduction 1.9.19(1) and the function whose definition is coded by d should be applied to arguments y_3. If $Red(1, y_1, y_2) = 0$ then we have $y_1 = \boldsymbol{cd}(k, n, m, e)\underline{\bullet}x_1\underline{\bullet}\ldots\underline{\bullet}x_i$ for $i + 1 < n$ and the value of $a\underline{\bullet}b$ is the code of the irreducible curried term:

$$y_3 = y_1\underline{\bullet}y_2 = \boldsymbol{cd}(k, n, m, e)\underline{\bullet}x_1\underline{\bullet}\ldots\underline{\bullet}x_i\underline{\bullet}y_2 \ .$$

1.9.23 Characterization of computation of curried terms. The introduction of curried function types increases so strongly the power of intensional functionals that PA does not prove the curried counterpart of the Main lemma 1.9.14(1):

$$(r = 0 \vee r = r_1 \mapsto r_2 \wedge c : r) \wedge (s = 0 \vee x : s) \wedge a[r; s]\,\underline{:}\,t \\ \rightarrow \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{a[c; x] = y}\right) \qquad (1)$$

although the formula is satisfied in the standard model of PA.

Only the following special forms are provable in PA:

$$x : s \wedge lev(s) = 0 \wedge \underline{n}_m[0; x]\,\underline{:}\,t \rightarrow \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{\underline{n}_m[0; x] = y}\right) \ .$$

As the code n gets more complex the induction axioms needed to prove the special case are of increasing quantifier complexity. It can be shown that every provably recursive function of PA can be introduced in this way by a suitable choice of the code n.

1.9.24 The theory PA(V). If one wishes to have the intensional functionals with the full power of lambda abstraction, and all adherents of functional programming in the style of Haskell would agree, then one needs to strengthen PA. The strengthening of PA is not needed because we wish to have more provably recursive functions but because we wish comfortable programming with curried functionals.

The strengthening of PA to the theory PA(V) is obtained by suitably extending PA with the auxiliary functions and predicates needed in the definition of the function V (see Par. 1.6.4) and then adding the three clauses for V as axioms. One also needs to extend the induction axioms so one can use the symbol V in the induction formulas. Since V is Δ_1-definable by a formula $\phi[x, y]$, it suffices to add one induction axiom for the formula $\phi[x, V(x)]$. The remaining induction axioms with V are then provable. The theory PA(V) is consistent (because the clauses for V are satisfied in the standard model of PA) but it is not a conservative extension of PA.

Nevertheless, the theory PA(V) is strong enough to prove the Main Lemma 1.9.23(1) for the computation of curried functionals.

1.9.25 Definition of the curried application operator. We proceed similarly as in Par. 1.9.15. We first introduce into PA(V) the binary predicate

$$Wtappl(c,x) \leftrightarrow \exists s \exists t (c : s \mapsto t \wedge x : s) \ .$$

The unbounded existential quantifiers have primitive recursive bounds and so the predicate is a primitive recursive one.

We then introduce into PA(V) the application operator $\bullet$ by the following contextual Σ_1-definition:

$$c \bullet x = y \leftrightarrow Wtappl(c,x) \wedge \exists p \boldsymbol{Ct}\left(\frac{p}{(c\underline{\bullet}\boldsymbol{x_0})[0;x] = y}\right) \vee \neg Wtappl(c,x) \wedge y = 0 \tag{1}$$

whose existence condition follows from the Main lemma 1.9.23(1) and the uniqueness condition holds because the values y and p are uniquely determined. Although the definition of the operator is Σ_1, it is not a provably recursive function of PA. This is because the Main lemma is provable only in PA(V).

1.9.26 Characterization of curried functionals. Functions definable by curried definitions (both typed and untyped) are clearly primitive recursive in the curried application operator $\bullet$ defined by 1.9.25(1) as provably recursive in PA(V) but not in PA.

The theory PA(V), but not PA, proves the typing property 1.9.3(1) of the curried application operator $\bullet$. For every well-typed curried definition of $f :: s_1, \ldots, s_n \to t$, PA(V) proves that $'f$ is the index of f by proving the properties 1.9.18(1) and 1.9.18(2). If the type levels of types $s_1, \ldots, s_n$ are 0 then both properties are provable even in PA and so the function f is provably recursive in PA.

1.9.27 Fast growing hierarchy defined as curried functionals. We will now demonstrate the power of curried functionals by defining the functions $F_\alpha(x)$ of the fast growing hierarchy (see Par. 1.6.6) as curried functionals with indices. We will be moving at the limits of definability by curried functionals and so we will be able to introduce at one time only the functions F_α where for some fixed number M we have $\alpha \prec \omega^1_M$.

We first define by primitive recursion the ternary *iteration* functional $I(k,x,y)$ of polymorphic type $I :: N, (t \mapsto t), t \mapsto t$ which we abbreviate to $x^{(k)}(y)$:

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

For each $n \leq M$ we define the function types T^k as abbreviations:

$$T^0 \equiv {}'N$$
$$T^{n+1} \equiv T^n \mapsto T^n$$

and also $(n+1)$-ary intensional functionals I^{n+1} of types

$$I^{n+1} :: T^n, T^{n-1}, \ldots, T^1, T^0 \mapsto T^0$$

explicitly defined by:

$$I^1(k) = k$$
$$I^2(x_1, k) = x_1 \bullet (k+1)$$
$$I^3_0(x_2, x_1, k) = x_2^{(k+1)}(x_1) \bullet 1$$
$$I^{n+4}_0(x_{n+3}, x_{n+2}, x_{n+1}, \ldots, x_1, k) = x_{n+3}^{(k+1)}(x_{n+2}) \bullet x_{n+1} \bullet \ldots \bullet x_1 \bullet k \ .$$

Since the definitions are well-typed, the functionals have indices ${}'I^{n+1}$ such that PA(V) proves ${}'I^{n+1} : T^{n+1}$ and

$$x_n : T^n \wedge \ldots x_1 : T^1 \rightarrow {}'I^{n+1} \bullet x_n \bullet \ldots \bullet x_1 \bullet k = I^{n+1}(x_n, \ldots, x_1, k) \ .$$

We now introduce into PA(V) a unary function in a: $K^{M+1}(a)$ by explicit definition:

$K^{M+1}(a) = {}'I^{M+1}$

and for $n = M-1, M-2, \ldots, 0$ a sequence of unary functions in a: $K^{n+1}(a)$ by course of values recursion:

$K^{n+1}(0) = {}'I^{n+1}$
$K^{n+1}(a \sharp \omega^b) = K^{n+2}(b) \bullet K^{n+1}(a)$.

We claim that every function F_α of the fast growing hierarchy where $\alpha \prec \omega^1_M$, with a the canonical code of the ordinal α, can be explicitly defined by:

$$F_\alpha(k) = K^1(0 \sharp \omega^a) \bullet k \ .$$

Moreover, the functions F_α are defined by well-typed curried definitions and have indices.

Preparatory to the proof of our claim we derive in PA(V) for each $n < M$ by induction on k:

$$K^{n+1}(a \sharp \omega^b \cdot k) = (K^{n+2}(b))^k)(K^{n+1}(a))$$
$$K^1(a \sharp \omega^0 \cdot k) \bullet m = K^1(a) \bullet (m+k)$$

and by complete induction on c:

$$x_{n+1} : T^{n+1} \wedge \ldots \wedge x_1 : T^1 \rightarrow$$
$$K^{n+2}(a \sharp \omega^{b \sharp \omega^c}) \bullet x_{n+1} \bullet \ldots \bullet x_1 \bullet k = K^{n+2}((a \sharp \omega^{b \sharp \omega^c})[k]) \bullet x_{n+1} \bullet \ldots \bullet x_1 \bullet k \ .$$

We then prove by $\prec w^1_M$ induction on a:

$$a \prec w^1_M \to V(a) = 0 \sharp \omega^0 \cdot (K^1(a) \bullet 0) \ .$$

We thus have

$$F_\alpha(k) = A(a,k) = L\,V(0 \sharp \omega^a \sharp \omega^0 \cdot k) = L(0 \sharp \omega^0 \cdot (K^1(0 \sharp \omega^a \sharp \omega^0 \cdot k) \bullet 0) = K^1(0 \sharp \omega^a \sharp \omega^0 \cdot k) \bullet 0 = K^1(0 \sharp \omega^a) \bullet k \ .$$

We now show that the definitions of K^{n+1} and F_α are well-typed and so the functionals have indices. To that end we introduce the type predicate $a : \mathcal{O}$ of a being the code of an ordinal $\prec \epsilon_0$ by

$0 : \mathcal{O}$
$a \sharp \omega^b : \mathcal{O} \leftarrow b : \mathcal{O} \wedge a : \mathcal{O} \quad .$

The reader will note that the type $\mathcal{O}$ is coextensive with the type $List(\mathcal{O})$ and so we have $a : \mathcal{O}$.

The three functions over the codes of ordinals defined in Paragraphs 1.6.2 and 1.6.3 have well-typed definitions and we have $\cdot \sharp \omega^{\cdot} :: \mathcal{O}, \mathcal{O} \mapsto \mathcal{O}$, $\cdot \sharp \omega^{\cdot \cdot} :: \mathcal{O}, \mathcal{O}, N \mapsto \mathcal{O}$, and $(\cdot)[\cdot] :: \mathcal{O}, N \mapsto \mathcal{O}$. It can be then shown that the curried functionals K^{n+1} have well-typed definitions such that $K^{n+1} :: \mathcal{O} \mapsto T^{n+1}$ and thus also F_α has a well-typed definition $F_\alpha : N \mapsto N$.

1.10 Second-Order Issues of Modularity

Conservative Second-Order Calculus of Extensions

Second-order calculus for the description of parameterized extensions of PA. The calulus is very week in that it is conservative over PA. Its purpose is to explicate parameterized modular construction of programs and the abstract data types.

Abstract Data Types

Extraction od efficient clausal definitions from proofs

1.11 Issues Open to Further Research

In-place Modification

Optimization by Computer-Aided Transformations

Concurrency

References

1. J. Barwise. *An Introduction to First-Order Logic.* In Handbook of Mathematical Logic, J. Barwise ed., North-Holland Publishing Co, 1977.
2. E. W. Beth. *The Foundations of Mathematics.* North Holland 1959.
3. G. Boolos and R. Jeffrey, *Computability and Logic*, Cambridge University Press, Cambridge, 1974.
4. M. Davis, *Computability and Unsolvability*, McGraw Hill, New York, 1958.
5. S. Feferman. *Theory of Finite Type Related to Mathematical Practice*, In Handbook of Mathematical Logic, J. Barwise ed., North-Holland Publishing Co, 1977.
6. K. Gödel, Üeber formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, Monatshefte Math. Phys. vol. 38 (1931) pages 173-198.
7. M. J. Gordon, R. Milner, C. P. Wadsworth, *Edinburgh LCF.* Lecture Notes in Computer Science, vol 78. Springer, Berlin 1979.
8. P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic.* Springer Verlag, 1993.
9. D. Hilbert, P. Bernays. *Grundlagen der Mathematik.* Springer 1970.
10. N. D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, 1997.
11. J. Komara, P. J. Voda. *Syntactic Reduction of Predicate Tableaux to Propositional Tableaux.* Workshop on Theorem Proving and Analytic Tableaux, May 1995. Lecture Notes in Artificial Intelligence vol 918, Springer Verlag 1995.
12. J. Komara, P. J. Voda. *On Quasitautologies.* Workshop on Theorem Proving and Analytic Tableaux, May 1997. Lecture Notes in Artificial Intelligence vol. 1227, Springer Verlag 1997.
13. J. Komara, P. J. Voda. *Theorems of Péter and Parson in Computer Programming.* Computer Science Logic Conf. (CSL-98), Brno The Czech Rep. 1998. Lecture Notes in Computer Science vol 1584, Springer Verlag 1999.
14. J. Komara, P. J. Voda. *Towards Provably Correct Programming.* www.fmph.uniba.sk/~voda. April 2000.
15. J. Komara. *Specification and Verification of Computer Programs.* http://dent.ii.fmph.uniba.sk/~voda/cl.html. May 2001.
16. R. Milner, *A Theory of Type Polymorphism in Programming.* J. Comput. System Sci. 17 (1978) 348-375.
17. J. C. Mitchell. *Type Systems for Programming Languages.* In Handbook of Theoretical Compute Science (vol B), J. van Leeuwen ed., Elsevier 1990.
18. R. Péter. *Konstruktion nichtrekursiver Funktionen.* Math. Ann. vol 111 (1935), pages 42-60.
19. H. E. Rose. *Subrecursion: Functions and Hierarchies.* Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.
20. H. Schwichtenberg, *Eine Klassifikation der ϵ_0-rekursiven Funktionen*, Zeitschrift für mathematische Logik und Grundlagen der Mathematik, vol. 17, (1971), pp. 91-74.

21. J. R. Shoenfield. *Mathematical Logic*, Addison-Wesley, 1967.
22. R. M. Smullyan *First-Order Logic*. Springer, 1968.
23. W. W. Tait. *Intensional interpretation of functionals of finite type*. J. of Symbolic Logic, 32:198–212, 1967.
24. G. Takeuti. *Proof Theory*. North-Holland, 1975.
25. A. S. Troelstra. *Aspects of Constructive Mathematics*. In Handbook of Mathematical Logic, J. Barwise ed., North-Holland Publishing Co, 1977.
26. P. J. Voda. *Subrecursion as a basis for a feasible programming language*, in L. Pacholski and J. Tiuryn, editors, Proceedings of CSL'94, number 933 in LNCS Springer Verlag, 1995, pages 324-338.
27. S. S. Wainer. *A classification of the ordinal recursive functions*, Archiv für mathematische Logik und Grundlagen Forschung, vol. 13, 1970, pp. 136-153.