Paul J. Voda

Meta-Mathematics of Computer Programming

(Preliminary Draft)

www.fmph.uniba.sk/∼voda
Institute of Informatics, Comenius University,
Bratislava

May 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 Issues Open to Further Research 88

Part I. First-Order Logic and Peano Arithmetic

2. First-Order Languages 101
- 2.1 Language of First-Order Logic 101

3. Propositional Logic 105
- 3.1 Tautologies 105
- 3.2 Propositional Tableaux 107
- 3.3 Admissible Expansion Rules 113
- 3.4 Tautological Consequence 122
- 3.5 Tableaux with Axioms 125

4. Identity Logic 131
- 4.1 Some Syntactic Concepts 131
- 4.2 Quasitautological Consequence 131
- 4.3 Identity Tableaux 137

5. Quantification Logic . . . 143
5.1 Some Syntactic Concepts . . . 143
5.2 Logical Consequence . . . 146
5.3 Quantification Tableaux . . . 154

6. First-order Theories . . . 167
6.1 Theorems of Predicate Calculus . . . 167
6.2 Extensions of Theories . . . 172
6.3 Extensions by Explicitly Defined Predicates . . . 174
6.4 Skolem Extensions . . . 178
6.5 Extensions by Contextually Defined Functions . . . 188
6.6 Extensions by Definitions . . . 190

7. Peano Arithmetic . . . 195
7.1 Basic Theorems in PA . . . 195
7.2 Extensions of PA . . . 200
7.3 Introduction of Basic Predicates into PA . . . 202
7.4 Introduction of Basic Functions into PA . . . 206
7.5 The Lattice of Divisibility . . . 210

8. Recursive Bootstrapping of PA . . . 219
8.1 Exponentiation Function . . . 219
8.2 Primitive Recursion . . . 233
8.3 Suitable Pairing Function . . . 240
8.4 Course of Values Recursion with Measure . . . 260

9. Proof Theory of PA . . . 277

Part II. Computer Programming

10. Clausal Language . . . 281
10.1 Clausal Definitions . . . 281
10.2 Regular Clausal Definitions . . . 291
10.3 The Strength of Clausal Definitions . . . 293

11. Computation of Clausal Programs . . . 295
11.1 Computation over Monadic Numerals . . . 295
11.2 Computation over Binary Numerals . . . 296
11.3 Computation over Pair Numerals . . . 297
11.4 Computation over Mixed Numerals . . . 298

12. Data Types . . . 301
12.1 Pascal-Style Typing . . . 301
12.2 ML-Style Typing . . . 303

13. Functional Programming 305

14. Modular Programming 307
14.1 Extraction of Programs from Proofs 307

References 309

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 $\varDelta_0$-definition of the graph $2^x = y$ of the exponentiation function,
5. a simple $\varDelta_1$-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* [16] 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 [12, 13]), on the finitary proof that Skolem axioms are conservative, we have investigated the use of Parson's rule in proving properties of programs [14], and the extraction of efficient clausal programs from proofs [15].

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 $1 = 0\mathbf{1} = 0\mathbf{01} = 0\mathbf{001}\ldots$ 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

$$x = 0 \lor \exists z(x = z\mathbf{0} \land z > 0) \lor \exists z\, x = z\mathbf{1}$$

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:

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

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*:

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

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

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

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:

$$\mathrm{gcd}(x,y) \mid x \wedge \mathrm{gcd}(x,y) \mid y \tag{1}$$
$$x+y>0 \wedge d \mid x \wedge d \mid y \rightarrow d \leq \mathrm{gcd}(x,y) \tag{2}$$
$$\mathrm{gcd}(0,0)=0\ . \tag{3}$$

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:

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

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 $\mathrm{gcd}(x,y) = \mathrm{gcd}(x_0,y_0)$ 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:

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

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 $d \le x = \gcd(x, 0)$. If $y > 0$ then, since $d \mid x \bmod y$ by 1.2.1(2) and $y < x \bmod y$, we get

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

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:

$$\begin{aligned}\mathrm{fib}(4) &\overset{(3)}{=} \mathrm{fib}(3) + \mathrm{fib}(2) \overset{2\times(3)}{=} \mathrm{fib}(2) + \mathrm{fib}(1) + \mathrm{fib}(1) + \mathrm{fib}(0) \overset{(3);2\times(2);(1)}{=} \\ &\mathrm{fib}(1) + \mathrm{fib}(0) + 1 + 1 + 0 \overset{(2);(1)}{=} 1 + 0 + 2 = 3\ .\end{aligned}$$

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*
    $n := n \dot{-} 1;\ c := a;\ a := a + b;\ b := c;$
```

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 **??** 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 $x_1, x_2, \ldots, x_n$ can be then coded as a word over the two element alphabet $a_1 \equiv |$, $a_2 \equiv \#$ as

$$\#|^{x_1}\#|^{x_2}\cdots\#|^{x_n}\# \,.$$

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, $S_1, S_2, \ldots, S_p$ 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*:

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

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 $a_{i_1} a_{i_2} \dots a_{i_m}$. 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:

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

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:

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

Clearly, the numbers x such that

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

and no others have the dyadic size n.

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:.

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

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

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

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 [28]. 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

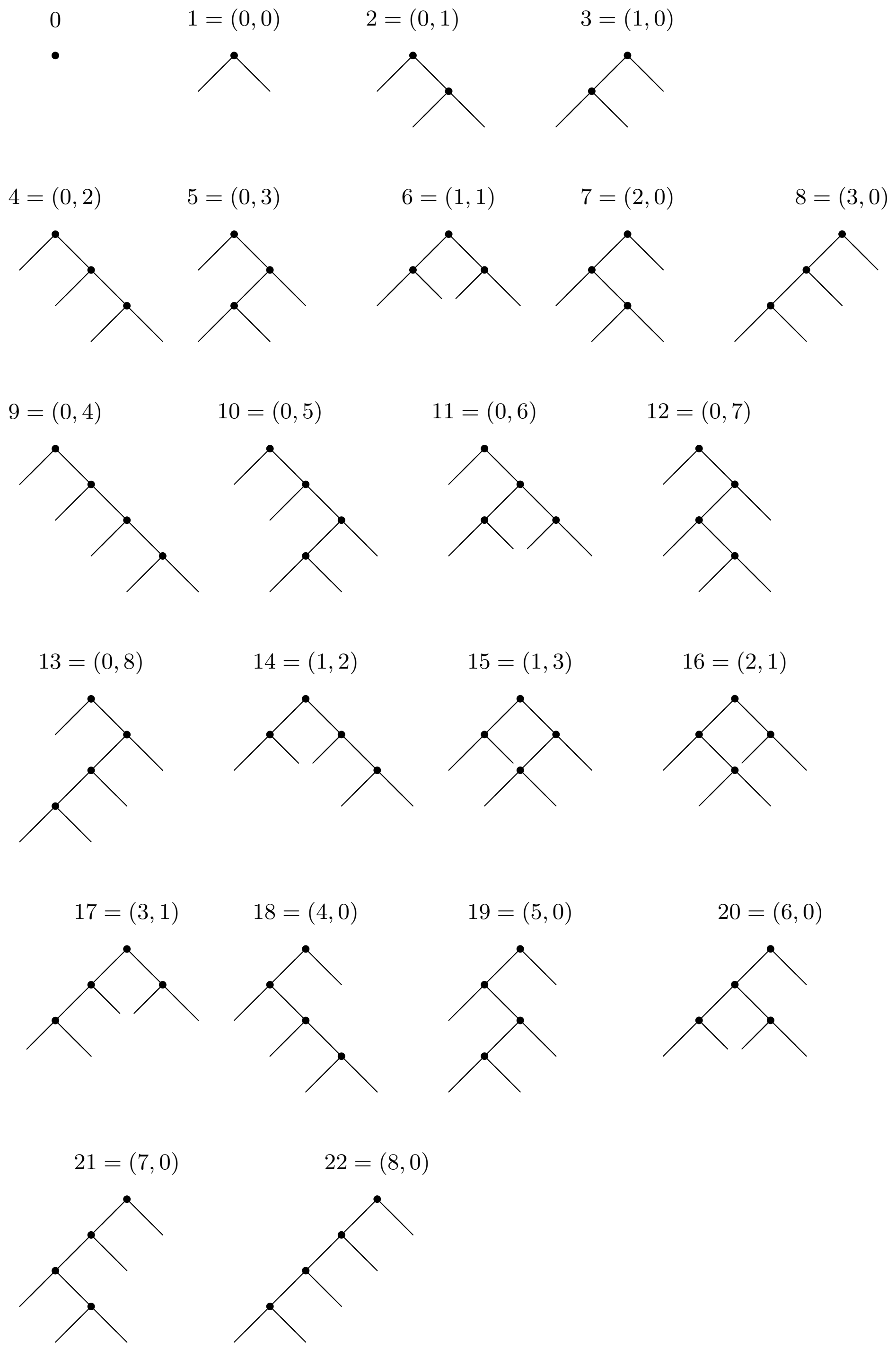


Fig. 1.2. Enumeration of binary trees

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 [28]).

(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:

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

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

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

It should be clear that we have

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

1.3.14 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 $\vec{\tau}$. For instance, when f is an n-ary function then the basic property of its contraction $\langle f\rangle$ is:

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

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

1.3.15 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.

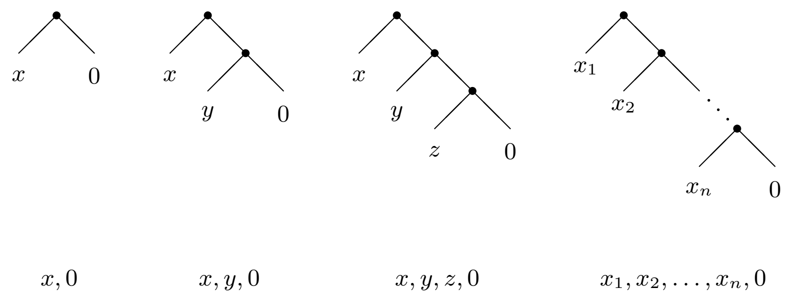


Fig. 1.4. Lists

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:

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

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:

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

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 $x \mathrel{\varepsilon} a$. The binary *list membership* predicate $x \mathrel{\varepsilon} a$ satisfying the specifications:

$$x \mathrel{\varepsilon} a \leftrightarrow \exists i(i < L(a) \wedge (a)_i = x)$$

can be defined by the following clausal definition:

$$x \mathrel{\varepsilon} y, a \leftarrow x = y$$
$$x \mathrel{\varepsilon} y, a \leftarrow x \neq y \wedge x \mathrel{\varepsilon} a \ .$$

The negative clause $x \not\mathrel{\varepsilon} 0$ 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:

$$\begin{aligned}&Bt(E)\\ &Bt\, Nd(x, t_1, t_2) \leftarrow Bt(t_1) \land Bt(t_2) \quad .\end{aligned}$$

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 $\mathit{Flat}(t)$ 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:

$$\begin{aligned}&\mathit{Flat}(E) = 0\\ &\mathit{Flat}\, Nd(x, t_1, t_2) = \mathit{Flat}(t_1) \oplus (x, \mathit{Flat}(t_2)) \ .\end{aligned}$$

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

$$\begin{aligned}&f(E, a) = a\\ &f(Nd(x, t_1, t_2), a) = f(t_1, x, f(t_2, a)) \quad .\end{aligned}$$

We can now explicitly define a fast flatten by:

$$\mathit{Flat}(t) = f(t, 0) \ .$$

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

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

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:

$$\begin{aligned}\mathit{Incr}(t) &\leftrightarrow \forall i \forall j (i < j < L(t) \rightarrow (a)_i < (a)_j)\\ \mathit{Bst}(t) &\leftrightarrow \mathit{Incr}\, \mathit{Flat}(t) \ .\end{aligned}$$

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:

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

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

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

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

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

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

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

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

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

Arithmetization of Numeric Domains

1.3.26 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:

$$\begin{array}{l}
\mathbb{Z}(+(n)) \\
\mathbb{Z}(-(n+1)) \ .
\end{array}$$

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

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

holds, has the following explicit clausal definition:

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

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

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

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:

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

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:

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

In general, for any formula $\phi[x_1, \dots, x_n]$ 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*:

$$R(x_1,\dots,x_n) \leftrightarrow \phi[x_1,\dots,x_n] \ . \qquad (1)$$

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 $\tau[x_1,\dots,x_n]$ 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:

$$f(x_1,\dots,x_n) = \tau[x_1,\dots,x_n] \ . \qquad (1)$$

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* $f(x_1,\dots,x_n) = y$ as an $(n+1)$-ary relation. For instance, the modified subtraction function $x \dot{-} y$ can be introduced by an explicit contextual definition as:

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

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

$$\exists y \phi[x_1,\dots,x_n,y] \tag{2}$$
$$\phi[x_1,\dots,x_n,y_1] \land \phi[x_1,\dots,x_n,y_2] \to y_1 = y_2 \tag{3}$$

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*:

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

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. $\psi[f(\tau_1,\dots,\tau_n)]$ is a formula of S the theory S proves

$$\psi[f(\tau_1,\dots,\tau_n)] \leftrightarrow \exists y(f(\tau_1,\dots,\tau_n) = y \land \psi[y])$$

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

$$\psi[f(\tau_1,\dots,\tau_n)] \leftrightarrow \exists y(\phi[\tau_1,\dots,\tau_n,y] \land \psi[y])$$

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:

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

This can be done because PA proves the existence condition:

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

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

$$q_1{\cdot}y + r_1 = q_2{\cdot}y + r_2 \land r_1 < y \land r_2 < y \to q_1 = q_2 \land r_1 = r_2 \ .$$

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:

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

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:

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

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:

$$(x {=_*} y) = z \leftrightarrow x = y \land z = 1 \lor x \neq y \land z = 0$$
$$(x {<_*} y) = z \leftrightarrow x < y \land z = 1 \lor x \geq y \land z = 0$$

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:

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

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:

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

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:

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

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

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

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.

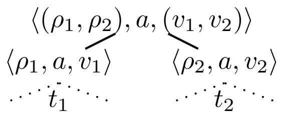
We now outline how to find a formula $\phi[\vec{x}; y]$ of T such that when T is extended by definition to S_1 with the axiom

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

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.

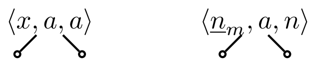
We will introduce the function f with the help of a predicate arithmetizing the computation of f in a *computation* tree. Without loss of generality, because we can always work with contractions, we assume that f is unary and that the term τ is built up from the variable x and from the numerals $\underline{n}_m$ 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:


$\langle(\rho_1,\rho_2),a,(v_1,v_2)\rangle$
$\langle\rho_1,a,v_1\rangle$
$\langle\rho_2,a,v_2\rangle$
t_1
t_2

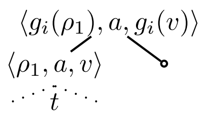
where the value v_1 of ρ_1 is computed in the left son and the value v_2 of ρ_2 in the right son. The value of (ρ_1, ρ_2) is then the pair (v_1, v_2).

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



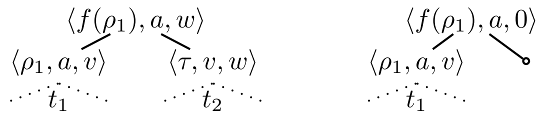
where there are no subcomputations because the values of x and $\underline{n}_m$ can be determined directly as a and n respectively.

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


$\langle g_i(\rho_1),a,g_i(v)\rangle$
$\langle\rho_1,a,v\rangle$
t

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:


$\langle f(\rho_1),a,w\rangle$
$\langle\rho_1,a,v\rangle$
$\langle\tau,v,w\rangle$
t_1
t_2
$\langle f(\rho_1),a,0\rangle$
$\langle\rho_1,a,v\rangle$
t_1

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:

$$\exists x(x \leq \tau \land \phi[x])$$
$$\exists x(x \leq \tau \to \phi[x])$$

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

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

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:

$$\begin{aligned}
&u \trianglelefteq t \leftarrow u = t \\
&u \trianglelefteq n, t_1, t_2 \leftarrow u \neq n, t_1, t_2 \wedge (u \trianglelefteq t_1 \vee u \trianglelefteq t_2) \ .
\end{aligned}$$

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 $\mathcal{O}(\log(z))$:

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

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.

```
$[\sqrt{z}] = y \leftrightarrow$ case
          $z < 4 \rightarrow$ case
                  $z = 0 \rightarrow 0 = y$
                  $z > 0 \rightarrow 1 = y$
          $z \geq 4 \rightarrow$ let
                  $z = 4{\cdot}x + i \land i < 4 \rightarrow_{x,i}$
                    let
                      $[\sqrt{x}] = s \rightarrow_s$ case
                                 $4{\cdot}x + i < (s\mathbf{1})^2 \rightarrow s\mathbf{0} = y$
                                 $4{\cdot}x + i \geq (s\mathbf{1})^2 \rightarrow s\mathbf{1} = y$
```

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:

```
case
  $\phi_1 \rightarrow \psi_1$
  $\phi_2[x, y] \rightarrow_{x,y} \psi_2[x, y]$
  $\phi_3[x] \rightarrow_x \psi_3[x]$
```

as standing for the formula

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

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:

$$\phi_1 \lor \exists x!\exists y!\, \phi_2[x,y] \lor \exists x!\, \phi_3[x]$$
$$\phi_2[x,y] \lor \phi_3[x] \to \neg\phi_1$$
$$\phi_2[x,y] \to \neg\exists x \phi_3[x]\ .$$

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

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

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
$z = 4{\cdot}x + i \land i < 4 \to_{x,i} \phi[x,i]$

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 $\phi \leftarrow \psi_1 \lor \psi_2$ and

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

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:

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

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.

```
$[\sqrt{z}] =$ **case**
    $\overline{sgn}(z <_* 4) = 0 \rightarrow$ **case**
                                  $\overline{sgn}(z =_* 0) = 0 \rightarrow 0$
                                  $\overline{sgn}(z =_* 0) = 1 \rightarrow 1$
    $\overline{sgn}(z <_* 4) = 1 \rightarrow$ **let**
                                  $0, qr(z) = 0, x, i \rightarrow_{x,i}$
                                    **let**
                                      $0, [\sqrt{x}] = 0, s \rightarrow_s$ **case**
                                          $\overline{sgn}(4{\cdot}x + i <_* (s\mathbf{1})^2) = 0 \rightarrow s\mathbf{0}$
                                          $\overline{sgn}(4{\cdot}x + i <_* (s\mathbf{1})^2) = 1 \rightarrow s\mathbf{1}$
```

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

```
**case**
  $\tau = 0 \rightarrow \alpha_0$
    $\vdots$
  $\tau = \underline{m-1}_m \rightarrow \alpha_{m-1}$
  $\tau = \underline{m}_m, x_1, \ldots, x_{n_m} \rightarrow_{x_1,\ldots,x_{n_m}} \alpha_m[x_1, \ldots, x_{n_m}]$
    $\vdots$
  $\tau = \underline{k-1}_m, x_1, \ldots, x_{n_{k-1}} \rightarrow_{x_1,\ldots,x_{n_{k-1}}} \alpha_{k-1}[x_1, \ldots, x_{n_{k-1}}]$
```

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:

$$\tau = 0 \lor \ldots \lor \tau = \underline{m-1}_m \lor \exists \vec{x}_i\, \tau = \underline{m}_m, \vec{x}_m \lor \ldots \lor \exists \vec{x}_{k-1}\, \tau = \underline{k-1}_m, \vec{x}_{k-1} \,. \qquad (1)$$

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*,* $\ldots$*,* 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

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

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

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.

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

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 ϕ_0, …, ϕ_{m-1}, $\phi_m[\vec{y}_m]$, …, $\phi_{k-1}[\vec{y}_{k-1}]$ 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.

$$\begin{array}{l}
D(\tau{=_*}\underline{0}_m, \alpha^\star_0, \\
\quad \ldots \\
\quad D(\tau{=_*}\underline{m-1}_m, \alpha^\star_{m-1}, \\
\qquad D(H(\tau){=_*}\underline{m}_m, \alpha^\star_m[H\,T^1(\tau), \ldots, H\,T^{n_m-1}(\tau), T^{n_m}(\tau)], \\
\qquad\quad \ldots \\
\qquad\quad D(H(\tau){=_*}\underline{k-2}_m, \alpha^\star_{k-2}[H\,T^1(\tau), \ldots, H\,T^{n_{k-2}-1}(\tau), T^{n_{k-2}}(\tau)], \\
\qquad\qquad\qquad\qquad\qquad \alpha^\star_{k-1}[H\,T^1(\tau), \ldots, H\,T^{n_{k-1}-1}(\tau), T^{n_{k-1}}(\tau)])\ldots))\ldots)
\end{array}$$

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

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

where ψ_1, ..., ψ_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

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

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:

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

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

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

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

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

and

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

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

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

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

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

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:

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

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:

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

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

$$
\begin{aligned}
x \oplus y = \mathbf{case}& \\
&p(x) = 0 \to y \\
&p(x) = 1, v, w \to_{v,w} v, w \oplus y
\end{aligned}
$$

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

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

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:

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

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

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

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:

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

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:

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

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:

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

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

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

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 ultra-modern 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 $z = 4{\cdot}x + i \land i < 4$. 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:

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

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

$$\begin{aligned}&\omega^i_0 = 0 \sharp \omega^0 {\cdot} i\\ &\omega^i_{k+1} = 0 \sharp \omega^{\omega^i_k} \ .\end{aligned}$$

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 $a \prec \omega^i_k$ which is introduced into PA by course of values recursion in a:

$$\begin{aligned}&0 \prec \omega^i_k \leftarrow k > 0 \lor i > 0\\ &a \sharp \omega^0 \prec \omega^{i+1}_0 \leftarrow a \prec \omega^i_0\\ &a \sharp \omega^b \prec \omega^i_{k+1} \leftarrow a \prec \omega^i_{k+1} \land b \prec \omega^i_k \ .\end{aligned}$$

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 [21]) 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

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

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:

$$\begin{aligned} Ack(0,n) &= n+1 && (1) \\ Ack(m+1,0) &= Ack(m,1) && (2) \\ Ack(m+1,n+1) &= Ack(m, Ack(m+1,n)) && (3) \end{aligned}$$

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 [29] and Schwichtenberg [22] (see also Rose [21]). 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:

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

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:

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

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

$$\begin{aligned}
&V^0_e(p) = p \\
&V^{n+1}_e(p) = V_e\,V^n_e(p)\quad .
\end{aligned}$$

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

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

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

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

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

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

simplify the discriminators of binary case terms. Reductions

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

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)

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

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 [28]), 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 [18]) 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

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

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

For instance, the predicate

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

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

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

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:

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

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

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

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:

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

which applies a binary *universal* typing predicate $x \mathbin{\boldsymbol{:}} t$. 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

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

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

$$x \mathbin{\boldsymbol{:}} {}'T \leftrightarrow x : T \ .$$

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, [18]).

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* $'T(\tau_1,\dots,\tau_n)$ for $n \geq 0$ in an abbreviated form $T(\tau_1,\dots,\tau_n)$. We can thus write $x : \mathit{List}(T)$ instead of $x : \mathit{List}('T)$. The reader will note that by (1) the last is equivalent to $x \mathbin{\pmb{:}} '\mathit{List}('T)$ which can be abbreviated to $x \mathbin{\pmb{:}} \mathit{List}(T)$.

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

$$\begin{aligned}&E : \mathit{Bt}(t)\\ &\mathit{Nd}(n,l,r) : \mathit{Bt}(t) \leftarrow n \mathbin{\pmb{:}} t \wedge l : \mathit{Bt}(t) \wedge r : \mathit{Bt}(t) \quad .\end{aligned}$$

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

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 $x : \mathit{Vect}_1(n,t)$ holding of *fixed* vectors is introduced into PA by primitive recursion:

$$\begin{aligned}&0 : \mathit{Vect}_1(0,t)\\ &v,w : \mathit{Vect}_1(n+1,t) \leftarrow v \mathbin{\pmb{:}} t \wedge w : \mathit{Vect}_1(n,t) \ .\end{aligned}$$

For every positive number n and every type predicate T fixed vectors $x : \mathit{Vect}_1(\underline{n}_b,'T)$ are represented in memory just like the Pascal arrays of type **array**$[0..(n{-}1)]$ **of** T.

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

$$n,x : \mathit{Vect}(t) \leftarrow x : \mathit{Vect}_1(n,t) \ .$$

For every type predicate T the flexible vectors $x : \mathit{Vect}('T)$ are represented in computer memory as pointers to *dependent* Cartesian pairs of type $N \times \mathit{Vect}_1(n,'T)$ 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 **??**) 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 $x \mathbin{\boldsymbol{:}} c$ which acts as the universal typing predicate and is such that PA proves

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

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

$$x \mathbin{\boldsymbol{:}} \ulcorner T \urcorner \circ 0 \leftrightarrow x : T \ .$$

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:

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

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:

$$x \mathbin{\boldsymbol{:}} t \to t = \boldsymbol{N} \lor \exists s\, t = \boldsymbol{vect}(s) \lor \exists s_1 \exists s_2\, t = s_1{\times}s_2 \lor \\ \exists s_1 \exists s_2\, t = s_1{\times^*}s_2 \lor \exists m \exists s\, t = \boldsymbol{Un}(m, s) \lor \exists c \exists p\, t = c{\circ}p$$

$$x \mathbin{\boldsymbol{:}} \boldsymbol{N}$$

$$x \mathbin{\boldsymbol{:}} \boldsymbol{vect}(t) \leftrightarrow \exists n \exists a (x = n+1, a \land L(a) = n+1 \land \forall i (i \le n \to (a)_i \mathbin{\boldsymbol{:}} t))$$

$$x \mathbin{\boldsymbol{:}} t_1{\times}t_2 \leftrightarrow \exists v \exists w (x = v, w \land v \mathbin{\boldsymbol{:}} t_1 \land w \mathbin{\boldsymbol{:}} t_2)$$

$$x \mathbin{\boldsymbol{:}} t_1{\times^*}t_2 \leftrightarrow x = 0 \lor x \mathbin{\boldsymbol{:}} t_1{\times}t_2$$

$$x \mathbin{\boldsymbol{:}} c{\circ}p \leftrightarrow x \mathbin{\boldsymbol{:}} sb(c, c, p)$$

$$x \mathbin{\boldsymbol{:}} \boldsymbol{Un}(m, s) \leftrightarrow x < m \lor \exists i \exists y (i < L(s) \land x = m + i, y \land y \mathbin{\boldsymbol{:}} (s)_i)\ .$$

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. $\ulcorner \mathit{Vect} \urcorner \equiv \boldsymbol{vect}\,\boldsymbol{par}(0)$, and so

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

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 $\ulcorner \mathit{List} \urcorner$ of the polymorphic list type predicate $\mathit{List}(t)$ (see Par. 1.8.6) is as in the following:

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

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:

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

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 $x \mathrel{\boldsymbol{:}} {}'List(t)$ 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 $x \mathrel{\boldsymbol{:}} c$ is primitive recursive.

The constructor $\boldsymbol{Un}$ codes union type predicates. The idea is that we have $x \mathrel{\boldsymbol{:}} \boldsymbol{Un}(m, \rho_m, \ldots, \rho_{k-1}, 0)$ iff x is a union value determined by m, k, n_m, ..., n_{k-1} (see Par. 1.5.6). Moreover, for every j s.t. $m \le j < k$ 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 $y_1 \mathrel{\boldsymbol{:}} \sigma_1$, ..., $y_{n_j} \mathrel{\boldsymbol{:}} \sigma_{n_j}$. The reader will note that the last is equivalent to $x = \underline{j}_b, y$ and $y \mathrel{\boldsymbol{:}} \rho_j$.

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:

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

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 $x \mathrel{\boldsymbol{:}} \rho$. Type terms are constructed from variables (say t_1, t_2, ...) 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 $'T(\rho_1,\dots,\rho_n)$ 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 $\ulcorner T\urcorner \circ (\rho_1,\dots,\rho_n,0)$.

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 $'T(t_1,\dots,t_n)$ We obtain thereby a type term $\rho[t_1,\dots,t_n]$ such that PA proves

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

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

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

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

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

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

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

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 $f(\tau_1,\dots,\tau_n)$ 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

$$\varDelta \Rightarrow \alpha \mathbin{\pmb{:}} \rho \tag{3}$$

where α is a generalized term and ρ a type term. $\varDelta$ is a sequence of finitely many (possibly empty) assumptions (guards). The assumptions are of the form $x \mathbin{\pmb{:}} \rho$ or $f :: \rho_1, \ldots, \rho_n {\mapsto} \rho$. The order of assumptions in $\varDelta$ is irrelevant and the reader can treat $\varDelta$ 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 $\varDelta$. We say that the typing calculus derives $\alpha \mathbin{\pmb{:}} \rho$ if it derives the sequent $\emptyset \Rightarrow \alpha \mathbin{\pmb{:}} \rho$.

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 $\varDelta \Rightarrow \alpha \mathbin{\pmb{:}} \rho$ then PA proves $\varDelta^\star \rightarrow \alpha^* \mathbin{\pmb{:}} \rho$ where $\varDelta^\star$ stands for the conjunction of translated assumptions. Here an assumption $x \mathbin{\pmb{:}} \rho$ is translated into itself and an assumption $f :: \rho_1, \ldots, \rho_n {\mapsto} \rho$ into

$$\forall x_1 \ldots \forall x_n (x_1 \mathbin{\pmb{:}} \rho_1 \land \ldots x_n \mathbin{\pmb{:}} \rho_n \rightarrow f(x_1, \ldots, x_n) \mathbin{\pmb{:}} \rho) \ .$$

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 meta-variables 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:

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

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

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:

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

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

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

$$\begin{array}{cc} \overline{sgn} :: {'N} \mapsto {'\mathit{Bool}} & \overline{sgn} :: {'\mathit{Bool}} \mapsto {'\mathit{Bool}} \\ b :: {'N} \mapsto \boldsymbol{Un}(0, {'N}, {'N}, 0) & c :: t_1 \times t_2 \mapsto \boldsymbol{Un}(0, (t_1 \times t_2), 0) \end{array}$$

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

$\mathit{Bool}(0)$
$\mathit{Bool}(1)$.

It should be clear that we have

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

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 $\ldots \mapsto {'\mathit{Bool}}$.

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

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

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 discriminator term τ by the union type (1). These considerations lead to the following inference rule:

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

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

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:

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

As an example of such rules we show in Fig. 1.10 the derivation of $0, 1, 0 : \mathit{List}(\mathit{Bool})$ 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:

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

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, $\ldots$, 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 $f :: t{\mapsto}t{\times}N{\times}N$, $f :: t_1{\times}^*t_2 \to \mathit{List}(t_1{\times}^*t_2)$, or $f :: N{\mapsto}\mathit{List}(N)$. 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 [17]. 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, ..., 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 [25], 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 written 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:

$$c : s{\mapsto}t \leftrightarrow \forall x(x \mathbin{\mathbf{:}} s \rightarrow \exists y(c{\bullet}x \simeq y \wedge y \mathbin{\mathbf{:}} t)) \ .$$

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, 27]. 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 $x \mathbin{\mathbf{:}} t$ 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 $c \mathbin{\mathbf{:}} s{\mapsto}t$ 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

$$x : s \mapsto t \land y : s \to x \bullet y : t \ . \tag{1}$$

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:

$$'f : s_1 \times \ldots \times s_n \mapsto t \tag{3}$$

$$x_1 : s_1 \land \ldots \land x_n : s_n \to {'f} \bullet (x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \ . \tag{4}$$

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

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

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:

$$f : s\mapsto t \land x : List(s) \rightarrow {}'Map\bullet(f,x) = Map(f,x)\ . \tag{1}$$

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:

$$x[0] = x$$
$$(v, w)[i\mathbf{1}] = v[i]$$
$$(v, w)[i\mathbf{2}] = w[i] \ .$$

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 $'f \boldsymbol{:} s_1 \times \ldots \times s_n \mapsto t$ 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:

$$\left(\frac{p}{a[r;s] : t}\right) = (a,r,s,t),p\ .$$

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:

$$\begin{aligned}
&\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{z}[r;s] : \boldsymbol{N}}\right)\\
&\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{z}[r;s] : t_1{\times^*}t_2}\right)\ .
\end{aligned}$$

Some clauses coding inference rules with one or two premises:

$$\begin{aligned}
&\boldsymbol{Dt}\left(\frac{p_1,0}{\boldsymbol{S}_1(a)[r;s] : \boldsymbol{N}}\right) &&\leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[r;s] : \boldsymbol{N}}\right)\\
&\boldsymbol{Dt}\left(\frac{p_1,p_2,0}{\boldsymbol{P}(a,b)[r;s] : t_1{\times}t_2}\right) &&\leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[r;s] : t_1}\right) \land\\
& &&\ \boldsymbol{Dt}(p_2) \land p_2 = \left(\frac{q_2}{b[r;s] : t_2}\right)\\
&\boldsymbol{Dt}\left(\frac{p_1,0}{\boldsymbol{P}(a,b)[r;s] : t_1{\times^*}t_2}\right) &&\leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{\boldsymbol{P}(a,b)[r;s] : t_1{\times}t_2}\right)\ .
\end{aligned}$$

Clauses for $\boldsymbol{Dt}$ making use of assumptions in the antecedents of sequents are:

$$\begin{aligned}
&\boldsymbol{Dt}\left(\frac{p_1}{\boldsymbol{rec}(a)[r_1{\mapsto}r_2;s] \mathbin{\boldsymbol{:}} t}\right) \leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[r_1{\mapsto}r_2;s] \mathbin{\boldsymbol{:}} r_1}\right) \land t = r_2\\
&\boldsymbol{Dt}\left(\frac{0}{\boldsymbol{x}_i[r;s] \mathbin{\boldsymbol{:}} t}\right) \qquad\qquad \leftarrow t = s[\![i]\!]
\end{aligned}$$

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:

$$\begin{aligned}
&t[\![0]\!] = t\\
&(t_1{\times}t_2)[\![i\mathbf{1}]\!] = t_1[\![i]\!]\\
&(t_1{\times}t_2)[\![i\mathbf{2}]\!] = t_2[\![i]\!] \ .
\end{aligned}$$

Clauses coding the inference rules concerned with the application operator are:

$$\begin{aligned}
&\boldsymbol{Dt}\left(\frac{p_1,p_2,0}{(a\underline{\bullet}b)[r;s] \mathbin{\boldsymbol{:}} t_2}\right) &&\leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[r;s] \mathbin{\boldsymbol{:}} t_1{\mapsto}t_2}\right) \land\\
& &&\quad\boldsymbol{Dt}(p_2) \land p_2 = \left(\frac{q_2}{b[r;s] \mathbin{\boldsymbol{:}} t_1}\right)\\
&\boldsymbol{Dt}\left(\frac{p_1,p_2,p_3,0}{\boldsymbol{cd}(a,n,m,e)[r;s] \mathbin{\boldsymbol{:}} t_1{\mapsto}t_2}\right) &&\leftarrow n > 0 \land Arity(t_1) \geq n \land\\
& &&\quad\boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[t_1{\mapsto}t_2;t_1] \mathbin{\boldsymbol{:}} t_2}\right) \land\\
& &&\quad\boldsymbol{Dt}(p_2) \land p_2 = \left(\frac{q_2}{m[0;t_1] \mathbin{\boldsymbol{:}} \boldsymbol{N}}\right) \land\\
& &&\quad\boldsymbol{Dt}(p_3) \land p_3 = \left(\frac{q_3}{e[0;t_1] \mathbin{\boldsymbol{:}} t_2}\right) \ .
\end{aligned}$$

Here the function $Arity(t)$, which counts the Cartesian types in the type t, is introduced into PA by:

$$\begin{aligned}
&Arity(t_1{\times}t_2) = Arity(t_2) + 1\\
&Arity(t) = 1 \leftarrow \neg\exists t_1 \exists t_2\, t = t_1{\times}t_2 \ .
\end{aligned}$$

One clause for $\boldsymbol{Dt}$ opens the type definitions:

$$\boldsymbol{Dt}\left(\frac{p_1}{a[r;s] \mathbin{\boldsymbol{:}} c{\circ}u}\right) \leftarrow \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[r;s] \mathbin{\boldsymbol{:}} sb(c,c,u)}\right)$$

1.9.10 Arithmetized bland typing predicate. The four-place predicate $a[r;s] \mathbin{\underline{\boldsymbol{:}}} t$ arithmetizing the relation of derivability in the typing calculus and holding whenever the sequent $a[r;s] \mathbin{\boldsymbol{:}} t$ is derivable is introduced into PA by the following explicit definition:

$$a[r;s] \mathbin{\underline{\boldsymbol{:}}} t \leftrightarrow \exists p \boldsymbol{Dt}\left(\frac{p}{a[r;s] \mathbin{\boldsymbol{:}} t}\right) \ .$$

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

$$x : s{\mapsto}t \leftrightarrow \exists a \exists n \exists m \exists e (x = \boldsymbol{cd}(a, n, m, e) \wedge x[0;0] \,\underline{:}\, s{\mapsto}t) \ . \qquad (1)$$

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, $\ldots$, 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\mathbf{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\mathbf{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 \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\mathbf{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\mathbf{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 (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 y_1=j,y_2\wedge j\geq m\wedge j<L(t)\wedge (t)_j=n,b\wedge\\
& &&\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 \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{m[0;y_1]=y_2}\right)\wedge\\
& &&\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 \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 \boldsymbol{Ct}(p_2)\wedge p_2=\left(\frac{q_2}{m[0;y_1]=y_2}\right)\wedge\\
& &&\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 \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.

$$\begin{aligned}\boldsymbol{Ct}\left(\frac{p_1, p_2, p_3, 0}{(a\underline{\bullet}b)[c;x] = y_3}\right) \qquad \leftarrow\ & \boldsymbol{Ct}(p_1) \land p_1 = \left(\frac{q_1}{a[c;x] = y_1}\right) \land y_1 = \boldsymbol{cd}(d,n,m,e) \land \\ & \boldsymbol{Ct}(p_2) \land p_2 = \left(\frac{q_2}{b[c;x] = y_2}\right) \land \\ & \boldsymbol{Ct}(p_3) \land p_3 = \left(\frac{q_3}{d[y_1;y_2] = y_3}\right)\ .\end{aligned}$$

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

$$(r = 0 \lor r = r_1{\mapsto}r_2 \land c \mathrel{:} r) \land (s = 0 \lor x \mathrel{:} s) \land a[r;s] \mathrel{\underline{:}} t$$
$$\to \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{a[c;x] = y}\right) . \qquad (1)$$

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 $c \mathrel{:} \boldsymbol{N}{\mapsto}\boldsymbol{N}$. Thus $(c\underline{\bullet}\boldsymbol{x}_0)[0;x] \mathrel{\underline{:}} \boldsymbol{N}$ 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:

$$x \mathrel{:} s \land lev(s) = 0 \land \underline{n}_m[0;x] \mathrel{\underline{:}} t \to \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{\underline{n}_m[0;x] = y}\right) .$$

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

$$Wtappl(c,x) \leftrightarrow \exists s \exists t (c \mathrel{:} s{\mapsto}t \land x \mathrel{:} s) .$$

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:

$$c{\bullet}x = y \leftrightarrow Wtappl(c,x) \land \exists p\, \boldsymbol{Ct}\left(\frac{p}{(c\underline{\bullet}\boldsymbol{x_0})[0;x] = y}\right) \lor \neg\, Wtappl(c,x) \land y = 0$$

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:

$A(0) = {}'S$
$A(n+1) = {}'F{\bullet}A(n) \quad .$

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*:

$$ {}'f : s_1{\mapsto}\ldots{\mapsto}s_n{\mapsto}t \tag{1}$$
$$ x_1 : s_1 \wedge \ldots \wedge x_n : s_n \to {}'f{\bullet}x_1{\bullet}\ldots{\bullet}x_n = f(x_1,\ldots,x_n) \quad . \tag{2}$$

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, …, 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

$$\frac{}{\Delta, f :: s_1,\ldots,s_n{\mapsto}t \Rightarrow {}'f : s_1{\mapsto}\ldots{\mapsto}s_n{\mapsto}t} \quad . \tag{3}$$

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):

$$\begin{aligned} \boldsymbol{Dt}\left(\frac{p_1, p_2, p_3, 0}{\boldsymbol{cd}(a, n, m, e)[r; s] \mathrel{\boldsymbol{:}} t}\right) \leftarrow n > 0 \land \mathit{Uncurry}(n, t) = t_1 {\mapsto} t_2 \land & \\ \boldsymbol{Dt}(p_1) \land p_1 = \left(\frac{q_1}{a[t_1{\mapsto}t_2; t_1] \mathrel{\boldsymbol{:}} t_2}\right) \land & \\ \boldsymbol{Dt}(p_2) \land p_2 = \left(\frac{q_2}{m[0; t_1] \mathrel{\boldsymbol{:}} \boldsymbol{N}}\right) \land & \\ \boldsymbol{Dt}(p_3) \land p_3 = \left(\frac{q_3}{e[0; t_1] \mathrel{\boldsymbol{:}} t_2}\right) . & \end{aligned}$$

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 $y_3 = x_1, \dots, x_{n-1}, y_2$. 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 $\mathit{Red}(1, y_1, y_2) = 0$ then we have $y_1 = \boldsymbol{cd}(k, n, m, e)\underline{\bullet}x_1\underline{\bullet}\dots\underline{\bullet}x_i$ for $i + 1 < n$ and the value of $a\underline{\bullet}b$ is the code of the irreducible curried term:

$$y_3 = y_1\underline{\bullet}y_2 = \boldsymbol{cd}(k, n, m, e)\underline{\bullet}x_1\underline{\bullet}\dots\underline{\bullet}x_i\underline{\bullet}y_2 \ .$$

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):

$$(r = 0 \lor r = r_1 {\mapsto} r_2 \land c \mathbin{\boldsymbol{:}} r) \land (s = 0 \lor x \mathbin{\boldsymbol{:}} s) \land a[r; s] \mathbin{\underline{\boldsymbol{:}}}\, t \\ \to \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{a[c; x] = y}\right) \qquad (1)$$

although the formula is satisfied in the standard model of PA.

Only the following special forms are provable in PA:

$$x \mathbin{\boldsymbol{:}} s \land \mathit{lev}(s) = 0 \land \underline{n}_m[0; x] \mathbin{\underline{\boldsymbol{:}}}\, t \to \exists y \exists p \boldsymbol{Ct}\left(\frac{p}{\underline{n}_m[0; x] = y}\right) \ .$$

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

$$\mathit{Wtappl}(c,x) \leftrightarrow \exists s \exists t (c : s {\mapsto} t \land x : s) \ .$$

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:

$$c{\bullet}x = y \leftrightarrow \mathit{Wtappl}(c,x) \land \exists p\, \boldsymbol{Ct}\left(\frac{p}{(c\underline{\bullet}\boldsymbol{x_0})[0;x]=y}\right) \lor \neg\, \mathit{Wtappl}(c,x) \land y = 0 \tag{1}$$

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

$$I^{n+1} :: T^n, T^{n-1}, \dots, T^1, T^0 \mapsto T^0$$

explicitly defined by:

$$\begin{aligned}
I^1(k) &= k \\
I^2(x_1, k) &= x_1 \bullet (k{+}1) \\
I_0^3(x_2, x_1, k) &= x_2^{(k+1)}(x_1) \bullet 1 \\
I_0^{n+4}(x_{n+3}, x_{n+2}, x_{n+1}, \dots, x_1, k) &= x_{n+3}^{(k+1)}(x_{n+2}) \bullet x_{n+1} \bullet \dots \bullet x_1 \bullet k \ .
\end{aligned}$$

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

$$x_n : T^n \wedge \dots x_1 : T^1 \to {}'I^{n+1} \bullet x_n \bullet \dots \bullet x_1 \bullet k = I^{n+1}(x_n, \dots, x_1, k) \ .$$

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 $n = M{-}1, M{-}2, \dots, 0$ a sequence of unary functions in a: $K^{n+1}(a)$ by course of values recursion:

$K^{n+1}(0) = {}'I^{n+1}$
$K^{n+1}(a \sharp \omega^b) = K^{n+2}(b) \bullet K^{n+1}(a) \quad .$

We claim that every function F_α of the fast growing hierarchy where $\alpha \prec \omega_M^1$, 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:

$$\begin{aligned}
K^{n+1}(a \sharp \omega^b \cdot k) &= (K^{n+2}(b))^k)(K^{n+1}(a)) \\
K^1(a \sharp \omega^0 \cdot k) \bullet m &= K^1(a) \bullet (m{+}k)
\end{aligned}$$

and by complete induction on c:

$$\begin{aligned}
&x_{n+1} : T^{n+1} \wedge \dots \wedge x_1 : T^1 \to \\
&\quad K^{n+2}(a \sharp \omega^{b \sharp \omega^c}) \bullet x_{n+1} \bullet \dots \bullet x_1 \bullet k = K^{n+2}((a \sharp \omega^{b \sharp \omega^c})[k]) \bullet x_{n+1} \bullet \dots \bullet x_1 \bullet k \ .
\end{aligned}$$

We then prove by $\prec w^1_M$ induction on a:

$$a \prec w^1_M \rightarrow V(a) = 0 \sharp \omega^0{\cdot}(K^1(a){\bullet}0) \; .$$

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

$$\begin{aligned} &0 : \mathcal{O} \\ &a \sharp \omega^b : \mathcal{O} \leftarrow b : \mathcal{O} \land a : \mathcal{O} \quad . \end{aligned}$$

The reader will note that the type $\mathcal{O}$ is coextensive with the type $\mathit{List}(\mathcal{O})$ and so we have $a \mathbin{\textbf{:}} \mathcal{O}$.

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 $\cdot \sharp \omega^{\cdot} :: \mathcal{O}, \mathcal{O} {\mapsto} \mathcal{O}$, $\cdot \sharp \omega^{\cdot}{\cdot}{\cdot} :: \mathcal{O}, \mathcal{O}, N {\mapsto} \mathcal{O}$, 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 Issues Open to Further Research

UNFINISHED we do not propose solutions and only outline possible explication in PA of issues which must be solved before declarative programming can replace imperative one. The most pressing are the destructive updates.

Destructive Updates

UNFINISHED this is the most important problem to be solved in declarative programming. It is usually tackled as a problem of arrays but it is actually a more general problem of destructive updates of recursive container data structures because the large databases which have to be updated without copying before declarative programs manipulating them become feasible.

1.10.1 Modification functions. UNFINISHED

1.10.2 Dags in PA. The problem of updating trees would be relatively simple were it not the case that during the computation of declarative programs the trees represented in memory become *directed acyclic graphs* (dags).

For instance, the function

$$f(x, y) = x, y, x$$

applied to two mixed numerals as $f(\rho_1, \rho_2)$ constructs a dag: ....

This dag can be represented by the generalized term

let
$\rho_1 = x \rightarrow_x x, \rho_2, x$.

Generalized terms bind variables and for that reason we did not introduce them into PA. There is another possibility of representing dags in PA with explicit 'pointers': $m = T\,T(m), \rho_2, \rho_1$.

1.10.3 Pointers as values. We prever to deal with pointers as values and to that end we introduce a binary 'memory pointing' function $m.\,p$ which is defined by dyadic induction:

$m.\,0 = m$
$(m_1, m_2).\,p\mathbf{1} = m_1.\,p$
$(m_1, m_2).\,p\mathbf{1} = m_1.\,p$
$(m_1, m_2).\,p\mathbf{2} = m_2.\,p$.

Dags which we are interested in will be represented by *dag terms* which are built up from 0 by pairing ρ_1, ρ_2 and by *pointers* $m.\,p$ where p is a dyadic numeral. **UNFINISHED** always exists the unique m which means no cycles. Perhaps comment on constraints when the uniqueness is relaxed.

If $\rho[m]$ is a subterm of the heap $\tau[m]$ s.t. $\exists m\, m = \tau[m]$ then the ternary *dag indexing* function $\rho[p]^\tau$ yields the *dereferenced* subterm (i.e. a subterm which is not a pointer) of τ selected from ρ by the pointer p:

$$\begin{aligned}
\rho[0]^\tau &\equiv \begin{cases} \rho & \text{if } \rho \equiv 0 \text{ or } \rho \equiv \rho_1, \rho_2 \\ \tau[p]^\tau & \text{if } \rho \equiv m.\, p \end{cases} \\
(\rho_1, \rho_2)[p\mathbf{1}]^\tau &\equiv \rho_1[p]^\tau \\
(\rho_1, \rho_2)[p\mathbf{2}]^\tau &\equiv \rho_2[p]^\tau \\
(m.\, r)[p]^\tau &\equiv \tau[r \star p]^\tau \ .
\end{aligned}$$

We clearly have

$$m = \tau[m] \rightarrow m.\, p = \tau[p]^\tau \ .$$

The indexing function is used in the binary *dag denotation* function ρ^τ which yields the denotation of a dag term $\rho[m]$ relatively to the heap τ s.t. $\exists m\, m = \tau[m]$. The denotation function satisfies:

$$\begin{aligned}
0^\tau &= 0 \\
(\rho_1, \rho_2)^\tau &= {\rho_1}^\tau, {\rho_2}^\tau \\
(m.\, p)^\tau &= (\tau[p]^\tau)^\tau \ .
\end{aligned}$$

1.10.4 Dag programs. For the sake of simplicity we do not deal with dyadic numerals and arithmetic. The dag programs we are interested in are obtained by the unfolding of certain regular clausal definitions with measure. The definitions are constructed from variables, 0, pairing, applications of previously defined (by dag programs) functions , and from the generalized pair case terms.

UNFINISHED what about the measures.

1.10.5 Reduction of dag programs. We will reduce terms $\tau[m]$ built up from 0, by pairing, applications of functions, generalized terms, and by pointers. Pointers $m.\, p$ in the terms point to the *heap* (store) m. Note that the terms τ may contain at most the variable m free. The state of the store is given by a dag term $\rho[m]$ such that $\exists m\, m = \rho[m]$.

We will reduce terms $\tau[m]$ in the context of the heap and we can visualize the reduction of the formula:

$$m = \rho[m] \land \tau = y$$

One reduction step is

$$m = \rho[m] \land \tau[m] = y \blacktriangleright m = \rho_1[m] \land \tau_1[m] = y \qquad (1)$$

where the leftmost and outermost redex in τ is replaced by its corresponding contractum whereby we obtain a term τ_1. The reduction may involve the change of the heap whose new state is given by the dag term ρ_1. The reduction terminates with an irreducible numeral τ which is either 0 or a pointer.

The invariant of the reduction step (1) is

$$\exists m(m = \rho[m] \wedge \tau[m] = y) \leftrightarrow \exists m(m = \rho_1[m] \wedge \tau_1[m] = y)$$

which means that although the term τ and the heap ρ change, the denotation y of $\tau_1[m]$ does not.

The initial term τ to be reduced is closed, which means that τ does not contain any pointers $m.p$. The initial formula is $m = 0 \wedge \tau = y$ with the heap m empty. The reduction of τ goes on until the term τ becomes an irreducible term τ_1 which will be either 0 or a pointer. We will have

$$m = 0 \wedge \tau = y \blacktriangleright m = \rho[m] \wedge \tau_1[m] = y$$

for some dga term ρ representing the heap and so

$$\tau = y \leftrightarrow \exists m(m = 0 \wedge \tau = y) \leftrightarrow \exists m(m = \rho[m] \wedge \tau_1[m] = y) \ .$$

1.10.6 Redexes. cons and applications. the role of the stack.

1.10.7 Comment on operational versus denotational. bisim.

Conservative Second-Order Calculus of Extensions

UNFINISHED Universal quantification: for all extensions such and such ϕ holds. Existential quantificaton: for specification: there exists an extension such that ϕ holds.

1.10.8 Universal second order quantification: $\forall_2 I$. $\forall P(\phi_1[P] \rightarrow \phi_2[P]$ is proved if by extending PA with P and a new axiom $\phi_1[P]$ we can prove $\phi_2[P]$.

The 'schema' of induction:

$$\forall P \forall p(P(0,p) \wedge \forall x(P(x,p) \rightarrow P(x',p)) \rightarrow \forall x P(x,p)) \ .$$

is not a good example because it is not provable.

1.10.9 Universal instantiation: $\forall_2 E$. UNFINISHED with a term or formulas with possibly empty parameters.

1.10.10 Existential second order quantification: $\exists_2 I$. Specification:

$$\exists f \forall x\, f(x)^2 \leq x < (f(x)+1)$$

is proved by (possibly empty extension of PA so a term $\tau[x]$ is in the language and proving

$$\tau[x]^2 \leq x < (\tau[x]+1) \ .$$

Note: Most often we will have $\tau[x] \equiv f(x)$.

1.10.11 Existence property: $\exists_2 E$. UNFINISHED inversion

1.10.12 Combined second order quantification.

$$\forall g \forall h \exists f \forall p \forall x (f(0) = g(p) \land f(x', p) = h(x, f(x,p), p))$$

Minimalization function:

$$\forall P (\forall p \exists y P(y,p) \to \exists f \forall p\, f(p) = \mu_y[P(y,p)])$$

where $f(\vec{x}) = \mu_y[P(y,\vec{x})]$ abbreviates

$$\forall y (f(\vec{x}) = y \leftrightarrow P(y,\vec{x}) \land \forall z (z < y \to \neg P(z,\vec{x})))$$

1.10.13 Conservativity. Every first-order property proved by a detour through a second order one is provable in PA.

1.10.14 Tree of extensions. UNFINISHED
UNFINISHED existential instantiation of functions by proofs as a generalization of extraction of programs.

Extraction of Programs from Proofs

UNFINISHED

Abstract Data Types

1.10.15 Abstraction Types. UNFINISHED

1.10.16 Representants. UNFINISHED Let $\sim$ be an equivalence over T:

$$\begin{gathered} x \sim y \to T(x) \land T(y) \\ T(x) \to x \sim x \\ x \sim y \to y \sim x \\ x \sim y \land y \sim z \to x \sim z \ . \end{gathered}$$

We define a predicate provably recursive in $\sim$:

$$\mathit{Isr}(x) \leftrightarrow \forall y (y \sim x \to y \geq x)$$

and a function provably recursive in $\sim$:

$r(0) = 0$
$r(x+1) = y + 1 \leftarrow \mathit{Isr}(x+1) \land r(x) = y \land \mathit{Isr}(y)$
$r(x+1) = y \leftarrow \mathit{Isr}(x+1) \land r(x) = y \land \neg \mathit{Isr}(y)$
$r(x+1) = r(x) \leftarrow \neg \mathit{Isr}(x+1)$.

If T is infinite, i.e. if

$$\exists y(y \geq x \land T(y))$$

then

$$x \sim y \to r(x) = r(y)$$
$$T(x) \land T(y) \land r(x) = r(y) \to x \sim y$$
$$\exists y(x = r(y) \land T(y)) \ .$$

1.10.17 Finite Sequences. Sequences and three specializations: lists, arrays, queues, first typeless, then typed.

1.10.18 Finite Sets. Language: $\emptyset$ empty set, $s \cup \{a\}$ insert into set, ch choice function, $|s|_{\boldsymbol{s}}$ size of the set, Set predicate of being a set. SET is abbreviation for the formulas of the following groups:

Properties of ch, $\emptyset$, and Set:

$$ch(\emptyset) = 0 \ .$$

Clausal property of set size (there is no measure which goes down):

$|s|_{\boldsymbol{s}} = 0 \leftarrow ch(s) = 0$
$|s|_{\boldsymbol{s}} = |t|_{\boldsymbol{s}} + 1 \leftarrow ch(s) = x, t \ .$

Clausal definition of predicate $x : Set$ ($|s|_{\boldsymbol{s}}$ is the measure):

$\emptyset : Set$
$s : Set \leftarrow ch(s) = y, t \land t : Set \ .$

Clausal definition of set membership ($|s|_{\boldsymbol{s}}$ is the measure):

$x \in s \leftarrow ch(s) = y, t \land (x = y \lor x \in t) \ .$

Extensionality:

$$s : Set \land t : Set \to (s = t \leftrightarrow \forall x(x \in s \leftrightarrow x \in t)) \ .$$

Properties of insert:

$$s : Set \to s \cup \{x\} : Set$$
$$s \cup \{x\} : Set \to (y \in s \cup \{x\} \leftrightarrow y = x \lor y \in s) \ .$$

1.10.19 Typing of sets. Every implementation of SET can be typed for every type T by defining:

$$s : Set(T) \leftrightarrow s : Set \land \forall x(x \in s \to x : T) \ .$$

We can then prove the typing properties:

$$\emptyset : Set(T)$$
$$s : Set(T) \land ch(s) = x, t \to x : T \land t : Set(T)$$
$$x : T \land s : Set(T) \to s \cup \{x\} : Set(T) \ .$$

1.10.20 Finite Sets II. Language: $\emptyset$ empty set, $s \cup \{a\}$ insert into set, ch choice function, Axioms: Properties of ch:

$$\begin{gathered} ch(\emptyset) = 0 \\ ch(s) = x, t \to t < s \ . \end{gathered}$$

Clausal definition of set membership

$$x \in s \leftarrow ch(s) = y, t \land (x = y \lor x \in t) \ .$$

Extensionality:

$$s = t \leftrightarrow \forall x (x \in s \leftrightarrow x \in t) \ .$$

Basic property of insert:

$$y \in s \cup \{x\} \leftrightarrow y = x \lor y \in s \ .$$

UNFINISHED three implementations, bitmap, ordered list, binary search tree, Note that membership can be faster than through choice.

1.10.21 Abstract binary trees. Language: $\emptyset$, $Nd(n, s, t)$. Axioms:

$$x = \emptyset \lor \exists n \exists s \exists t \, x = Nd(n, s, t) \tag{1}$$

$$\emptyset \neq Nd(n, s, t) \tag{2}$$

$$Nd(n_1, s_1, t_1) = Nd(n_2, s_2, t_2) \to n_1 = n_2 \land s_1 = s_2 \land t_1 = t_2 \tag{3}$$

$$s < Nd(n, s, t) \land t < Nd(n, s, t) \ . \tag{4}$$

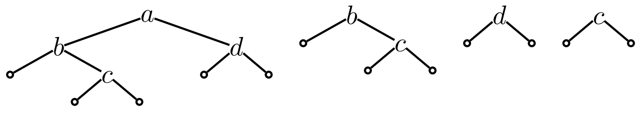
Optimization by Computer-Aided Transformations

Some ten years ago there was a considerable research into the techniques of computer-assisted program transformations in order to optime programs. This was before the semantics of programming languages was sufficiently simplified so the languages could be designed together with formal proof systems. The following problem shows how a rather deep knowledge of properties of underlying data structures is required before programs can be transformed in computer assisted way.

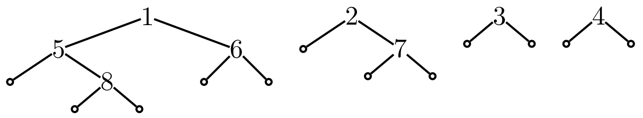
1.10.22 Breadth-first Numbering. We present a so called *functional programming pearl* posed by John Launchberry and discussed by Chris Okasaki in [19]:

> Given a tree, create a new tree of the same shape, but with the values at the nodes replaced by the numbers 1, 2,. . . in breadth-first order.

Okasaki's solution in the programming language ML can be formulated for our binary trees (see Par. 1.3.20) by designing an auxiliary function $B(n, a) = b$ taking a number n and a forest a of type $List\,Bt(t)$ into the forest b of type $List\,Bt(N)$, such that $L(a) = L(b)$, the trees of b are of the same shape as the corresponding trees in a, and the entire forest is numbered starting with n in a breadth-first order. For instance, if a is the forest:



Then $b = B(1, a)$ should be the forest:



The function B looks at its second argument as a queue and yields and works with the output forest as a backward queue to which one adds in the front and removes from the end:

$$\begin{aligned}
&B(n, 0) = 0\\
&B(n, [E] \oplus a) = [E] \oplus B(n, a)\\
&B(n, [Nd(x, t, s)] \oplus a) = [Nd(n, t_1, s_1)] \oplus a_1 \leftarrow\\
&\qquad B(n + 1, a \oplus [t] \oplus [s]) = a_1 \oplus [t_1] \oplus [s_1] \ .
\end{aligned}$$

In the third clause of B, the subtrees t and s are placed at the end of the argument queue and after the recursive application $B(n+1, a \oplus [t] \oplus [s]) = b$ they will be correctly numbered as t_1 and s_1 respectively in the output forest b which is destructed from the end as $b = a_1 \oplus [t_1] \oplus [s_1]$. The forest a_1 is now correctly numbered to follow the tree $Nd(n, t_1, s_1)$ in the forest yielded by $B(n, [Nd(x, t, s)] \oplus a)$.

The clausal definition of $B(n, a)$ is regular in the measure $\sum_{s \varepsilon a} 2{\cdot}|t|_t + 1$ where $|s|_t$ is the *size* of the tree s clausally defined as:

$$\begin{aligned}
&|E|_t = 0\\
&|Nd(x, s_1, s_2)|_t = |s_1|_t + |s_2|_t + 1 \ .
\end{aligned}$$

This is because **UNFINISHED**

The function $Bf(t)$ breadth-first numbering the tree t is now explicitly defined as:

$$Bf(t) = t_1 \leftarrow B(1, [t]) = [t_1]$$

because the breadth-first numbering of t is the same as the breadth-first numbering of the single element forest $[t]$.

1.10.23 Turning the backwards queue into a normal one. Okasaki notes that the output backwards queue yielded by B can be turned into an ordinary queue in a function $B_1(n, a)$ yielding the same forest as $B(n, a)$ but in the reversed order. We can view the problem of designing the function B_1 as a program transformation problem where we are looking for a recursive clausal definition of the function B_1 given its explicit 'definition'

$$B_1(n, a) = \mathit{Rev} B(n, a) \tag{1}$$

which should be viewed as the desired property of B_1 rather than as a definition. The clausal definition of B_1 can be semi-mechanically obtained from the clasuses for B when we note that we have $a = b \leftrightarrow \mathit{Rev}(a) = \mathit{Rev}(b)$. We accordingly modify the clauses of B to obtain:

$$\begin{aligned}
&\mathit{Rev}\, B(n, 0) = \mathit{Rev}(0)\\
&\mathit{Rev}\, B(n, [E] \oplus a) = \mathit{Rev}(E \oplus B(n, a))\\
&\mathit{Rev}\, B(n, [\mathit{Nd}(x, t, s)] \oplus a) = \mathit{Rev}([\mathit{Nd}(n, t_1, s_1)] \oplus a_1) \leftarrow\\
&\quad \mathit{Rev}\, B(n + 1, a \oplus [t] \oplus [s]) = \mathit{Rev}(a_1 \oplus [t_1] \oplus [s_1])\ .
\end{aligned}$$

We now use the obvious properties

$$\mathit{Rev}(0) = 0 \tag{2}$$
$$\mathit{Rev}([a]) = [a] \tag{3}$$
$$\mathit{Rev}(a \oplus b) = \mathit{Rev}(b) + \mathit{Rev}(a) \tag{4}$$

of the list reversal function Rev and obtain:

$$\begin{aligned}
&\mathit{Rev}\, B(n, 0) = 0\\
&\mathit{Rev}\, B(n, [E] \oplus a) = \mathit{Rev}\, B(n, a) \oplus [E]\\
&\mathit{Rev}\, B(n, [\mathit{Nd}(x, t, s)] \oplus a) = \mathit{Rev}(a_1) \oplus [\mathit{Nd}(n, t_1, s_1)] \leftarrow\\
&\quad \mathit{Rev}\, B(n + 1, a \oplus [t] \oplus [s]) = [s_1] \oplus [t_1] \oplus \mathit{Rev}(a_1)\ .
\end{aligned}$$

As the final step, we replace the auxiliary result $\mathit{Rev}(a_1)$ by a_1 and use the property (1) to get the desired clausal definition of the function B_1:

$$\begin{aligned}
&B_1(n, 0) = 0\\
&B_1(n, [E] \oplus a) = B_1(n, a) \oplus [E]\\
&B_1(n, [\mathit{Nd}(x, t, s)] \oplus a) = a_1 \oplus [\mathit{Nd}(n, t_1, s_1)] \leftarrow\\
&\quad B_1(n + 1, a \oplus [t] \oplus [s]) = [s_1] \oplus [t_1] \oplus a_1\ .
\end{aligned}$$

Both input and output queues are now standard where we add at the end and remove from the front. The new breadth-first numbering function Bf_1 has an explicit definition:

$$\mathit{Bf}_1(t) = t_1 \leftarrow B_1(1, [t]) = [t_1]\ .$$

1.10.24 Speeding up the en-queuing. We note that our function $B_1(t)$ operates in time $\mathcal{O}(|t|_s^2)$ because of the naive implementation of the en-queuing operation $q \oplus [x]$ by concatenation. Okasaki improves this naive solution by using a fast implementation of the ADT queue. One such implementation represents a queue $q = a \oplus Rev(b)$ as a pair of lists (a, b). The en-queuing and de-queing functions have the following definitions:

$$\begin{aligned}
&Enq(x,(a,b)) = a, x, b\\
&Deq(0,0) = 0\\
&Deq(0,b) = x, a, 0 \leftarrow b > 0 \wedge Rev(b) = [x] \oplus a\\
&Deq((x,a),b) = x, a, b\ .
\end{aligned}$$

The reader will note that the de-queuing function yields a pair consisting of the removed element and the new shortened queue provided that the queue is not empty. For empty queues the function yields 0.

Instead of modifying B_1 for the ADT queue, we transform it into a ternary function $B_2(n, a, b)$ where (a, b) represents the input queue $a \oplus Rev(b)$ and B_2 yields the pair (b_1, a_1) representing the output queue $b_1 \oplus Rev(a_1)$. In other words, we are looking for a recursive clausal definition of the function B_2 satisfying:

$$B_2(n,a,b) = b_1, a_1 \leftrightarrow B_1(n, a \oplus Rev(b)) = b_1 \oplus Rev(a_1) \wedge L(a) = L(a_1)\ . \tag{1}$$

Towards that end we rewrite the clauses of B_1 in an equivalent homogeneous form:

$$\begin{aligned}
&B_1(n,a) = 0 \leftarrow a = 0\\
&B_1(n,a) = b \oplus [E] \leftarrow a = [E] \oplus a_1 \wedge B_1(n,a_1) = b\\
&B_1(n,a) = a_2 \oplus [Nd(n,t_1,s_1)] \leftarrow a = [Nd(x,t,s)] \oplus a_1 \wedge\\
&\qquad B_1(n+1, a_1 \oplus [t] \oplus [s]) = [s_1] \oplus [t_1] \oplus a_2\ .
\end{aligned}$$

Substituting $a_0 \oplus Rev$ for a yields:

$$B_1(n, a_0 \oplus Rev(b)) = 0 \leftarrow a_0 \oplus Rev(b) = 0 \tag{2}$$
$$B_1(n, a_0 \oplus Rev(b)) = c \oplus [E] \leftarrow a_0 \oplus Rev(b) = [E] \oplus a_1 \wedge B_1(n,a_1) = c \tag{3}$$
$$\begin{aligned}
&B_1(n, a_0 \oplus Rev(b)) = a_2 \oplus [Nd(n,t_1,s_1)] \leftarrow a_0 \oplus Rev(b) = [Nd(x,t,s)] \oplus a_1 \wedge\\
&\qquad B_1(n+1, a_1 \oplus [t] \oplus [s]) = [s_1] \oplus [t_1] \oplus a_2\ .
\end{aligned}\tag{4}$$

Clause (2) can be equivalently written as

$$B_1(n, 0 \oplus Rev(0)) = 0 \oplus Rev(0)$$

and then by property (1) as

$$B_2(n,0,0) = 0, 0\ . \tag{5}$$

We will now transform the clause (4) under the assumption that $a_0 > 0$, i.e. $a_0 = [x] \oplus a$ for some x and a. We then have

$$a_0 \oplus Rev(b) = [Nd(x,t,s)] \oplus a_1 \leftrightarrow x = Nd(x,t,s) \wedge a \oplus Rev(b) = a_1$$

and so we can simplify the clause (4) to

$$\begin{aligned}&B_1(n,([Nd(x,t,s)]\oplus a)\oplus Rev(b)) = a_2\oplus[Nd(n,t_1,s_1)] \leftarrow\\ &\quad B_1(n+1, a\oplus Rev(b)\oplus[t]\oplus[s]) = [s_1]\oplus[t_1]\oplus a_2 \quad .\end{aligned}$$

Since

$$a\oplus Rev(b)\oplus[t]\oplus[s] = a\oplus Rev([s]\oplus[t]\oplus b) \ ,$$

we further get

$$\begin{aligned}&B_1(n,([Nd(x,t,s)]\oplus a)\oplus Rev(b)) = a_2\oplus[Nd(n,t_1,s_1)] \leftarrow\\ &\quad B_1(n+1, a\oplus Rev([s]\oplus[t]\oplus b)) = [s_1]\oplus[t_1]\oplus a_2 \quad .\end{aligned}$$

We always have $a_2 = b_1\oplus Rev(a_1)$ for some b_1 and a_1 and, since

$$L(a_2){+}2 = L\,B_1(n{+}1, a{\oplus}Rev([s]{\oplus}[t]{\oplus}b)) = L(a{\oplus}Rev([s]{\oplus}[t]{\oplus}b)) = L(a){+}L(b){+}2 \ ,$$

we can also assume that $L(a_1) = L(a)$ (and $L(b_1) = L(b)$). The last clause can be then written as

$$\begin{aligned}&B_1(n,([Nd(x,t,s)]\oplus a)\oplus Rev(b)) = (b_1\oplus Rev(a_1))\oplus[Nd(n,t_1,s_1)] \leftarrow\\ &\quad B_1(n+1, a\oplus Rev([s]\oplus[t]\oplus b)) = [s_1]\oplus[t_1]\oplus(b_1\oplus Rev(a_1))\wedge L(a) = L(a_1)\end{aligned}$$

which simplifies to:

$$\begin{aligned}&B_1(n,([Nd(x,t,s)]\oplus a)\oplus Rev(b)) = b_1\oplus Rev([Nd(n,t_1,s_1)]\oplus a_1) \leftarrow\\ &\quad B_1(n+1, a\oplus Rev([s]\oplus[t]\oplus b)) = ([s_1]\oplus[t_1]\oplus b_1)\oplus Rev(a_1)\wedge L(a) = L(a_1) \quad .\end{aligned}$$

Using the property (1) we get from this a clause for B_2:

$$\begin{aligned}&B_2(n,[Nd(x,t,s)]\oplus a, b) = b_1, [Nd(n,t_1,s_1)]\oplus a_1 \leftarrow\\ &\quad B_2(n+1, a, [s]\oplus[t]\oplus b) = [s_1]\oplus[t_1]\oplus b_1, a_1 \ .\end{aligned} \tag{6}$$

By similar transformations we obtain under the assumption $a_0 > 0$ from the clause (3) the clause

$$B_2(n,[E]\oplus a,b) = b_1,[E]\oplus a_1 \leftarrow B_2(n,a,b) = b_1,a_1 \ . \tag{7}$$

We now transform the clauses (3) and (4) under the assumption that $a_0 = 0$. The clauses simplify to:

$$\begin{aligned}&B_1(n,0\oplus Rev(b)) = c\oplus[E] \leftarrow Rev(b) = [E]\oplus a_1 \wedge B_1(n,a_1) = c\\ &B_1(n,0\oplus Rev(b)) = a_2\oplus[Nd(n,t_1,s_1)] \leftarrow Rev(b) = [Nd(x,t,s)]\oplus a_1 \wedge\\ &\quad B_1(n+1, a_1\oplus[t]\oplus[s]) = [s_1]\oplus[t_1]\oplus a_2 \ .\end{aligned}$$

Both clauses have in their antecedents $Rev(b)\neq 0$ from which we get $b\neq 0$ and we can collapse both clauses into one ‘trivial’ clause:

$$B_1(n,0\oplus Rev(b)) = c \leftarrow b\neq 0\wedge B_1(n,Rev(b)) = c \ .$$

We substitute $c := Rev(b_1)$ to get:

$$B_1(n,0\oplus Rev(b)) = Rev(b_1) \leftarrow b\neq 0\wedge B_1(n,Rev(b)) = Rev(b_1) \ .$$

We have

$$L(b) = L\,Rev(b) = L\,B_1(n,Rev(b)) = L\,Rev(b_1) = L(b_1)$$

and so we can equivalently write the clause as:

$$\begin{aligned}&B_1(n, 0 \oplus \mathit{Rev}(b)) = \mathit{Rev}(b_1) \oplus \mathit{Rev}(0) \leftarrow b \neq 0 \wedge \\ &\quad B_1(n, \mathit{Rev}(b) \oplus \mathit{Rev}(0)) = 0 \oplus \mathit{Rev}(b_1) \wedge L\,\mathit{Rev}(b) = L\,\mathit{Rev}(b_1)\ .\end{aligned}$$

Using the property (1) we get from this a clause for B_2:

$$B_2(n, 0, b) = \mathit{Rev}(b_1), 0 \leftarrow b \neq 0 \wedge B_2(n, \mathit{Rev}(b), 0) = 0, b_1\ . \tag{8}$$

The clauses (5), (8), (7), and (6) constitute a clausal definition for the function B_2 which we write with the list notation x, y instead of the identical sequence notation $[x] \oplus y$ as follows:

$$\begin{aligned}&B_2(n, 0, 0) = 0, 0 \\ &B_2(n, 0, b) = \mathit{Rev}(b_1), 0 \leftarrow b \neq 0 \wedge B_1(n, \mathit{Rev}(b), 0) = 0, b_1 \\ &B_2(n, (E, a), b) = b_1, E, a_1 \leftarrow B_2(n, a, b) = b_1, a_1 \\ &B_2(n, (\mathit{Nd}(x, t, s), a), b) = b_1, \mathit{Nd}(n, t_1, s_1), a_1 \leftarrow \\ &\quad B_2(n + 1, a, s, t, b) = (s_1, t_1, b_1), a_1\ .\end{aligned}$$

It remains to explicitly define the function Bf_2 numbering one tree as

$$\mathit{Bf}_2(t) = t_1 \leftarrow B_2(1, (t, 0), 0) = (t_1, b), a\ .$$

The function $\mathit{Bf}_2(s)$ performs in time $\mathcal{O}(|s|_t)$ because the second clause for B_2 is taken once for each level of the tree s with the corresponding forest of children of t as b. This means that the function Rev performs total $2{\cdot}|s|_t$ applications and the maximal length.

is probably one the fastest functional algorithms for the breadth 3 times through.

Concurrency

Part I

First-Order Logic and Peano Arithmetic

2. First-Order Languages

2.1 Language of First-Order Logic

2.1.1 First order languages. A *first-order language* $\mathcal{L}$ is given by at most countable sets of *function symbols* and *predicate symols*. Every function and predicate symbol has *arity* $n \geq 0$ which is the number of arguments the symbol expects. We require that there are effective procedures to decide whether p is an n-ary function or predicate symbol of $\mathcal{L}$.

Expressions in $\mathcal{L}$ will be *terms* and *formulas* and they will be finite sequences of symbols. Expressions will be *metamathematical* objects in contrast to mathemathical objects such as natural numbers, real numbers, topological spaces, etc. The expressions of $\mathcal{L}$ will have have no meaning by themselves. They will be defined in such a way that it will be effectively decidable whether an expression is correctly formed.

We write $\tau_1 \equiv \tau_2$ to mean that the expressions τ_1 and τ_2 are identical sequences of symbols.

In these notes we will deal exclusively with first order languages logic so from now on we will use the term *language* to denote a first order language.

2.1.2 Definition of terms of $\mathcal{L}$. The set of *terms* of a language $\mathcal{L}$ is the smallest set of finite sequences of symbols satisfying:

1. variables v_0, v_1, v_2, ... are terms,
2. if τ_1, ..., τ_n are terms and f is an n-ary function symbol of $\mathcal{L}$ then also $f(\tau_1, \ldots, \tau_n)$ is a term called *function application.*

Nullary function symbols are *constant* symbols and their applications $f()$, which will be abbrevaited to f, are *constants.*

We will use possibly subscripted symbols x, y, z as *syntactic* (meta) variables ranging over variables, symbols f, g as syntactic variables ranging over function symbols, and τ, ρ as syntactic variables ranging over terms.

Some binary function symbols, such as $+$, are customarily written in the *infix* form $\tau_1 + \tau_2$ as an abbreviation for $+(\tau_1, \tau_2)$.

2.1.3 Definition of formulas of $\mathcal{L}$. The set of *formulas* of a language $\mathcal{L}$ is the smallest set of finite sequences of symbols satisfying:

1. if τ_1 and τ_2 are terms of $\mathcal{L}$ then $\tau_1 = \tau_2$ is a formula called *identity*,
2. if $\tau_1, \ldots, \tau_n$ are terms and P is an n-ary predicate symbol of $\mathcal{L}$ then $P(\tau_1, \ldots, \tau_n)$ is a formula called *predicate application*,
3. the symbol $\top$, called the *truth symbol*, is a formula,
4. the symbol $\bot$, called *falsehood symbol*, is a formula,
5. if ϕ is a formula so is $\neg\phi$ called *negation*,
6. if ϕ_1 and ϕ_2 are formulas so is $(\phi_1 \lor \phi_2)$, (called *disjunction*), $(\phi_1 \land \phi_2)$ (called *conjunction*), $(\phi_1 \rightarrow \phi_2)$ (called *implication*), and $(\phi_1 \leftrightarrow \phi_2)$ (called *equivalence*),
7. if ϕ is a formula and x a variable then so are $\forall x\phi$ (called *universal quantification*) and $\exists x\phi$ (called *existential quantification*).

Formulas formed by the rules (1) and (2) are *atomic formulas.* Formulas formed by the rules (3) though (6) are *propositional* formulas. Formulas formed by the rule (7) are *quantifier* formulas.

The formula ϕ_1 in the implication $(\phi_1 \rightarrow \phi_2)$ is called *antecedent* and ϕ_2 *consequent.* The symbols used in propositional formulas are *propositional connectives.* Nullary predicate symbols are *propositional constants* and their applications $P()$, which will be abbreviated to P, are *propositional constants.*

We use the possibly subscripted symbols ϕ, ψ as syntactic variables ranging over formulas.

In order to increase the readability of formulas we can drop the topmost pair of parentheses around a formula. All binary propositional connectives associate to the right, for instance $\phi_1 \rightarrow \phi_2 \rightarrow \phi_3$ abbreviates $\phi_1 \rightarrow (\phi_2 \rightarrow \phi_3)$. Negations and quantification have larger precedence (bind stronger) than conjunctions, which have larger precedence than disjunctions, which have larger precedence than implications and equivalences. We often abbreviate $\neg\tau_1 = \tau_2$ to $\tau_1 \neq \tau_2$.

2.1.4 Propositional atoms. Atomic and quantified formulas of $\mathcal{L}$ are called the *propositional atoms* of $\mathcal{L}$. The metamathematical function $FPA(\alpha)$ is defined to yield the set of *free* propositional atoms of α, i.e. propositional atoms of α outside of quantifiers. The function is defined for formulas $\alpha \equiv \phi$ and sets of formulas $\alpha \equiv T$ to satisfy the identities given in Fig. 2.1.

2.1.5 Extension of languages. The language $\mathcal{L}_1$ is an *extension* of the language $\mathcal{L}$ if every function and predicate symbol of $\mathcal{L}$ is a symbol of $\mathcal{L}_1$. It should be clear that every term or formula of $\mathcal{L}$ is a term or formula of $\mathcal{L}_1$. Note that every $\mathcal{L}$ is an extension of itself.

2.1.6 Metamathematical implication and equivalence. In order to shorten the metamathematical discursion in English we will use the symbol $\Rightarrow$ in context $\cdots \Rightarrow \cdots$ as abbreviation for *if* $\cdots$ *then* $\cdots$ and the symbol $\Leftrightarrow$ in context $\cdots \Leftrightarrow \cdots$ as abbreviation for $\cdots$ *if and only if* $\cdots$. Sometimes we will write the last also as $\cdots$ *iff* $\cdots$. We use the metamathematical sym-

$$
\begin{aligned}
FPA(\tau_1 = \tau_2) &= \{\tau_1 = \tau_2\} \\
FPA(P(\tau_1, \ldots, \tau_n)) &= \{P(\tau_1, \ldots, \tau_n)\} \\
FPA(\forall x \phi) &= \{\forall x \phi\} \\
FPA(\exists x \phi) &= \{\exists x \phi\} \\
FPA(\top) &= \emptyset \\
FPA(\bot) &= \emptyset \\
FPA(\neg \phi) &= FPA(\phi) \\
FPA(\phi_1 \vee \phi_2) &= FPA(\phi_1) \cup FPA(\phi_2) \\
FPA(\phi_1 \wedge \phi_2) &= FPA(\phi_1) \cup FPA(\phi_2) \\
FPA(\phi_1 \rightarrow \phi_2) &= FPA(\phi_1) \cup FPA(\phi_2) \\
FPA(\phi_1 \leftrightarrow \phi_2) &= FPA(\phi_1) \cup FPA(\phi_2) \\
FPA(T) &= \bigcup \{FPA(\phi) \mid \phi \in T\} \ .
\end{aligned}
$$

Fig. 2.1. Function yielding sets of propositional atoms

bols $\Rightarrow$ and $\Leftrightarrow$ in order to distinguish them from the logical symbols $\rightarrow$ and $\leftrightarrow$ wchich are from the *object*, i.e. first order, language.

3. Propositional Logic

3.1 Tautologies

In this section we investigate formulas which are always true only on the strength of their propositional connectives.

3.1.1 Propositional interpretations. A *propositional interpretation* $\mathcal{I}$ for $\mathcal{L}$ is a subset of propositional atoms of $\mathcal{L}$. $\mathcal{I}$ is *finite* or *infinite* if the set $\mathcal{I}$ is finite or infinite. The intention is that a propositional atom ϕ is true in the propositional interpretation $\mathcal{I}$ iff $\phi \in \mathcal{I}$.

We denote by $\mathcal{I}^T$ the *restriction of* $\mathcal{I}$ *to the free propositional atoms of the set* T: $\mathcal{I}^T = \mathcal{I} \cap FPA(T)$.

3.1.2 Satisfaction relation for propositional interpretations. For a given propositional interpretation $\mathcal{I}$ for $\mathcal{L}$ and a formula ϕ we define the unary relation $\mathcal{I}$ *satisfies* ϕ, written as $\mathcal{I} \vDash \phi$, as shown in Fig. 3.1 where ϕ, ϕ_1 and ϕ_2 are arbitrary formulas and ψ are propositional atoms of $\mathcal{L}$:

$$\begin{aligned}
\mathcal{I} \vDash \psi &\Leftrightarrow \psi \in \mathcal{I} \\
\mathcal{I} &\vDash \top \\
\mathcal{I} &\nvDash \bot \\
\mathcal{I} \vDash \neg\phi &\Leftrightarrow \mathcal{I} \nvDash \phi \\
\mathcal{I} \vDash \phi_1 \lor \phi_2 &\Leftrightarrow \mathcal{I} \vDash \phi_1 \text{ or } \mathcal{I} \vDash \phi_2 \\
\mathcal{I} \vDash \phi_1 \land \phi_2 &\Leftrightarrow \mathcal{I} \vDash \phi_1 \text{ and } \mathcal{I} \vDash \phi_2 \\
\mathcal{I} \vDash \phi_1 \to \phi_2 &\Leftrightarrow \mathcal{I} \nvDash \phi_1 \text{ or } \mathcal{I} \vDash \phi_2 \\
\mathcal{I} \vDash \phi_1 \leftrightarrow \phi_2 &\Leftrightarrow \mathcal{I} \vDash \phi_1 \to \phi_2 \text{ and } \mathcal{I} \vDash \phi_2 \to \phi_1 \ .
\end{aligned}$$

Fig. 3.1. Propositional satisfaction relation

Note that the propositional atoms obtain meaning directly from $\mathcal{I}$ whereas the meaning of other kinds of propositional formulas is uniquely determined by the meaning of its subformulas.

We write $\mathcal{I} \models T$ as an abbreviation for $\mathcal{I} \models \phi$ *for all* $\phi \in T$ and say that T *is satisfied in* $\mathcal{I}$.

Two (propositional) interpretations $\mathcal{I}$ and $\mathcal{J}$ for $\mathcal{L}$ are (elementarily) T-*equivalent*, in writing $\mathcal{I} \equiv_T \mathcal{J}$, if

$$\mathcal{I} \models T \Leftrightarrow \mathcal{J} \models T \ .$$

$\mathcal{I}$ and $\mathcal{J}$ are (elementarily) *equivalent*, in writing $\mathcal{I} \equiv \mathcal{J}$, if they are $\{\phi \mid \phi \in \mathcal{L}\}$-equivalent, i.e. $\mathcal{I} \models \phi \Leftrightarrow \mathcal{J} \models \phi$ holds for all formulas ϕ.

3.1.3 Sequents. Fix a language $\mathcal{L}$. For a finite sequence of formulas $\mathit{\Lambda}$: ϕ_1, $\ldots$, ϕ_n we write $\bigwedge \mathit{\Lambda}$ as an abbreviation for the conjunction $\phi_1 \wedge \cdots \wedge \phi_n$ if $n \geq 1$ and for $\top$ if $n = 0$. For any propositional interpretation $\mathcal{I}$ we clearly have $\mathcal{I} \models \bigwedge \mathit{\Lambda}$ iff $\mathcal{I} \models \phi$ for all $\phi \in \mathit{\Lambda}$. Note that this holds also for $\mathit{\Lambda} \equiv \emptyset$ because $\bigwedge \emptyset \equiv \top$ for which $\mathcal{I} \models \top$ holds just as $\mathcal{I} \models \phi$ vacuously holds for all $\phi \in \emptyset$.

For the same sequence $\mathit{\Lambda}$ we write $\bigvee \mathit{\Lambda}$ as an abbreviation for the disjunction $\phi_1 \vee \cdots \vee \phi_n$ if $n \geq 1$ and for $\perp$ if $n = 0$. For any propositional interpretation $\mathcal{I}$ we clearly have $\mathcal{I} \models \bigvee \mathit{\Lambda}$ iff $\mathcal{I} \models \phi$ for some $\phi \in \mathit{\Lambda}$.

For two finite sequences of formulas $\mathit{\Lambda}$ and $\mathit{\Gamma}$ we call the formula $\bigwedge \mathit{\Lambda} \rightarrow \bigvee \mathit{\Gamma}$ a *sequent*. For any propositional interpretation $\mathcal{I}$ we clearly have $\mathcal{I} \models \bigwedge \mathit{\Lambda} \rightarrow \bigvee \mathit{\Gamma}$ iff $\mathcal{I} \not\models \phi$ for some $\phi \in \mathit{\Lambda}$ or $\mathcal{I} \models \phi$ for some $\phi \in \mathit{\Gamma}$. Thus $\mathcal{I} \not\models \bigwedge \mathit{\Lambda} \rightarrow \bigvee \mathit{\Gamma}$ iff $\mathcal{I} \models \phi$ for all $\phi \in \mathit{\Lambda}$ and $\mathcal{I} \not\models \phi$ for all $\phi \in \mathit{\Gamma}$. We have $\mathcal{I} \not\models \bigwedge \emptyset \rightarrow \bigvee \emptyset$ because this stands for $\mathcal{I} \not\models \top \rightarrow \perp$.

3.1.4 Tautologies. A formula ϕ of $\mathcal{L}$ is a *tautology*, in symbols $\models_p \phi$, if $\mathcal{I} \models \phi$ holds for all propositional interpretations $\mathcal{I}$ for $\mathcal{L}$.

We say that two formulas ϕ_1 and ϕ_2 are *propositionally equivalent* if we have $\mathcal{I} \models \phi_1 \Leftrightarrow \mathcal{I} \models \phi_2$ for all propositional interpretations $\mathcal{I}$. But this means that $\mathcal{I} \models \phi_1 \leftrightarrow \phi_2$ holds for all propositional interpretations $\mathcal{I}$ and so $\phi_1 \leftrightarrow \phi_2$ is a tautology. Note that we then also have $\models_p \phi_1$ iff $\models_p \phi_2$.

Because $\mathcal{I} \models \phi_1 \wedge \phi_2$ holds iff $\mathcal{I} \models \phi_1$ and $\mathcal{I} \models \phi_2$ hold we have $\models_p \phi_1 \wedge \phi_2$ iff $\models_p \phi_1$ and $\models_p \phi_2$.

3.1.5 Equivalence lemma.

$$\mathcal{I}^T = \mathcal{J}^T \Rightarrow \mathcal{I} \equiv_T \mathcal{J} \ .$$

Proof. Assume $\mathcal{I}^T = \mathcal{J}^T$ and for any $\phi \in T$ prove $\mathcal{I} \models \phi \Leftrightarrow \mathcal{J} \models \phi$ by a straightforward induction on the construction of ϕ because for every $\phi \in \mathit{FPA}(\phi)$ we have $\phi \in \mathcal{I} \Leftrightarrow \phi \in \mathcal{J}$. □

3.1.6 Decidability of tautologies. Fix a language $\mathcal{L}$. We call a set S of formulas of $\mathcal{L}$ *decidable* if we can effectively decide for every formula ϕ whether or not $\phi \in S$ holds. Finite sets S are clearly decidable.

For every decidable propositional interpretation $\mathcal{I}$ and for every formula ϕ of $\mathcal{L}$ we can effectively decide whether $\mathcal{I} \vDash \phi$ holds by simply using the properties Par. 3.1.2 of propositional truth.

We can effectively decide whether a formula ϕ is a tautology because the set $FPA\phi$ of all of its free propositional atoms contains finitely many, say n, atoms. We claim that $\vDash_p \phi$ holds iff $\mathcal{I} \vDash_p \phi$ holds for all 2^n subsets $\mathcal{I}$ of $FPA(\phi)$ and the last is clearly effectively decidable. The claim holds because if ϕ is a tautology then it is true in all subsets of $FPA(\phi)$. On the other hand, if ϕ is true in all subsets of $FPA(\phi)$ then if $\mathcal{I}$ is any propositional interpretation for $\mathcal{L}$ then so is $\mathcal{I}^\phi \subseteq FPA(\phi)$ and we have $\mathcal{I}^\phi \vDash \phi$ and hence $\mathcal{I} \vDash \phi$ by Lemma 3.1.5. Thus ϕ is a tautology.

The just described method of deciding whether ϕ is a tautology is called the *truth table method* because every propositional interpretation $\mathcal{I} \subseteq FPA(\phi)$ can be viewed as a row in a table containing t or f according to whether $\mathcal{I} \vDash \phi$ holds or not.

3.2 Propositional Tableaux

We now introduce a method of testing for tautologies by propositional *tableau* proofs. Tableaux were first used by Beth [2] and refined by R. Smullyan [24]. We use the signed tableaux of Smullyan but in a positive way which proves goals rather than refuting them. We will show that tableau proofs prove exactly the tautologies. We introduce the tableaux because they can be extended to deal with identity and quantifiers whereas the truth table method cannot, at least not directly.

3.2.1 Signed formulas. A formula ϕ of a language $\mathcal{L}$ is *signed* if it is written in one of two forms: ϕ or $\phi*$. The signed formula of the first form is called *assumption* and of the second form *goal*. We will use ϕ^+ as a syntactic variable ranging over signed formulas. We will denote by Δ finite nonempty sequences of signed formulas. Every sequence Δ of signed formulas can be associated with a sequent $\bigwedge\Lambda \to \bigvee\Gamma$ where the sequence Λ contains exactly the assumptions in Δ and the sequence Γ contains exactly the goals in Δ (with the signs $*$ deleted).

3.2.2 Propositional tableaux. Fix a language $\mathcal{L}$. Propositional tableaux for $\mathcal{L}$ are finitely branching trees built by to two kinds of expansion rules. A *unary tableau expansion* is of the form

$$\frac{\Delta_1}{\phi_1^+} \tag{1}$$

where Δ_1 is a possibly empty sequence of signed formulas and ϕ_1^+ a signed formula. A *binary tableau expansion rule* is of the form

$$\frac{\Delta_1}{\phi_1^+ \mid \phi_2^+} \tag{2}$$

where also ϕ_2^+ is a signed formula. The signed formulas of Δ are called *premises* and the signed formulas ϕ_1^+ and ϕ_2^+ *conclusions* of the expansion rules.

Let Δ be a non-empty sequence of signed formulas and $\bigwedge\Lambda \to \bigvee\Gamma$ a sequent associated with Δ. A binary tree π with signed formulas as labels is a *tableau for* Δ:

$$\frac{\Delta}{\cdots\pi\cdots}$$

if the tree π is either empty or it is obtained by an expansion rule. In the graphical representation we place the tableau π under the sequence of signed formulas Δ. The tableau is separated from the sequence by a solid line.

If the tableau π for Δ is empty then it has the form

$$\frac{\Delta}{\circ}$$

If the tableau π for Δ is obtained by a unary rule (1) then it has the form

$$\begin{array}{c}\Delta\\ \hline \phi_1^+\\ \cdots\pi_1\cdots\end{array} \quad \text{where } \Delta_1 \subseteq \Delta \text{ and } \quad \begin{array}{c}\Delta\\ \phi_1^+\\ \hline \cdots\pi_1\cdots\end{array}$$

Note that π_1 is a tableau for the sequence Δ, ϕ_1^+. The unary propositional rules will be such that

$$\models_p (\bigwedge\Lambda \to \bigvee\Gamma) \leftrightarrow (\bigwedge\Lambda_1 \to \bigvee\Gamma_1) \tag{3}$$

where the sequent on the right is associated with Δ, ϕ_1^+.

If the tableau π for Δ is obtained by a binary rule (2) then it has the form

$$\begin{array}{cc}\multicolumn{2}{c}{\Delta}\\ \hline \phi_1^+ & \phi_2^+\\ \cdots\pi_1\cdots & \cdots\pi_2\cdots\end{array} \quad \text{where } \Delta_1 \subseteq \Delta, \quad \begin{array}{c}\Delta\\ \phi_1^+\\ \hline \cdots\pi_1\cdots\end{array} \text{, and } \quad \begin{array}{c}\Delta\\ \phi_2^+\\ \hline \cdots\pi_2\cdots\end{array}$$

Note that π_1 is a tableau for the sequence Δ, ϕ_1 and π_2 a tableau for the sequence Δ, ϕ_2. The binary rules will be such that

$$\models_p (\bigwedge\Lambda \to \bigvee\Gamma) \leftrightarrow (\bigwedge\Lambda_1 \to \bigvee\Gamma_1) \wedge (\bigwedge\Lambda_2 \to \bigvee\Gamma_2) \tag{4}$$

where the two sequents on the right are associated with Δ, ϕ_1^+ and Δ, ϕ_2^+ respectively.

3.2.3 Propositional tableau expansion rules. Fix a language $\mathcal{L}$. The following unary propositional tableau expansion rules are called *flatten rules*:

$$\frac{\phi_1 \land \phi_2}{\phi_1}\,(\land_1) \quad \frac{\phi_1 \land \phi_2}{\phi_2}\,(\land_2) \quad \frac{\phi_1 \leftrightarrow \phi_2}{\phi_1 \to \phi_2}\,(\leftrightarrow_1) \quad \frac{\phi_1 \leftrightarrow \phi_2}{\phi_2 \to \phi_1}\,(\leftrightarrow_2)$$
$$\frac{\phi_1 \to \phi_2 *}{\phi_1}\,(\to_1 *) \quad \frac{\phi_1 \to \phi_2 *}{\phi_2 *}\,(\to_2 *) \quad \frac{\phi_1 \lor \phi_2 *}{\phi_1 *}\,(\lor_1 *) \quad \frac{\phi_1 \lor \phi_2 *}{\phi_2 *}\,(\lor_2 *)\ .$$

The following binary propositional tableau expansion rules are called *split rules*:

$$\frac{\phi_1 \lor \phi_2}{\phi_1 \mid \phi_2}\,(\lor) \quad \frac{\phi_1 \to \phi_2}{\phi_2 \mid \phi_1 *}\,(\to) \quad \frac{\phi_1 \land \phi_2 *}{\phi_1 * \mid \phi_2 *}\,(\land *) \quad \frac{\phi_1 \leftrightarrow \phi_2 *}{\phi_1 \to \phi_2 * \mid \phi_2 \to \phi_1 *}\,(\leftrightarrow *)\ .$$

The following unary propositional tableau expansion rules are called *inversion rules*:

$$\frac{\neg\phi}{\phi *}\,(\neg) \quad \frac{\neg\phi *}{\phi}\,(\neg *)\ .$$

Note that we have for each propositional connective two rules, one when the connective is the premise formula as an assumption and one as a goal.

We will now convince ourselves by the truth table method that, for instance, the rule $\to$ satisfies 3.2.2(4) which has the form:

$$\vDash_p ((\phi_1 \to \phi_2) \land \bigwedge \Lambda \to \bigvee \Gamma) \leftrightarrow (\phi_2 \land \bigwedge \Lambda \to \bigvee \Gamma) \land (\bigwedge \Lambda \to \phi_1 \lor \bigvee \Gamma)\ .$$

Indeed, for every propositional interpretation $\mathcal{I}$ we have

$$\mathcal{I} \vDash (\phi_1 \to \phi_2) \land \bigwedge \Lambda \to \bigvee \Gamma$$

iff $\mathcal{I} \nvDash \phi_1 \to \phi_2$ or $\mathcal{I} \vDash \bigwedge \Lambda \to \bigvee \Gamma$ iff ($\mathcal{I} \vDash \phi_1$ and $\mathcal{I} \nvDash \phi_2$) or $\mathcal{I} \vDash \bigwedge \Lambda \to \bigvee \Gamma$ iff $\mathcal{I} \vDash \bigwedge \Lambda \to \phi_1 \lor \bigvee \Gamma$ and $\mathcal{I} \vDash \phi_2 \land \bigwedge \Lambda \to \bigvee \Gamma$ iff

$$\mathcal{I} \vDash (\phi_2 \land \bigwedge \Lambda \to \bigvee \Gamma) \land (\bigwedge \Lambda \to \phi_1 \lor \bigvee \Gamma)\ .$$

3.2.4 Proofs with propositional tableaux. Fix a language $\mathcal{L}$. Let Δ be a non-empty finite sequence of signed formulas of $\mathcal{L}$, and π a propositional tableau for Δ.

A *branch* of the tableau π is a sequence of signed formulas Δ, Δ_1 where Δ_1 is read off some branch of the tree π. We say that the branch is *closed on* ϕ if both ϕ and $\phi *$ are in the branch Δ, Δ_1. The branch is *closed* if Δ, Δ_1 contains either the goal $\top *$, or the assumption $\bot$, or the branch is closed on some ϕ different from $\top$ and $\bot$. The tableau π is *closed* if all of its branches are closed.

We write $\pi\colon \vdash_p [\Delta]$ when π is a closed propositional tableau for Δ. We write $\vdash_p [\Delta]$ if there is a propositional tableau π such that $\pi\colon \vdash_p [\Delta]$. By a simple induction proof on the structure of π we can show:

$$\pi : \vdash_p [\Delta] \text{ and } \Delta \subseteq \Delta_1 \Rightarrow \pi : \vdash_p [\Delta_1]$$

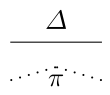
where by $\Delta \subseteq \Delta_1$ we mean $\phi^+ \in \Delta \Rightarrow \phi^+ \in \Delta_1$ *for all* ϕ^+.

For a formula ϕ we say that the tableau π *proves* ϕ, in symbols $\pi\colon \vdash_p \phi$, if $\pi\colon \vdash_p [\phi*]$ holds. The same conventions as for sequences Δ apply also to formulas ϕ. In particular, we say that ϕ *is provable* if $\vdash_p \phi$ holds. For a set of of formulas S we write $\vdash_p S$ if $\vdash_p \phi$ holds for all $\phi \in S$.

3.2.5 Lemma (Soundness of propositional tableaux). *If* $\bigwedge\Lambda \to \bigvee\Gamma$ *is associated to* Δ *then*

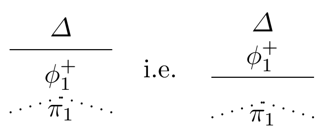
$$\pi\colon \vdash_p [\Delta] \Rightarrow\ \models_p \bigwedge\Lambda \to \bigvee\Gamma\ .$$

Proof. By induction on the structure of π. So assume that the tableau π for Δ is closed:



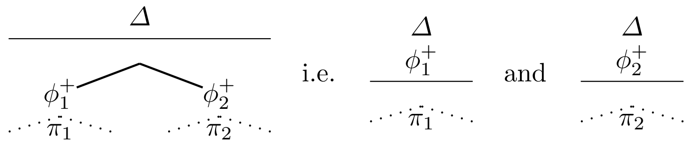
and perform a case analysis on π. If π is empty then either $\top* \in \Delta$, i.e. $\top \in \Gamma$, or $\bot \in \Delta$, i.e. $\bot \in \Lambda$, or else $\psi, \psi* \in \Delta$, i.e. $\psi \in \Lambda$ and $\psi \in \Gamma$ for some propositional atom ψ. For any propositional interpretation $\mathcal{I}$ we then have $\mathcal{I} \models \bigwedge\Lambda \to \bigvee\Gamma$ and so the sequent is a tautology.

If the first expansion in π is by a unary rule 3.2.2(1) such that $\Delta_1 \subseteq \Delta$ then we have



for some tableau π_1 such that $\pi_1\colon \vdash_p [\Delta, \phi_1^+]$. Let $\bigwedge\Lambda_1 \to \bigvee\Gamma_1$ be a sequent associated with Δ, ϕ_1^+. We get $\models_p \bigwedge\Lambda_1 \to \bigvee\Gamma_1$ by IH and so $\models_p \bigwedge\Lambda \to \bigvee\Gamma$ by 3.2.2(3).

If the first expansion in π is by a binary rule 3.2.2(2) such that $\Delta_1 \subseteq \Delta$ then we have



for some tableaux π_1 and π_2 such that $\pi_1\colon \vdash_p [\Delta, \phi_1^+]$ and $\pi_2\colon \vdash_p [\Delta, \phi_2^+]$. Let $\bigwedge\Lambda_1 \to \bigvee\Gamma_1$ and $\bigwedge\Lambda_2 \to \bigvee\Gamma_2$ be sequents associated with Δ, ϕ_1^+ and

Δ, ϕ_2^+ respectively. We get $\models_p \bigwedge \Lambda_1 \rightarrow \bigvee \Gamma_1$ and $\models_p \bigwedge \Lambda_2 \rightarrow \bigvee \Gamma_2$ by two IH's. Hence $\models_p \bigwedge \Lambda \rightarrow \bigvee \Gamma$ by 3.2.2(4). □

3.2.6 Lemma (Completeness of propositional tableaux). *If* $\bigwedge \Lambda \rightarrow \bigvee \Gamma$ *is associated with* Δ *then*

$$\models_p \bigwedge \Lambda \rightarrow \bigvee \Gamma \Rightarrow \ \vdash_p [\Delta] \ .$$

Proof. Assume $\models_p \bigwedge \Lambda \rightarrow \bigvee \Gamma$ and prove $\vdash_p [\Delta]$ by induction on the total number n of propositional connectives in the formulas of Δ (not counting those within quantifiers).

If $n = 0$ then Δ consists at most of propositional atoms $\bot$ and $\top$. If $\bot \in \Lambda$ or $\top \in \Gamma$ then the branch Δ is closed. If neither of the two cases applies and the sequences Λ and Γ have no propositional atom in common then we have a contradiction because $\mathcal{I} \not\models \bigwedge \Lambda \rightarrow \bigvee \Gamma$ for $\mathcal{I} = \{\phi \mid \phi \in \Lambda\}$. Thus Δ must be closed on some ϕ. Hence, in both cases it suffices to take π empty.

If $n > 0$ then select a signed propositional formula ϕ^+ of Δ which is not $\top$ or $\bot$ and denote by Δ_1 the sequence obtained from Δ by omitting the selected formula from it. Denote by Λ_1 the sequence obtained from Λ by deleting ϕ if the selected signed formula is an assumption and the sequence Λ otherwise and denote by Γ_1 the sequence obtained from Γ by deleting ϕ if the selected formula is a goal and the sequence Γ otherwise. Thus if the selected formula is a goal then $\bigwedge \Lambda_1 \rightarrow \phi \vee \bigvee \Gamma_1$ is a tautology and if the selected formula is an assumption then $\phi \wedge \bigwedge \Lambda_1 \rightarrow \bigvee \Gamma_1$ is a tautology. We wish to prove $\vdash_p [\Delta]$ by the case analysis of the signed formula ϕ^+.

If $\phi^+ \equiv \neg \phi_1 * \in \Delta$ then, since $\models_p \bigwedge \Lambda_1 \rightarrow \neg \phi_1 \vee \bigvee \Gamma_1$, we also have $\models_p \phi_1 \wedge \bigwedge \Lambda_1 \rightarrow \bigvee \Gamma_1$. The associated sequence Δ_1, ϕ_1 has $n-1$ connectives and so $\pi_1 \colon \vdash_p [\Delta_1, \phi_1]$ for some π_1 by IH. This is shown on the left and the constructed tableau is shown on the right:

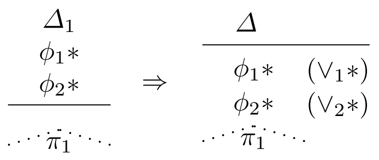
$$\begin{array}{c} \Delta_1 \\ \phi_1 \\ \hline \cdots \pi_1 \cdots \end{array} \quad \Rightarrow \quad \begin{array}{c} \Delta \\ \hline \phi_1 \quad (\neg *) \\ \cdots \pi_1 \cdots \end{array}$$

The constructed tableau is closed and so $\vdash_p [\Delta]$.

If $\phi^+ \equiv \phi_1 \vee \phi_2 * \in \Delta$ then, since $\models_p \bigwedge \Lambda_1 \rightarrow (\phi_1 \vee \phi_2) \vee \bigvee \Gamma_1$, we also have

$$\models_p \bigwedge \Lambda_1 \rightarrow \phi_1 \vee \phi_2 \vee \bigvee \Gamma_1 \ .$$

The associated sequence $\Delta_1, \phi_1 *, \phi_2 *$ has $n-1$ propositional connectives and so $\pi_1 \colon \vdash_p [\Delta_1, \phi_1 *, \phi_2 *]$ for some π_1 by IH. This is shown in the following on the left and the constructed tableau is shown on the right:

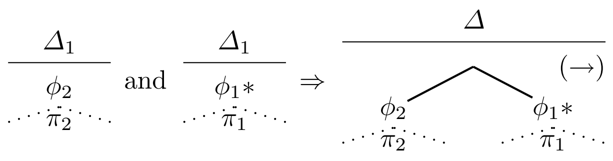


The constructed tableau is closed and so $\vdash_p [\Delta]$.

If $\phi^+ \equiv \phi_1 \rightarrow \phi_2 \in \Delta$ then, since $\models_p (\phi_1 \rightarrow \phi_2) \wedge \bigwedge \Lambda_1 \rightarrow \bigvee \Gamma_1$, we also have

$$\models_p \phi_2 \wedge \bigwedge \Lambda_1 \rightarrow \bigvee \Gamma_1 \text{ and } \models_p \bigwedge \Lambda_1 \rightarrow \phi_1 \vee \bigvee \Gamma_1 \ .$$

The sequences Δ_1, ϕ_2 and Δ_1, ϕ_1* have $n-1$ propositional connectives each and so $\pi_2\colon \vdash_p [\Delta_1, \phi_2]$ and $\pi_1\colon \vdash_p [\Delta_1, \phi_1*]$ for some π_2 and π_1 by two IH's. The two tableaux are shown on the left and the constructed tableau is shown on the right:



The constructed tableau is closed and so $\vdash_p [\Delta]$. The remaining cases are similar. □

3.2.7 Theorem (Soundness and completeness of propositional tableaux).

$$\vdash_p \phi \Leftrightarrow \models_p \phi \ .$$

Proof. We have $\vdash_p \phi$ iff by definition $\vdash_p [\phi*]$ iff by Lemmas 3.2.5 and 3.2.6 $\models_p \top \rightarrow \phi$ iff $\models_p \phi$. □

3.2.8 Falsification of open branches in propositional tableaux. A branch Δ of a propositional tableau π for $\phi*$ is *propositionally complete* if with every expansion rule with a premise in Δ the branch contains also at least one of its conclusions.

If a tableau for $\phi*$ does not close then it contains an open branch Δ and this can be clearly propositionally completed by finitely many expansions because the conclusions of expansion rules are formulas with a lesser number of propositional connectives than the premises.

For every propositionally complete and not closed branch Δ of a tableau for ϕ we can construct a propositional interpretation $\mathcal{I}$ by collecting all propositional atoms in assumptions:

$$\mathcal{I} = \{\psi \mid \psi \in \Delta \text{ and } \psi \text{ is a propositional atom}\} \ .$$

We prove by induction on the number of connectives in ψ that the following holds:

$$(\psi \in \mathit{\Delta} \Rightarrow \mathcal{I} \models \psi) \text{ and } (\psi{*} \in \mathit{\Delta} \Rightarrow \mathcal{I} \not\models \psi)\ .$$

If ψ is a propositional atom then the claim holds directly from the definition of $\mathcal{I}$. If $\psi \equiv \neg\psi_1$ then if $\neg\psi_1 \in \mathit{\Delta}$ we have $\psi_1{*} \in \mathit{\Delta}$ because the branch is propositionally complete and so $\mathcal{I} \not\models \psi_1$ by IH, i.e. $\mathcal{I} \models \neg\psi_1$. The case $\neg\psi_1{*} \in \mathit{\Delta}$ is similar.

If $\psi \equiv \psi_1 \wedge \psi_2$ then if $\psi_1 \wedge \psi_2 \in \mathit{\Delta}$ we have $\psi_1, \psi_2 \in \mathit{\Delta}$ because of saturation and $\mathcal{I} \models \psi_1$, $\mathcal{I} \models \psi_2$ by two IH's. Hence $\mathcal{I} \models \psi_1 \wedge \psi_2$. If $\psi_1 \wedge \psi_2{*} \in \mathit{\Delta}$ then one of the subformulas is a goal in $\mathit{\Delta}$, say $\psi_1{*} \in \mathit{\Delta}$. Thus $\mathcal{I} \not\models \psi_1$ by IH and hence $\mathcal{I} \not\models \psi_1 \wedge \psi_2$. The remaining cases for ψ are similar.

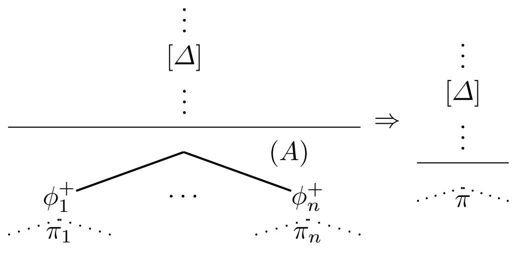
The goal $\phi{*}$ cannot be a tautology because we have $\mathcal{I} \not\models \phi$ by the just proved property.

3.3 Admissible Expansion Rules

3.3.1 Admissible expansion rules. We can often considerably shorten a tableau proof by the use of *admissible* expansion rules. Admissible rules can be reduced to the *basic* expansion rules, which are in the propositional case given in Par. 3.2.3), and so they do not add any strength to the proof system. Precisely, the n-ary expansion rule ($n \geq 1$)

$$\frac{\mathit{\Delta}}{\phi_1^{+} \mid \cdots \mid \phi_n^{+}}\ (A)$$

is *admissible* if for all sequences of signed formulas $\mathit{\Delta}_1$ s.t. $\mathit{\Delta} \subseteq \mathit{\Delta}_1$ and all tableaux $\pi_1, \ldots, \pi_n$ s.t. π_1: $\vdash_p [\mathit{\Delta}_1, \phi_1^{+}]$, …, π_n: $\vdash_p [\mathit{\Delta}_1, \phi_n^{+}]$ we can effectively construct a basic tableau π such that π: $\vdash_p [\mathit{\Delta}_1]$. The situation can be visualized as follows:



where we have indicated by $[\mathit{\Delta}]$ that the premises of A are in the branch above. An application of the admissible rule and the subsequent closure by tableaux $\pi_1, \ldots, \pi_n$ on the left can be effectively replaced by the tableau π.

The reader will note that we can permit in the tableau π also expansions by previously justified admissible rules of inference because they can be always replaced by basic inferences. After we have demonstrated the effective reduction of tableaux $\pi_1, \ldots, \pi_n$ to the tableau π we may freely use the admissible rule (A) as if it were a basic rule.

3.3.2 Theorem (Generalized flatten rules). *Following* generalized flatten *rules are admissible in propositional tableaux for any $n \geq 2$ and $1 \leq i \leq n$:*

$$\frac{\phi_1 \wedge \cdots \wedge \phi_i \wedge \cdots \wedge \phi_n}{\phi_i}\ (G\wedge_i)$$
$$\frac{\phi_1 \vee \cdots \vee \phi_i \vee \cdots \vee \phi_n *}{\phi_i *}\ (G\vee_i *)\ .$$

Proof. We prove the admissibility of the rule $(G\wedge_i)$ by induction on i; the admissibility of $(G\vee_i *)$ is proved similarly. In the base case when $i = 1$ there is nothing to prove as $(G\wedge_1)$ is the basic flatten rule $(\wedge_1)$. For $2 \leq i + 1 \leq n$ we consider an expansion by the rule:

$$\begin{array}{lc}
\quad\vdots & \\
\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n & \\
\quad\vdots & \\
\hline
\phi_{i+1} & (G\wedge_{i+1}) \\
\cdots\pi\cdots &
\end{array}$$

If $n = 2$ then this is the basic $(\wedge_2)$ rule and we are done. If $n > 2$ then we replace the rule $(G\wedge_{i+1})$ by the basic rule $(\wedge_2)$ followed by the $(G\wedge_i)$ rule which is admissible by IH. The new tableau is as follows:

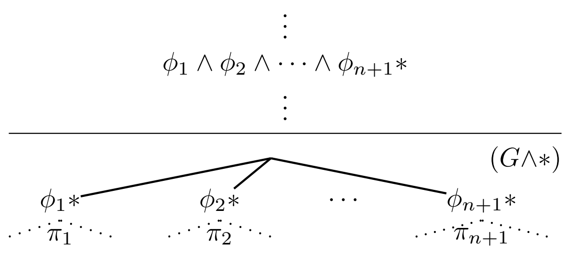
$$\begin{array}{lc}
\quad\vdots & \\
\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n & \\
\quad\vdots & \\
\hline
\phi_2 \wedge \cdots \wedge \phi_n & (\wedge_2) \\
\phi_{i+1} & (G\wedge_i) \\
\cdots\pi\cdots &
\end{array}$$

□

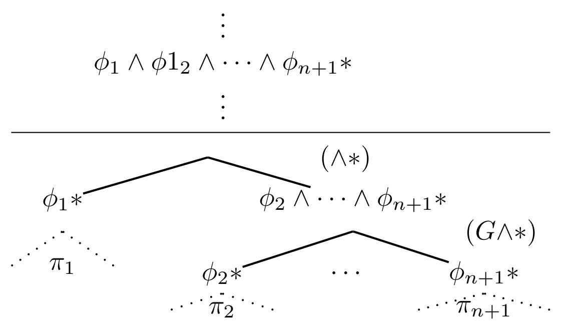
3.3.3 Theorem (Generalized split rules). *The following generalizations of rules $(\vee)$ and $(\wedge *)$ are admissible in propositional tableaux for any $n \geq 1$:*

$$\frac{\phi_1 \vee \cdots \vee \phi_n}{\phi_1 \mid \cdots \mid \phi_n} \quad (G\vee)$$
$$\frac{\phi_1 \wedge \cdots \wedge \phi_n *}{\phi_1 * \mid \cdots \mid \phi_n *} \quad (G\wedge *)$$

Proof. We prove here only $(G\wedge *)$ by induction on n. In the base case when $n = 1$ there is nothing to prove as $(G\wedge *)$ is the basic split rule $(\wedge)$. In the inductive case when $n + 1 \geq 2$ we consider an expansion by the rule:



We replace the rule $(G\wedge *)$ by $(\wedge *)$ followed on the right by an n-ary $(G\wedge *)$ rule which is admissible by IH. The new tableau is as follows:



$\phi_1 \wedge \phi 1_2 \wedge \cdots \wedge \phi_{n+1} *$
$(\wedge *)$
$\phi_1 *$
$\phi_2 \wedge \cdots \wedge \phi_{n+1} *$
$(G\wedge *)$
π_1
$\phi_2 *$
$\cdots$
$\phi_{n+1} *$
π_2
π_{n+1}

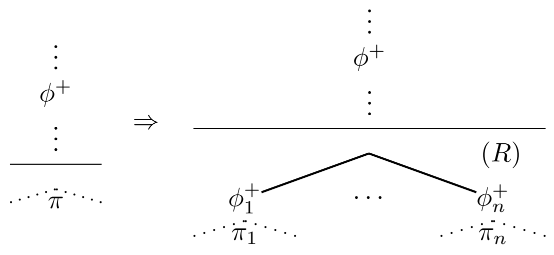
□

3.3.4 Inversion of expansion rules. Let

$$\frac{\phi^+}{\phi_1^+ \mid \cdots \mid \phi_n^+} \quad (R)$$

be an expansion rule with $n \geq 1$. The rule R can be an inversion or split rule as well as an identity or quantifier rule which will be introduced later. We say that the rule (R) is *invertible* if for every sequence of signed formulas Δ such that $\pi : \vdash_p [\Delta, \phi^+]$ there are tableaux π_i for $1 \leq i \leq n$ such that

$\pi_i : \vdash_p [\varDelta, \phi_i^+]$. Moreover the rule (R) is not applied with ϕ^+ as a premise, and the formula ϕ^+ is not used for the closing of a branch, in neither of tableaux π_i. The inversion of the rule R can be visualized as follows:

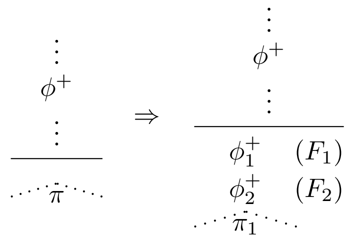


where we can in effect assume that the first expansion in the closed tableau on the right is by the rule (R) and that the premise ϕ^+ is used only once as indicated. This also means that the premise ϕ^+ is not used to close a branch.

Propositional flatten rules have the following schematic form:

$$\frac{\phi^+}{\phi_1^+}\,(F_1) \qquad \frac{\phi^+}{\phi_2^+}\,(F_2)\ .$$

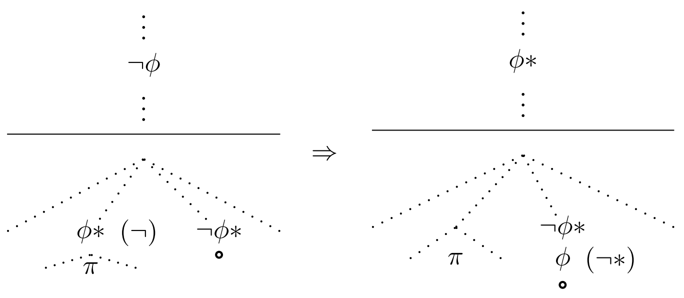
We say that the flatten rules (F_1) and (F_2) are *invertible* if from $\pi\colon \vdash_p [\varDelta, \phi^+]$ we can form a tableau π_1 such that $\pi_1\colon \vdash_p [\varDelta, \phi_1^+, \phi_2^+]$ where the rules (F_1) and (F_2) are not used in π_1 with ϕ^+ as a premise and neither is the formula ϕ^+ used for closing of a branch. The inversion can be visualized as follows:



where we can in effect assume that the first two expansions in the closed tableau on the right are by the rules (F_1) and (F_2) and that the premise ϕ^+ is used only once as indicated. This also means that the premise ϕ^+ is not used to close a branch.

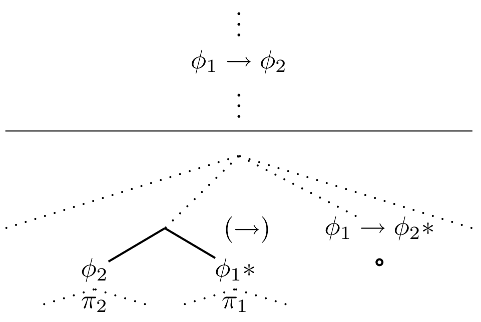
3.3.5 Inversion theorem. *Inversion, split, and flatten rules are invertible in propositional tableaux.*

Proof. The following diagram illustrates the inversion of a $(\neg)$ inversion rule:

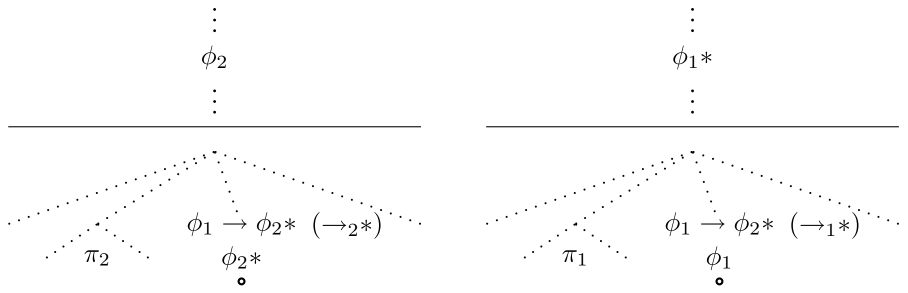


The closed tableau on the left shows just two of possibly many uses of the premise $\neg\phi$. The first one uses the premise in a $(\neg)$ rule and the second one closes the branch with $\neg\phi*$. The closed tableau on the right shows the inversion where the first used formula has been removed (because it is now in the top sequence) and the second one is now used as a premise of a $(\neg*)$ rule after which the branch closes. The two indicated transformations should be applied to all uses of the premise $\neg\phi$. The inversion of $\neg*$ inversion rules is similar.

The following diagram shows a closed tableau before the inversion of a $(\rightarrow)$ split rule:



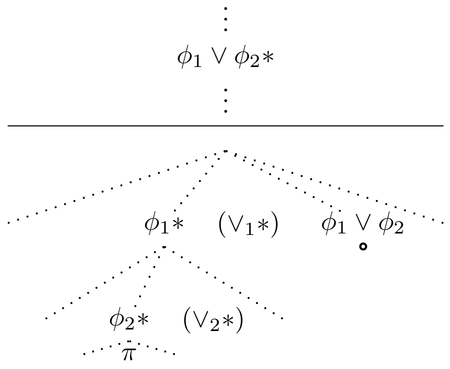
We have indicated two possible uses of the premise $\phi_1 \rightarrow \phi_2$. The first use is in a $(\rightarrow)$ rule and the second use is in the closure of a branch. We form the two inverted tableux as follows:



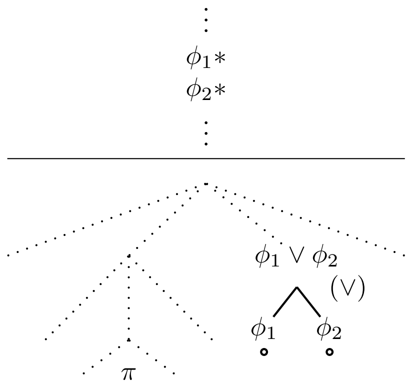
ϕ_2
$\phi_1 \rightarrow \phi_2*$ $(\rightarrow_2*)$
π_2
ϕ_2*
ϕ_1*
$\phi_1 \rightarrow \phi_2*$ $(\rightarrow_1*)$
π_1
ϕ_1

The expansions by the $(\rightarrow)$ rule have been replaced in the inverted tableau by tableaux π_2 and π_1 respectively and the closing formulas $\phi_1 \rightarrow \phi_2*$ have been expanded by the corresponding flatten rules after which the branches close. The inversion of other split rules is similar.

The following diagram shows a tableau before the inversion of a $(\vee_i*)$ flatten rules:



We have indicated three possible uses of the premise $\phi_1 \vee \phi_2*$. The first use is in a $(\vee_1*)$ rule after which there is a use in a $(\vee_2*)$ rule and third use is in the closure of a branch. We form the closed inverted tableau as follows:



In the inverted tableau the expansions by the $(\vee_i)$ rules have been removed and the closing formulas $\phi_1 \vee \phi_2$ have been expanded by $(\vee)$ split rules after which the branches close. The inversion of other flatten rules is similar. □

3.3.6 Cut rules. For any formula π the following is a *cut rule on* ϕ:

$$\frac{}{\phi \mid \phi*}\ (C)\ .$$

That cut rules are admissible in propositional tableaux, i.e. that

$$\pi_1 \colon \vdash_p [\Delta, \phi] \text{ and } \pi_2 \colon \vdash_p [\Delta, \phi *] \Rightarrow \vdash_p [\Delta] \ .$$

holds can be proved by the following semantic argument. Assume $\pi_1 \colon \vdash_p [\Delta, \phi]$ and $\pi_2 \colon \vdash_p [\Delta, \phi *]$ and let $\bigwedge \Lambda \to \bigvee \Gamma$ be a sequent associated with Δ. By the Soundness lemma (3.2.5) we have $\models_p \phi \land \bigwedge \Lambda \to \bigvee \Gamma$ and $\models_p \bigwedge \Lambda \to \phi \lor \bigvee \Gamma$. By the truth table method we can see that also

$$\models_p (\phi \land \bigwedge \Lambda \to \bigvee \Gamma) \land (\bigwedge \Lambda \to \phi \lor \bigvee \Gamma) \to \bigwedge \Lambda \to \bigvee \Gamma$$

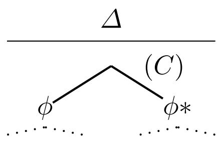
holds and hence $\models_p \bigwedge \Lambda \to \bigvee \Gamma$. By the Completeness lemma (3.2.6) we then get $\vdash_p [\Delta]$.

Unfortunately, this argument, which depends on the soundness and completeness of propositional tableaux, cannot be extended to tableaux with quantifier rules without first proving the soundness and completeness theorem for such tableaux. Our intention is to reduce the quantificational logic to the propositional logic and to obtain the soundness and completeness of quantificational tableaux through the reduction and so the above semantical argument proving the admissibility of cut rules cannot be used.

We will now prove the admissibility of cuts by a syntactic (proof-theoretic) argument in a way which extends to the quantificational case.

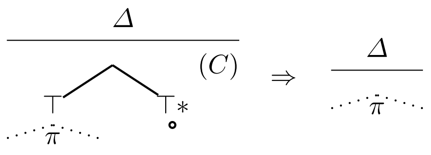
3.3.7 Lemma (Admissibility of cuts on propositional formulas). *If the cut rules on all propositional atoms in a formula ϕ are admissible in propositional tableaux then also the cut rule on ϕ is admissible.*

Proof. Assume that the cuts on the propositional atoms in $FPA(\phi)$ are admissible and prove by induction on the structure of ϕ that the cut on ϕ is admissible. We perform the case analysis of ϕ used in a cut as follows:



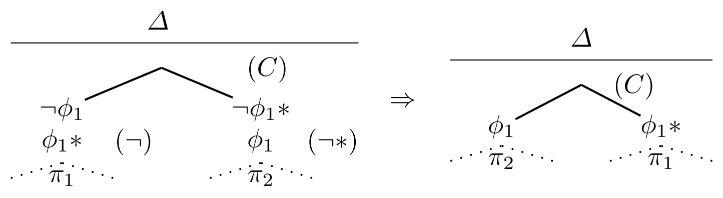
If ϕ is a propositional atom then the cut on ϕ is admissible from the assumption.

If $\phi \equiv \top$ then the expansion by the cut on $\top$ is shown in the following on the left:



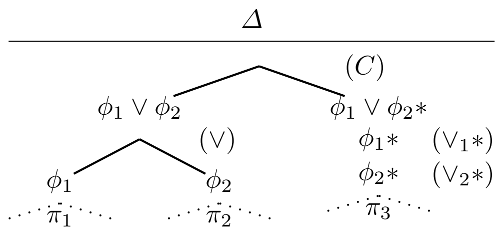
The assumption $\top$ is not used for anything in π and so the tableau π for Δ is closed as shown on the right. The case $\phi \equiv \bot$ is similar.

If $\phi \equiv \neg\phi_1$ then the expansion by the cut on $\neg\phi_1$ is shown in the following on the left:

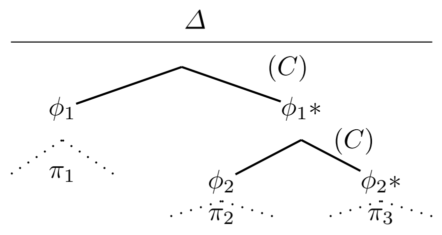


where we may assume without loss of generality that both inversion rules have been inverted. As a consequence the assumption $\neg\phi_1$ is not used for anything in π_1 and the goal $\neg\phi_1*$ is not used in π_2. We transform the tableau as shown on the right where the cut on $\neg\phi_1$ has been removed and the inversion rules replaced by a cut on ϕ_1 which is admissible by IH.

If $\phi \equiv \phi_1 \vee \phi_2$ then the expansion by the cut on $\phi_1 \vee \phi_2$ is as follows:



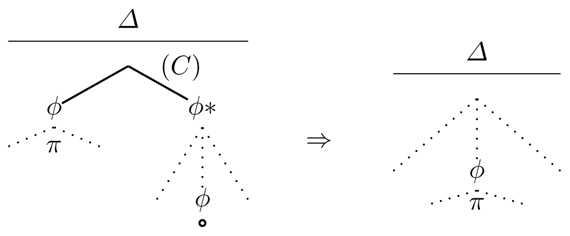
where we may assume without loss of generality that the $(\vee)$ split rule on the left and the $(\vee_i*)$ flatten rules on the right have been inverted. As a consequence the assumption $\phi_1 \vee \phi_2$ is not used for anything in π_1 and π_2 and the goal $\phi_1 \vee \phi_2*$ is not used in π_3. We transform the tableau as follows:



where the formula ϕ has been removed and the disjunctive rules replaced by two cuts on ϕ_1 and ϕ_2 respectively which are admissible by IH. The cases when the main propositional connective of ϕ is $\wedge$, $\rightarrow$, or $\leftrightarrow$ are similar. □

3.3.8 Lemma (Admissibility of cut rules on propositional atoms). *Cut rules on propositional atoms are admissible in propositional tableaux.*

Proof. Consider a closed propositional tableaux with a cut on a propositional atom ϕ shown in the following on the left:



We have shown in the tableau for $\Delta, \phi*$ one of possibly many branches closed on the pair of propositional atoms $\phi, \phi*$. Because the goal $\phi*$ cannot be used in propositional tableaux for anything else we can form a closed tableau for Δ shown on the right where we perform the indicated transformation for all branches closed on $\phi, \phi*$. □

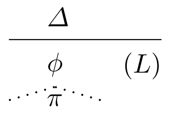
3.3.9 Theorem (Admissibility of cuts). *Cut rules are admissible in propositional tableaux on arbitrary formulas.*

Proof. This is a direct consequence of Lemmas 3.3.7 and 3.3.8. □

3.3.10 Theorem (Lemma rule). *If $\vdash_p \phi$ then the following unary* lemma *rule*

$$\frac{}{\phi}\quad (L)$$

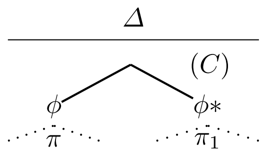
is admissible in propositional tableaux, i.e. for any closed propositional tableau for Δ such that



(1)

we have $\vdash_p [\Delta]$.

Proof. Assume π_1: $\vdash_p \phi$ and that the tableau (1) is closed and form the following closed tableau witnessing $\vdash_p [\Delta]$:



□

3.4 Tautological Consequence

Important extension of the notion of tautology is the notion of *tautological consequence* where we ask whether a formula ϕ follows from a finite or infinite set of formulas T by the laws of propositional logic alone.

3.4.1 Tautological consequence. Fix a language $\mathcal{L}$ and let T be a set of formulas from $\mathcal{L}$. We define the relation *formula ϕ is a tautological consequence of T*, in symbols $T \models_p \phi$ as follows:

$$T \models_p \phi \Leftrightarrow (\mathcal{I} \models T \Rightarrow \mathcal{I} \models \phi) \text{ for all propositional interpretations } \mathcal{I}.$$

From this we can see that for $T = \emptyset$ we have $\emptyset \models_p \phi \Leftrightarrow\ \models_p \phi$.

Tautological consequence is a generalization of implication where we so to speak permit infinite many formulas in antecedent. If T is finite then we have $T \models_p \phi \Leftrightarrow\ \models_p \bigwedge T \to \phi$ by Lemma 3.4.2. If the set T is infinite then it can contain infinitely many propositional atoms and we cannot replace infinite propositional interpretations by finite ones as we did in the truth table method. Thus it seems that that the testing of $T \models_p \phi$ is a hard problem having to do with quantification over uncountably many propositional interpretations. Fortunately, this kind of quantification can be replaced by existential quantification over countably many finite sets of formulas. This is a consequence of the fundamental theorem 3.4.4.

3.4.2 Semantic deduction lemma. *For every finite set S of formulas we have:*

$$S \models_p \phi \Leftrightarrow\ \models_p \bigwedge S \to \phi\ .$$

Proof. We have $\mathcal{I} \models S$ iff $\mathcal{I} \models \psi$ for all $\psi \in S$ iff $\mathcal{I} \models \bigwedge S$. Thus $S \models_p \phi$ iff $\mathcal{I} \models S \Rightarrow \mathcal{I} \models_p \phi$ for all $\mathcal{I}$ iff $\mathcal{I} \models \bigwedge S \Rightarrow \mathcal{I} \models_p \phi$ for all $\mathcal{I}$ iff $\models_p \bigwedge S \to \phi$. □

3.4.3 Semantic weakening lemma. *If $S \subseteq T$ then*

$$S \models_p \phi \Rightarrow T \models_p \phi\ .$$

Proof. Assume $S \models_p \phi$ and take any propositional interpretation such that $\mathcal{I} \models T$ holds. Since this means $\mathcal{I} \models \phi$ for all $\phi \in T$ we also have $\mathcal{I} \models S$ and thus $\mathcal{I} \models \phi$ from the assumption. □

3.4.4 Compactness theorem.

$$T \models_p \phi \Rightarrow S \models_p \phi \text{ for a finite } S \subseteq T.$$

Proof. Assume $T \vDash_p \phi$. T is a set of formulas of a first order language $\mathcal{L}$ which has countably many formulas. Thus the set T is at most countable and we can write it in a form $T = \bigcup_{i \in N} S_i$ where each of the sets S_i is finite and we have $S_0 = \emptyset$ and $S_i \subseteq S_{i+1}$ (This is trivial if T is finite; otherwise, for instance, enumerate T and define S_i to be the set of the first i formulas in the enumeration).

We assign by the function $W(\mathcal{I})$ to every propositional interpretation $\mathcal{I}$ one of the finite sets S_i by:

$$W(\mathcal{I}) = S_i \quad \text{where } i \text{ is the least such that } \mathcal{I} \vDash S_i \Rightarrow \mathcal{I} \vDash \phi.$$

This is a legal definition because if $\mathcal{I} \vDash \phi$ then we have $\mathcal{I} \vDash S_0 \Rightarrow \mathcal{I} \vDash \phi$ and if $\mathcal{I} \nvDash \phi$ then we have $\mathcal{I} \nvDash T$. Thus $\mathcal{I} \nvDash \psi$ for some $\psi \in T$ and, since then $\psi \in S_i$ for some i, we have $\mathcal{I} \nvDash S_i$ for some i and there is a least such i. Define the set S to satisfy:

$$S := \bigcup \{W(\mathcal{I}) \mid \mathcal{I} \text{ is a propositional interpretation.}\}$$

We clearly have $S \subseteq T$. We also have $S \vDash_p \phi$ because for any propositional interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash S$ we have $W(\mathcal{I}) \subseteq S$ and so $\mathcal{I} \vDash W(\mathcal{I})$. We then obtain $\mathcal{I} \vDash \phi$ from the definition of $W(\mathcal{I})$. Thus the theorem will be proved if we demonstrate by an indirect proof that the set S is finite.

So suppose that S is infinite. We will construct an increasing chain of propositional interpretations $\mathcal{I}_i$ in which the infiniteness of S will be propagated. Enumerate towards that end all propositional atoms of $\mathcal{L}$ in an infinite sequence $\psi_1, \psi_2, \psi_3, \ldots$ and define the finite sets of propositional atoms A_i for $i \in \mathbb{N}$ by

$$A_i = \bigcup_{1 \le j \le i} \{\psi_j\} \ .$$

Note that $A_0 = \emptyset$. Define an infinite sequence $\{\mathcal{I}_i\}_{i \in \mathbb{N}}$ of finite propositional interpretations $\mathcal{I}_i$ to satisfy:

$$\mathcal{I}_0 = \emptyset$$
$$\mathcal{I}_{i+1} = \begin{cases} \mathcal{I}_i & \text{if } \bigcup \{W(\mathcal{I}) \mid \mathcal{I} \cap A_{i+1} = \mathcal{I}_i\} \text{ is infinite} \\ \mathcal{I}_i \cup \{\psi_{i+1}\} & \text{otherwise.} \end{cases}$$

We clearly have $\mathcal{I}_i \subseteq \mathcal{I}_{i+1}$. Define the sets I_i for $i \in \mathbb{N}$ to satisfy:

$$I_i = \bigcup \{W(\mathcal{I}) \mid \mathcal{I} \cap A_i = \mathcal{I}_i\}$$

and prove by induction on i:

$$M_i \subseteq A_i \text{ and } I_i \text{ is infinite.}$$

In the base case we have $\mathcal{I}_0 = \emptyset = A_0$ and the set

$$I_0 = \bigcup\{W(\mathcal{I}) \mid \mathcal{I} \cap A_0 = \mathcal{I}_0\} = \bigcup\{W(\mathcal{I}) \mid \emptyset = \emptyset\} = S \ .$$

is infinite. In the inductive case we have

$$\mathcal{I}_{i+1} \subseteq \mathcal{I}_i \cup \{\psi_{i+1}\} \overset{\text{IH}}{\subseteq} A_i \cup \{\psi_{i+1}\} = A_{i+1} \ .$$

We note that if $\mathcal{I} \cap A_i = \mathcal{I}_i$ then

$$\begin{aligned}\mathcal{I} \cap A_{i+1} =& \mathcal{I} \cap (A_i \cup \{\psi_{i+1}\}) = (\mathcal{I} \cap A_i) \cup (\mathcal{I} \cap \{\psi_{i+1}\}) = \\ & \mathcal{I}_i \cup (\mathcal{I} \cap \{\psi_{i+1}\}) = \begin{cases}\mathcal{I}_i & \text{if } \psi_{i+1} \notin \mathcal{I} \\ \mathcal{I}_i \cup \{\psi_{i+1}\} & \text{if } \psi_{i+1} \in \mathcal{I}\end{cases}\end{aligned}$$

and so

$$\begin{aligned}I_n =& \bigcup\{W(\mathcal{I}) \mid \mathcal{I} \cap A_i = \mathcal{I}_i\} = \\ & \bigcup\{W(\mathcal{I}) \mid \mathcal{I} \cap A_{i+1} = \mathcal{I}_i\} \cup \bigcup\{W(\mathcal{I}) \mid \mathcal{I} \cap A_{i+1} = \mathcal{I}_i \cup \{\psi_{i+1}\}\} \ .\end{aligned}$$

One of the two sets on the right must be infinite because I_n is by IH. We consider two cases. If the first set is infinite then $\mathcal{I}_{i+1} = \mathcal{I}_i$ by definition and the first set is I_{n+1}. If the first set is finite then $\mathcal{I}_{i+1} = \mathcal{I}_i \cup \{\psi_{i+1}\}$ and so the second set, which must be infinite, is I_{n+1}. Thus in both cases I_{n+1} is an infinite set.

We construct a propositional interpretation $\mathcal{I} = \bigcup\{\mathcal{I}_i \mid i \in \mathbb{N}\}$ and we have $W(\mathcal{I}) = S_k$ for the least k such that $\mathcal{I} \vDash S_k \Rightarrow \mathcal{I} \vDash \phi$. Let i be the least number such that $FPA(S_k \cup \{\phi\}) \subseteq A_i$ and let $\mathcal{J}$ be any propositional interpretation such that $\mathcal{J} \cap A_i = \mathcal{I}_i$. We have

$$\begin{aligned}\mathcal{J}^{S_k \cup \{\phi\}} = & \mathcal{J} \cap FPA(S_k \cup \{\phi\}) = \mathcal{J} \cap A_i \cap FPA(S_k \cup \{\phi\}) = \\ & \mathcal{I}_i \cap FPA(S_k \cup \{\phi\}) = \mathcal{I}_i^{S_k \cup \{\phi\}}\end{aligned}$$

and so $\mathcal{J}$ and $\mathcal{I}_i$ are $S_k \cup \{\phi\}$-equivalent by the Equivalence lemma (see 3.1.5). From this we get $W(\mathcal{J}) = W(\mathcal{I}_i)$ and hence

$$I_i = \bigcup\{W(\mathcal{J}) \mid \mathcal{J} \cap A_i = \mathcal{I}_i\} = \bigcup\{W(\mathcal{I}_i) \mid \mathcal{J} \cap A_i = \mathcal{I}_i\} = W(\mathcal{I}_i)$$

which is a contradiction because the set $W(\mathcal{I}_i)$ is finite (the set is actually equal to S_k). $\square$

3.4.5 Corollary (Tautological reduction).

$$T \vDash_p \phi \Leftrightarrow \ \vDash_p \bigwedge S \rightarrow \phi$$

for a finite subset S of T.

Proof. From $T \vDash_p \phi$ we obtain $S \vDash_p \phi$ for a finite subset S of T by Thm. 3.4.4. The opposite direction follows from Lemma 3.4.3 and we have $S \vDash_p \phi$ iff $\vDash_p \bigwedge S \rightarrow \phi$ by Lemma 3.4.2. $\square$

3.4.6 Remark. The reader familiar with set theory will recognize the construction of the sequence $\{\mathcal{I}_i\}_{i\mathbb{N}}$ in the proof of the Compactness theorem as the construction of an infinite branch of an infinite tree with finitely branching nodes in the proof of the König's lemma. The lemma says that in every finitely branching tree with an infinite number of nodes there is an infinite branch.

3.4.7 Semidecidability of tautological consequence. Let $\mathcal{L}$ be a language and T an infinite decidable set of its formulas. We have noted in Par. 3.4.1 that in order to decide the unary relation ϕ *is a tautological consequence of* T, i.e. $T \vDash_p \phi$, one would expect to test $\mathcal{I} \vDash T \Rightarrow \mathcal{I} \vDash \phi$ for uncountably many propositional interpretations $\mathcal{I}$. By the Compactness theorem and Lemma 3.4.3 we however know that it is sufficient to decide whether $S \vDash_p \phi$ holds for a finite subset S of T. As there are countably many finite subsets S of the countable T we have to test by Lemma 3.4.2 countably many times whether $\bigwedge S \rightarrow \phi$ is a tautology.

One of the ways to organize the tests is as follows. We encode all formulas of $\mathcal{L}$ into $\mathbb{N}$. Starting from 0 we then successively test for all natural numbers i whether i is a list such that for every $j \in i$ (there are finitely many such j's) the number j codes a formula ϕ (this can be effectively done). If so, then we test whether $\phi \in T$. If this holds for all $j \in i$ we can effectively form the finite set S of all formulas of $\mathcal{L}$ coded by i. We then test whether $\bigwedge S \rightarrow \phi$ is a tautology. If this is the case we stop the testing and we know that ϕ is a tautological consequence of T. If not then we have to continue with the next number $i+1$ and if ϕ is not a tautological consequence then by Lemmas 3.4.3 and 3.4.2 there is no finite $S \subset T$ such that $\bigwedge S \rightarrow \phi$ is a tautology and we will never discover the fact.

Predicates $P(x)$ such that if there is an x satisfying P we can effectively find it but we go on searching forever if for all x not $P(x)$ are called *semidecidable* predicates. It can be shown by the methods of recursion theory that the unary predicate $T \vDash_p \phi$ is semidecidable and that the above described method of crude search cannot be improved upon. Later in this text we will show how to reduce the general questions of logical validity of formulas and of logical consequence to the questions of tautological consequence from decidable sets. This will mean that the best we can do in first order logic is to find a proof of a formula from given axioms if there is one. If the formula is unprovable we might never discover it.

3.5 Tableaux with Axioms

3.5.1 Axiom rules. Fix a language $\mathcal{L}$. We now extend propositional tableaux so we can prove that ϕ is a tautological consequence of a decidable set T of formulas $\mathcal{L}$. For every $\psi \in T$ we add a unary *axiom rule*:

$$\frac{}{\psi}\quad (Ax)$$

which means that we can extend a branch of a tableau at an arbitrary position with the assumption ψ. Note that the decidability of T is crucial for this because we can use the axiom rule only if $\psi \in T$. Without decidability we would not be able to recognize whether a given tree of signed formulas is a legal tableau or not.

3.5.2 Proofs with propositional tableaux with axioms. A propositional tableau π for a sequence of signed formulas Δ which possibly uses axiom rules from T is a *propositional tableau for Δ from axioms T*. If π is closed then we assert this by writing $\pi : T \vdash_p [\Delta]$. Note that if $T = \emptyset$ then we have $\pi : \emptyset \vdash_p [\Delta] \Leftrightarrow \pi \!: \vdash_p [\Delta]$. When $\Delta \equiv \phi *$ then we say that π *propositionally proves ϕ from axioms T* and write it as $\pi : T \vdash_p \phi$. We use abbreviations similar to those discussed in Par. 3.2.4 also for the proofs from axioms.

3.5.3 Theorem (Admissible expansion rules in propositional tableaux with axioms). *All rules proved admissible for propositional tableaux in Sect. 3.3 are also admissible in propositional tableaux with axioms.*

Proof. Inspection of the proofs of admissibility of expansion rules in Sect. 3.3 reveals that the proofs remain correct also for tableaux with axiom rules because their presence does not affect the proofs of admissibility. □

3.5.4 Syntactic weakening lemma. *If $S \subseteq T$ then*

$$S \vdash_p \phi \Rightarrow T \vdash_p \phi \ .$$

Proof. Assume $\pi : S \vdash_p \phi$. Every every axiom rule for $\psi \in S$ used in π is also an axiom rule for $\psi \in T$ and so $\pi : T \vdash_p \phi$. □

3.5.5 Theorem on syntactic compactness.

$$T \vdash_p \phi \Rightarrow S \vdash_p \phi \text{ for a finite } S \subseteq T.$$

Proof. Assume $\pi : T \vdash_p \phi$ and construct the finite set S to consist of all conclusions ψ of axiom rules for $\psi \in T$ used in the tableau π. The same rules are axiom rules for $\psi \in S$ and so $\pi : S \vdash_p \phi$. □

3.5.6 Deduction theorem. For a finite set of formulas S:

$$S \vdash_p \phi \Leftrightarrow \ \vdash_p \bigwedge S \to \phi \ .$$

Proof. Let $S = \{\psi_1, \ldots, \psi_n\}$. In the direction $(\Rightarrow)$ assume $\pi : S \vdash_p \phi$, i.e. $\pi : S \vdash_p [\phi *]$. If $n = 0$ then $\bigwedge S \equiv \top$ and we derive $\vdash_p \top \to \phi$ as follows:

$$\begin{array}{cc} \multicolumn{2}{c}{\top \to \phi*} \\ \hline \phi* & (\to_2*) \\ \multicolumn{2}{c}{\cdots\pi\cdots} \end{array}$$

If $n > 0$ then denote by π_1 the tableau formed from π by deleting all expansions by axiom rules. We clearly have $\pi_1\colon \vdash_p [\phi*, \psi_1, \ldots, \psi_n]$. which is shown in the following on the left:

$$\begin{array}{c} \phi* \\ \psi_1 \\ \vdots \\ \psi_n \\ \hline \cdots\pi_1\cdots \end{array} \quad \Rightarrow \quad \begin{array}{cc} \multicolumn{2}{c}{\psi_1 \wedge \cdots \wedge \psi_n \to \phi*} \\ \hline \phi* & (\to_2*) \\ \psi_1 \wedge \cdots \wedge \psi_n & (\to_1*) \\ \psi_1 & (G\wedge_1) \\ \vdots & \\ \psi_n & (G\wedge_n) \\ \multicolumn{2}{c}{\cdots\pi_1\cdots} \end{array}$$

We form the closed tableau on the right which is expanded with the help of the generalized flatten rules (see Thm. 3.3.2) and so $\vdash_p \bigwedge S \to \phi$.

In the direction $(\Leftarrow)$ assume $\pi\colon \vdash_p [\psi_1 \wedge \cdots \wedge \psi_n \to \phi*]$. We can assume by inversion that the first two expansions in π are by $(\to_i*)$ flatten rules such that the goal $\bigwedge S \to \phi*$ is not used in π_1. If $n = 0$ then the situation is shown on the left:

$$\begin{array}{cc} \multicolumn{2}{c}{\top \to \phi*} \\ \hline \phi* & (\to_2*) \\ \top & (\to_1*) \\ \multicolumn{2}{c}{\cdots\pi_1\cdots} \end{array} \quad \Rightarrow \quad \begin{array}{c} \phi* \\ \hline \cdots\pi_1\cdots \end{array}$$

and we can form a closed tableau π for $\phi*$ because the assumption $\top$ cannot be used in π_1. If $n > 0$ then we can assume that the inversion of $(\to_i*)$ flatten rules is followed by by $n-1$ inversions of $(\wedge)$ flatten rules which is shown in the following on the left:

$$\begin{array}{cc} \multicolumn{2}{c}{\psi_1 \wedge \cdots \wedge \psi_n \to \phi*} \\ \hline \phi* & (\to_2*) \\ \psi_1 \wedge \cdots \wedge \psi_n & (\to_1*) \\ \psi_1 & (\wedge_1) \\ \psi_2 \wedge \cdots \wedge \psi_n & (\wedge_2*) \\ \vdots & \\ \psi_n & (\wedge_2) \\ \multicolumn{2}{c}{\cdots\pi_1\cdots} \end{array} \quad \Rightarrow \quad \begin{array}{cc} \multicolumn{2}{c}{\phi*} \\ \hline \psi_1 & (Ax) \\ \cdots & \\ \psi_n & (Ax) \\ \multicolumn{2}{c}{\cdots\pi_1\cdots} \end{array}$$

We form the closed tableau on the right which is expanded with the help of axiom rules from S and in which we have omitted the assumptions $\psi_i \wedge \cdots \wedge \psi_n$ for $1 \leq i < n$ because they cannot be used in π_1. We thus have $S \vdash_p \phi$. $\square$

3.5.7 Theorem (Introduction/elimination of axioms).

$$T \vdash_p \phi \Leftrightarrow \; \vdash_p \bigwedge S \to \phi$$

for a finite subset S of T.

Proof. From $T \vdash_p \phi$ we obtain $S \vdash_p \phi$ for a finite subset S of T by Thm. 3.5.5. The opposite direction follows from Lemma 3.5.4 and we have $S \vdash_p \phi$ iff $\vdash_p \bigwedge S \to \phi$ by Thm. 3.5.6. □

3.5.8 Corollary (Soundness and completeness of propositional tableaux). *For any fromula ϕ and set of formulas T there is in each direction a finite subset S of T such that the following holds:*

$$\begin{tikzcd}
T \models_p \phi \arrow[d, Rightarrow, shift left, "3.4.4"] & & T \vdash_p \phi \arrow[d, Rightarrow, shift left, "3.5.5"] \\
S \models_p \phi \arrow[u, Rightarrow, shift left, "3.4.3"] \arrow[d, Leftrightarrow, "3.4.2"] & & S \vdash_p \phi \arrow[u, Rightarrow, shift left, "3.5.4"] \arrow[d, Leftrightarrow, "3.5.6"] \\
\models_p \bigwedge S \to \phi \arrow[rr, Rightarrow, shift left, "3.2.6"] & & \vdash_p \bigwedge S \to \phi \arrow[ll, Rightarrow, shift left, "3.2.5"]
\end{tikzcd}$$

□

3.5.9 Semidecidability of tautological consequence revisited. One of the consequences of the Corollary 3.5.8 is that we can semidecide the relation $T \models_p \phi$ by tableaux instead of testing for tautologies as outlined in Par. 3.4.7. We do this by constructing possibly infinite branches which are *axiomatically complete*, i.e. which apply the axiom rules for all axioms from T.

In order to decide the relation $T \models_p \phi$ we enumerate the axioms of T into a sequence ψ_1, ψ_2, …, and construct a propositional tableau for the goal $\phi*$. We select an open propositionally complete branch of it (if any) and we extend it by an axiom rule by taking ψ_1 into assumptions. We then construct a propositional tableau under the assumption. We select a not closed propositionally complete branch again (if any) and expand it with the axiom rule for ψ_2. We continue in this way in the hope of closing the branch. If this happens we apply the same procedure of systematically applying one axiom after another to the remaining open branches. If all branches close then $T \models_p \phi$ holds.

Otherwise, if T is finite we stop with at least one open branch Δ which is both propositionally and axiomatically complete. The branch is finite. If T is infinite we will go on extending a branch forever because there is at least one infinite branch which is open and propositionally and axiomatically complete.

In both cases there is a propositional interpretation $\mathcal{I}$ constructed by collecting all propositional atoms in the assumptions. We have proved in Par. 3.2.8 that we have $\mathcal{I} \not\models \phi$ and $\mathcal{I} \models \phi_1$ for every assumption ϕ_1 in the branch. Since the branch contains all axioms T as assumptions we have in particular $\mathcal{I} \models T$. But this means that we have $T \not\models_p \phi$. Note that we can effectively determine this fact only when T is finite.

4. Identity Logic

In this chapter we investigate formulas always true on the strength of propositional logic and of properties of identity.

4.1 Some Syntactic Concepts

4.1.1 Free terms. *Free terms* of a set of formulas S of a language $\mathcal{L}$ are terms occurring outside of quantifiers in the formulas of S. This is made precise by a metamathematical function $FT(\alpha)$ defined on terms, formulas, and sets of formulas to yield the set of free terms of α. The function FT satisfies:

$$\begin{aligned}
FT(x) &= \{x\} \\
FT(f(\tau_1, \ldots, \tau_n)) &= \{f(\tau_1, \ldots, \tau_n)\} \cup FT(\tau_1) \cup \ldots \cup FT(\tau_n) \\
FT(\tau_1 = \tau_2) &= FT(\tau_1) \cup FT(\tau_2) \\
FT(P(\tau_1, \ldots, \tau_n)) &= FT(\tau_1) \cup \ldots \cup FT(\tau_n) \\
FT(\forall x \phi) &= \emptyset \\
FT(\exists x \phi) &= \emptyset \\
FT(\top) &= \emptyset \\
FT(\bot) &= \emptyset \\
FT(\neg \phi) &= FT(\phi) \\
FT(\phi_1 \vee \phi_2) &= FT(\phi_1) \cup FT(\phi_2) \\
FT(\phi_1 \wedge \phi_2) &= FT(\phi_1) \cup FT(\phi_2) \\
FT(\phi_1 \rightarrow \phi_2) &= FT(\phi_1) \cup FT(\phi_2) \\
FT(\phi_1 \leftrightarrow \phi_2) &= FT(\phi_1) \cup FT(\phi_2) \\
FT(T) &= \bigcup \{FT(\phi) \mid \phi \in T\} \ .
\end{aligned}$$

4.2 Quasitautological Consequence

4.2.1 Structures. A structure $\mathcal{M}$ for a language $\mathcal{L}$ is given by

1. a non-empty set D, called the *domain* of the structure,
2. for every n-ary function symbol f of $\mathcal{L}$ an n-ary function $f^{\mathcal{M}}$ over D, i.e. $f^{\mathcal{M}} : D^n \mapsto D$, called the *interpretation of* f,
3. for every n-ary predicate symbol P of $\mathcal{L}$ a subset $P^{\mathcal{M}}$ of D^n called the *interpretation of* P.

Here we define $D^0 = \{\emptyset\}$ and identify nullary functions over D with elements of D. Thus for every constant c of $\mathcal{L}$ we have $c^{\mathcal{M}} \in D$ and for every propositional constant P of $\mathcal{L}$ we have $P^{\mathcal{M}} \subseteq \{\emptyset\}$. We agree to identify the value $P^{\mathcal{M}} = \emptyset$ with falsehood and the value $P^{\mathcal{M}} = \{\emptyset\}$ with truth.

Structures are mathematical objects (triples of sets) whose purpose is to assign meaning to terms and formulas of $\mathcal{L}$. A structure $\mathcal{M}$ is *finite* if its domain D is a finite set and *infinite* otherwise. $\mathcal{M}$ is a *numeric* structure if its domain is a subset of natural numbers.

4.2.2 Assignments. For a structure $\mathcal{M}$ for $\mathcal{L}$ with the domain D we call a function a from $\mathbb{N}$ to D *assignment in* $\mathcal{M}$. The idea is that the assignment a assigns the value $a(i) \in D$ to the variable v_i.

4.2.3 Identity interpretations. An *identity interpretation* $\mathcal{I}$ *for* $\mathcal{L}$ is a triple $\langle Q, \mathcal{M}, a \rangle$ where Q, called the *quantifier set*, is a subset of quantifier formulas of $\mathcal{L}$, $\mathcal{M}$ is a structure for $\mathcal{L}$ and a is an assignment in $\mathcal{M}$. An identity $\mathcal{I}$ is *finite* if its structure is finite and *numeric* if its structure is numeric.

Identity interpretations uniquely determine the meaning of terms and formulas as shown in the following two paragraphs.

4.2.4 Denotation of terms. For a given identity interpretation $\mathcal{I} = \langle Q, \mathcal{M}, a \rangle$ for $\mathcal{L}$ with D the domain of $\mathcal{M}$ we assign to every term τ of $\mathcal{L}$ its *denotation*, designated as $\tau^{\mathcal{I}}$, to be the element of D satisfying the following:

$$\begin{aligned} v_i^{\mathcal{I}} &= a(i) \\ f(\tau_1, \ldots, \tau_n)^{\mathcal{I}} &= f^{\mathcal{M}}(\tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}}) \quad f \text{ is } n\text{-ary function symbol of } \mathcal{L}. \end{aligned}$$

If f is a constant symbol, i.e. a nullary function symbol, then we abbreviate $f^{\mathcal{M}}()$ to $f^{\mathcal{M}}$.

The identity interpretation $\mathcal{I}$ is *canonical* if it is a numeric interpretation and for every element d in the domain of its structure we have $d = \tau^{\mathcal{I}}$ for some term τ.

4.2.5 Satisfaction relation for identity interpretations. Let $\mathcal{I} = \langle Q, \mathcal{M}, a \rangle$ be an identity interpretation for $\mathcal{L}$. For every formula ϕ of $\mathcal{L}$ we define the unary relation $\mathcal{I} \vDash \phi$, read as $\mathcal{I}$ *satisfies* ϕ, to be similar to the satisfaction relation for propositional interpretations (see Par. 3.1.2) when ϕ is a propositional formula. If ϕ is a propositional atom then we define

$$\begin{aligned}
\mathcal{I} &\models \forall x\phi \Leftrightarrow \forall x\phi \in Q \\
\mathcal{I} &\models \exists x\phi \Leftrightarrow \exists x\phi \in Q \\
\mathcal{I} &\models \tau_1 = \tau_2 \Leftrightarrow \tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}} \\
\mathcal{I} &\models P(\tau_1, \ldots, \tau_n) \Leftrightarrow \langle \tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}} \rangle \in P^{\mathcal{M}} \ .
\end{aligned}$$

Here the 'zero-tuple' $\langle\rangle$ is defined as the empty set $\emptyset$. This means that for a propositional constant P we have $\mathcal{I} \models P$ iff $P^{\mathcal{M}} = \{\emptyset\}$. This should explain why we have identified the set $\{\emptyset\}$ with the truth.

Assignments in $\mathcal{I}$ are used only in atomic formulas which obtain the meaning from the interpretation of function and predicate symbols specified by the points (2) and (3) in Par. 4.2.1. Specifically, the meaning of identity $\tau_1 = \tau_2$ is determined as the identity of the denotations $\tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}}$. The reader will note that the symbol of identity in the formula $\tau_1 = \tau_2$ is just a symbol whereas the same symbol in $\tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}}$ stands for the relation of identity over the domain of $\mathcal{I}$. The meaning of quantifier formulas is determined from the quantifier set Q similarly as the meaning of propositional atoms is determined from propositional interpretations.

4.2.6 Lemma (Reduction to propositional interpretations). *To every identity interpretation $\mathcal{I}$ for $\mathcal{L}$ there is an equivalent propositional interpretation $\mathcal{J}$.*

Proof. Take an identity interpretation $\mathcal{I} = \langle Q, \mathcal{M}, a\rangle$ for $\mathcal{L}$. We construct the propositional interpretation $\mathcal{J}$ as follows:

$$\mathcal{J} = \{\psi \mid \mathcal{I} \models \psi \text{ for propositional atoms } \psi\} \ .$$

We prove the equivalence by induction on the construction of ϕ. If ϕ is a propositional atom then $\mathcal{J} \models \phi$ iff $\phi \in \mathcal{J}$ iff $\mathcal{I} \models \phi$. If $\phi \equiv \phi_1 \lor \phi_2$ then we have $\mathcal{J} \models \phi_1 \lor \phi_2$ iff $\mathcal{J} \models \phi_1$ or $\mathcal{J} \models \phi_2$ iff, by IH, $\mathcal{I} \models \phi_1$ or $\mathcal{I} \models \phi_2$ iff $\mathcal{I} \models \phi_1 \lor \phi_2$. The remaining cases are similar. □

4.2.7 Equality axioms. Let $\mathcal{L}$ be a first-order language. For all terms τ_1, $\ldots, \tau_n, \rho_1, \ldots, \rho_n$ the following formulas, designated by Eq, are called *equality axioms for $\mathcal{L}$*:

$$\tau_1 = \tau_1 \tag{1}$$

$$\tau_1 = \tau_2 \to \tau_2 = \tau_1 \tag{2}$$

$$\tau_1 = \tau_2 \land \tau_2 = \tau_3 \to \tau_1 = \tau_3 \tag{3}$$

$$\begin{gathered}\tau_1 = \rho_1 \land \cdots \land \tau_n = \rho_n \to f(\tau_1, \ldots, \tau_n) = f(\rho_1, \ldots, \rho_n) \\ f \text{ is } n\text{-ary function symbol, } n > 0\end{gathered} \tag{4}$$

$$\begin{gathered}\tau_1 = \rho_1 \land \cdots \land \tau_n = \rho_n \land P(\tau_1, \ldots, \tau_n) \to P(\rho_1, \ldots, \rho_n) \\ P \text{ is } n\text{-ary predicate symbol, } n > 0.\end{gathered} \tag{5}$$

Formulas (1) are axioms of *reflexivity*, (2) are axioms of *symmetry*, (3) are axioms of *transitivity*, (4) and (5) are axioms of *function* and *predicate substitution* respectively.

We designate by Eq^T the *restriction* of equality axioms to the free terms of a set of formulas T, i.e.:

$$Eq^T = \{\phi \mid \phi \in Eq \land FT(\phi) \subseteq FT(T)\} \ .$$

4.2.8 Lemma (Expansion of propositional interpretations). *To every set of formulas T of $\mathcal{L}$ and every propositional interpretation $\mathcal{I}$ for $\mathcal{L}$ such that $\mathcal{I} \vDash Eq^T$ there is a T-equivalent canonical identity interpretation $\mathcal{J}$. $\mathcal{J}$ is finite if the set $FT(T)$ is finite. Every element of the domain of the structure of $\mathcal{J}$ is denoted by a term of $FT(T)$ if the last set is not empty.*

Proof. Take any T and any propositional interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash Eq^T$. We wish to define the identity interpretation $\mathcal{J} = \langle Q, \mathcal{M}, a\rangle$. We define the quantifier set Q as follows:

$$Q = \{\psi \mid \psi \text{ is a quantifier formula and } \psi \in \mathcal{I} \ .\}$$

We enumerate the set $\mathcal{T} = FT(T)$ by a possibly finite or even empty sequence:

$$\sigma_0,\ \sigma_1,\ \sigma_2,\ \ldots$$

Clearly, the sequence has as many elements as is the cardinality of $\mathcal{T}$. We define the domain D of the structure $\mathcal{M}$ with the help of a *representant* function r mapping the terms of $\mathcal{L}$ to $\mathbb{N}$ by

$$r(\tau) = \min\{i \mid \tau \in \mathcal{T} \Rightarrow \mathcal{I} \vDash \tau = \sigma_i\} \ .$$

This is a legal definition only if the argument set to min is non-empty for every τ. That this is so can be seen by considering two cases. If $\tau \notin \mathcal{T}$ then the argument to min is $\mathbb{N}$ and so $r(\tau) = 0$. If $\tau \in \mathcal{T}$ then we have $\tau \equiv \sigma_i$ for some i and, since the reflexivity axiom $\tau = \sigma_i$ is in Eq^T, we have $\mathcal{I} \vDash \tau = \sigma_i$. Hence the argument to min contains i. We now define the domain D of $\mathcal{J}$ as the range of the function r:

$$D = \{r(\tau) \mid \tau \text{ is a term of } \mathcal{L}\} \ .$$

We have $D \subseteq \mathbb{N}$ and if $\mathcal{T} = \emptyset$ then $D = \{0\}$. Otherwise $\sigma_0 \in \mathcal{T}$ and $r(\sigma_0) = 0 \in D$. Note that D has no more elements than $\mathcal{T}$ if $\mathcal{T} \neq \emptyset$ is finite. We prove

$$\tau_1, \tau_2 \in \mathcal{T} \Rightarrow (r(\tau_1) = r(\tau_2) \Leftrightarrow \mathcal{I} \vDash \tau_1 = \tau_2) \ . \tag{1}$$

Assume $\tau_1, \tau_2 \in \mathcal{T}$. In the direction ($\Rightarrow$) also assume $r(\tau_1) = r(\tau_2)$. From this we get $\mathcal{I} \vDash \tau_1 = \sigma_i$ and $\mathcal{I} \vDash \tau_2 = \sigma_i$ for some i. Since $\sigma_i \in \mathcal{T}$, the symmetry axiom $\tau_2 = \sigma_i \to \sigma_i = \tau_2$ and the transitivity axiom $\tau_1 = \sigma_i \to \sigma_i = \tau_2 \to$

$\tau_1 = \tau_2$ are in the set Eq^T. Thus also $\mathcal{I} \vDash \tau_1 = \tau_2$ holds. In the direction ($\Leftarrow$) assume $\mathcal{I} \vDash \tau_1 = \tau_2$. We have $r(\tau_1) = i$ for the least i such that $\sigma_i \in \mathcal{T}$ and $\mathcal{I} \vDash \tau_1 = \sigma_i$ holds. By appropriate symmetry and transitivity axioms in Eq^T we obtain $\mathcal{I} \vDash \tau_2 = \sigma_i$. We have $r(\tau_2) = j$ for the least $j \leq i$ such that $\sigma_j \in \mathcal{T}$ and $\mathcal{I} \vDash \tau_2 = \sigma_j$ holds. If $j < i$ then by symmetry and transitivity axioms in Eq^T we would get $\mathcal{I} \vDash \tau_1 = \sigma_j$ contradicting the definition of i. Hence $i = j$ and so $r(\tau_1) = r(\tau_2)$.

We now take any n-ary function symbol f of $\mathcal{L}$ and define the interpretation $f^{\mathcal{M}} : D^n \mapsto D$ of f as follows:

$$f^{\mathcal{M}}(d_1, \ldots, d_n) = r(f(\tau_1, \ldots, \tau_n)) \quad \text{where } r(\tau_1) = d_1, \ldots, r(\tau_n) = d_n.$$

We must convince ourselves first that this is a legal definition. If $n = 0$ then we have $f^{\mathcal{M}} = r(f)$. If $n > 0$ and $\mathcal{T} = \emptyset$ then, since $D = \{0\}$, we always have $r(f(\tau_1, \ldots, \tau_n)) = 0$. If $n > 0$ and $\mathcal{T} \neq \emptyset$ then, since for every $d \in D$ we have $d = r(\rho)$ for some $\rho \in \mathcal{T}$, the definition determines $f^{\mathcal{M}}(d_1, \ldots, d_n)$ uniquely if the following holds:

$$r(\tau_1) = r(\rho_1) \wedge \cdots \wedge r(\tau_n) = r(\rho_n) \rightarrow r(f(\tau_1, \ldots, \tau_n)) = r(f(\rho_1, \ldots, \rho_n))$$

for every $\tau_1, \ldots, \tau_n, \rho_1, \ldots, \rho_n$ in $\mathcal{T}$. From the assumptions we obtain $\mathcal{I} \vDash \tau_1 = \rho_1, \ldots, \mathcal{I} \vDash \tau_n = \rho_n$ by (1). Since the function substitution axiom 4.2.7(4) is in Eq^T and $\mathcal{I} \vDash Eq^T$, we have $\mathcal{I} \vDash f(\tau_1, \ldots, \tau_n) = f(\rho_1, \ldots, \rho_n)$ and so $r(f(\tau_1, \ldots, \tau_n)) = r(f(\rho_1, \ldots, \rho_n))$ by (1).

For every n-ary predicate symbol P of $\mathcal{L}$ we define its interpretation $P^{\mathcal{M}} \subseteq D^n$:

$$P^{\mathcal{M}} = \{\langle r(\tau_1), \ldots, r(\tau_n)\rangle \mid \mathcal{I} \vDash P(\tau_1, \ldots, \tau_n) \text{ and } \tau_1, \ldots, \tau_n \in \mathcal{T}\}\ .$$

We prove

$$\rho_1, \ldots, \rho_n \in \mathcal{T} \Rightarrow (\mathcal{I} \vDash P(\rho_1, \ldots, \rho_n) \Leftrightarrow \langle r(\rho_1), \ldots, r(\rho_n)\rangle \in P^{\mathcal{M}}) \qquad (2)$$

by taking any $\rho_1, \ldots, \rho_n \in \mathcal{T}$. In the direction ($\Rightarrow$) the property follows from the definition of $P^{\mathcal{M}}$ (even when $n = 0$ because $\rho_1, \ldots, \rho_n \in \mathcal{T}$ holds vacuously and we then have $P^{\mathcal{M}} = \{\langle\rangle\} = \{\emptyset\}$). In the direction ($\Leftarrow$) assume $\langle r(\rho_1), \ldots, r(\rho_n)\rangle \in P^{\mathcal{M}}$. If $n = 0$ then this means $\langle\rangle \in P^{\mathcal{M}}$ and so $\mathcal{I} \vDash P$. If $n > 0$ we have $r(\tau_1) = r(\rho_1), \ldots, r(\tau_n) = r(\rho_n)$ and $\mathcal{I} \vDash P(\tau_1, \ldots, \tau_n)$ for some $\tau_1, \ldots, \tau_n \in \mathcal{T}$. We get $\mathcal{I} \vDash \tau_1 = \rho_1, \ldots, \mathcal{I} \vDash \tau_n = \rho_n$ from (1). Since the predicate substitution axiom 4.2.7(5) $\in Eq^T$ and $\mathcal{I} \vDash Eq^T$, we obtain $\mathcal{I} \vDash P(\rho_1, \ldots, \rho_n)$.

The structure $\mathcal{M}$ is now defined and we define the assignment $a : \mathbb{N} \mapsto D$ by $a(i) = r(v_i)$. Thus also the identity interpretation $\mathcal{J}$ is defined and we prove:

$$\tau \in \mathcal{T} \Rightarrow \tau^{\mathcal{J}} = r(\tau) \qquad (3)$$

by induction on the construction of the term τ. So assume $\tau \in \mathcal{T}$ and continue by the case analysis of τ. If $\tau \equiv v_i$ then $v_i^{\mathcal{J}} = a(i) = r(v_i)$. If $\tau \equiv f(\tau_1, \ldots, \tau_n)$ then

$$f(\tau_1, \ldots, \tau_n)^{\mathcal{J}} = f^{\mathcal{M}}(\tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}}) \overset{\text{IH}}{=} \\ f^{\mathcal{M}}(r(\tau_1), \ldots, r(\tau_n)) = r(f(\tau_1, \ldots, \tau_n)) \ .$$

Note that $\mathcal{J}$ is a canonical interpretation because if $d \in D$ then $d = r(\tau) = \tau^{\mathcal{J}}$ for some τ which is one of $\sigma_i \in FT(T)$ if the last set is not empty.

The lemma will be proved when we prove for every formula $\phi \in T$ the property:

$$\mathcal{J} \vDash \phi \Leftrightarrow \mathcal{I} \vDash \phi \ .$$

So take any $\phi \in T$. Note that $FT(\phi) \subseteq \mathcal{T}$ and proceed by induction on the construction of ϕ. If ϕ is a quantifier formula then we have $\mathcal{J} \vDash \phi$ iff $\phi \in Q$ iff $\phi \in \mathcal{I}$ iff $\mathcal{I} \vDash \phi$.

If $\phi \equiv \tau_1 = \tau_2$ then we must have $\mathcal{T} \neq \emptyset$ and so $\mathcal{J} \vDash \tau_1 = \tau_2$ iff $\tau_1^{\mathcal{J}} = \tau_2^{\mathcal{J}}$ iff, by (3), $r(\tau_1) = r(\tau_2)$ iff, by (1), $\mathcal{I} \vDash \tau_1 = \tau_2$.

If $\phi \equiv P(\tau_1, \ldots, \tau_n)$ then we must have $\mathcal{T} \neq \emptyset$ and so $\mathcal{J} \vDash P(\tau_1, \ldots, \tau_n)$ iff $\langle \tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}} \rangle \in P^{\mathcal{N}}$ iff, by (3), $\langle r(\tau_1), \ldots, r(\tau_n) \rangle \in P^{\mathcal{N}}$ iff, by (2), $\mathcal{I} \vDash P(\tau_1, \ldots, \tau_n)$.

If $\phi \equiv \phi_1 \vee \phi_2$ then $\mathcal{J} \vDash \phi_1 \vee \phi_2$ iff $\mathcal{J} \vDash \phi_1$ or $\mathcal{J} \vDash \phi_2$ iff, by IH, $\mathcal{I} \vDash \phi_1$ or $\mathcal{I} \vDash \phi_2$ iff $\mathcal{I} \vDash \phi_1 \vee \phi_2$. The remaining cases are similar. □

4.2.9 Quasitautological consequence. Fix a language $\mathcal{L}$ and let T be a set of formulas from $\mathcal{L}$. We define the relation ϕ *is a quasitautological consequence of* T, in symbols $T \vDash_i \phi$, as follows:

$$T \vDash_i \phi \Leftrightarrow (\mathcal{I} \vDash T \Rightarrow \mathcal{I} \vDash \phi) \text{ for all identity interpretations } \mathcal{I}.$$

We write $\vDash_i \phi$ for $\emptyset \vDash_i \phi$ and such a ϕ is *quasitautology*. Note that we have

$$\vDash_i \phi \Leftrightarrow \mathcal{I} \vDash \phi \text{ for all identity interpretations } \mathcal{I}.$$

4.2.10 Lemma. $\vDash_i$ *Eq*.

Proof. Take any identity interpretation $\mathcal{I} = \langle Q, \mathcal{M}, a \rangle$ for $\mathcal{L}$. We wish to prove $\mathcal{I} \vDash \phi$ for an equality axiom ϕ. If ϕ is a predicate substitution axiom 4.2.7(5) then $\mathcal{I} \vDash \phi$ iff from $\mathcal{I} \vDash \tau_1 = \rho_1, \ \ldots, \ \mathcal{I} \vDash \tau_n = \rho_n$, and $\mathcal{I} \vDash P(\tau_1, \ldots, \tau_n)$ follows $\mathcal{I} \vDash P(\rho_1, \ldots, \rho_n)$. If the assumptions are satisfied we have $\tau_1^{\mathcal{I}} = \rho_1^{\mathcal{I}}, \ \ldots, \ \tau_n^{\mathcal{I}} = \rho_n^{\mathcal{I}}$, and $\langle \tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}} \rangle \in P^{\mathcal{M}}$ and so $\langle \rho_1^{\mathcal{I}}, \ldots, \rho_n^{\mathcal{I}} \rangle \in P^{\mathcal{M}}$, i.e. $\mathcal{J} \vDash P(\rho_1, \ldots, \rho_n)$.

The remaining cases are similar. □

4.2.11 Theorem (Quasitautological reduction).

$$T \vDash_i \phi \Rightarrow T, Eq^{T \cup \{\phi\}} \vDash_p \phi \tag{1}$$

$$T \vDash_i \phi \Leftrightarrow T, Eq \vDash_p \phi\ . \tag{2}$$

Proof. (1): Assume $T \vDash_i \phi$ and take any propositional interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash T \cup Eq^{T \cup \{\phi\}}$ holds. From Lemma 4.2.8 we obtain a $T \cup \{\phi\}$-equivalent identity interpretation $\mathcal{J}$ and so $\mathcal{J} \vDash T$. We then get $\mathcal{J} \vDash \phi$ from the assumption and $\mathcal{I} \vDash \phi$ from the equivalence.

(2): The direction ($\Rightarrow$) follows from (1) by weakening. In the direction ($\Leftarrow$) assume $T, Eq \vDash_p \phi$ and take any identity interpretation $\mathcal{I}$ for $\mathcal{L}$ such that $\mathcal{I} \vDash T$ holds. We have $\mathcal{I} \vDash Eq$ by Lemma 4.2.10. From Lemma 4.2.6 we obtain an equivalent propositional interpretation $\mathcal{J}$. We thus have $\mathcal{J} \vDash T \cup Eq$, we get $\mathcal{J} \vDash \phi$ from the assumption, and $\mathcal{I} \vDash \phi$ from the equivalence. □

4.2.12 Decidability of quasitautologies. One of the consequences of the Reduction theorem is that the predicate $\vDash_i \phi$ of being a quasitautology is decidable. We namely have

$$\vDash_i \phi \Leftrightarrow\ \vDash_p \bigwedge Eq^{\{\phi\}} \rightarrow \phi\ . \tag{1}$$

First note that the set $FT(\phi)$ is finite and so there are only finitely many equality axioms with free terms from $FT(\phi)$ which means that the set $Eq^{\{\phi\}}$ is finite. We have $\vDash_i \phi$ iff, by 4.2.11(2), $Eq \vDash_p \phi$ iff, by Semantic deduction lemma 3.4.2, $\vDash_p \bigwedge Eq^{\{\phi\}} \rightarrow \phi$.

4.2.13 Theorem. $T \vDash_p \phi \Rightarrow T \vDash_i \phi$.

Proof. If $T \vDash_p \phi$ holds then we have $T, Eq \vDash_p \phi$ by weakening and $T \vDash_i \phi$ by 4.2.11(2). □

4.2.14 Theorem. *For every set of formulas T of some $\mathcal{L}$ and for every identity interpretation $\mathcal{I}$ for $\mathcal{L}$ there is a T-equivalent canonical identity interpretation $\mathcal{J}$. Every element of the domain of the structure of $\mathcal{J}$ is denoted by a term of $FT(T)$ if the last set is not empty.*

Proof. For a given identity interpretation $\mathcal{I}$ for $\mathcal{L}$ we obtain an equivalent propositional interpretation $\mathcal{I}_1$ by Lemma 4.2.6. By Lemma 4.2.10 we have $\mathcal{I} \vDash Eq$ and so $\mathcal{I}_1 \vDash Eq^T$ by the equivalence. From Lemma 4.2.8 we obtain a T-equivalent canonical identity interpretation $\mathcal{J}$ such that every element of the domain of the structure of $\mathcal{J}$ is denoted by a term of $FT(T)$ if the last set is not empty. □

4.3 Identity Tableaux

4.3.1 Identity tableau expansion rules. Fix a language $\mathcal{L}$. All expansion rules for identity tableaux are unary with premises in assumptions. In the following $\tau, \tau_1, \ldots, \tau_n, \rho_1, \ldots, \rho_n$ are arbitrary terms. *Reflexivity, symmetry,* and *transitivity* rules are in that order:

$$\frac{}{\tau = \tau}\,(\mathit{Refl}) \quad \frac{\tau_1 = \tau_2}{\tau_2 = \tau_1}\,(\mathit{Sym}) \quad \frac{\tau_1 = \tau_2 \;\; \tau_2 = \tau_3}{\tau_1 = \tau_3}\,(\mathit{Trans})$$

Function substitution rules are for every n-ary function symbol f of $\mathcal{L}$ with $n > 0$ as follows:

$$\frac{\tau_1 = \rho_1 \quad \cdots \quad \tau_n = \rho_n}{f(\tau_1, \ldots, \tau_n) = f(\rho_1, \ldots, \rho_n)}\,(\mathit{Fsub})$$

Predicate substitution rules are for every n-ary predicate symbol P of $\mathcal{L}$ with $n > 0$ as follows:

$$\frac{\tau_1 = \rho_1 \quad \cdots \quad \tau_n = \rho_n \quad P(\tau_1, \ldots, \tau_n)}{P(\rho_1, \ldots, \rho_n)}\,(\mathit{Psub})$$

We can see that every identity rule *corresponds* to exactly one equality axiom.

4.3.2 Proofs with identity tableaux. An identity tableau π for a sequence of signed formulas Δ from axioms in T is called an *identity tableau for Δ from axioms T*. If π is closed then we assert this by writing $\pi : T \vdash_i [\Delta]$. When $\Delta \equiv \phi*$ then we say that π *is an identity tableau proving ϕ from axioms T* and write it as $\pi : T \vdash_i \phi$. We use additional abbreviations similar to those discussed in Par. 3.2.4.

4.3.3 Theorem (Admissible expansion rules in identity tableaux). *All rules proved admissible for propositional tableaux in Sect. 3.3 are also admissible in identity tableaux.*

Proof. Inspection of the proofs of admissibility of expansion rules in Sect. 3.3 reveals that the proofs remain correct also for identity tableaux, the presence of identity rules does not affect the proofs.

The only potential problem could be in the admissibility of cuts on propositional atoms (see Lemma 3.3.8) because we now have identity rules on propositional atoms $\tau_1 = \tau_2$ in assumptions. Fortunately, the elimination of cuts on such formulas removes the goals $\tau_1 = \tau_2*$ which are not affected by the identity expansion rules. □

4.3.4 Lemma. $\vdash_i Eq$.

Proof. An axiom of reflexivity 4.2.7(1) has the following proof:

$$\frac{\tau = \tau*}{\tau = \tau \quad (\mathit{Refl})}$$
$$\circ$$

An axiom of predicate substitution 4.2.7(5) has the following proof:

$$\begin{array}{c}
\tau_1 = \rho_1 \wedge \cdots \wedge \tau_n = \rho_n \wedge P(\tau_1, \ldots, \tau_n) \to P(\rho_1, \ldots, \rho_n)* \\
\hline
P(\rho_1, \ldots, \rho_n)* \qquad (\to_2 *) \\
\tau_1 = \rho_1 \wedge \cdots \wedge \tau_n = \rho_n \wedge P(\tau_1, \ldots, \tau_n) \qquad (\to_2 *) \\
\tau_1 = \rho_1 \qquad (G\wedge_1) \\
\cdots \\
\tau_n = \rho_n \qquad (G\wedge_n) \\
P(\tau_1, \ldots, \tau_n) \qquad (G\wedge_{n+1}) \\
P(\rho_1, \ldots, \rho_n) \qquad (Psub) \\
\circ
\end{array}$$

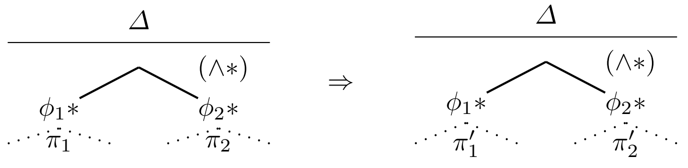
Remaining axioms have similar proofs. □

4.3.5 Lemma (Introduction of identity rules).

$$\pi : T, Eq \vdash_p [\Delta] \Rightarrow T \vdash_i [\Delta]$$

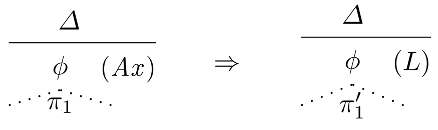
Proof. By induction on the structure of the tableau π. Assume $\pi : T, Eq \vdash_p [\Delta]$ and consider the form of π. If π is empty then we trivially have $T \vdash_i [\Delta]$.

If the first expansion in π is by a propositional rule, say $(\wedge *)$, then $\phi_1 \wedge \phi_2 * \in \Delta$ and π has a form shown in the following on the left:



We obtain identity tableaux π_1' and π_2' such that $\pi_1' : T \vdash_i [\Delta, \phi_1 *]$ and $\pi_2' : T \vdash_i [\Delta, \phi_2 *]$ by two IH's and construct the closed identity tableau for Δ from axioms T shown on the right. The remaining expansions by propositional rules and by axiom rules from T are similar.

If the first expansion in π is by an axiom rule for $\phi \in Eq$ then π has a form shown in the following on the left:



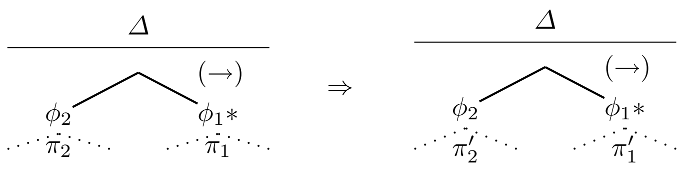
We obtain an identity tableau π_1' such that $\pi_1' : T \vdash_i [\Delta, \phi]$ by IH, note that $\vdash_i \phi$ by Lemma 4.3.4, and construct by a lemma rule the closed identity tableau for Δ from axioms T shown on the right. □

4.3.6 Lemma (Elimination of identity rules).

$$\pi : T \vdash_i [\varDelta] \Rightarrow T, Eq \vdash_p [\varDelta] \ .$$

Proof. By induction on the structure of the tableau π. Assume $\pi : T \vdash_i [\varDelta]$ and consider the form of π. If π is empty then we trivially have $T \vdash_p [\varDelta]$ and we obtain $T, Eq \vdash_p [\varDelta]$ by weakening.

If the first expansion in π is by a propositional rule, say $(\rightarrow)$, then $\phi_1 \rightarrow \phi_2 \in \varDelta$ and π has a form shown in the following on the left:



We obtain propositional tableaux π_1' and π_2' such that $\pi_1' : T, Eq \vdash_p [\varDelta, \phi_1 *]$ and $\pi_2' : T, Eq \vdash_p [\varDelta, \phi_2]$ by two IH's and construct the closed propositional tableau for $\varDelta$ from axioms T, Eq shown on the right. The remaining propositional and axiom expansions are similar.

If the first expansion in π is by a reflexivity rule then π has a form shown in the following on the left:

$$\cfrac{\varDelta}{\begin{array}{c}\tau = \tau \quad (\mathit{Refl}) \\ \pi_1\end{array}} \quad \Rightarrow \quad \cfrac{\varDelta}{\begin{array}{c}\tau = \tau \quad (Ax) \\ \pi_1'\end{array}}$$

We obtain a propositional tableau π_1' such that $\pi_1' : T, Eq \vdash_p [\varDelta, \tau = \tau]$ by IH and construct the closed propositional tableau for $\varDelta$ from axioms T, Eq shown on the right.

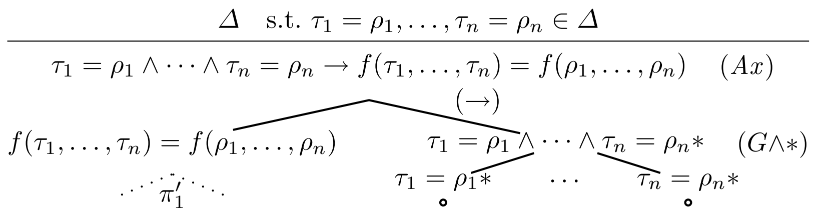
If the first expansion in π is by a function substitution rule then π has a following form:

$$\cfrac{\varDelta}{\begin{array}{c}f(\tau_1, \ldots, \tau_n) = f(\rho_1, \ldots, \rho_n) \quad (Fsub) \\ \pi_1\end{array}} \quad \text{where } \tau_1 = \rho_1, \ldots, \tau_n = \rho_n \in \varDelta$$

We obtain a propositional tableau π_1' such that

$$\pi_1' : T, Eq \vdash_p [\varDelta, f(\tau_1, \ldots, \tau_n) = f(\rho_1, \ldots, \rho_n)]$$

by IH and construct the following closed propositional tableau for $\varDelta$ from axioms T, Eq:


Δ s.t. $\tau_1 = \rho_1, \dots, \tau_n = \rho_n \in \Delta$
$\tau_1 = \rho_1 \land \dots \land \tau_n = \rho_n \to f(\tau_1, \dots, \tau_n) = f(\rho_1, \dots, \rho_n)$ (Ax)
$(\to)$
$f(\tau_1, \dots, \tau_n) = f(\rho_1, \dots, \rho_n)$
$\tau_1 = \rho_1 \land \dots \land \tau_n = \rho_n *$ $(G\land *)$
π_1
$\tau_1 = \rho_1 *$
$\dots$
$\tau_n = \rho_n *$

The remaining identity expansion rules are similar. □

4.3.7 Theorem (Introduction/elimination of identity rules).

$$T \vdash_i \phi \Leftrightarrow T, Eq \vdash_p \phi \ .$$

Proof. In the direction ($\Rightarrow$) assume $\pi : T \vdash_i \phi$, i.e $\pi : T \vdash_i [\phi *]$, and obtain $T, Eq \vdash_p [\phi *]$, i.e. $T, Eq \vdash_p \phi$, by Lemma 4.3.6.

In the direction ($\Leftarrow$) assume $T, Eq \vdash_p \phi$, i.e. $T, Eq \vdash_p [\phi *]$, and obtain $T \vdash_i [\phi *]$, i.e. $T \vdash_i \phi$, by Lemma 4.3.5. □

4.3.8 Corollary (Soundness and completeness of identity tableaux). *For any fromula ϕ and a set of formulas T from $\mathcal{L}$ we have*

$$\begin{array}{ccc} T \models_i \phi & & T \vdash_i \phi \\ \Updownarrow \text{ 4.2.11(2)} & & \Updownarrow \text{ 4.3.7} \\ T, Eq \models_p \phi & \Leftrightarrow \text{ 3.5.8} & T, Eq \vdash_p \phi \end{array}$$

□

4.3.9 Decidability of quasitautologies revisited. Property 4.2.12(1) reduces the problem of deciding whether a formula ϕ of some $\mathcal{L}$ is quasitautology to the problem of deciding whether the formula $\bigwedge Eq^{\{\phi\}} \to \phi$ is a tautology. This decision procedure can be called the method of *associates* because the conjunction of equality axioms $\bigwedge Eq^{\{\phi\}}$ is an *identity associate* of ϕ. Quantifier associates were introduced by R. Smullyan [24].

Lemma 4.2.6 and property 4.2.11(1) give us another method of deciding quasitautologies: test $\mathcal{J} \models \phi$ in all finite identity interpretations $\mathcal{J}$ for a finite language consisting of function and predicate symbols in ϕ. Note that the quantifier sets Q of such interpretations can be restricted to subsets of finitely many quantifier formulas among $FPA(\phi)$. Also note that there are only finitely many assignments a to be considered because the only values which matter are the finitely many values $a(i)$ where $v_i \in FT(\phi)$. The formula ϕ is a quasitautology iff $\mathcal{J} \models \phi$ for all such finite identity interpretations $\mathcal{J}$. Such a test for quasitautologies seems to be much less convenient than the method of associates. We mention it here because this is the method used by Boolos and Jeffrey [3] to recognize proofs in their proof calculus for predicate logic.

Identity tableaux offer the most convenient test for quasitautologies. We call a branch $\varDelta$ of an identity tableau *identically complete* if whenever an identity rule has its premises $\varDelta_1$ in $\varDelta$ then also its conclusion ϕ is an assumption in $\varDelta$ provided $FT(\phi) \subseteq FT(\varDelta_1)$. The reader will note that the last condition can be violated only by function substitution rules whose conclusions can introduce new terms into identity tableaux.

In order to test whether ϕ is a quasitautology we construct a tableau for $\phi*$ and in every open propositionally complete branch we saturate the branch by the conclusions of finitely many identity rules which can be applied on the branch and which do not introduce new terms. If all branches close then ϕ is a quasitautology by Corollary 4.3.8. If there is an open propositionally and identically complete branch then we construct a propositional interpretation $\mathcal{I}$ by collecting all propositional atoms in the assumptions of the branch. We have proved in Par. 3.2.8 that we have $\mathcal{I} \not\models \phi$ and $\mathcal{I} \models \psi$ for all assumptions ψ in the branch. Now, the branch is identically complete, and so we must have $\mathcal{I} \models Eq^{\{\phi\}}$. But this means that $Eq^{\{\phi\}} \not\models_p \phi$ holds and so we have $\not\models_i \phi$ by 4.2.11(1).

Identity tableaux facilitate also a semidecidable test for quasitautological consequence from decidable axioms T: $T \models_i \phi$. This is an extension of the procedure described in Par. 3.5.9 where we identically complete every open branch which is propositionally complete just before we add the next axiom to the assumptions of the branch. By the same reasoning as above we can show that if there is an open branch which is propositionally, identically, and axiomatically complete then we have a propositional interpretation $\mathcal{I}$ such that $\mathcal{I} \models T \cup Eq^{T\cup\{\phi\}}$ and $\mathcal{I} \not\models \phi$. Hence $\mathcal{I}$ witnesses $T, Eq^{T\cup\{\phi\}} \not\models_p \phi$ and we get $T \not\models_i \phi$ by 4.2.11(1).

5. Quantification Logic

In this section we investigate the notion of *logical consequence* where a formula ϕ follows from a set of axioms T by the laws of logic alone. This means that ϕ follows from T because of the properties of propositional connectives, identitity relation, and quantifiers and without taking into considerations the extralogical properties of its function and predicate symbols beyond those which are captured by the axioms T.

5.1 Some Syntactic Concepts

In contrast to the preceding chapters where the variables played no special role we will need in this chapter to pay the attention to the variables occurring in formulas of first order languages.

5.1.1 Free variables. *Free variables* of a set of formulas S of a language $\mathcal{L}$ are variables x occurring in the formulas of S outside of quantifiers $\exists x$ and $\forall x$. This is made precise by a metamathematical function $FV(\alpha)$ defined on terms, formulas, and sets of formulas to yield the set of free variables of α. The function FV satisfies the set identities given in Fig. 5.1.

5.1.2 Sentences. A term τ (formula ϕ) of some $\mathcal{L}$ such that $FV(\tau) = \emptyset$ ($FV(\phi) = \emptyset$) is *closed* and *open* otherwise. Closed formulas are called *sentences*. Sets T of closed formulas are *closed*. *Universal instantiation* of a formula ϕ is any sentence $\forall x_1 \ldots \forall x_n \phi$ such that $x_1, \ldots, x_n \in FV(\phi)$. We usually denote by $\forall\phi$ any universal instantiation of ϕ.

A variable x is *bound* in a formula ϕ if $\exists x\psi$ or $\forall x\psi$ occurs as a subformula in ϕ.

5.1.3 Substitution. If α is a term, formula, or a set of formulas of a language $\mathcal{L}$ then we will denote by $\alpha_x[\tau]$ the application of the ternary metamathematical *substitution* function which yields term, formula, or set similar to α but with every free occurrence of the variable x replaced by the term τ. The substitution function is defined to satisfy the term identities given in Fig. 5.2.

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(f(\tau_1, \ldots, \tau_n)) &= FV(\tau_1) \cup \ldots \cup FV(\tau_n) \\
FV(\tau_1 = \tau_2) &= FT(\tau_1) \cup FT(\tau_2) \\
FV(P(\tau_1, \ldots, \tau_n)) &= FV(\tau_1) \cup \ldots \cup FV(\tau_n) \\
FV(\forall x \phi) &= FV(\phi) \setminus \{x\} \\
FV(\exists x \phi) &= FV(\phi) \setminus \{x\} \\
FV(\top) &= \emptyset \\
FV(\bot) &= \emptyset \\
FV(\neg \phi) &= FV(\phi) \\
FV(\phi_1 \vee \phi_2) &= FV(\phi_1) \cup FV(\phi_2) \\
FV(\phi_1 \wedge \phi_2) &= FV(\phi_1) \cup FV(\phi_2) \\
FV(\phi_1 \rightarrow \phi_2) &= FV(\phi_1) \cup FV(\phi_2) \\
FV(\phi_1 \leftrightarrow \phi_2) &= FV(\phi_1) \cup FV(\phi_2) \\
FV(T) &= \bigcup \{FV(\phi) \mid \phi \in T\} \ .
\end{aligned}
$$

Fig. 5.1. Function yielding sets of free variables

$$
\begin{aligned}
y_x[\tau] &\equiv \begin{cases} \tau & \text{if } y \equiv x \\ y & \text{otherwise} \end{cases} \\
f(\tau_1, \ldots, \tau_n)_x[\tau] &\equiv f(\tau_{1x}[\tau], \ldots, \tau_{nx}[\tau]) \\
(\tau_1 = \tau_2)_x[\tau] &\equiv \tau_{1x}[\tau] = \tau_{2x}[\tau] \\
P(\tau_1, \ldots, \tau_n)_x[\tau] &\equiv P(\tau_{1x}[\tau], \ldots, \tau_{nx}[\tau]) \\
(\forall y \phi)_x[\tau] &\equiv \begin{cases} \forall y \phi & \text{if } y \equiv x \\ \forall y \phi_x[\tau] & \text{otherwise} \end{cases} \\
(\exists y \phi)_x[\tau] &\equiv \begin{cases} \exists y \phi & \text{if } y \equiv x \\ \exists y \phi_x[\tau] & \text{otherwise} \end{cases} \\
\top_x[\tau] &\equiv \top \\
\bot_x[\tau] &\equiv \bot \\
\neg \phi_x[\tau] &\equiv \phi_x[\tau] \\
(\phi_1 \vee \phi_2)_x[\tau] &\equiv \phi_{1x}[\tau] \vee \phi_{2x}[\tau] \\
(\phi_1 \wedge \phi_2)_x[\tau] &\equiv \phi_{1x}[\tau] \wedge \phi_{2x}[\tau] \\
(\phi_1 \rightarrow \phi_2)_x[\tau] &\equiv \phi_{1x}[\tau] \rightarrow \phi_{2x}[\tau] \\
(\phi_1 \leftrightarrow \phi_2)_x[\tau] &\equiv \phi_{1x}[\tau] \leftrightarrow \phi_{2x}[\tau] \\
T_x[\tau] &\equiv \{\phi_x[\tau] \mid \phi \in T\} \ .
\end{aligned}
$$

Fig. 5.2. Substitution function

For reasons discussed in Par. 5.2.11 we will use the the substitution function $\alpha_x[\tau]$ only with terms τ which are *free for x in α* meaning that no variable occurring in τ becomes bound in the result of the substitution. In other words, whenever the recursion reaches a quantified formula, say, $(\forall y \phi)_x[\tau]$ with $y \not\equiv x$ we have $y \notin FV(\tau)$. Note that a closed term τ is free for x in any ϕ.

We now agree on a convention that whenever we will write $\alpha_x[\tau]$ without explicitly mentioning that τ is free for x in α this restriction will be tacitly assumed.

5.1.4 Indication of variables. We will often *indicate* by writing $\phi[x]$ that the variable x is possibly free in ϕ. When we later write $\phi[\tau]$ we are designating the same formula as $\phi_x[\tau]$.

The method of indication is useful when we indicate more variables at once: $\phi[\vec{x}]$. In such situations we always tacitly assume that the n-tuple of variables $\vec{x}$ consists of pairwise distinct variables. By writing $\phi[\vec{\tau}]$ we designate the same formula as $\phi_{x_1}[\tau_1] \cdots_{x_n}[\tau_n]$.

5.1.5 Cantor's diagonal pairing function. We will need the well-known Cantor's 'diagonal' pairing function J defined as

$$J(x, y) = \frac{(x + y + 1)\cdot(x + y)}{2} + x \ .$$

It is not hard to see that every natural number z can be uniquely written as $z = J(x, y)$. Moreover, for any numbers x and y the value $J(x, y)$ can be efectively computed and also for a given number z the unique numbers x and y such that $z = J(x, y)$ can be effectively computed.

5.1.6 Witnessing extensions of first-order languages. Let $\mathcal{L}$ be a first-order language. We denote by $\mathcal{L}_c$ its *witnessing extension* by the addition of *Henkin* (witnessing) constants c_k where $k = J(i + 1, j)$ for some i, j. This c_k is said to be of *rank* $(i + 1)$. Henkin constants are always chosen as new constants not occurring in $\mathcal{L}$.

A formula ϕ of $\mathcal{L}_c$ is of *rank i* if ϕ is of $\mathcal{L}$ and $i = 0$ or if for some Henkin constant c_k with rank i we have $c_k \in FT(\phi)$ and no Henkin constant of higher rank is in $FT(\phi)$.

We now assign the Henkin constants to the existentially quantified sentences of $\mathcal{L}_c$ as follows. We can clearly enumerate the formulas of any language into a sequence. Moreover, by ommiting certain formulas, we can also enumerate for every number i all existentially quantified sentences of $\mathcal{L}_c$ of rank i:

$$\psi_0^i \ \psi_1^i \ \psi_2^i \ \ldots \ , \tag{1}$$

i.e. sentences where $\psi_j^i \equiv \exists x \phi$ for some ϕ of rank i from $\mathcal{L}_c$. We fix one such enumeration relatively to every $\mathcal{L}$ (and thus also relatively to $\mathcal{L}_c$) and i.

The Henkin constant $c_{J(i+1,j)}$ is said to *belong* to the sentence ψ^i_j.

5.1.7 Quantifier axioms. Fix a language $\mathcal{L}$ and consider its witnessing extension $\mathcal{L}_c$. We call the sentences

$$\phi_x[\tau] \to \exists x\phi \tag{1}$$
$$\forall x\phi \to \phi_x[\tau] \tag{2}$$

where $\exists x\phi$ and $\forall x\phi$ are sentences of $\mathcal{L}_c$ and τ a closed term of $\mathcal{L}_c$ *instantiation* axioms.

We call the sentence

$$\exists x\phi \to \phi_x[c_k] \tag{3}$$

where $\exists x\phi$ is a sentence of $\mathcal{L}_c$ and c_k the Henkin constant belonging to it the (Henkin) *witnessing axiom for* $\exists x\phi$.

We call the sentence

$$\phi_x[c_k] \to \forall x\phi \tag{4}$$

where $\forall x\phi$ is a sentence of $\mathcal{L}_c$ and c_k the Henkin constant belonging to the sentence $\exists x\neg\phi$ the (Henkin) *counterexample axiom for* $\forall x\phi$. Witnessing and counterexample axioms are together called *Henkin* axioms. We designate the witnessing axioms by Ha.

Note that the Henkin axioms for sentences $\exists x\phi$ and $\forall x\phi$ of rank i do not need to have ranks $i+1$ because it can happen that $x \notin FV(\phi)$.

The set Qa of *quantifier* axioms is the set of sentences of $\mathcal{L}_c$ consisting of the instantiation and Henkin axioms.

5.2 Logical Consequence

5.2.1 Interpretations. An *interpretation* $\mathcal{I}$ *for* $\mathcal{L}$ is a pair $\langle \mathcal{M}, a\rangle$ where $\mathcal{M}$ is a structure for $\mathcal{L}$ with domain D and a is an assignment in $\mathcal{M}$.

Denotations $\tau^{\mathcal{I}}$ of terms τ of $\mathcal{L}$ are defined exactly as in Par. 4.2.4.

For the definition of the satisfaction relation we will need an operator taking the assignment a, variable x, and $d \in D$ to the assigment designated by $a(\genfrac{}{}{0pt}{}{d}{x})$ and such that

$$a(\genfrac{}{}{0pt}{}{d}{x})(i) = \begin{cases} d & \text{if } x \equiv v_i \\ a(i) & \text{if } x \not\equiv v_i. \end{cases}$$

For the above interpretation $\mathcal{I}$ we designate by $\mathcal{I}(\genfrac{}{}{0pt}{}{d}{x})$ the interpretation $\langle \mathcal{M}, a(\genfrac{}{}{0pt}{}{d}{x})\rangle$.

The satisfaction relation $\mathcal{I}$ *satisfies* ϕ, written as $\mathcal{I} \vDash \phi$, is similar to the satisfaction relation for identity interpretations (see Par. 4.2.5) when ϕ

is an atomic or propositional formula. For quantifier formulas we define the relation as follows:

$$\mathcal{I} \models \forall x\phi \Leftrightarrow \mathcal{I}(\tfrac{d}{x}) \models \phi \text{ for all } d \in D$$
$$\mathcal{I} \models \exists x\phi \Leftrightarrow \mathcal{I}(\tfrac{d}{x}) \models \phi \text{ for some } d \in D.$$

Interpretations $\mathcal{I}$ for $\mathcal{L}_c$ such that $\mathcal{I} \models Ha$ are called *Henkin interpretations.*

5.2.2 Agreement of interpretations. If S is a set of terms then two interpretations $\mathcal{I} = \langle \mathcal{M}, a\rangle$ and $\mathcal{J} = \langle \mathcal{M}, b\rangle$ sharing the same structure *agree on* S if the assignments a and b *agree on* S, i.e. if for every variable $v_i \in S$ we have $a(i) = b(i)$.

5.2.3 Equivalence lemma. *If the interpetations* $\mathcal{I}$ *and* $\mathcal{J}$ *for* $\mathcal{L}$ *agree on*

1. $FV(\tau)$ *then* $\tau^{\mathcal{I}} = \tau^{\mathcal{J}}$,
2. $FV(T)$ *then* $\mathcal{I} \equiv_T \mathcal{J}$.

Proof. 1) Assume $\mathcal{I} = \langle \mathcal{M}, a\rangle$ and $\mathcal{J} = \langle \mathcal{M}, b\rangle$ and prove the property by induction on the construction of τ. If $\tau \equiv v_i$ then $v_i^{\mathcal{I}} = a(i) = b(i) = v_i^{\mathcal{J}}$. If $\tau \equiv f(\tau_1, \ldots, \tau_n)$ then

$$f(\tau_1, \ldots, \tau_n)^{\mathcal{I}} = f^{\mathcal{M}}(\tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}}) \overset{\text{IH}}{=} f^{\mathcal{M}}(\tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}}) = f(\tau_1, \ldots, \tau_n)^{\mathcal{J}} .$$

2) Take any $\phi \in T$ and prove

$$\mathcal{I} \models \phi \Leftrightarrow \mathcal{J} \models \phi \text{ for } \mathcal{I} \text{ and } \mathcal{J} \text{ agreeing on } FV(T)$$

by induction on the construction of ϕ. If $\phi \equiv \tau_1 = \tau_2$ then we have $\mathcal{I} \models \tau_1 = \tau_2$ iff $\tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}}$ iff, by 1), $\tau_1^{\mathcal{J}} = \tau_2^{\mathcal{J}}$ iff $\mathcal{J} \models \tau_1 = \tau_2$. The case when $\phi \equiv P(\tau_1, \ldots, \tau_n)$ is similar.

If $\phi \equiv \exists x\phi_1$ then $\mathcal{I} \models \exists x\phi_1$ iff $\mathcal{I}(\tfrac{d}{x}) \models \phi_1$ for some $d \in D$ where D is the domain of the structure shared by $\mathcal{I}$ and $\mathcal{J}$ iff, by IH, $\mathcal{J}(\tfrac{d}{x}) \models \phi_1$ for some $d \in D$ iff $\mathcal{J} \models \exists x\phi_1$. The case when $\phi \equiv \forall x\phi_1$ is similar.

If $\phi \equiv \neg\phi_1$ then $\mathcal{I} \models \neg\phi_1$ iff $\mathcal{I} \not\models \phi_1$ iff, by IH, $\mathcal{J} \not\models \phi_1$, iff $\mathcal{J} \models \neg\phi_1$. The remaining propositional cases for ϕ are similar.

5.2.4 Substitution lemma. For every interpretation $\mathcal{I}$ for $\mathcal{L}$, any terms ρ_1, ρ_2, any formula ϕ, and any term τ free for x in ϕ we have:

$$(\rho_{1x}[\rho_2])^{\mathcal{I}} = \rho_1^{\mathcal{I}(\frac{\rho_2^{\mathcal{I}}}{x})} \tag{1}$$
$$\mathcal{I} \models \phi_x[\tau] \Leftrightarrow \mathcal{I}(\tfrac{\tau^{\mathcal{I}}}{x}) \models \phi\ . \tag{2}$$

Proof. Take any interpretation $\mathcal{I} = \langle \mathcal{M}, a\rangle$ and let D be the domain of $\mathcal{M}$. (1): By induction on the construction of ρ_1. If $\rho_1 \equiv v_i$ then if $v_i \equiv x$ we have

$$(x_x[\rho_2])^{\mathcal{I}} = \rho_2^{\mathcal{I}} = a(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})(i) = x^{\mathcal{I}(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})}$$

and if $v_i \not\equiv x$ we have

$$(v_{ix}[\rho_2])^{\mathcal{I}} = v_i^{\mathcal{I}} = a(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})(i) = v_i^{\mathcal{I}(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})} .$$

If $\rho_1 \equiv f(\tau_1, \ldots, \tau_n)$ then

$$\begin{aligned}(f(\tau_1, \ldots, \tau_n)_x[\rho_2])^{\mathcal{I}} = (f(\tau_{1x}[\rho_2], \ldots, \tau_{nx}[\rho_2]))^{\mathcal{I}} = \\ f^{\mathcal{M}}((\tau_{1x}[\rho_2])^{\mathcal{I}}, \ldots, (\tau_{1x}[\rho_2])^{\mathcal{I}}) \overset{\text{IH}}{=} \\ f^{\mathcal{M}}(\tau_1^{\mathcal{I}(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})}, \ldots, \tau_n^{\mathcal{I}(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})}) = (f(\tau_1, \ldots, \tau_n))^{\mathcal{I}(\genfrac{}{}{0pt}{}{\rho_2^{\mathcal{I}}}{x})} .\end{aligned}$$

(2): By induction on the construction of ϕ. If $\phi \equiv \tau_1 = \tau_2$ then we have $\mathcal{I} \vDash (\tau_1 = \tau_2)_x[\tau]$ iff $\mathcal{I} \vDash \tau_{1x}[\tau] = \tau_{2x}[\tau]$ iff $(\tau_{1x}[\tau])^{\mathcal{I}} = (\tau_{2x}[\tau])^{\mathcal{I}}$ iff, by (1), $\tau_1^{\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})} = \tau_2^{\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})}$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \tau_1 = \tau_2$.

If $\phi \equiv P(\tau_1, \ldots, \tau_n)$ then $\mathcal{I} \vDash P(\tau_1, \ldots, \tau_n)_x[\tau]$ iff $\mathcal{I} \vDash P(\tau_{1x}[\tau], \ldots, \tau_{nx}[\tau])$ iff $\langle (\tau_{1x}[\tau])^{\mathcal{I}}, \ldots, (\tau_{nx}[\tau])^{\mathcal{I}} \rangle \in P^{\mathcal{M}}$ iff, by (1), $\langle \tau_1^{\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})}, \ldots, \tau_n^{\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})} \rangle \in P^{\mathcal{M}}$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash P(\tau_1, \ldots, \tau_n)$.

If $\phi \equiv \exists y \phi_1$ then if $x \equiv y$ then we have $\mathcal{I} \vDash (\exists y \phi_1)_x[\tau]$ iff $\mathcal{I} \vDash \exists y \phi_1$ iff by Lemma 5.2.3, since $\mathcal{I}$ and $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})$ agree on $FV(\exists y \phi_1)$, $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \exists y \phi_1$.

If $x \not\equiv y$ then we have $\mathcal{I} \vDash (\exists y \phi_1)_x[\tau]$ iff $\mathcal{I} \vDash \exists y \phi_{1x}[\tau]$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{d}{y}) \vDash \phi_{1x}[\tau]$ for some $d \in D$ iff by IH, since τ is free for x in ϕ_1, $\mathcal{I}(\genfrac{}{}{0pt}{}{d}{y})(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}(\genfrac{}{}{0pt}{}{d}{y})}}{x}) \vDash \phi_1$ for some $d \in D$ iff, by Lemma 5.2.3, since $y \notin FV(\tau)$, $\mathcal{I}(\genfrac{}{}{0pt}{}{d}{y})(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \phi_1$ for some $d \in D$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x})(\genfrac{}{}{0pt}{}{d}{y}) \vDash \phi_1$ for some $d \in D$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \exists y \phi_1$. If $\phi \equiv \forall y \phi_1$ then the proof is similar.

If $\phi \equiv \phi_1 \vee \phi_2$ then we have $\mathcal{I} \vDash (\phi_1 \vee \phi_2)_x[\tau]$ iff $\mathcal{I} \vDash \phi_{1x}[\tau] \vee \phi_{2x}[\tau]$ iff $\mathcal{I} \vDash \phi_{1x}[\tau]$ or $\mathcal{I} \vDash \phi_{2x}[\tau]$ iff by IH, since τ is free for x in ϕ_1, ϕ_2, $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \phi_1$ or $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \phi_2$ iff $\mathcal{I}(\genfrac{}{}{0pt}{}{\tau^{\mathcal{I}}}{x}) \vDash \phi_1 \vee \phi_2$. The remaining propositional cases are similar. □

5.2.5 Models. Let $\mathcal{M}$ be a structure for $\mathcal{L}$ with a domain D and $\phi[\vec{x}]$ a formula of $\mathcal{L}$ with all of its free variables among the indicated ones. By Lemma 5.2.3 any two assignments in $\mathcal{M}$ a and b which coincide on $\vec{x}$ are such that

$$\langle \mathcal{M}, a \rangle \vDash \phi \Leftrightarrow \langle \mathcal{M}, b \rangle \vDash \phi$$

and so for $\vec{d} \in D$ we can write $\mathcal{M} \vDash \phi[\vec{d}]$ as an abbreviation for $\langle \mathcal{M}, a \rangle \vDash \phi$ where a is any assignment such that $a(x_1) = d_1, \ldots, a(x_n) = d_n$.

We define $\mathcal{M} \vDash \phi$ as

$$\mathcal{M} \vDash \phi[\vec{d}] \quad \text{for all } \vec{d} \in D .$$

If ϕ is a sentence then we have $\mathcal{M} \models \phi$ iff $\langle \mathcal{M}, a \rangle \models \phi$ for some assignment a (and equivalently for all assignments a) and in that case we say that $\mathcal{M}$ is a *model* of ϕ, or that ϕ is *true* in $\mathcal{M}$.

For a set T of formulas of $\mathcal{L}$ we write $\mathcal{M} \models T$ for $\mathcal{M} \models \phi$ for all $\phi \in T$. $\mathcal{M}$ is a *model* of a set T of sentences of $\mathcal{L}$ if $\mathcal{M} \models T$. T is *satisfiable* if it is true in some structure, i.e. if T has a model.

5.2.6 Expansions of structures. Let $\mathcal{L}_1$ be an extension of a language $\mathcal{L}$ and let $\mathcal{M}$ be a structure for $\mathcal{L}$ with a domain D. The structure $\mathcal{N}$ for $\mathcal{L}_1$ is an *expansion* of $\mathcal{M}$ if the domain of $\mathcal{N}$ is D and the interpretation in $\mathcal{N}$ of every function and predicate symbol from $\mathcal{L}$ is the same as in $\mathcal{M}$.

The interpretation $\mathcal{J}$ for $\mathcal{L}_1$ is an *expansion* of the interpretation $\mathcal{I}$ for $\mathcal{L}$ if the structure of $\mathcal{J}$ is an expansion of the structure for $\mathcal{I}$ and both interpretations have the same assignments.

Two interpretations $\mathcal{I}$ for $\mathcal{L}$ and $\mathcal{J}$ for $\mathcal{L}_1$ where $\mathcal{L}_1$ is an extension of $\mathcal{L}$ are *equivalent on* T, which we write as $\mathcal{I} \equiv_T \mathcal{J}$, if T is a subset of formulas of $\mathcal{L}$ and $\mathcal{I} \models \phi \Leftrightarrow \mathcal{J} \models \phi$ for all $\phi \in T$. We write $\mathcal{I} \equiv_{\mathcal{L}} \mathcal{J}$ if $\mathcal{I}$ and $\mathcal{J}$ are equivalent on the formulas of $\mathcal{L}$.

5.2.7 Expansion theorem. *If $\mathcal{J}$ for $\mathcal{L}_1$ is an expansion of $\mathcal{I}$ for $\mathcal{L}$ then*

$$\tau^{\mathcal{J}} = \tau^{\mathcal{I}} \qquad \text{for all terms } \tau \text{ of } \mathcal{L} \tag{1}$$

$$\mathcal{J} \equiv_{\mathcal{L}} \mathcal{I} \; . \tag{2}$$

Proof. Assume that $\mathcal{J} = \langle \mathcal{N}, a \rangle$ with the domain D and $\mathcal{I} = \langle \mathcal{M}, a \rangle$.

(1): By induction on the construction of τ. If $\tau \equiv v_i$ then $v_i^{\mathcal{J}} = a(i) = v_i^{\mathcal{I}}$. If $\tau \equiv f(\tau_1, \ldots, \tau_n)$ for f in $\mathcal{L}$ then

$$f(\tau_1, \ldots, \tau_n)^{\mathcal{J}} = f^{\mathcal{N}}(\tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}}) \overset{\text{IH}}{=} f^{\mathcal{N}}(\tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}}) = \\ f^{\mathcal{M}}(\tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}}) = f(\tau_1, \ldots, \tau_n)^{\mathcal{I}} \; .$$

(2): The property follows from

$$\phi \text{ in } \mathcal{L}; \text{ for all } \mathcal{J} \text{ for } \mathcal{L} \text{ and } \mathcal{I} \text{ for } \mathcal{L}_1 \text{ s.t. } \mathcal{J} \text{ expands } \mathcal{I} \Rightarrow (\mathcal{J} \models \phi \Leftrightarrow \mathcal{I} \models \phi)$$

proved by induction on the construction of ϕ. If $\phi \equiv \tau_1 = \tau_2$ then we have $\mathcal{J} \models \tau_1 = \tau_2$ iff $\tau_1^{\mathcal{J}} = \tau_2^{\mathcal{J}}$ iff, by (1), $\tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}}$ iff $\mathcal{I} \models \tau_1 = \tau_2$. The case when $\phi \equiv P(\tau_1, \ldots, \tau_n)$ is similar.

If $\phi \equiv \exists x \phi_1$ then $\mathcal{J} \models \exists x \phi_1$ iff $\mathcal{J}(\frac{d}{x}) \models \phi_1$ for some $d \in D$ where D is the shared domain iff, by IH, $\mathcal{I}(\frac{d}{x}) \models \phi_1$ for some $d \in D$ iff $\mathcal{I} \models \exists x \phi_1$. The case when $\phi \equiv \forall x \phi_1$ is similar.

If $\phi \equiv \neg \phi_1$ then $\mathcal{J} \models \neg \phi_1$ iff $\mathcal{J} \not\models \phi_1$ iff, by IH, $\mathcal{I} \not\models \phi_1$, iff $\mathcal{I} \models \neg \phi_1$. The remaining propositional cases for ϕ are similar. $\square$

5.2.8 Lemma (Reduction to identity interpretations). *To every interpretation $\mathcal{I}$ for $\mathcal{L}$ there is an equivalent identity interpretation $\mathcal{J}$ for $\mathcal{L}$.*

Proof. Take any interpretation $\mathcal{I} = \langle \mathcal{M}, a \rangle$ for $\mathcal{L}$. We construct the identity interpretation $\mathcal{J} = \langle Q, \mathcal{M}, a \rangle$ by defining

$$Q = \{\psi \mid \mathcal{I} \vDash \psi \text{ for quantifier formulas } \psi\} \ .$$

Note that the assignment a in $\mathcal{I}$ is also an assignment in $\mathcal{J}$ because both interpretations share the structure $\mathcal{M}$. For the same reason we have

$$\tau^{\mathcal{J}} = \tau^{\mathcal{I}} \tag{1}$$

for every term τ of $\mathcal{L}$.

We prove the equivalence of $\mathcal{J}$ and $\mathcal{I}$ by proving $\mathcal{J} \vDash \phi \Leftrightarrow \mathcal{I} \vDash \phi$ by induction on construction of formulas ϕ of $\mathcal{L}$. If ϕ is a quantifier formula then $\mathcal{J} \vDash \phi$ iff $\phi \in Q$ iff $\mathcal{I} \vDash \phi$. If $\phi \equiv \tau_1 = \tau_2$ then $\mathcal{J} \vDash \tau_1 = \tau_2$ iff $\tau_1^{\mathcal{J}} = \tau_2^{\mathcal{J}}$ iff, by (1), $\tau_1^{\mathcal{I}} = \tau_2^{\mathcal{I}}$ iff $\mathcal{I} \vDash \tau_1 = \tau_2$. If $\phi \equiv P(\tau_1, \ldots, \tau_n)$ then $\mathcal{J} \vDash P(\tau_1, \ldots, \tau_n)$ iff $\langle \tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}} \rangle \in P^{\mathcal{M}}$ iff, by (1), $\langle \tau_1^{\mathcal{I}}, \ldots, \tau_n^{\mathcal{I}} \rangle \in P^{\mathcal{M}}$ iff $\mathcal{I} \vDash P(\tau_1, \ldots, \tau_n)$. If $\phi \equiv \phi_1 \vee \phi_2$ then we have $\mathcal{J} \vDash \phi_1 \vee \phi_2$ iff $\mathcal{J} \vDash \phi_1$ or $\mathcal{J} \vDash \phi_2$ iff, by IH, $\mathcal{I} \vDash \phi_1$ or $\mathcal{I} \vDash \phi_2$ iff $\mathcal{I} \vDash \phi_1 \vee \phi_2$. The remaining cases are similar. □

5.2.9 Logical consequence. Fix a language $\mathcal{L}$ and let T be a set of formulas from $\mathcal{L}$. We define the relation ϕ *is a logical consequence of* T, in symbols $T \vDash \phi$, as follows:

$$T \vDash \phi \Leftrightarrow (\mathcal{I} \vDash T \Rightarrow \mathcal{I} \vDash \phi) \text{ for all interpretations } \mathcal{I}.$$

We write $\vDash \phi$ for $\emptyset \vDash \phi$ and such a ϕ is a *valid* formula. Note that we have

$$\vDash \phi \Leftrightarrow \mathcal{I} \vDash \phi \text{ for all interpretations } \mathcal{I}.$$

If T is closed then we have $T \vDash \phi$ iff $\mathcal{M} \vDash \phi$ for all models of T.

The following lemma asserts that the instantiation axioms are valid and that Henkin counterexample axioms are logical consequences of *Ha*. Hence, *Qa* is staisfied in every Henkin interpretation.

5.2.10 Lemma.

$$\vDash \phi_x[\tau] \to \exists x \phi \qquad \tau,\ \phi \textit{ of any } \mathcal{L},\ \tau \textit{ free for } x \textit{ in } \phi \tag{1}$$

$$\vDash \forall x \phi \to \phi_x[\tau] \qquad \tau,\ \phi \textit{ of any } \mathcal{L},\ \tau \textit{ free for } x \textit{ in } \phi \tag{2}$$

$$Ha \vDash Qa \ . \tag{3}$$

Proof. (1): Take any interpretation $\mathcal{I}$ for $\mathcal{L}$ with the domain D and any formula $\phi_x[\tau] \to \exists x \phi$ satisfying the assumptions. Assume $\mathcal{I} \vDash \phi_x[\tau]$ and obtain $\mathcal{I}(\,{}^{\tau^{\mathcal{I}}}_{x}\,) \vDash \phi$ by Lemma 5.2.4. Since $\tau^{\mathcal{I}} \in D$ we get $\mathcal{I} \vDash \exists x \phi$.

(2): the proof is similar to that of (1).

(3): Take any Henkin interpretation $\mathcal{I}$ for $\mathcal{L}_c$ with the domain D and any $\psi \in Qa$. If $\psi \in Ha$ we have $\mathcal{I} \models \psi$ trivially and if ψ is an instantiation axiom we get the same by (1) or (2) because the term τ is closed and so it is free for x in ϕ. If ψ is a counterexample axiom of a form $\phi_x[c] \to \forall x\phi$ then $\exists x\neg\phi \to \neg\phi_x[c] \in Ha$ and so $\mathcal{I} \models \exists x\neg\phi \to \neg\phi_x[c]$. We now assume $\mathcal{I} \models \phi_x[c]$ and obtain $\mathcal{I} \not\models \neg\phi_x[c]$ and then $\mathcal{I} \not\models \exists x\neg\phi$. But then $\mathcal{I}(\frac{d}{x}) \not\models \neg\phi$, i.e. $\mathcal{I}(\frac{d}{x}) \models \phi$, for all $d \in D$ and so $\mathcal{I} \models \forall x\phi$. □

5.2.11 Remark. Why did we impose the freeness restriction on substitutions into quantified formulas. We did this because we wish the instantiation formulas 5.2.10(1) and 5.2.10(2) to be valid.

For instance, consider the formula $\exists y\, y \neq x$. We have $(\exists y\, y \neq x)_x[y] \equiv \exists y\, y \neq y$. The formula $\forall x\exists y\, y \neq x \to (\exists y\, y \neq x)_x[y]$, i.e. the formula

$$\forall x\exists y\, y \neq x \to \exists y\, y \neq y$$

has the form of 5.2.10(2) but the term y is not free for x in $\exists y\, y \neq x$. This violation of the substitution condition prevents the formula from being valid. This can be seen by noting that $\mathcal{I} \models \forall x\exists y\, y \neq x$ holds for any interpretation $\mathcal{I}$ with at least two elements in its domain whereas $\mathcal{I} \models \exists y\, y \neq y$ is satisfied for no interpretation $\mathcal{I}$.

5.2.12 Lemma (Henkin expansion). *For every interpretation $\mathcal{I}$ for $\mathcal{L}$ there is a Henkin expansion $\mathcal{J}$ for $\mathcal{L}_c$.*

Proof. Take any interpretation $\mathcal{I} = \langle \mathcal{M}, a\rangle$ for $\mathcal{L}$ and let D be the domain of $\mathcal{M}$. We wish to construct the interpretation $\mathcal{J} = \langle \mathcal{N}, a\rangle$ as an expansion of $\mathcal{I}$ satisfying Ha. To that end we define for $i \geq 0$ a sequence of interpretations $\mathcal{J}_i = \langle \mathcal{N}_i, a\rangle$ with the structures $\mathcal{N}_i$ for languages $\mathcal{L}_i$. We define $\mathcal{L}_0 = \mathcal{L}$ and $\mathcal{L}_{i+1}$ as the extension of $\mathcal{L}_i$ with the Henkin constants of rank $i+1$. Thus $\mathcal{L}_i$ contains all Henkin constants with ranks $\leq i$. We define the structures $\mathcal{N}_i$ and thus also the interpretations $\mathcal{J}_i$ by induction on i in such a way that $\mathcal{J}_{i+1}$ will be an expansion of $\mathcal{J}_i$ and for all Henkin constants c of rank $i+1$ belonging to sentences $\exists x\phi$ of $\mathcal{L}_i$ of rank i we will have

$$\mathcal{J}_{i+1} \models \exists x\phi \to \phi_x[c]\ . \tag{1}$$

In the base case we set $\mathcal{N}_0 = \mathcal{M}$ and so $\mathcal{J}_0 = \mathcal{I}$. In the inductive case we take any Henkin constant c of rank $i+1$. The constant belongs to a sentence $\exists x\phi$ of $\mathcal{L}_i$ which is of rank i and we consider two cases. If $\mathcal{J}_i \models \exists x\phi$ holds then $\mathcal{J}_i(\frac{d}{x}) \models \phi$ for some $d \in D$ and we interpret $c^{\mathcal{N}_{i+1}} = d$. If $\mathcal{J}_i \not\models \exists x\phi$ then we interpret $c^{\mathcal{N}_{i+1}} = d$ where d is arbitrary element of D. This ends the definition of $\mathcal{N}_{i+1}$. The interpreation $\mathcal{J}_{i+1}$ is an expansion of $\mathcal{J}_i$ and they are equivalent on the formulas of $\mathcal{L}_i$ by Lemma 5.2.7. For every Henkin constant c if rank $i+1$ we prove (1) by considering the same two cases again. If $\mathcal{J}_i \models \exists x\phi$

holds then we have $\mathcal{J}_{i+1} \models \exists x\phi$ and $\mathcal{J}_{i+1}(\frac{c^{\mathcal{N}_i}}{x}) \models \phi$ by the interpretation of c. We obtain $\mathcal{J}_{i+1} \models \phi_x[c]$ by Lemma 5.2.4. This satisfies (1). If $\mathcal{J}_i \not\models \exists x\phi$ then we have $\mathcal{J}_{i+1} \not\models \exists x\phi$ and so (1) holds again.

With the construction of interpretations $\mathcal{J}_i$ done we note that whenever $0 \leq i < j$ the language $\mathcal{L}_j$ is an extension of $\mathcal{L}_i$, $\mathcal{J}_j$ is an expansion of $\mathcal{J}_i$, and the interpretations are equivalent on $\mathcal{L}_i$ by Lemma 5.2.7. Thus $\mathcal{J}_j$ satisfies the axioms of Ha whose Henkin constants have ranks $\leq j$. We also have $\mathcal{J}_i \equiv_{\mathcal{L}} \mathcal{I}$. We now construct the interpretation $\mathcal{J} = \langle \mathcal{N}, a \rangle$ for $\mathcal{L}_c$ as an expansion of $\mathcal{I}$ where the structure $\mathcal{N}$ is an expansion of $\mathcal{M}$ interpreting Henkin constants of rank i in the same way as $\mathcal{J}_i$. Thus $\mathcal{J} \equiv_{\mathcal{L}} \mathcal{I}$ and, since $\mathcal{J}$ is also an expansion of every $\mathcal{J}_i$ and thus $\mathcal{J} \equiv_{\mathcal{L}_i} \mathcal{J}_i$ by Lemma 5.2.7, we have $\mathcal{J} \models Ha$. □

5.2.13 Lemma (Expansion of identity interpretations). *For every $\mathcal{L}$ and every identity interpretation $\mathcal{I}$ for $\mathcal{L}_c$ such that $\mathcal{I} \models Qa$ there is a canonical Henkin interpretation $\mathcal{J}$ for $\mathcal{L}_c$ equivalent to $\mathcal{I}$ on the sentences of $\mathcal{L}_c$.*

Proof. Take any identity interpretation $\mathcal{I}$ for $\mathcal{L}_c$ such that $\mathcal{I} \models Qa$. Let T be the set of sentences of $\mathcal{L}_c$. We obtain a T-equivalent canonical identity interpretation $\mathcal{I}_1 = \langle Q, \mathcal{M}, a \rangle$ for a structure $\mathcal{M}$ with the domain D and an assignment a by Lemma 4.2.14. The set $FT(T)$ is not empty because it contains at least the Henkin constants and its elements are closed terms. Thus for every $d \in D$ there is a closed term $\tau \in FT(T)$ such that $\tau^{\mathcal{I}_1} = d$. Since $Qa \subseteq T$ we have $\mathcal{I}_1 \models Qa$. We construct the interpretation $\mathcal{J} = \langle \mathcal{M}, a \rangle$ which is canonical because

$$\tau^{\mathcal{J}} = \tau^{\mathcal{I}_1} \tag{1}$$

holds. By induction on the structure of formulas ϕ we prove

$$\phi \in T \Rightarrow (\mathcal{J} \models \phi \Leftrightarrow \mathcal{I}_1 \models \phi)\ . \tag{2}$$

Thus assume $\phi \in T$ and if $\phi \equiv \tau_1 = \tau_2$ then we have $\mathcal{J} \models \tau_1 = \tau_2$ iff $\tau_1^{\mathcal{J}} = \tau_2^{\mathcal{J}}$ iff, by (1), $\tau_1^{\mathcal{I}_1} = \tau_2^{\mathcal{I}_1}$ iff $\mathcal{I}_1 \models \tau_1 = \tau_2$.

If $\phi \equiv P(\tau_1, \ldots, \tau_n)$ then we have $\mathcal{J} \models P(\tau_1, \ldots, \tau_n)$ iff $\langle \tau_1^{\mathcal{J}}, \ldots, \tau_n^{\mathcal{J}} \rangle \in P^{\mathcal{M}}$ iff, by (1), $\langle \tau_1^{\mathcal{I}_1}, \ldots, \tau_n^{\mathcal{I}_1} \rangle \in P^{\mathcal{M}}$ iff $\mathcal{I}_1 \models P(\tau_1, \ldots, \tau_n)$.

If $\phi \equiv \exists x\phi_1$ then if $\mathcal{J} \models \exists x\phi_1$ holds we have $\mathcal{J}(\frac{d}{x}) \models \phi_1$ for some $d \in D$. There is a closed term τ of $\mathcal{L}_c$ such that $\tau^{\mathcal{J}} \overset{(1)}{=} \tau^{\mathcal{I}_1} = d$ and so $\mathcal{J}(\frac{\tau^{\mathcal{J}}}{x}) \models \phi_1$ and we obtain $\mathcal{J} \models \phi_{1x}[\tau]$ by Lemma 5.2.4. Because $\phi_{1x}[\tau] \in T$ we obtain $\mathcal{I}_1 \models \phi_{1x}[\tau]$ by IH and, since $\phi_{1x}[\tau] \to \exists x\phi \in Qa$, we get $\mathcal{I}_1 \models \exists x\phi_1$.

Vice versa, if $\mathcal{I}_1 \models \exists x\phi_1$ holds then we have $\exists x\phi \to \phi_{1x}[c] \in Qa$ for the Henkin constant c belonging to $\exists x\phi$. From $\mathcal{I}_1 \models \exists x\phi \to \phi_{1x}[c]$ we get $\mathcal{I}_1 \models \phi_{1x}[c]$ and then $\mathcal{J} \models \phi_{1x}[c]$ by IH. Hence $\mathcal{J}(\frac{c^{\mathcal{J}}}{x}) \models \phi_1$ by Lemma 5.2.4 and so $\mathcal{J} \models \exists x\phi_1$ holds. The case when $\phi \equiv \forall x\phi_1$ is similar.

If $\phi \equiv \phi_1 \vee \phi_2$ then $\mathcal{J} \models \phi_1 \vee \phi_2$ iff $\mathcal{J} \models \phi_1$ or $\mathcal{J} \models \phi_2$ iff, since $\phi_1, \phi_2 \in T$, we have by IH $\mathcal{I}_1 \models \phi_1$ or $\mathcal{I}_1 \models \phi_2$ iff $\mathcal{I}_1 \models \phi_1 \vee \phi_2$. The remaining cases for ϕ are similar.

We have $\mathcal{I} \equiv_T \mathcal{I}_1$ from the construction of $\mathcal{I}_1$. From (2) we get $\mathcal{I}_1 \equiv_T \mathcal{J}$ and so $\mathcal{I} \equiv_T \mathcal{J}$ holds.

The reader will note that the use of Henkin axioms in the inductive proof of (2) works only when T consists of sentences and so the lemma cannot be extended to the formulas of $\mathcal{L}$. □

5.2.14 Theorem. *If $\mathcal{I}$ is an interpretation for $\mathcal{L}$ then there is a canonical interpretation $\mathcal{J}$ for $\mathcal{L}$ equivalent to $\mathcal{I}$ on the sentences of $\mathcal{L}$.*

Proof. For a given interpretation $\mathcal{I}$ for $\mathcal{L}$ we obtain a Henkin expansion $\mathcal{I}_1$ for $\mathcal{L}_c$ by Lemma 5.2.12 and we have $\mathcal{I}_1 \equiv_{\mathcal{L}} \mathcal{I}$ by Lemma 5.2.7. We then get $\mathcal{I}_1 \vDash Qa$ by 5.2.10(3) and obtain an identity interpretation $\mathcal{I}_2$ for $\mathcal{L}_c$ equivalent to $\mathcal{I}_1$ by Lemma 5.2.8. Since then $\mathcal{I}_2 \vDash Qa$ holds, we get a canonical interpretation $\mathcal{J}$ for $\mathcal{L}_c$ equivalent to $\mathcal{I}_2$ on the sentences of $\mathcal{L}_c$ by Lemma 5.2.13. Since the sentences of $\mathcal{L}$ are sentences of $\mathcal{L}_c$, the interpretation $\mathcal{I}$ is equivalent to $\mathcal{J}$ on the sentences of $\mathcal{L}$. □

5.2.15 Theorem (Henkin reduction). *If ϕ and T consist of sentences of $\mathcal{L}$ then*

$$T \vDash \phi \Leftrightarrow T, Qa \vDash_i \phi \ . \tag{1}$$

Proof. In the direction ($\Rightarrow$) assume $T \vDash \phi$ and take any identity interpretation $\mathcal{I}$ for $\mathcal{L}_c$ such that $\mathcal{I} \vDash T \cup Qa$. There is an interpretation $\mathcal{J}$ for $\mathcal{L}_c$ equivalent to $\mathcal{I}$ on the sentences of $\mathcal{L}_c$ by Lemma 5.2.13. Since the sentences of $\mathcal{L}$ are sentences of $\mathcal{L}_c$, we have $\mathcal{J} \vDash T$ and hence $\mathcal{J} \vDash \phi$ from the assumption. But then $\mathcal{I} \vDash \phi$ by the equivalence.

In the direction ($\Leftarrow$) assume $T, Qa \vDash_i \phi$ and take any interpretation $\mathcal{I}$ for $\mathcal{L}$ such that $\mathcal{I} \vDash T$ holds. There is a Henkin expansion $\mathcal{I}_1$ for $\mathcal{L}_c$ by Lemma 5.2.12 which is equivalent to $\mathcal{I}$ on the formulas of $\mathcal{L}$ by Lemma 5.2.7. Thus $\mathcal{I}_1 \vDash T$ and we also have $\mathcal{I}_1 \vDash Qa$ by 5.2.10(3). There is an identity interpretation $\mathcal{J}$ for $\mathcal{L}_c$ equivalent to $\mathcal{I}_1$ by Lemma 5.2.8. Hence $\mathcal{J} \vDash T \cup Qa$ and we get $\mathcal{J} \vDash \phi$ from the assumption and $\mathcal{I} \vDash \phi$ by the equivalence. □

5.2.16 Theorem. *If ϕ and T consist of sentences of $\mathcal{L}$ then*

$$T \vDash_i \phi \Rightarrow T \vDash \phi \ .$$

Proof. If $T \vDash_i \phi$ holds then we have $T, Qa \vDash_i \phi$ by weakening and $T \vDash \phi$ by Thm. 5.2.15. □

5.2.17 Theorem (Generalization). *If T is closed and ϕ a formula of some $\mathcal{L}$ then*

$$T \models \phi \Leftrightarrow T \models \forall\phi\ .$$

Proof. The theorem is proved by a repeated application of an auxiliary assertion

$$T \models \phi \Leftrightarrow T \models \forall x\phi$$

which is proved in the direction ($\rightarrow$) by assuming $T \models \phi$ and taking any interpretation for $\mathcal{L}$ with a domain D such that $\mathcal{I} \models T$. For any $d \in D$ interpretations $\mathcal{I}(\frac{d}{x})$ and $\mathcal{I}$ agree on $FV(T)$ because T is closed and so $\mathcal{I}(\frac{d}{x}) \models T$ by Lemma 5.2.3. Thus $\mathcal{I}(\frac{d}{x}) \models \phi$ from the assumption and so $\mathcal{I} \models \forall x\phi$.

In the direction ($\leftarrow$) we assume $T \models \forall x\phi$ and take any interpretation for $\mathcal{L}$ such that $\mathcal{I} \models T$. Thus $\mathcal{I} \models \forall x\phi$ and, since $\phi_x[x] \equiv \phi$, we obtain $\mathcal{I} \models \phi$ by 5.2.10(2). ☐

5.2.18 Theorem (Skolem-Löwenheim). *If T is a set of sentences of some $\mathcal{L}$ and ϕ a formula of $\mathcal{L}$ then*

1. *if T is satisfiable then T has a numerical model,*
2. *$T \models \phi$ iff ϕ is true in all numerical models of T.*

Proof. 1) If T is is satisfiable then it has a model $\mathcal{M}$, i.e. $\mathcal{M} \models T$. Since T is a set of sentences we have $\langle \mathcal{M}, a\rangle \models T$ for any assignment a in $\mathcal{M}$. There is a canonical interpretation $\mathcal{J}$ such that $\mathcal{J} \models T$ by Thm. 5.2.14. Thus $\mathcal{J} = \langle \mathcal{N}, b\rangle$ for a numerical structure $\mathcal{N}$ and an assignment b in $\mathcal{N}$. But then $\mathcal{N} \models T$.

2) The direction ($\rightarrow$) is trivial. In the direction ($\leftarrow$) assume $T \not\models \phi$. Thus T has a model $\mathcal{M}$ in which $\mathcal{M} \not\models \phi$, i.e. $\mathcal{M} \models \neg\forall\phi$ by Thm. 5.2.17 and so there is a numerical model $\mathcal{N}$ of $T, \neg\forall\phi$ by 1). Thus $\mathcal{N}$ is a model of T s.t. $\mathcal{N} \not\models \forall\phi$, i.e. $\mathcal{N} \not\models \phi$. ☐

5.2.19 Import of Skolem-Löwenheim's theorem. This paragraph is not yet finished. Numeric structures are the most important ones. The next two parts of this text are devoted to the study of theorems true in the most important numeric structure: the standard model of Peano Arithmetic.

5.3 Quantification Tableaux

5.3.1 Tableau expansion rules. Fix a language $\mathcal{L}$. All expansion rules for (quantifier) tableaux are unary. *Quantifier instantiation* rules are

$$\frac{\forall x\phi}{\phi_x[\tau]}\ (\forall) \qquad \frac{\exists x\phi *}{\phi_x[\tau] *}\ (\exists *)$$

for all formulas ϕ, variables x, and terms τ free for x in ϕ. *Eigen-variable* rules are

$$\frac{\exists x\phi}{\phi_x[y]}\,(\exists) \qquad\qquad \frac{\forall x\phi*}{\phi_x[y]*}\,(\forall*)$$

for all formulas ϕ, variables x, and *eigen-variables* y free for x in ϕ.

The reader will note that the instantiation rules corrrespond to the valid formulas 5.2.10(2)(2) and the eigen-variable rules to Henkin axioms.

5.3.2 Proofs with tableaux. A tableau π for a sequence of signed formulas Δ from axioms in T is called a *tableau for Δ from axioms T* if every eigen-variable y in π satisfies the *eigen-variable condition* that it is not free in any signed formula in the branch above the conclusion of the corresponding eigen-variable rule and neither it is free in T. If π is closed then we assert this by writing $\pi : T \vdash [\Delta]$. The requirement that the eigen-variables be not free in axioms are not that severe as they might look because our axioms will be mostly closed.

When $\Delta \equiv \phi*$ then we say that π *is a tableau proving ϕ from axioms T* and write it as $\pi : T \vdash \phi$. We use additional abbreviations similar to those discussed in Par. 3.2.4.

We first adapt to quantifier tableaux some of the theorems formulated for propositional tableaux.

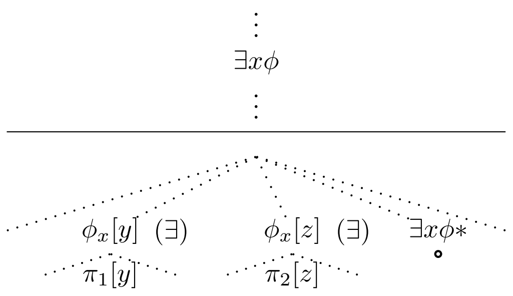
5.3.3 Theorem (Admissible rules in tableaux). *Generalized flatten (see Thm. 3.3.2), generalized split (see Thm. 3.3.3), and propositional inversion (see Thm. 3.3.5) are admissible in any tableaux. Inversion rules for* $(\exists)$ *and* $(\forall*)$*:*

$$T \vdash [\Delta, \exists x\phi] \Rightarrow \pi : T \vdash [\Delta, \phi_x[y]] \quad \textit{for a } y \textit{ and } \pi \textit{ with no } (\exists) \textit{ on } \exists x\phi$$
$$T \vdash [\Delta, \forall x\phi*] \Rightarrow \pi : T \vdash [\Delta, \phi_x[y]*] \quad \textit{for a } y \textit{ and } \pi \textit{ with no } (\forall*) \textit{ on } \forall x\phi* \,.$$

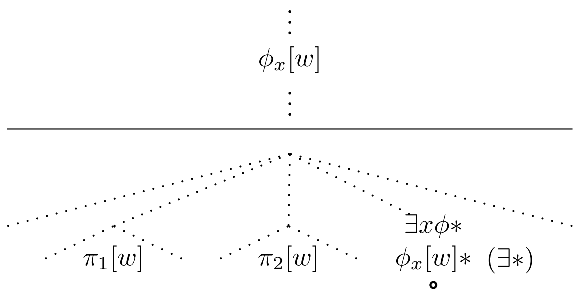
are admissible in tableaux with closed axioms T.

Proof. Inspection of proofs of admissibility of expansion generalized flatten and split rules as well as of propositional inversion rules reveals that the proofs remain correct also for (quantifier) tableaux; the presence of quantifier and identity rules does not affect the proofs.

We prove just that the inversion of $(\exists)$ rule is admissible, the proof of the inversion of $(\forall*)$ is similar. Consider the following closed tableau



where we have indicated three out of possibly many uses of the assumption $\exists x\phi$. The first two uses are by $(\exists)$ rules with the eigen-variables y and z and the third use closes the branch. The following closed tableau shows the inverted rule:



where we have chosen a new variable w, removed the two conclusions of $(\exists)$ rules (since they occur above), systematically renamed the variables y and z to w in the tableaux $\phi_1[y]$ and $\phi_2[z]$, and closed the branch containing the goal $\exists x\phi*$ with a $(\exists*)$ rule. Note that we can always find a new variable w because there are only finitely many of them free or bound in the tableau. This makes w free for x in ϕ and free for y and z in the quantified formulas of π_1 and π_2. Also note that the renaming of variables does not affect any axiom expansions because they are sentences. □

5.3.4 Syntactic compactness and deduction theorems.

$$T \vdash \phi \Rightarrow S \vdash \phi \qquad \text{finite } S \subseteq T \tag{1}$$

$$S \vdash \phi \Leftrightarrow\ \vdash \bigwedge S \to \phi \qquad S \text{ finite.} \tag{2}$$

Proof. (1): This is the theorem on syntactic compactness 3.5.5 whose proof directly lifts up to the general tableaux.

(2): This is the Deduction theorem 3.5.6 whose proof directly lifts up to the general tableaux. □

5.3.5 Renaming lemma. *If T in $\mathcal{L}$ is closed, $\Delta[\vec{x}]$ is a sequence of formulas of $\mathcal{L}$ with all free variables among the indicated ones, and the variables in $\vec{y}$ are free for the corresponding variables in Δ then*

$$\pi : T \vdash \Delta[\vec{x}] \Rightarrow T \vdash \Delta[\vec{y}]\ .$$

Proof. By induction on the number of expansions in π. If π is empty then whatever closes $\Delta[\vec{x}]$ must close $\Delta[\vec{y}]$. If the last expansion in π is by a rule which does not introduce new variables, say $(\neg*)$, then, for a premise $\neg\phi[\vec{x}]* \in \Delta[\vec{x}]$, we show the tableau π on the left:

$$\begin{array}{c} \Delta[\vec{x}] \\ \hline \phi[\vec{x}] \quad (\neg *) \\ \cdots\bar{\pi}_1\cdots \end{array} \quad\Rightarrow\quad \begin{array}{c} \Delta[\vec{y}] \\ \hline \phi[\vec{y}] \quad (\neg *) \\ \cdots\bar{\pi}'_1\cdots \end{array}$$

We have $\pi_1 : T \vdash [\Delta[\vec{x}], \phi[\vec{x}]]$ and we obtain $\pi'_1 : T \vdash [\Delta[\vec{y}], \phi[\vec{y}]]$ for some π'_1 by IH. We construct the tableu π' shown on the right. Note that $\neg\phi[\vec{y}]* \in \Delta[\vec{y}]$ and thus $\pi' : T \vdash \Delta[\vec{y}]$.

$(\neg *)$ rules, remaining propositional rules, as well as all identity rules except reflexivity, are rules which do not introduce new variables and they are invariant to systematic renaming of free variables in their premises and conclusions.

If the first expansion in π is by a rule which may introduce new variables, say $(\exists)$, then, for a premise $\exists v \phi[v, \vec{x}]* \in \Delta[\vec{x}]$ we show the tableau π on the left:

$$\begin{array}{c} \Delta[\vec{x}] \\ \hline \phi[\tau[\vec{z}, \vec{x}], \vec{x}] \quad (\exists) \\ \cdots\bar{\pi}_1\cdots \end{array} \quad\Rightarrow\quad \begin{array}{c} \Delta[\vec{y}] \\ \hline \phi[\tau[\vec{z}, \vec{y}], \vec{y}] \quad (\exists) \\ \cdots\bar{\pi}'_1\cdots \end{array}$$

We have $\pi_1 : T \vdash [\Delta[\vec{x}], \phi[\tau[\vec{z}, \vec{x}], \vec{x}]]$ an we obtain $\pi'_1 : T \vdash [\Delta[\vec{y}], \phi[\tau[\vec{z}, \vec{y}], \vec{y}]]$ for some π'_1 by IH. We construct the tableau π' shown on the right. Note that $\exists v \phi[v, \vec{y}]* \in \Delta[\vec{y}]$ and thus $\pi' : T \vdash [\Delta[\vec{y}]]$.

$(\exists)$ rules, remaining quantifier rules, axiom rules, as well as reflexivity rules, may introduce new variables into tableaux. All but the axiom rules are invariant to systematic renaming of free variables in their premises and conclusions. We have assumed that the axioms in T are closed which makes the axiom rules invariant to renaming too. $\square$

5.3.6 Lemma (Admissibility of cuts on propositional formulas). *If the cut rules on all propositional atoms in a formula ϕ are admissible in (quantifier) tableaux then also the cut rule on ϕ is admissible.*

Proof. Inspection of the proof of the similar Lemma 3.3.7 for propositional tableaux reveals that the lemma holds also for tableaux (with identity and quantifier rules). $\square$

5.3.7 Theorem (Admissibility of cuts). *Cut rules on arbitrary formulas are admissible in tableaux with closed axioms.*

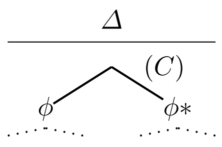
Proof. For the duration of this proof we call a tableau $\pi : T \vdash [\Delta]$ *normal* if

1. the axioms T are sentences,
2. free variables in the formulas of Δ and π are disjoint with all bound variables used in Δ and π,
3. all new variables introduced into π by the reflexivity, instantiation, and eigen-variable rules are pairwise distinct.

We prove

> if π is a normal closed tableau from T for Δ with a cut on the formula ϕ as the first and only expansion by a cut in π then there is a tableau π' such that $\pi' : T \vdash [\Delta]$

by induction on the number of propositional connectives and quantifiers in the cut formula ϕ. The tableau π can be visualized as follows

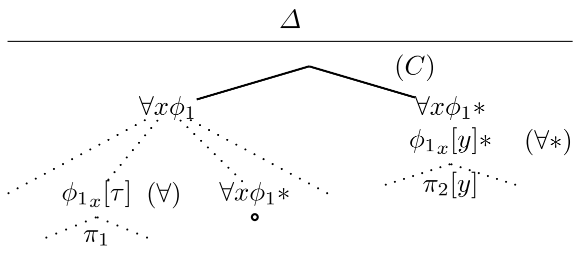


(1)

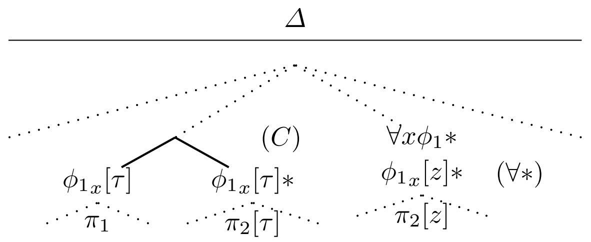
If ϕ is an atomic formula then the cut on ϕ is admissible by the same argument as in Lemma 3.3.8 (taking into account the comment on cuts on identity formulas in Thm. 4.3.3).

If ϕ is a propositional formula then cuts on all propositional atoms in ϕ are admissible by IH and so the cut on ϕ is admissible by Lemma 5.3.6.

If $\phi \equiv \forall x \phi_1$ then (1) looks as follows:



where we may assume without loss of generality that the $(\forall *)$ rule on the right has been inverted. Thus the goal $\forall x \phi_1 *$ is not used for anything in the tableau $\pi_2[y]$. We have also indicated on the left one of possibly many uses of the assumption $\forall x \phi_1$ in an $(\forall)$ rule and in the closing of a branch. We transform π into π' by modifying the tableau under the assumption $\forall x \phi_1$:



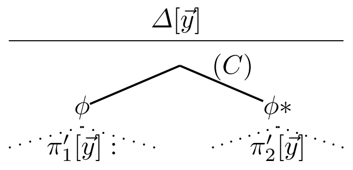
where we replace every use of a $(\forall)$ rule with the conclusion $\phi_{1x}[\tau]$ by a cut on this formula. The branch leading to the goal $\phi_{1x}[\tau]*$ is closed by the tableau

$\pi_2[\tau]$ obtained from $\pi_2[y]$ by substituting for the eigen-variable y the term τ. The substitution cannot affect axiom expansions in $\pi_2[y]$ because they are closed by 1). Furthermore, no variable free in τ is bound anywhere in $\pi_2[y]$ by 2) and so all substitutions are free. Finally, no eigen-variable condition on any eigen-variable w in $pi_2[\tau]$ is violated because w cannot be introduced into the branch leading to ${\phi_1}_x[\tau]*$ by 3).

We also apply a $(\forall *)$ rule with new eigen-variables z everywhere where the assumption $\forall x \phi_1$ was used for closing. We restore the normality of π' if needed by renaming with new variables all variables introduced in the possibly many copies of $\pi_2[y]$. The tableau π' clearly does not use the assumption $\forall x \phi_1$ and the cuts on all formulas ${\phi_1}_x[\tau]$ are admissible by IH. Hence $\pi' : T \vdash [\Delta]$ holds.

The case when $\phi \equiv \exists x \phi_1$ is proved similarly.

In order to finish the proof we must show how to convert the tableau (1) into a normal tableau. The tableau consists of two branches such that $\pi_1[\vec{x}] : T \vdash [\Delta[\vec{x}], \phi[\vec{x}]]$ and $\pi_2[\vec{x}] : T \vdash [\Delta[\vec{x}], \phi[\vec{x}]*]$. We have indicated all free variables in Δ and ϕ. The condition 1) is satisfied by the assumption of the theorem that T is closed. We satisfy the conditions 2) and 3) by choosing new variables $\vec{y}$ and systematically renaming to new variables the free variables introduced into π_1 and π_2. In this way we obtain $\pi_1'[\vec{y}] : T \vdash [\Delta[\vec{y}], \phi[\vec{y}]]$, $\pi_2'[\vec{y}] : T \vdash [\Delta[\vec{y}], \phi[\vec{y}]*]$ for some π_1' and π_2' by Lemma 5.3.5. The tableau



is normal, its cut is admissible, and we obtain $\pi' : T \vdash \Delta[\vec{y}]$ for a cut-free π'. We now apply Lemma 5.3.5 again to get $T \vdash \Delta[\vec{x}]$. □

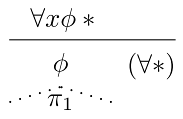
5.3.8 Theorem (Admissibility of the generalization rule). *If T is closed and ϕ a formula of some $\mathcal{L}$ then*

$$T \vdash \phi \Leftrightarrow T \vdash \forall \phi \ .$$

Proof. The theorem is proved by a repeated application of an auxiliary assertion

$$T \vdash \phi \Leftrightarrow T \vdash \forall x \phi$$

which is proved in the direction $(\rightarrow)$ by assuming $\pi_1 : T \vdash \phi$ for some π_1 and constructing the tableau π':



for the eigen-variable x where we note that $\phi_x[x] \equiv \phi$. Thus $\pi' : T \vdash \forall x\phi$.

In the direction $(\leftarrow)$ we assume $\pi : T \vdash \forall\phi$. The tableau can be presented by $(\forall)*$ inversion for some w as follows:

$$\begin{array}{ll}
\forall x\phi\, * & \\
\hline
\phi_x[w] & (\forall *) \\
\cdots\vdots\pi_1\cdots &
\end{array}$$

We have $\pi_1 : T \vdash \phi_x[w]$ from which we obtain $T \vdash \phi$ by Lemma 5.3.5 because $\phi_x[x] \equiv \phi$. □

5.3.9 Lemma. $Ha \vdash Qa$.

Proof. Take any $\psi \in Q$ and consider four cases. If ψ is an existential instantiation axiom $\phi_x[\tau] \to \exists x\phi$ then we can prove it even without Henkin axioms:

$$\begin{array}{ll}
\phi_x[\tau] \to \exists x\phi\, * & \\
\hline
\phi_x[\tau] & (\to_1 *) \\
\exists x\phi * & (\to_2 *) \\
\phi_x[\tau] * & (\exists *) \\
\circ &
\end{array}$$

If ψ is an universal instantiation axiom $\forall x\phi \to \phi_x[\tau]$ then the proof is similar.

If ψ is a witnessing axiom $\exists x\phi \to \phi_x[c]$ then we construct the following one expansion tableau:

$$\begin{array}{ll}
\exists x\phi \to \phi_x[c]\, * & \\
\hline
\exists x\phi \to \phi_x[c] & (Ax) \\
\circ &
\end{array}$$

If ψ is a counterexample axiom $\phi_x[c] \to \forall x\phi$ then the Henkin constant is shared with the witnessing axiom: $\exists x\neg\phi \to \neg\phi_x[c]$. We prove ψ with a new eigen-variable z as follows:

$$\begin{array}{llll}
\multicolumn{4}{c}{\phi_x[c] \to \forall x\phi *} \\
\hline
\multicolumn{3}{c}{\exists x\neg\phi \to \neg\phi_x[c]} & (Ax) \\
 & & & (\to) \\
\neg\phi_x[c] & & \exists x\neg\phi * & \\
\phi_x[c] * & (\neg) & \forall x\phi * & (\to_2 *) \\
\phi_x[c] & (\to_1 *) & \phi_x[z] * & (\forall *) \\
\circ & & \neg\phi_x[z] * & (\exists *) \\
 & & \phi_x[z] & (\neg *) \\
 & & \circ &
\end{array}$$

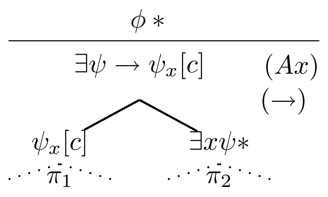
5.3.10 Theorem (Elimination of Henkin witnessing axioms). *If T and ϕ consist of sentences of a language $\mathcal{L}$ and $\mathcal{L}_c$ is its witnessing extension then*

$$T, Ha \vdash \phi \Rightarrow T \vdash \phi \ .$$

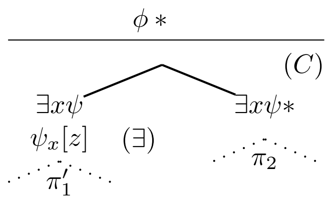
Proof. Assume $\pi : T, Ha \vdash \phi$ for closed T and ϕ. We have $\pi : T, Ha_1 \vdash \phi$ for a finite subset Ha_1 of Ha by 5.3.4(1). The theorem follows from the following assertion:

if $\pi : T, Ha_1 \vdash \phi$ for a finite subset Ha_1 of Ha then $T \vdash \phi$.

which is proved by induction on the size of Ha_1. If $Ha_1 = \emptyset$ there is nothing to prove. Otherwise select from Ha_1 a witnessing axiom $\psi_0 \equiv \exists\psi \to \psi_x[c]$ with the Henkin variable of maximal rank and denote by Ha_2 the set of remaining axioms. We may assume without loss of generality that π has the following form



where neither π_1 nor π_2 contain expansions by the axiom ψ_0. This is because we can delete all expansions by the axiom ψ_0 from π and put it as the first expansion in π. The inversion of ψ_0 can only affect eigen-variable rules but the condition that no eigen-variable occurs free in ψ_0, which is a sentence anyway, assures that the inversion is always possible. We can also invert the $(\to)$ expansion for ψ_0. We construct a tableau π' for ϕ as follows:



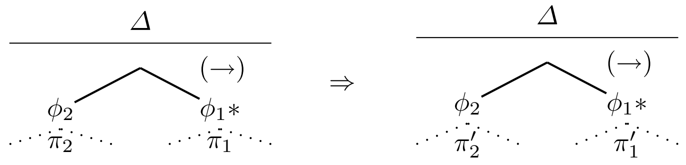
where we have replaced the expansion by ψ_0 by an admissible cut on the formula $\exists x\psi$. Its right branch is closed by π_2 and the left branch is expanded by the eigen-variable rule with a new variable z after which the branch is closed with the tableau π_1' obtained from π_1 by replacing everywhere the constant c by z. The crucial fact is that the Henkin constant c occurs neither in T nor in ϕ because these are in the language $\mathcal{L}$. Thus the axioms from T and the subformulas of ϕ occurring in π_1 are not affected by the replacement. Since c is of maximal rank in Ha_1, it occurs neither in Ha_2 nor in $\exists x\psi$. Thus neither the axioms from Ha_2 used in π_1 are affected by the replacement. We have $\pi' : T, Ha_2 \vdash \phi$ from which we obtain $T \vdash \phi$ by IH. □

5.3.11 Lemma (Elimination of quantifier rules). *If T and $\mathit{\Delta}$ consist of sentences of a language $\mathcal{L}$ and $\mathcal{L}_c$ is its witnessing extension then*

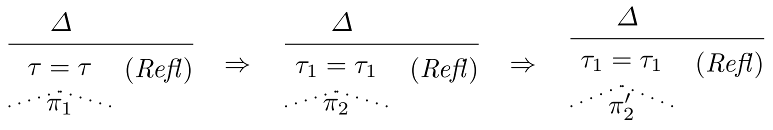
$$\pi : T \vdash [\mathit{\Delta}] \Rightarrow T, Qa \vdash_i [\mathit{\Delta}] \ .$$

Proof. By induction on the number of expansions in the tableau π. Assume $\pi : T \vdash [\mathit{\Delta}]$ with closed T, $\mathit{\Delta}$ and consider the form of π. If π is empty then we trivially have $T \vdash_i [\mathit{\Delta}]$ and we obtain $T, Qa \vdash_i [\mathit{\Delta}]$ by weakening.

If the first expansion in π is by a propositional rule, say $(\rightarrow)$, then we have a closed $\phi_1 \rightarrow \phi_2 \in \mathit{\Delta}$ and π has the form shown in the following on the left:

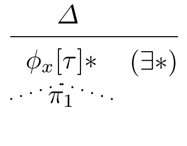


Since ϕ_1, ϕ_2 are closed, we obtain identity tableaux π_1' and π_2' such that $\pi_1' : T, Qa \vdash_i [\mathit{\Delta}, \phi_1*]$ and $\pi_2' : T, Qa \vdash_i [\mathit{\Delta}, \phi_2]$ by two IH's and construct the closed identity tableau for $\mathit{\Delta}$ from axioms T, Qa shown on the right. The remaining propositional, axiom, and all identity expansions except reflexivity are similar. This is because neither of the mentioned expansions introduces new free variables (recall axioms in T are closed). The exception is the reflexivity rule shown on the left:

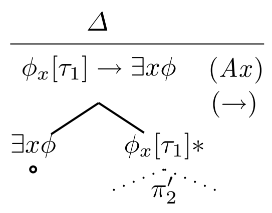


If the term τ is not closed then we substitute for its free variables some Henkin constant whereby we obtain a closed term τ_1. We perform the same substitutions in the formulas of π_1 whereby we obtain a tableau π_2. Note that the substitutions do not affect the structure of π_1 because T and $\mathit{\Delta}$ are closed. The new tableau π' for $\mathit{\Delta}$ with the same number of expansions as π is shown above in the middle. We have $\pi' : T \vdash [\mathit{\Delta}]$ and also $\pi_2 : T \vdash [\mathit{\Delta}, \tau_1 = \tau_2]$. We apply IH to the last tableau and obtain an identity tableau π_2' s.t. $\pi_2' : T, Qa \vdash_i [\mathit{\Delta}, \tau_1 = \tau_2]$, We then construct the closed identity tableau π'' for $\mathit{\Delta}$ from axioms T, Qa shown on the right.

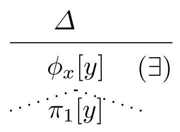
If the first expansion in π is by quantifier instantiation rule, say $(\exists*)$, then we have a closed $\exists x \phi* \in \mathit{\Delta}$ and π has the following form:



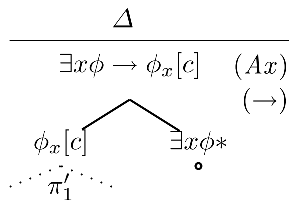
If the term τ contains free variables then we substitute for them, similarly as in the preceding case, some Henkin constant and we perform the same substitutions in the tableau π_1 whereby we obtain a closed term τ_1 and a tableau π_2 s.t. $\pi_2 : T \vdash [\Delta, \phi_x[\tau_1]*]$. We then obtain an identity tableau π_2' such that $\pi_2' : T, Qa \vdash_i [\Delta, \phi_x[\tau_1]*]$ by IH and construct the following closed identity tableau for Δ from axioms T, Qa as follws:



If the first expansion in π is by an eigen-variable rule, say $(\exists)$, then we have a closed $\exists x\phi \in \Delta$ and π has the following form:



Note that the eigen-variable y cannot freely occur in T or in Δ because both are closed. Let c be the Henkin constant belonging to $\exists x\phi$. We substitute c for y in $\phi_x[y]$ and $\pi_1[y]$ whereby we obtain a tableau $\pi_1[c]$ such that $\pi_1[c] : T \vdash [\Delta, \phi_x[c]]$. Since $\phi_x[c]$ is closed, we obtain an identity tableau π_1' such that $\pi_1' : T, Qa \vdash [\Delta, \phi_x[c]]$ by IH and we construct the closed identity tableau π' for Δ from axioms T, Qa as follows:

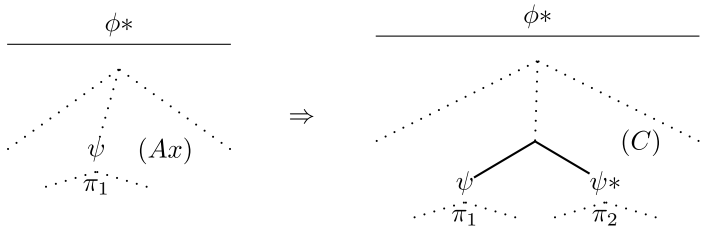


The remaining quantifier expansion rules are similar. □

5.3.12 Theorem (Introduction/elimination of quantifier rules). *If T and ϕ consist of sentences of a language $\mathcal{L}$ and $\mathcal{L}_c$ is its witnessing extension then*

$$T \vdash \phi \Leftrightarrow T, Qa \vdash_i \phi \ .$$

Proof. In the direction $(\Rightarrow)$ assume $\pi : T \vdash \phi$, i.e. $\pi : T \vdash [\phi*]$, and apply Lemma 5.3.11 to get $T, Qa \vdash_i [\phi*]$, i.e. $T, Qa \vdash_i \phi$. In the direction $(\Leftarrow)$ assume $\pi : T, Qa \vdash_i \phi$ which is shown in the following on the left



where we have indicated just one out of possibly many (or none) expansions by the axiom rule with $\psi \in Qa$. We construct a new tableau π' shown on the right where we replace every such expansion by a cut on ψ whose right branch is closed by the tableau π_2 obtained by Lemma 5.3.9 to satisfy $\pi_2 : Ha \vdash \psi$. Cuts are admissible by Lemma 5.3.7 and so $\pi' : T, Ha \vdash \phi$ from which we obtain $T \vdash \phi$ by Thm. 5.3.10. □

5.3.13 Theorem (Soundness and completeness of tableaux). *If T of some $\mathcal{L}$ is closed and ϕ is a formula of $\mathcal{L}$ then*

$$T \vDash \phi \Leftrightarrow T \vdash \phi \ . \tag{1}$$

If also ϕ is a sentence then Fig. 5.3 shows the complete semantic and syntactic reductions.

$$\begin{tikzcd}
T \vDash \phi \arrow[rr, Leftrightarrow, "\textbf{(a)}"] \arrow[d, Leftrightarrow, "\text{(Henkin reduction: 5.2.15)}"] & & T \vdash \phi \arrow[d, Leftrightarrow, "\text{(Intro/elim of q-rules: 5.3.12)}"] \\
T, Qa \vDash_i \psi \arrow[rr, Leftrightarrow, "\textbf{(b)}"] \arrow[d, Leftrightarrow, "\text{(Quasitautological reduction: 4.2.11)}"] & & T, Qa \vdash_i \phi \arrow[d, Leftrightarrow, "\text{(Intro/elim of i-rules: 4.3.7)}"] \\
T, Qa, Eq \vDash_p \phi \arrow[rr, Leftrightarrow, "\textbf{(c)}"] \arrow[d, Leftrightarrow, "\text{(Tautological reduction: 3.4.5)}"] & & T, Qa, Eq \vdash_p \phi \arrow[d, Leftrightarrow, "\text{(Intro/elim of axioms: 3.5.7)}"] \\
\vDash_p \psi_1 \wedge \cdots \wedge \psi_n \to \phi \text{ for some } \psi_1, \ldots, \psi_n \in T \cup Qa \cup Eq \arrow[rr, Leftrightarrow, "\textbf{(d)}"] & & \vdash_p \psi_1 \wedge \cdots \wedge \psi_n \to \phi \text{ for some } \psi_1, \ldots, \psi_n \in T \cup Qa \cup Eq
\end{tikzcd}$$

(a): Soundness and completeness of tableaux: 5.3.13

(b): Soundness and completeness of identity tableaux: 4.3.8

(c): Soundness and completeness of propositional tableaux: 3.5.8

(d): Soundness and completeness of propositional tableaux without axioms: 3.2.7

Fig. 5.3. Soundness and completeness of tableaux for closed T and ϕ.

Proof. When ϕ is a sentence then we have $T \vDash \phi$ iff, by Thm. 5.2.15, $T, Qa \vDash_i \phi$ iff, by Corollary 4.3.8, $T, Qa \vdash_i \phi$ iff, by Thm. 5.3.12, $T \vdash \phi$. This is shown in Fig. 5.3.

If ϕ is a formula then we have $T \vDash \phi$ iff, by Thm. 5.2.17, $T \vDash \forall\phi$ for a universal closure $\forall\phi$ of ϕ iff, by the just proved special case, $T \vdash \forall\phi$ iff, by Thm. 5.3.8, $T \vdash \phi$. $\square$

5.3.14 Semidecidability of validity. This paragraph is not yet finished. We can find a proof if valid. If not we may search forever for an counterexample. This is the best, but the proof of it requires a knowledge of the theory of computability.

6. First-order Theories

A *first-order theory in $\mathcal{L}$*, or simply theory in $\mathcal{L}$, is a set T of sentences of the first-order language $\mathcal{L}$. *Theorems* of T are formulas ϕ provable from T: $T \vdash \phi$.

6.1 Theorems of Predicate Calculus

For every language $\mathcal{L}$ the theorems of the *empty* theory $\emptyset$ in $\mathcal{L}$ are called theorems of *predicate calculus.*

6.1.1 Tableau vs. informal proofs. This paragraph is not finished. It will deal with proofs proofs of basic theorems of predicate calculus formally and informally. The theorems deal with quantifiers, prenex operations, instantiations (substitution), etc.

We now prove two *rules of Leibnitz* which are a generalization of function *Fsub* and predicate *Psub* substitution rules to arbitrary terms and formulas.

6.1.2 Theorem (Rules of Leibnitz). *For a term $\sigma[x_1,\dots,x_n]$ and a formula $\phi[x_1,\dots,x_n]$ with of a language $\mathcal{L}$ and with the free variables among the indicated ones the following are admissible rules of inference:*

$$\frac{\tau_1=\rho_1 \;\dots\; \tau_n=\rho_n}{\sigma[\tau_1,\dots,\tau_n]=\sigma[\rho_1,\dots,\rho_n]}\,(L_1) \qquad \frac{\tau_1=\rho_1 \;\dots\; \tau_n=\rho_n \;\; \phi[\tau_1,\dots,\tau_n]}{\phi[\rho_1,\dots,\rho_n]}\,(L_2)$$

Proof. (L_1): An application of the rule is shown on the left:

$$\dfrac{\begin{array}{c}[\vec\tau=\vec\rho]\\ \vdots\end{array}}{\begin{array}{c}\sigma[\tau_1,\dots,\tau_n]=\sigma[\rho_1,\dots,\rho_n]\\ \pi\end{array}}\,(L_1) \quad\Rightarrow\quad \dfrac{\begin{array}{c}[\vec\tau=\vec\rho]\\ \vdots\end{array}}{\pi'}$$

where we have indicated by $[\vec\tau=\vec\rho]$ the premises $\tau_1=\rho_1$, $\dots$, $\tau_n=\rho_n$. We wish to find a closed tableau π' shown on the right by induction on the structure of the term σ.

If $\sigma \equiv y$ where y is not among the indicated variables then the conclusion of (L_1) is $y = y$ and we start π' with a reflexivity rule:

$$\begin{array}{c} [\vec{\tau} = \vec{\rho}] \\ \vdots \\ \hline y = y \qquad (\mathit{Refl}) \\ \cdots\bar{\pi}\cdots \end{array}$$

If $\sigma \equiv x_i$ where $1 \leq i \leq n$ then the conclusion of (L_1) is $\tau_i = \rho_i$ and we can omit it altogether because it is already among the premises. Thus $\pi' \equiv \pi$:

$$\begin{array}{c} [\vec{\tau} = \vec{\rho}] \\ \vdots \\ \hline \cdots\bar{\pi}\cdots \end{array}$$

If $\sigma \equiv f(\sigma_1, \ldots, \sigma_m)$ then we construct π' by applying (L_1) m-times to the terms σ_i because this is admissible by IH and then by using a function substitution rule for f:

$$\begin{array}{c} [\vec{\tau} = \vec{\rho}] \\ \vdots \\ \hline \sigma_1[\vec{\tau}] = \sigma_1[\vec{\rho}] \qquad (L_1) \\ \vdots \\ \sigma_m[\vec{\tau}] = \sigma_m[\vec{\rho}] \qquad (L_1) \\ f(\sigma_1[\vec{\tau}], \ldots, \sigma_m[\vec{\tau}]) = f(\sigma_1[\vec{\rho}], \ldots, \sigma_m[\vec{\rho}]) \qquad (\mathit{Fsub}) \\ \cdots\bar{\pi}\cdots \end{array}$$

(L_2): An application of the rule is shown on the left:

$$\begin{array}{c} [\vec{\tau} = \vec{\rho}] \\ \vdots \\ \phi[\tau_1, \ldots, \tau_n] \\ \hline \phi[\rho_1, \ldots, \rho_n] \qquad (L_2) \\ \cdots\bar{\pi}\cdots \end{array} \quad \Rightarrow \quad \begin{array}{c} [\vec{\tau} = \vec{\rho}] \\ \vdots \\ \phi[\tau_1, \ldots, \tau_n] \\ \hline \cdots\bar{\pi}'\cdots \end{array}$$

We wish to find a closed tableau π' shown on the right by induction on the number of propositional connectives and quantifiers in the formula ϕ.

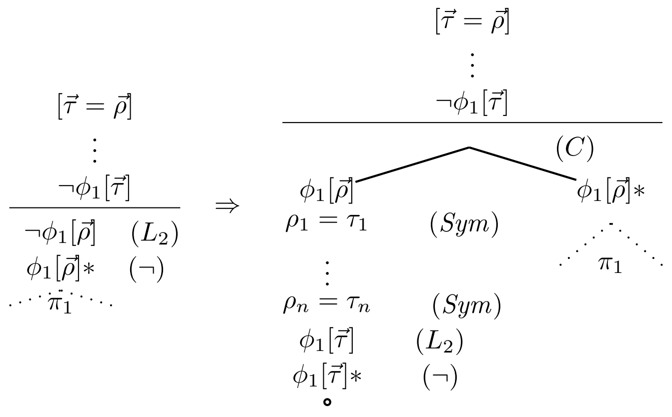
If $\phi \equiv \top$ or $\phi \equiv \bot$ then both $\phi[\vec{\tau}]$ and $\phi[\vec{\rho}]$ are identical and we can omit the conclusion altogether by setting $\pi' \equiv \pi$ as in the case $\sigma \equiv x_i$ of (L_1).

If $\phi \equiv P(\sigma_1, \ldots, \sigma_m)$ then we construct π' similarly as in the case $\sigma \equiv R(\sigma_1, \ldots, \sigma_m)$ of (L_1) and use a predicate substitution rule for P.

If $\phi \equiv \sigma_1 = \sigma_2$ then we construct π' by applying (L_1) to terms σ_1 and σ_2 and then by using the rules of identity as shown:

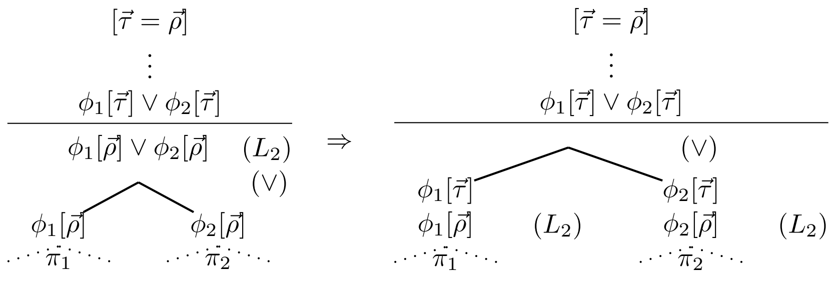
$$\begin{array}{ll}
\multicolumn{2}{c}{[\vec{\tau} = \vec{\rho}]} \\
\multicolumn{2}{c}{\vdots} \\
\multicolumn{2}{c}{\sigma_1[\vec{\tau}] = \sigma_2[\vec{\tau}]} \\
\hline
\sigma_2[\vec{\tau}] = \sigma_2[\vec{\rho}] & (L_1) \\
\sigma_1[\vec{\tau}] = \sigma_2[\vec{\rho}] & (\mathit{Trans}) \\
\sigma_1[\vec{\tau}] = \sigma_1[\vec{\rho}] & (L_1) \\
\sigma_1[\vec{\rho}] = \sigma_1[\vec{\tau}] & (\mathit{Sym}) \\
\sigma_1[\vec{\rho}] = \sigma_2[\vec{\rho}] & (\mathit{Trans}) \\
\cdots\bar{\pi}\cdots &
\end{array}$$

If $\phi \equiv \neg\phi_1$ then we may assume that π starts with an inversion of $(\neg)$-rule applied to $\neg\phi_1[\vec{\rho}]$ as shown in the following on the left:


$[\vec{\tau} = \vec{\rho}]$
$\neg\phi_1[\vec{\tau}]$
$\neg\phi_1[\vec{\rho}]$ (L_2)
$\phi_1[\vec{\rho}]*$ $(\neg)$
$\bar{\pi}_1$
$\Rightarrow$
$[\vec{\tau} = \vec{\rho}]$
$\neg\phi_1[\vec{\tau}]$
(C)
$\phi_1[\vec{\rho}]$
$\phi_1[\vec{\rho}]*$
$\rho_1 = \tau_1$ (Sym)
π_1
$\rho_n = \tau_n$ (Sym)
$\phi_1[\vec{\tau}]$ (L_2)
$\phi_1[\vec{\tau}]*$ $(\neg)$

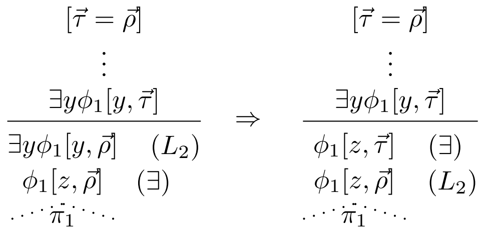
We construct π' as shown on the right by introducing a cut on $\phi_1[\vec{\rho}]$ and on the assumption side by applying IH to $\phi[\vec{\rho}]$ to obtain $\phi[\vec{\tau}]$ after inserting n symmetry rules. The branch is then closed by applying $(\neg)$ to $\neg\phi_1[\vec{\tau}]$ in the premises.

If $\phi \equiv \phi_1 \vee \phi_2$ then we may assume that π starts with an inversion of $(\vee)$-rule applied to $\phi_1[\vec{\rho}] \vee \phi_2[\vec{\rho}]$ as shown in the following on the left:


$[\vec{\tau} = \vec{\rho}]$
$\phi_1[\vec{\tau}] \vee \phi_2[\vec{\tau}]$
$\phi_1[\vec{\rho}] \vee \phi_2[\vec{\rho}]$ (L_2)
$(\vee)$
$\phi_1[\vec{\rho}]$
$\phi_2[\vec{\rho}]$
$\bar{\pi}_1$
$\bar{\pi}_2$
$\Rightarrow$
$[\vec{\tau} = \vec{\rho}]$
$\phi_1[\vec{\tau}] \vee \phi_2[\vec{\tau}]$
$(\vee)$
$\phi_1[\vec{\tau}]$
$\phi_2[\vec{\tau}]$
$\phi_1[\vec{\rho}]$ (L_2)
$\phi_2[\vec{\rho}]$ (L_2)
$\bar{\pi}_1$
$\bar{\pi}_2$

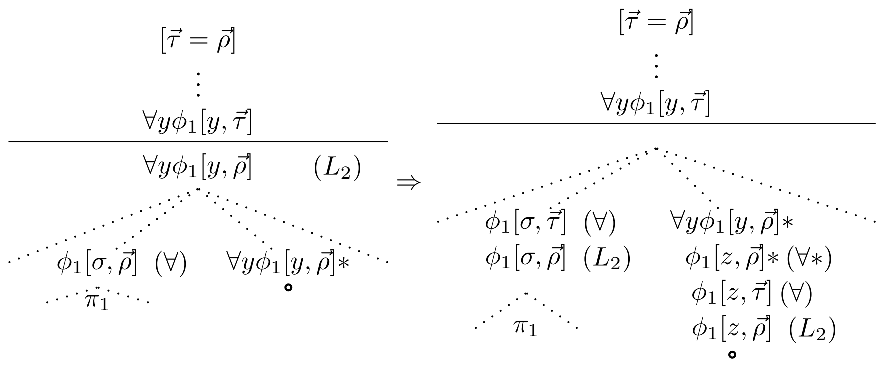
We construct π' as shown on the right by applying $(\vee)$-rule to the assumption $\phi_1[\vec{\tau}] \vee \phi_2[\vec{\tau}]$ and then by applying (L_2) on both sides by IH.

If $\phi \equiv \exists y \phi_1[y, \vec{x}]$ then we may assume that π starts with an inversion of $(\exists)$-rule with an eigen-variable z applied to $\exists y \phi_1[y, \vec{\rho}]$ as shown in the following on the left:


$[\vec{\tau} = \vec{\rho}]$
$\exists y \phi_1[y, \vec{\tau}]$
$\exists y \phi_1[y, \vec{\rho}]$ (L_2)
$\phi_1[z, \vec{\rho}]$ $(\exists)$
π_1
$\Rightarrow$
$[\vec{\tau} = \vec{\rho}]$
$\exists y \phi_1[y, \vec{\tau}]$
$\phi_1[z, \vec{\tau}]$ $(\exists)$
$\phi_1[z, \vec{\rho}]$ (L_2)
π_1

We construct π' as shown on the right by applying the $(\exists)$-rule with an eigen-variable z to the premise $\exists y \phi_1[y, \vec{\tau}]$ and then by using (L_2) by IH.

If $\phi \equiv \forall y \phi_1[y, \vec{x}]$ then in the tableau π shown in the following on the left we show just one of possibly many expansions of a $(\forall)$-rule applied to the assumption $\forall y \phi_1[y, \vec{\rho}]$. We also show a possible closure of a branch in π by this assumption.


$[\vec{\tau} = \vec{\rho}]$
$\forall y \phi_1[y, \vec{\tau}]$
$\forall y \phi_1[y, \vec{\rho}]$ (L_2)
$\phi_1[\sigma, \vec{\rho}]$ $(\forall)$
π_1
$\forall y \phi_1[y, \vec{\rho}]*$
$\Rightarrow$
$[\vec{\tau} = \vec{\rho}]$
$\forall y \phi_1[y, \vec{\tau}]$
$\phi_1[\sigma, \vec{\tau}]$ $(\forall)$
$\phi_1[\sigma, \vec{\rho}]$ (L_2)
π_1
$\forall y \phi_1[y, \vec{\rho}]*$
$\phi_1[z, \vec{\rho}]*$ $(\forall *)$
$\phi_1[z, \vec{\tau}]$ $(\forall)$
$\phi_1[z, \vec{\rho}]$ (L_2)

We construct π' as shown on the right by reconstructing all branches similar to the shown ones as follows. In the branch on the left we instantiate the premise $\forall y \phi_1[y, \vec{\tau}]$ with $y := \sigma$ and then apply (L_2) by IH. In the branch on the right we apply the eigen-variable to $\forall y \phi_1[y, \vec{\rho}]*$ with a new eigen-variable z. We then instantiate the premise $\forall y \phi_1[y, \vec{\tau}]$ with $y := z$ and then close the branch by applying (L_2) by IH.

The remaining propositional cases for ϕ are dealt with similarly. □

6.1.3 Theorems of Leibnitz. For any terms $\sigma[x_1, \ldots, x_n]$ and formulas $\phi[x_1, \ldots, x_n]$ we obtain as an immediate consequence of rules of Leibnitz:

$$\vdash \tau_1 = \rho_1 \wedge \ldots \wedge \tau_n = \rho_n \rightarrow \sigma[\tau_1, \ldots, \tau_n] = \sigma[\rho_1, \ldots, \rho_n] \qquad (1)$$

$$\vdash \tau_1 = \rho_1 \wedge \ldots \wedge \tau_n = \rho_n \wedge \phi[\tau_1, \ldots, \tau_n] \rightarrow \phi[\rho_1, \ldots, \rho_n] \, . \qquad (2)$$

6.1.4 Identity theorem. If the variable x is possibly free in a formula $\phi[x]$ and does not occur in a term τ then

$$\vdash \exists x(x = \tau \land \phi[x]) \leftrightarrow \phi[\tau] \tag{1}$$

$$\vdash \forall x(x = \tau \rightarrow \phi[x]) \leftrightarrow \phi[\tau]\ . \tag{2}$$

(1): In the direction ($\rightarrow$) assume $x = \tau$ and $\phi[x]$ for some x. We obtain $\phi[\tau]$ by the theorem of Leibnitz 6.1.3(2). In the direction ($\leftarrow$) assume $\phi[\tau]$. Since $\tau = \tau$, we get $\exists x(x = \tau \land \phi[x])$ by setting $x := \tau$.

(2): This is similar.

6.1.5 Equivalence theorem. This paragraph is not yet finished. If a formula ϕ contains an occurrence of a formula $\psi[\vec{x}]$:

$$\psi \equiv \ldots \psi[\vec{x}] \ldots$$

where we have indicated that all bound variables of ϕ in whose scope lies the occurrence of ψ are among $\vec{x}$. If the formula ϕ_1 is obtained from ϕ by replacing this occurrence of ψ by a formula ψ_1. then

$$\vdash \forall \vec{x}(\psi \leftrightarrow \psi_1) \land \phi \rightarrow \phi_1\ . \tag{1}$$

Note that we have as an immediate consequence by the Generalization rule 5.3.8:

$$\vdash \psi \leftrightarrow \psi_1 \Rightarrow\ \vdash \phi \leftrightarrow \phi_1 \tag{2}$$

6.1.6 Variant theorem. We say that a formula ϕ_1 is a *variant* of ϕ if ϕ_1 differs from ϕ only in the names of its bound variables. More precisely, if ϕ_1 is obtained from ϕ by a sequence of replacements of its subformulas $\exists x\psi[x]$ or $\forall x\psi[x]$ by the corresponding subformulas $\exists y\psi[y]$ or $\forall x\psi[y]$ for a variable y not free in ψ.

If ϕ_1 is a variant of ϕ we have

$$\vdash \phi \leftrightarrow \phi_1\ . \tag{1}$$

Thus follows from the following properties by 6.1.5(2):

$$\vdash \exists x\psi[x] \leftrightarrow \exists y\psi[y] \tag{2}$$

$$\vdash \forall x\psi[x] \leftrightarrow \forall y\psi[y]\ . \tag{3}$$

(2): In the direction ($\rightarrow$) assume $\psi[x]$ for some x and obtain $\exists y\phi[x]$ by setting $y := x$. The direction ($\leftarrow$) is similar.

(3): In the direction ($\rightarrow$) assume $\forall x\psi[x]$ and take any y. We get $\phi[y]$ by instantiating the assumption with $x := y$. The direction ($\leftarrow$) is similar.

6.2 Extensions of Theories

6.2.1 Theories. A *theory* T *in* $\mathcal{L}$ is a set of sentences of some first-order language $\mathcal{L}$. We call $\mathcal{L}$ *the language of* T and designate it by $\mathcal{L}_T$.

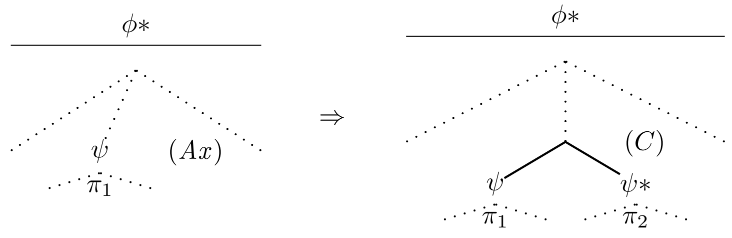
Sentences of T are called *axioms* of the theory T. A theory is *open* if its axioms are universal closures of quantifier-free formulas, i.e. if every axiom has a form $\forall\vec{x}\phi$ with ϕ a formula without quantifiers.

A formula ϕ of $\mathcal{L}_T$ is a *theorem* of T if $T \vdash \phi$. Axioms $\phi \in T$ are trivially theorems of T.

6.2.2 Lemma. *If T, T_1, S are theories in $\mathcal{L}$ and ϕ a formula of $\mathcal{L}$ then*

$$S \vdash T \textit{ and } T, T_1 \vdash \phi \Rightarrow S, T_1 \vdash \phi\ .$$

Proof. Assume $S \vdash T$ and $\pi : T, T_1 \vdash \phi$ where π is shown in the following on the left:



with one of possibly many expansions by an axiom $\psi \in T$. We construct the tableau π' shown on the right by replacing every such expansion by a cut on ψ whose right branch is closed by a tableau π_2 such that $\pi_2 : S \vdash \psi$. We assume that the eigen-variables of π_2 were systematically renamed so they do not occur freely in the branch above $\psi*$. We clearly have $\pi' : S, T_1 \vdash \phi$. □

6.2.3 Consistency of theories. A theory T is *consistent* if $T \nvdash \bot$. Because $\bot \to \phi$ is a tautology, every formula ϕ is a theorem of an inconsistent theory. An inconsistent theory is thus worthless and so the consistency is the minimal requirement on theories. Because the consistence is defined by provability it is a syntactic concept. The consistency of open theories is syntactically characterized by the theorem of Hilbert-Ackermann Thm. 6.2.4 and the consistency of all theories is semantically characterized in Thm. 6.2.5.

If the reader finds the proof of the first characterization theorem trivial he should bear in mind that our tableau system has a subformula property where the proofs proceed without any detours such as the formulas introduced by the cut rules. Hilbert [9] (see also Shoenfield [23]) used a formal system based on modus ponens which is a form of cut. The theorem becomes non-trivial in cut-based proof systems. The difficulty of its proof is comparable to that of the proof of our non-trivial theorem on the admissibility of cuts.

6.2.4 Theorem (Hilbert-Ackermann). *An open theory is inconsistent iff $\vdash_i \bigwedge S \to \bot$ for a finite set S of instances of its axioms.*

Proof. Let T be an open theory. If T is inconsistent then we have $\pi : T \vdash \bot$ for some basic tableau π. The only quantifier rules in π are $(\forall)$-rules instantiating the axioms of T. We remove from π all formulas of a form $\forall\phi$, which can be only axioms from T and its instantiations, whereby we obtain a tableau π_1 s.t. $\pi_1 : S \vdash_i \bot$ where S is a finite sets of all quantifier-free instances of axioms in T. We then get $\vdash_i \bigwedge S \to \bot$ by the Deduction theorem 5.3.4.

Vice versa, if $\vdash_i \bigwedge S \to \bot$ then $\pi : S \vdash_i \bot$ by Thm. 5.3.4 and we can insert into π in front of every application of an axiom $\phi \in S$ the appropriate axiom $\forall\phi \in T$ followed by instantiations by $(\forall)$-rules leading to S whereby we obtain $T \vdash \bot$. □

6.2.5 Theorem. *A theory is consistent iff it has a model.*

Proof. We have $T \vdash \bot$ iff $T \models \bot$, i.e. a theory T is consistent iff $T \not\models \bot$ by the Completeness theorem 5.3.13. If T has no model, i.e. if $\mathcal{M} \not\models T$ for all structures $\mathcal{M}$ for $\mathcal{L}_T$, then $T \models \bot$ and so T is inconsistent. Vice versa, if T is inconsistent, i.e. if $T \models \bot$, then take any structure $\mathcal{M}$ for $\mathcal{L}_T$. We have $\mathcal{M} \not\models \bot$ and so $\mathcal{M} \not\models T$. Hence T has no model. □

6.2.6 Extensions of theories. A theory S is an *extension* of a theory T if $\mathcal{L}_S$ is an extension of $\mathcal{L}_T$ and every theorem of T is a theorem of S, i.e.

$$T \vdash \phi \Rightarrow S \vdash \phi \quad \text{for all formulas } \phi \text{ of } \mathcal{L}_T \ .$$

Note that T is an extension of itself. Also S is an extension of T iff $\mathcal{L}_S$ extends $\mathcal{L}_T$ and $S \vdash T$ by Lemma 6.2.2 but this does not imply that the axioms of T are among those of S, i.e. $S \subseteq T$.

Let S be an extension of T and K a set of formulas of $\mathcal{L}_T$. We say that T and S are *equivalent on* K if $S \vdash K \Rightarrow T \vdash K$. We designate this by writing $T \equiv_K S$. If K consists of all formulas of $\mathcal{L}_T$ we write $T \equiv_{\mathcal{L}_T} S$.

Note that if T and S are equivalent on K as above then, since $T \vdash K \Rightarrow S \vdash K$ because S extends T, a formula of K is a theorem of S iff it is a theorem of T.

The theories T and S are *equivalent* if S extends T and T extends S. Note that we then have $\mathcal{L}_T = \mathcal{L}_S$ and $T \equiv_{\mathcal{L}_T} S$ which we write simply as $T \equiv S$.

The following theorem characterizes the situation when we can consistently extend a theory T by adding a single axiom without extending the language. We can, namely, add the axiom iff its negation is unprovable in T.

6.2.7 Theorem. *Let T be a theory and ϕ a sentence of $\mathcal{L}_T$. The extended theory T, ϕ is consistent iff $T \nvdash \neg\phi$.*

Proof. If $T \vdash \neg\phi$ then, since $T, \phi \vdash \phi$, we have $T \vdash \bot$. Vice versa, if $T, \phi \vdash \bot$ then $T \vdash \phi \to \bot$ by the Deduction theorem and so $T \vdash \neg\phi$. □

6.2.8 Conservative extensions. Not all extensions are interesting. Uninteresting extensions are extensions turning consistent theories into inconsistent ones.

For instance, take a formalized theory of natural numbers which we will study in a form of so called Peano arithmetic, shortly PA, in the second part of this text. The square root of 2 is not a natural number (not even a rational one as was already known to Greeks). An attempt to extend PA with a new constant symbol $\sqrt{2}$ and a new axiom $2 = \sqrt{2}\cdot\sqrt{2}$ yields an extension PA_1 of PA because we trivially have $\mathrm{PA}_1 \vdash \mathrm{PA}$. Unfortunately PA_1 is inconsistent whereas PA is consistent. This is because we trivially have $\mathrm{PA}_1 \vdash 2 = \sqrt{2}\cdot\sqrt{2}$ but $\mathrm{PA} \vdash \forall x\, 2 \neq x\cdot x$, hence $\mathrm{PA}_1 \vdash \forall x\, 2 \neq x\cdot x$, and so $\mathrm{PA}_1 \vdash 2 \neq \sqrt{2}\cdot\sqrt{2}$. Because $\phi \land \neg\phi \to \bot$ is a tautology we then get $\mathrm{PA}_1 \vdash \bot$.

This leads us to extensions S of T which are equivalent with T on $\mathcal{L}_T$, i.e. such that $T \equiv_{\mathcal{L}_T} S$. Such extensions are called *conservative* extensions because they do not add any new theorems in the language of T. Specifically if T is consistent then $\bot$, which is in $\mathcal{L}_T$, is not a theorem of T and we cannot have $S \vdash \bot$. Hence also S is consistent.

In order to prove that the extension S of T is conservative, which is more simply put as *S is conservative over T*, it suffices to prove that whenever $S \vdash \phi$ where ϕ is a formula of $\mathcal{L}_T$ then also $T \vdash \phi$. Note that the converse holds because S extends T and so every theorem of T is a theorem of S. By the Generalization rule 5.3.8 it suffices to prove the above for the sentences ϕ of $\mathcal{L}$.

Note that the extension of T to T, ϕ when $T \nvdash \neg\phi$ is not in general conservative because the extended theory proves ϕ which can be *undecidable* in T, i.e. neither ϕ nor $\neg\phi$ are provable in T.

6.3 Extensions by Explicitly Defined Predicates

Mathematicians often introduce new predicates as abbreviations for larger formulas. They do it in order to increase the readability of their theorems and to shorten their proofs. A typical example is the introduction of the predicate $x < y$ into a theory T which contains the binary function of addition and the constant 1. Formulas of a form $\tau_1 < \tau_2$ can be viewed as abbreviations for formulas

$$\exists z(\tau_1 + (z + 1) = \tau_2)$$

where the variable z is new. One can then proceed to prove properties of $<$, say the transitivity:

$$T \vdash x < y \land y < z \to x < z\ ,$$

always keeping on mind that this is just an abbreviation for a larger and less readable theorem

$$T \vdash \exists a(x+1+(a+1)=y) \land \exists b(y+(b+1)=z) \to \exists c(x+(c+1)=z)\ .$$

The problem with this approach is that when one wants to prove a metatheoretical, theorem on provability in T, which often happens in logic but rarely in mathematics, one has to eliminate all abbreviations. A cleaner approach from a logical point of view is the extension of $\mathcal{L}_T$ with a new binary predicate symbol $<$ and the addition to T of a new axiom for $<$ which is any universal closure of:

$$x < y \leftrightarrow \exists z(x+(z+1)=y)\ .$$

We then wish to know that the extended theory S is conservative over T, i.e. that it does not add any power to T beyond notational convenience. One can actually prove more than this by defining a translation $\phi^\star$ into $\mathcal{L}_T$ of every formula ϕ of $\mathcal{L}_S$ such that we have

$$S \vdash \phi \Leftrightarrow T \vdash \phi^\star\ .$$

6.3.1 Extensions by explicitly defined predicates. Let T be a theory, $\phi[\vec{x}]$ a formula of $\mathcal{L}_T$ with the free variables among the indicated ones, and P a new n-ary predicate symbol ($n \geq 0$). Consider the formula

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

which is in the extension $\mathcal{L}_T + P$ of $\mathcal{L}_T$. Designate by S the theory $T, \forall(1)$ whose language is the extension of $\mathcal{L}_T$ with the symbol P and whose axioms are T plus any universal closure of (1) which is called the *defining axiom* for P.

We trivially have $S \vdash T$ and so S is an extension of T which we call *extension by explicitly defined predicate*. The term 'explicit' refers to the fact that the defining axiom is not 'recursive', i.e. that P is not applied in ϕ. This is obviously so because ϕ is in a language which does not contain the predicate symbol P. We then have

$$S \vdash P(\vec{x}) \leftrightarrow \phi[\vec{x}] \tag{2}$$

by Thm. 5.3.8 because S trivially proves its axiom $\forall(1)$.

6.3.2 Translation function. Let S be the extended theory $T, \forall 6.3.1(1)$. For every formula ψ of $\mathcal{L}_S$ we designate by $\psi^\star$ any formula of $\mathcal{L}_T$ obtained from ψ by replacing in it every application $P(\vec{\tau})$ by a formula $\phi'[\vec{\tau}]$ where ϕ' is a variant of ϕ such that the substitution of $\vec{\tau}$ in ϕ' is free for $\vec{x}$. We call any such formula $\psi^\star$ a *translation* of ψ.

6.3.3 Translation lemma. *If S is the extended theory $T, \forall 6.3.1(1)$ and ψ a formula of $\mathcal{L}_S$ then we have*

$$S \vdash \psi \leftrightarrow \psi^*\ . \tag{1}$$

Proof. It is clearly sufficient to show

$$S \vdash P(\vec{\tau}) \leftrightarrow \phi'[\vec{\tau}] \tag{2}$$

where $\vec{\tau}$ are terms of $\mathcal{L}_T$ (and thus of $\mathcal{L}_S$) and ϕ' a variant of ϕ because we can then repeatedly apply the Equivalence theorem (6.1.5) to the theorem of S: $\psi \leftrightarrow \psi$ until we eliminate all applications of P on the right.

We prove (2) by working in S where we have $P(\vec{x}) \leftrightarrow \phi'[\vec{x}]$ from the defining axiom of P by the Variant theorem (6.1.6). From this we get the property by instantiating $\vec{x} := \vec{\tau}$. □

6.3.4 Theorem. *The extension $S = T, \forall(P(\vec{x}) \leftrightarrow \phi[\vec{x}])$ of T by the explicitly defined predicate P is conservative and for any formula ψ of $\mathcal{L}_T$ we have*

$$S \vdash \psi \Leftrightarrow T \vdash \psi^\star \ . \tag{1}$$

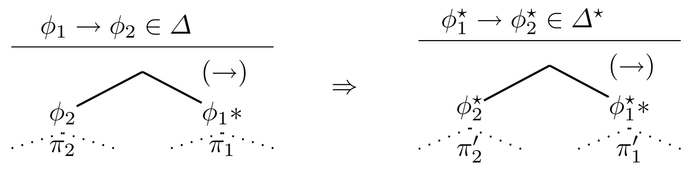
Proof. Assume $S \vdash \psi$ for a formula ψ of $\mathcal{L}_T$. Then $T \vdash \psi^\star$ by 6.3.3(2) and, since $\psi^\star \equiv \psi$, we have $T \vdash \psi$. This proves the conservation. Property (1) is proved in the direction ($\leftarrow$) by assuming $T \vdash \psi^\star$. We have $S \vdash \psi^\star$, since S extends T, and so $S \vdash \psi$ by Lemma 6.3.3.

In the direction ($\rightarrow$) Property (1) follows from an auxiliary property

$$\pi : S \vdash [\varDelta] \Rightarrow T \vdash [\varDelta^\star]$$

where $\varDelta$ is a sequence of formulas of $\mathcal{L}_S$ and $\varDelta^\star$ is the sequence of formulas of $\mathcal{L}_T$ obtained by translating the corresponding formulas of $\varDelta$. The auxiliary property is proved by induction on the structure of π. If π is empty then $\varDelta$ is closed. This happens if $\top* \in \varDelta$ but then $\top* \equiv \top^\star* \in \varDelta^\star$, or $\bot \in \varDelta$ but then $\bot \equiv \bot^\star \in \varDelta^\star$, or $\phi, \phi* \in \varDelta$ but then $\phi^\star, \phi^\star* \in \varDelta^\star$. In any case $\varDelta^\star$ is closed and we even have $\vdash [\varDelta^\star]$.

If the first expansion in π is by a propositional rule, say ($\rightarrow$), then π has the form shown in the following on the left:



Since $(\phi_1 \rightarrow \phi_2)^\star \equiv \phi_1^\star \rightarrow \phi_2^\star$, we obtain identity tableaux π_1' and π_2' such that $\pi_1' : T \vdash [\varDelta^\star, \phi_1^\star*]$ and $\pi_2' : T \vdash [\varDelta^\star, \phi_2^\star]$ by two IH's from $\pi_1 : S \vdash [\varDelta, \phi_1*]$ and $\pi_2' : S \vdash [\varDelta, \phi_2]$. We then construct the closed tableau for $\varDelta^\star$ from axioms T shown on the right. The remaining propositional, quantifier, identity expansions except by predicate substitution rule applied to P are similar. This is because the translation is recursively applied, for instance, $(\psi_x[\tau] \to \exists x\psi)^\star \equiv {\psi^\star}_x[\tau] \to \exists x\psi^\star$.

If the first expansion in π is by a predicate substitution rule applied to P then π looks as follows:

$$\cfrac{\Delta \qquad \tau_1 = \rho_1, \dots, \tau_n = \rho_n, P(\tau_1, \dots, \tau_n) \in \Delta}{\begin{array}{c} P(\rho_1, \dots, \rho_n) \qquad (Psub) \\ \cdots \pi_1 \cdots \end{array}}$$

We have $(\tau_i = \rho_i)^\star \equiv \tau_i = \rho_i$, $P(\tau_1, \dots, \tau_n)^\star \equiv \phi'[\tau_1, \dots, \tau_n]$, and $P(\rho_1, \dots, \rho_n)^\star \equiv \phi'[\rho_1, \dots, \rho_n]$ for a variant $\phi'[\vec{x}]$ of $\phi[\vec{x}]$ where we may assume that the bound variables of ϕ' are such that both substitutions are free for $\vec{x}$. For $\pi_1 : S \vdash [\Delta, P(\rho_1, \dots, \rho_n)]$ we obtain $\pi_1' : T \vdash [\Delta^\star, \phi'[\rho_1, \dots, \rho_n]]$ by IH. We have

$$\pi_2 : \vdash [\tau_1 = \rho_1, \dots, \tau_n = \rho_n, \phi'[\tau_1, \dots, \tau_n], \phi'[\rho_1, \dots, \rho_n]*]$$

for some π_2 by a rule of Leibnitz. We then construct the following closed tableau for $\Delta^\star$ from axioms T:

$$\cfrac{\Delta^\star \qquad \tau_1 = \rho_1, \dots, \tau_n = \rho_n, \phi'[\tau_1, \dots, \tau_n] \in \Delta^\star}{\begin{array}{cc} \multicolumn{2}{c}{(C)} \\ \phi'[\rho_1, \dots, \rho_n] & \phi'[\rho_1, \dots, \rho_n]* \\ \cdots \pi_1' \cdots & \cdots \pi_2 \cdots \end{array}}$$

If the first expansion in π is by an axiom rule for $\psi \in T$ then, since $\psi^\star \equiv \psi$, we construct $\pi' : T \vdash [\Delta^\star]$ similarly as for the propositional cases above. If the axiom is the defining axiom for P then π looks as follows:

$$\cfrac{\Delta}{\begin{array}{c} \forall\vec{x}(P(\vec{x}) \leftrightarrow \phi[\vec{x}]) \qquad (Ax) \\ \cdots \pi_1 \cdots \end{array}}$$

We have

$$(\forall\vec{x}(P(\vec{x}) \leftrightarrow \phi[\vec{x}]))^\star \equiv \forall\vec{x}(\phi[\vec{x}] \leftrightarrow \phi[\vec{x}])$$

and for $\pi_1 : S \vdash [\Delta, \forall\vec{x}(P(\vec{x}) \leftrightarrow \phi[\vec{x}])]$ we obtain $\pi_1' : T \vdash [\Delta^\star, \forall\vec{x}(\phi[\vec{x}] \leftrightarrow \phi[\vec{x}])]$ by IH. We clearly have $\pi_2 : \vdash \forall\vec{x}(\phi[\vec{x}] \leftrightarrow \phi[\vec{x}])$ for some π_2. We then construct the following closed tableau for $\Delta^\star$ from axioms T:

$$\cfrac{\Delta^\star}{\begin{array}{cc} \multicolumn{2}{c}{(C)} \\ \forall\vec{x}(\phi[\vec{x}] \leftrightarrow \phi[\vec{x}]) & \forall\vec{x}(\phi[\vec{x}] \leftrightarrow \phi[\vec{x}])* \\ \cdots \pi_1' \cdots & \cdots \pi_2 \cdots \end{array}} \qquad \Box$$

6.4 Skolem Extensions

6.4.1 Skolem axioms. Suppose that T is a theory whose language does not contain the n-ary function symbol f $(n \geq 0)$. Further suppose that $\phi[\vec{x}, y]$ is a formula of $\mathcal{L}_T$ with the free variables among the $n+1$ indicated ones. The sentence

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

is called the *Skolem axiom for ϕ and f*. The reader will note that in view of prenex theorems of predicate calculus we could have equivalently written the sentence (1) also as a universal closure of

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

Extension of T to S by the addition of the function symbol f to its language and of the Skolem axiom (1) to its axioms is a *Skolem extension* of T. We will prove in this section that S is conservative over T both by semantic and finitary proofs. For that we use until the end of this section the abbreviation

$$\phi_1[\vec{x}, y] \equiv \exists y\phi[\vec{x}, y] \to \phi[\vec{x}, y])$$

and call the formulas

$$\forall x_i \ldots \forall x_n \phi_1[\tau_1, \ldots, \tau_{i-1}, x_i, \ldots, x_n, \tau_1, \ldots, f(\tau_{i-1}, x_i, \ldots, x_n)] \tag{3}$$

where $1 \leq i \leq n$ *partial* instances of the Skolem axiom and formulas $\phi_1[\vec{\tau}, f(\vec{\tau})]$ *full* instances.

Terms of the form $f(\vec{\tau})$ are called *f-terms* and function substitution expansion rules for f are called *f-substitution rules.* If the conclusion of an f-substitution rule is $f(\vec{\tau}) = f(\vec{\tau})$ then the rule is called *trivial.* Trivial f-substitution rules are not needed because their conclusions $f(\vec{\tau}) = f(\vec{\tau})$ can be obtained also by reflexivity rules.

We will need below the following notation. We designate by $\mathcal{L}_t$ the witnessing extension of $\mathcal{L}_T$, by $\mathcal{L}_s$ the witnessing extension of $\mathcal{L}_S$. A formula of $\mathcal{L}_S$ is *free for f* if it does not contain any f-terms whose arguments contain a bound occurrence of a variable. We denote by H_s all Henkin witnessing and counterexample axioms for $\mathcal{L}_S$ which are free for f, and by H_t all Henkin witnessing and counterexample axioms for $\mathcal{L}_T$. We designate by Sk_s the set of of full instances of the Skolem axiom: $\phi_1[\vec{\tau}, f(\vec{\tau})]$ where $\vec{\tau}$ are closed terms of $\mathcal{L}_s$.

6.4.2 Semantic proof of conservativity of Skolem extensions. Skolem axioms 6.4.1(1) are a generalization of Henkin witnessing axioms where the term $f(\vec{\tau})$ acts as a witness for the formula $\exists y\phi[\vec{\tau}, y]$. If f is a constant, i.e. if $n = 0$, then the Skolem axiom is

$$\exists y \phi[y] \to \phi[f] \tag{1}$$

and it looks like the Henkin witnessing axiom for $\exists y\phi[y]$ except that the constant symbol f is not fixed as it is for Henkin constants. The reader will note that the theorem on the elimination of Henkin witnessing axioms (5.3.10) asserts that the extension of any theory T in $\mathcal{L}$ to the theory T, Ha in $\mathcal{L}_c$ is conservative.

We will now prove that Skolem extensions are conservative. When the new function symbol f is a constant then this follows Thm. 5.3.10. Indeed, if $\pi : T, (1) \vdash \psi$ for a sentence ψ of $\mathcal{L}_T$ then $\pi' : T, Ha \vdash \psi$ where π' is formed by systematically replacing in π the constant symbol f by the Henkin constant c belonging to $\exists y\phi[y]$. We then obtain $T \vdash \psi$ by Thm. 5.3.10.

The above proof is called *finitary* in mathematical logic. For our purposes it suffices to say that finitary proofs are such when one manipulates by constructive means tableaux which are syntactic objects. Finitary proofs are considered more convincing than the semantic arguments by means of models. As it happens, the extension Thm. 6.4.5 of the above finitary proof to the case when $n > 0$, which is the main result of this section, is extremely non-trivial. On the other hand, the semantic proof is almost trivial.

The semantic proof that the theory S from Par. 6.4.1 is conservative over T is by assuming $S \vdash_1 \psi$ for a sentence ψ of $\mathcal{L}_T$. We have $S \vDash \psi$ by Thm. 5.3.13 and it suffices to prove $T \vDash \psi$. So let $\mathcal{M}$ for $\mathcal{L}_T$ be a model of T with a domain D such that $d_0 \in D$. We expand $\mathcal{M}$ to a structure $\mathcal{N}$ for $\mathcal{L}_S$ by defining

$$f^{\mathcal{N}}(d_1, \ldots, d_n) = \begin{cases} d & \text{if for some } d \in D \text{ we have } \mathcal{M} \vDash \phi[d_1, \ldots, d_n, d] \\ d_0 & \text{otherwise.} \end{cases}$$

We have $\mathcal{M} \vDash \psi_1$ iff $\mathcal{N} \vDash \psi_1$ for all sentences ψ_1 of $\mathcal{L}_T$ by Thm. 5.2.7. Hence $\mathcal{N} \vDash T$ and we can easily prove $\mathcal{N} \vDash \forall$6.4.1(1). Thus $\mathcal{N} \vDash \psi$ from the assumption and then $\mathcal{M} \vDash \phi$, since ψ is a sentence of $\mathcal{L}_T$.

6.4.3 Invariance of tableau rules under replacement of f-terms. Let the theories T and S be as in Par. 6.4.1. For the finitary proof of conservativity of S over T we need to investigate the effect of systematic replacements of f-terms by another terms in tableaux $\pi : T, Sk_s, H_s \vdash \psi$ where ψ is a sentence of $\mathcal{L}_T$. We assume that the tableau π, which is in the language $\mathcal{L}_s$, consists of sentences only. This means that there are no eigen-variable rules in π. We will see from the proof of Lemma 6.4.4 that π might contain, in addition to the basic expansion rules, also cut rules applied to identities $\tau_1 = \tau_2$ and applications of Leibnitz rules in the following form:

$$\frac{\tau_1 = \rho_1 \ \ldots\ \tau_n = \rho_n \ \ \phi_1[\vec{\tau}, \vec{\sigma}]}{\phi_1[\vec{\tau}, \vec{\sigma}]} \,(L_2)\ .$$

The f-terms can be introduced into π by axioms H_s and Sk_s, by quantifier instantiation, cut, reflexivity, and by f-substitution rules.

Quantifier instantiation rules are invariant under the replacement of f-terms by other terms unless arguments of f-terms contain bound variables, i.e. they are not free fro f. For instance, take the partial instance

$$\forall x_n \phi_1[\tau_1, \ldots, \tau_{n-1}, x_n, f(\tau_1, \ldots, \tau_{n-1}, x_n)]$$

of the Skolem axiom 6.4.1(1). If it used as a premise to a $(\forall)$-instantiation rule with the conclusion $\phi_1[\tau_1, \ldots, \tau_{n-1}, x_n, f(\tau_1, \ldots, \tau_{n-1}, \tau_n)]$ then the replacement of the closed term $f(\tau_1, \ldots, \tau_{n-1}, \tau_n)$ by a different term invalidates the rule. Note that an replacement of an f-term occurring, say, in τ_{n-1} does not destroy the character of the rule because the same change is done in both the premise and in the conclusion. The reader will note that the tableau π can apply only invariant quantifier instantiation rules.

An f-substitution rule with the conclusion $f(\vec{\tau}) = f(\vec{\rho})$ is not invariant under the replacement of the f-terms $f(\vec{\tau})$ and $f(\vec{\rho})$. It is, however, invariant under the replacement of an f-term anywhere in the arguments $\vec{\tau}$ and $\vec{\rho}$.

An axiom $\phi_1[\vec{\tau}, f(\vec{\tau})]$ from Sk_s is not invariant under the replacement of $f(\vec{\tau})$ but it is invariant under the replacement of any other f-terms applied in $\vec{\tau}$.

An axiom $\exists x \psi_1[x] \to \psi_1[c_1]$ or $\psi_2[c_2] \to \forall x \psi_2[x]$ from H_s is not invariant under the replacement of an f term even though it is free for f. If, however, the replacement is *deep* in the sense that it also systematically modifies the Henking constants c_1 and c_2 the invariance can be restored. In order to explain this we recall that the Henkin constant c_j of rank $i+1$ from $\mathcal{L}_s$ belongs to the sentence $\exists x \psi_3[x]$ which is at the j-th in the enumeration of existential sentences 5.1.6(1) of $\mathcal{L}_s$ of rank i. We can thus visualize the Henkin constant c_j written with *symbolic index*: $c_{\exists x \psi_3[x]}$. The deep replacement of an f-term in a Henkin witnessing $\exists x \psi_1[x] \to \psi_1[c_{\exists x \psi_1[x]}]$ or counterexample $\psi_2[c_{\exists x \neg \psi_2[x]}] \to \forall x \psi_2[x]$ axiom means that we replace the f-term also in the symbolic indices of Henkin constants. Note that the symbolic indices $\exists x \psi_1[x]$ and $\exists x \neg \psi_2[x]$ may again contain Henkin constants (of lower rank) and we must perform the deep replacement also in them. After the deep replacement we look up the symbolic indices in the corresponding enumerations 5.1.6(1) and replace them by ordinary indices. It should be clear that the deep replacement changes an axiom from H_s into an axiom from H_s and so its use in the tableau π can be visualized to be invariant under the deep replacement of f-terms.

All other expansion rules, including the cut and Leibnitz rules in the above form, are invariant under systematic replacements of f-terms by different terms. Since a Henkin constant can be replaced by another Henkin constant without invalidating any rules except possibly those in H_s, the deep replacement of f-terms in π may invalidate at most the f-substitution rules and the axioms from Sk_s.

6.4.4 Lemma (Elimination of f-substitution rules). *If for a sentence ψ of $\mathcal{L}_T$ as described in Par. 6.4.1 we have $\pi : T, Sk_s, H_s \vdash \psi$ where π*

consists of sentences only and its expansion rules, except the axioms Sk_s and f-substitution rules, are invariant under the deep replacement of f-terms then there is a similar tableau π_0 such that $\pi_0 : T, Sk_s, H_s \vdash \psi$ but π_0 does not apply non-trivial f-substitution rules.

Proof. Take any π satisfying the assumption os the lemma. We determine the *weight* of tableaux with f-terms those occurring in π as follows. We order all distinct f-terms occurring in the tableau π into a finite sequence

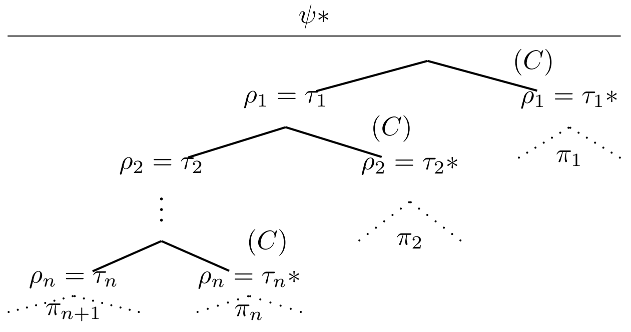
$$\sigma_1 \; \sigma_2 \; \ldots \; \sigma_k$$

ordered by the non-decreasing number of applications of the symbol f. The *weight* of the f-term τ_i is i. Note that the f-terms with higher weight cannot occur as subterms of f-terms with lesser weight.

A conclusion $\sigma_i = \sigma_j$ of a non-trivial f-substitution rule is assigned the *weight* $k \cdot \max(i, j) + \min(i, j)$. The *weight* of a tableau with all f-terms some σ_i is the maximum of weights of its non-trivial conclusions of f-substitution rules.

The lemma is proved by induction on the weight of π. If there are at most trivial f-substitution rules in π then we set $\pi_0 \equiv \pi$ and we are done. Note that this includes the case when $n = 0$ because then there are no f-rules.

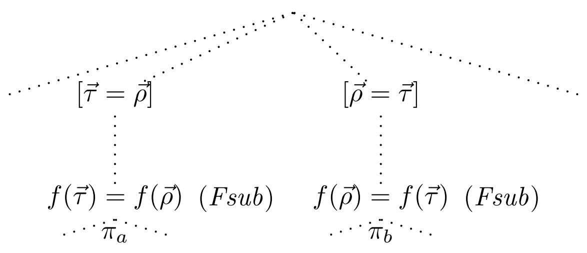
If the weight of π is $m > 0$ then we select all non-trivial conclusions $f(\vec{\tau}) = f(\vec{\rho})$ and $f(\vec{\rho}) = f(\vec{\tau})$ of f-substitution rules with the weight m. We may assume without loss of generality that the weight of $f(\vec{\tau})$ is higher than the weight of $f(\vec{\rho})$. We intend to eliminate such rules, and thereby decrease the weight of π, by turning the f-substitution rules into trivial ones by deep replacing all occurrences of terms $f(\vec{\tau})$ by the term $f(\vec{\rho})$. As such a replacement may invalidate some f-substitution rules and axioms Sk_s we have to proceed with caution. We construct the tableau $\pi' : T, Sk_s, H_s \vdash \psi$ with weight $< m$ as:



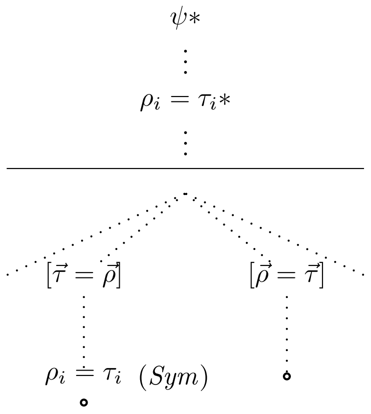
where we will now determine the subtableaux $\pi_1, \ldots, \pi_n$, and π_{n+1}.

The tableau π can be visualized as follows:

(1)


$[\vec{\tau} = \vec{\rho}]$
$[\vec{\rho} = \vec{\tau}]$
$f(\vec{\tau}) = f(\vec{\rho})$ $(Fsub)$
$f(\vec{\rho}) = f(\vec{\tau})$ $(Fsub)$
π_a
π_b

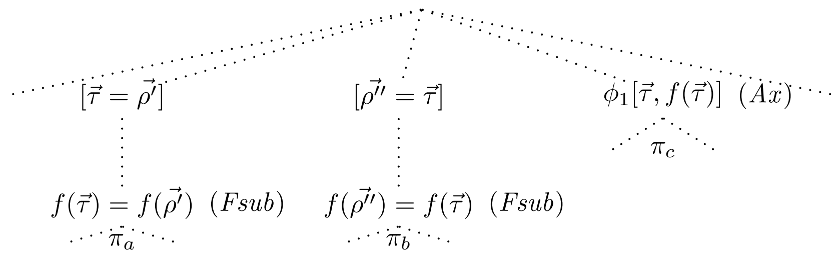
where we have indicated two typical conclusions of the f-substitution rules with the weight m. The notation $[\vec{\tau} = \vec{\rho}]$ is just an indication that the n-assumptions $\tau_1 = \rho_1, \ldots, \tau_n = \rho_n$ occur somewhere along the branch. For any i s.t. $1 \leq i \leq n$ we form the tableau π_i from π by the following replacements:


$\psi*$
$\rho_i = \tau_i*$
$[\vec{\tau} = \vec{\rho}]$
$[\vec{\rho} = \vec{\tau}]$
$\rho_i = \tau_i$ (Sym)

The reader will note that the tableau π_i is in the tableau π' at the end of the branch with the goal $\rho_i = \tau_i*$. In the transformation on the right the branch thus closes against the assumption $\rho_i = \tau_i$ which is among the ones indicated by $[\vec{\rho} = \vec{\tau}]$. In the transformation on the left we apply the symmetry rule to the assumption $\tau_i = \rho_i$ which is among the ones indicated by $[\vec{\tau} = \vec{\rho}]$. The conclusion $\rho_i = \tau_i$ of the symmetry rule thus closes the branch against the goal $\rho_i = \tau_i*$. We apply the shown transformation to all topmost conclusions in π of the f-substitution rule with the weight m. This means that the tableau π_i is without such rules and it has a weight $< m$.

For the construction of the tableau π_{n+1} we can visualize the tableau π as follows:

(2)


$[\vec{\tau} = \vec{\rho'}]$
$[\vec{\rho''} = \vec{\tau}]$
$\phi_1[\vec{\tau}, f(\vec{\tau})]$ (Ax)
π_c
$f(\vec{\tau}) = f(\vec{\rho'})$ $(Fsub)$
$f(\vec{\rho''}) = f(\vec{\tau})$ $(Fsub)$
π_a
π_b

where we have not shown the conclusions of f-substitution rules with the weight m as visualized in 6.4.4(1). We have instead shown two typical conclusions of non-trivial f-substitution rules containing $f(\vec{\tau})$ and which are with lesser weights. We have also application $\phi_1[\vec{\tau}, f(\vec{\tau})]$ of an axiom from Sk_s.

We now perform the deep replacement in the above copy of π of all f-terms $f(\vec{\tau})$ by the f-terms $f(\vec{\rho})$. The replacement turns all conclusions of f-substitution rules with weights m into conclusions of rules of reflexivity but it invalidates the conclusions of f-substitution rules similar to the two shown. It also invalidates the axiom rule from Sk_s The tableau-like tree after the replacement looks as shown in Fig. 6.1 where we have shown that the tree after a correction will be located in the tableau π' at the position of π_{n+1}.

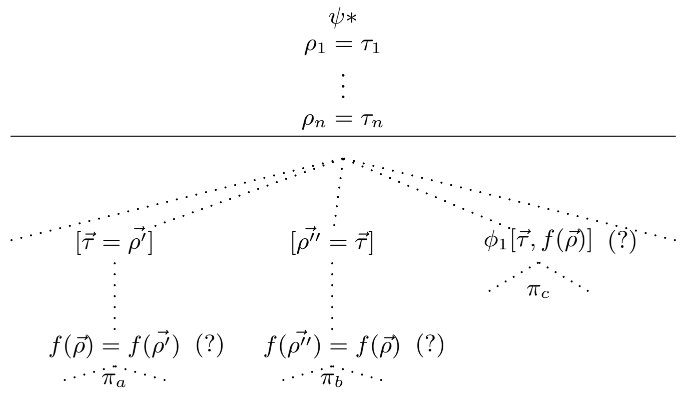

$\psi*$
$\rho_1 = \tau_1$
$\rho_n = \tau_n$
$[\vec{\tau} = \vec{\rho'}]$
$[\vec{\rho''} = \vec{\tau}]$
$\phi_1[\vec{\tau}, f(\vec{\rho})]$ $(?)$
π_c
$f(\vec{\rho}) = f(\vec{\rho'})$ $(?)$
$f(\vec{\rho''}) = f(\vec{\rho})$ $(?)$
π_a
π_b

Fig. 6.1. Tableau-like figure after the deep replacement $f(\vec{\tau}) := f(\vec{\rho})$

Note that the additional terms shown in the figure are not affected by the replacement because the f-term $f(\vec{\tau})$ cannot occur in them. Hovewever, there can be f-terms in the tableau π with higher weights than $f(\vec{\tau})$ and which can contain the last term as subterms. Since there are no non-trivial f-substitution rules for those higher weight f-terms in π, the replacement does not invalidate any of the rules containing the higher weight f-terms although the terms themselves change.

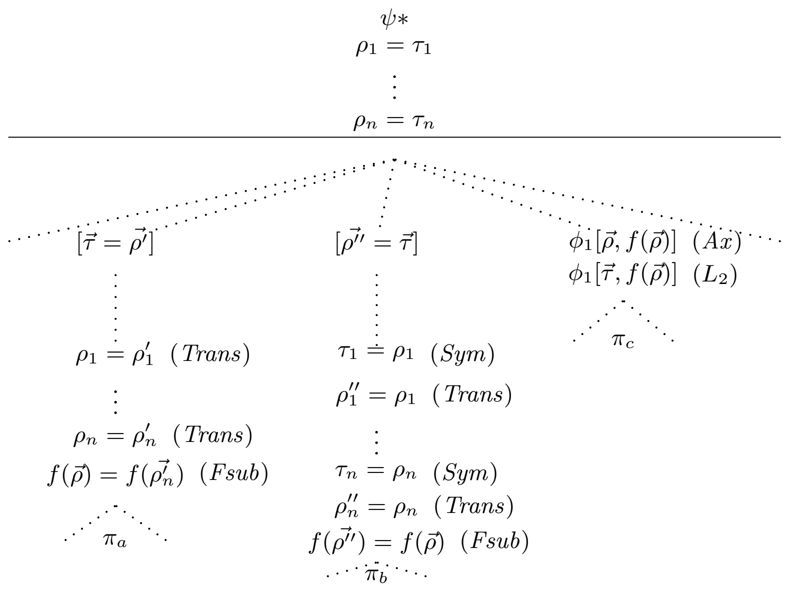

$\psi*$
$\rho_1 = \tau_1$
$\rho_n = \tau_n$
$[\vec{\tau} = \vec{\rho'}]$
$[\vec{\rho''} = \vec{\tau}]$
$\phi_1[\vec{\rho}, f(\vec{\rho})]$ (Ax)
$\phi_1[\vec{\tau}, f(\vec{\rho})]$ (L_2)
π_c
$\rho_1 = \rho'_1$ (Trans)
$\tau_1 = \rho_1$ (Sym)
$\rho''_1 = \rho_1$ (Trans)
$\rho_n = \rho'_n$ (Trans)
$f(\vec{\rho}) = f(\vec{\rho'_n})$ (Fsub)
π_a
$\tau_n = \rho_n$ (Sym)
$\rho''_n = \rho_n$ (Trans)
$f(\vec{\rho''}) = f(\vec{\rho})$ (Fsub)
π_b

Fig. 6.2. Corrected tableau π_{n+1}

We now correct the tableau-like tree in Fig. 6.1 whereby we obtain the tableau π_{n+1} shown in Fig. 6.2. The branch above the tableau π_a is corrected by the insertion of n conclusions $\rho_i = \rho'_i$ of transitivity rules after which the formula $f(\vec{\rho}) = f(\vec{\rho'})$ is a conclusion of an f-substitution rule with weight $< m$. The branch above the tableau π_b is corrected by the insertion of conclusions of symmetry rules $\tau_i = \rho_i$ followed by conclusions of transitivity rules $\rho''_i = \rho_i$. This is repeated n-times after which the formula $f(\vec{\rho''}) = f(\vec{\rho})$ is a conclusion of an f-substitution rule with a weight $< m$. The branch above the tableau π_c is corrected by an axiom $\phi_1[\vec{\rho}, f(\vec{\rho})]$ from Sk_s. We then use an admissible Leibnitz rule (L_2) to introduce the conclusion $\phi_1[\vec{\tau}, f(\vec{\rho})]$.

We perform the just described corrections for all conclusions similar to typical conclusions visualized in (2). The tableau π_{n+1} has a weight $< m$ and so does the tableau π'. We now apply the induction hypothesis to π' whereby we obtain a tableau π_0 s.t. $\pi_0 : T, Sk_s, H_s \vdash \psi$ without any non-trivial f-substitution rules. $\square$

6.4.5 Theorem. *Skolem extensions are conservative.*

Proof. Let S be a Skolem extension of T as in Par. 6.4.1 and assume $\pi : S \vdash \psi$ for a basic tableau π and a formula ψ of $\mathcal{L}_T$. Because of the Generalization rule 5.3.8 we may assume without loss of generality that ψ is a sentence. We now replace in π all free variables other than eigen-variables by arbitrary

Henkin constants from $\mathcal{L}_S$ whereby we obtain a closed tableau π_1 for ψ which is in the language $\mathcal{L}_s$. Note that variables other than eigen-variables can be introduced into π only by quantifier instantiation and reflexivity rules.

The next transformation is the elimination of all eigen-variable rules from π_1. The elimination is similar as in the proof of the Lemma 5.3.11 on the elimination of quantifier rules except that we do not eliminate the quantifier instantiation rules. Since the premises of all eigen-variable rules in π_1 are free for f, the introduced Henkin witnessing and counterexample axioms are from H_s and so we obtain a tableau π_2 consisting of sentences and such that $\pi_2 : S, H_s \vdash \psi$.

We now delete from π_2 all partial instances of the Skolem axiom 6.4.1(1) whereby we obtain a tableau π_3 such that $\pi_3 : T, Sk_s, H_s \vdash \psi$. The tableau satisfies the assumptions of Lemma 6.4.4 and we obtain from it a similar tableau $\pi_4 : T, Sk_s, H_s \vdash \psi$ which is without non-trivial f-substitution rules.

We intend to eliminate from π_4 the applications of axioms from Sk_s:

$$\phi_1[\vec{\tau}, f(\vec{\tau})] \equiv \exists y \phi[\vec{\tau}, y] \rightarrow \phi[\vec{\tau}, f(\vec{\tau})] \ .$$

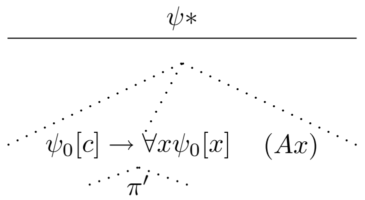
by deeply replacing the closed term $f(\vec{\tau})$ by the Henkin constant c belonging to the sentence $\exists y \phi[\vec{\tau}, y]$. This constant can be written with symbolic index as $c_{\exists y \phi[\vec{\tau}, y]}$ and the deep replacement turns the above axiom to a Henkin witnessing axiom from H_s:

$$\exists y \phi[\vec{\tau}, y] \rightarrow \phi[\vec{\tau}, c_{\exists x \phi[\vec{\tau}, x]}] \ .$$

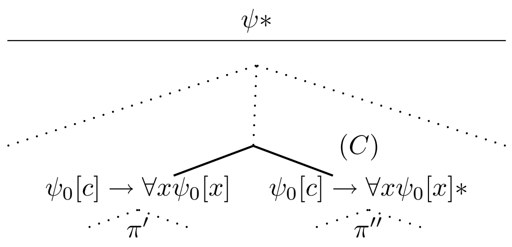
Toward that end we deeply replace all f-terms $f(\vec{\rho})$ in π_4 by the Henkin constants $c_{\exists y \phi[\vec{\rho}, y]}$ whereby we obtain a tableau π_5. The order of elimination of f-terms is immaterial because it leads to identical results but it is crucial that we perform the replacements not only in the formulas of π_4 but also in the indices of all Henking constants occurring in π_4. It should be clear that the elimination of f-terms turns the full instances of Skolem axioms into Henkin witnessing axioms from Ha_s and does not change the character of any other rules in π_4. Specifically, Henkin witnessing axioms are turned into Henkin witnessing axioms without f-terms.

There are no f-terms in the sentences of π_5, nor in the indices of its Henkin constants and we have $\pi_5 : T, H_s \vdash \psi$. We now take every Henkin constant occurring in π_5 and write in the symbolic form $c_{\exists x \psi_0[x]}$ where also the Henkin constants in ψ_0 are with symbolic indices and also the Henkin constants in these symbolic indices are such .... We change the symbolic indices back to the numerical form by looking up the existential sentences in the appropriate enumerations 5.1.6(1) but this time for the language $\mathcal{L}_t$. The look up is done first for the formulas of the lowest rank and then for the formulas with the next higher ranks, ... We thus obtain a tableau π_6 such that $\pi_6 : T, H_t \vdash \psi$.

We eliminate next the Henkin counterexample axioms from π_6 just as it was done in the proof of the theorem on the Introduction/elimination of quantifier rules 5.3.12. We, namely, take any counterexample axiom in π_6:



and replace it by a cut:


$\psi*$
(C)
$\psi_0[c] \to \forall x \psi_0[x]$
$\psi_0[c] \to \forall x \psi_0[x]*$
π'
π''

whose right branch is closed by the tableau π'' obtained by Lemma 5.3.9 to satisfy $\pi'' : Ha \vdash \psi_0[c] \to \forall x \psi_0[x]$. We denote by Ha the Henkin witnessing axioms for $\mathcal{L}_T$. We thus obtain a tableau π_7 such that $\pi_7 : T, Ha \vdash \psi$ to which we apply the Theorem on the conservativity of Henkin witnessing axioms 5.3.10 and obtain $T \vdash \psi$. ☐

6.4.6 Skolemization and Herbrand's theorem. One of the uses of Skolem axioms is to eliminate, by a process called *Skolemization*, quantifiers from formulas at the price of introducing *Skolem* functions. Since we will not need this in the further development, we just illustrate the process with an example.

In order to eliminate the quantifiers from a sentence we bring it to a prenex form, say, $\exists x \forall y \exists z \forall w \phi[x, y, z, w]$ where the formula ϕ is quantifier free. We consider the following Skolem axioms:

$$\forall x(\exists y \forall z \exists w \neg\phi[x, y, z, w] \to \forall z \exists w \neg\phi[x, f(x), z, w])$$
$$\forall x \forall z(\exists w \neg\phi[x, f(x), z, w] \to \neg\phi[x, f(x), z, g(x, z)])$$

for new function symbols f and g. It is not hard to see that we have

$$\vdash \forall x \forall z \neg\phi[x, f(x), z, g(x, z)] \to \forall x \exists y \forall z \exists w \neg\phi[x, y, z, w] \tag{1}$$

in the predicate calculus. For the proof of converse we need the two Skolem axioms and we have

$$T \vdash \forall x \exists y \forall z \exists w \neg\phi[x, y, z, w] \to \forall x \forall z \neg\phi[x, f(x), z, g(x, z)] \tag{2}$$

in the theory T consisting of the two Skolem axioms.

The celebrated theorem of Herbrand asserts

$$\begin{aligned}&\vDash \exists x\forall y\exists z\forall w\phi[x,y,z,w] \Leftrightarrow\\ &\qquad \vDash_i \phi[\tau_1, f(\tau_1), \rho_1, g(\tau_1,\rho_1)] \lor \ldots \lor \phi[\tau_n, f(\tau_n), \rho_n, g(\tau_n,\rho_n)]\\ &\qquad \text{for some } n \geq 1 \text{ and terms } \tau_1,\, \rho_1,\, \ldots,\, \tau_n,\, \rho_n.\end{aligned}$$

We first show

$$\vdash \exists x\forall y\exists z\forall w\phi[x,y,z,w] \Leftrightarrow\ \vdash \exists x\exists z\phi[x,f(x),z,g(x,z)] \tag{3}$$

where the sentence $\exists x\exists z\phi[x,f(x),z,g(x,z)]$ is called a *Herbrand normal form* of the sentence $\exists x\forall y\exists z\forall w\phi[x,y,z,w]$.

(3): In the direction $\Rightarrow$ we assume $\vdash \exists x\forall y\exists z\forall w\phi[x,y,z,w]$. Thus $\vdash \neg\forall x\exists y\forall z\exists w\neg\phi[x,y,z,w]$ and we get $\vdash \neg\forall x\forall z\neg\phi[x,f(x),z,g(x,z)]$ by (1), i.e. $\vdash \exists x\exists z\phi[x,f(x),z,g(x,z)]$.

In the direction $\Leftarrow$ we assume $\vdash \exists x\exists z\phi[x,f(x),z,g(x,z)]$ and get $T \vdash \neg\forall x\forall z\neg\phi[x,f(x),z,g(x,z)]$. We then have $T \vdash \neg\forall x\exists y\forall z\exists w\neg\phi[x,y,z,w]$ by (2). We then get $\vdash \neg\forall x\exists y\forall z\exists w\neg\phi[x,y,z,w]$ by the conservativity of T and hence $\vdash \exists x\forall y\exists z\forall w\phi[x,y,z,w]$.

We show next

$$\begin{aligned}&\vdash \exists x\exists z\phi[x,f(x),z,g(x,z)] \Leftrightarrow\\ &\qquad \vdash_i \phi[\tau_1, f(\tau_1), \rho_1, g(\tau_1,\rho_1)] \lor \ldots \lor \phi[\tau_n, f(\tau_n), \rho_n, g(\tau_n,\rho_n)]\\ &\qquad \text{for some } n \geq 1 \text{ and terms } \tau_1,\, \rho_1,\, \ldots,\, \tau_n,\, \rho_n.\end{aligned} \tag{4}$$

In the direction ($\Rightarrow$) we assume $\pi :\ \vdash \exists x\exists z\phi[x,f(x),z,g(x,z)]$. We remove from the tableau π all conclusions $\exists z\phi[\tau,f(\tau),z,g(\tau,z)]*$ of $(\exists*)$-instantiation rules applied to $\exists x\exists z\phi[x,f(x),z,g(x,z)]*$ and then remove also all conclusions $\phi[\tau,f(\tau),\rho,g(\tau,\rho)]*$ of $(\exists*)$-instantiation rules applied to $\exists z\phi[\tau,f(\tau),z,g(\tau,z)]*$ whereby we obtain a tableau π_1 such that

$$\pi_1 :\ \vdash [\phi[\tau_1, f(\tau_1), \rho_1, g(\tau_1,\rho_1)]*, \ldots, \phi[\tau_n, f(\tau_n), \rho_n, g(\tau_n,\rho_n)]*]$$

where the shown n formulas are all conclusions removed in the second step. Form this we clearly get

$$\pi_2 :\ \vdash \phi[\tau_1, f(\tau_1), \rho_1, g(\tau_1,\rho_1)] \lor \ldots \lor \phi[\tau_n, f(\tau_n), \rho_n, g(\tau_n,\rho_n)]$$

for a tableau π_2 obtained from π_1 by n generalized flatten rules $(G\lor_i*)$. Since the tableau π_2 cannot apply any quantifier rules, it must be an identity tableau.

In the direction ($\Leftarrow$) we assume the right-hand-side of (4). We clearly have

$$\vdash \phi[\tau_i, f(\tau_i), \rho_i, g(\tau_i,\rho_i)] \to \exists x\exists z\phi[x,f(x),z,g(x,z)]$$

for all $1 \leq i \leq n$ from which we get the left-hand-side.

The above instance of the Herbrand's theorem now follows by combining the results (3) and (4) and by using soundness and completeness theorems for both quantifier and identity tableaux.

6.5 Extensions by Contextually Defined Functions

We can extend theories by explicit definitions of functions: $f(\vec{x}) = \tau[\vec{x}]$. That this is conservative can be then proved similarly as the conservativity of explicit definitions of predicates. In contrast to such definitions of predicates, explicitly defined functions do not fully use the power of first-order logic because the terms τ do not posses the full expressiveness of formulas. Contextually defined functions utilize the full power of formulas.

As an example consider a formal theory of arithmetic which has multiplication, addition, and constant 1. We have seen in Sect. 6.3 how to extend such a theory by explicit definition of the predicate $<$. We can further define the binary predicate $x \leq y \leftrightarrow x < y \lor x = y$ and the unary square function $x^2 = x{\cdot}x$ by explicit definitions. Let us call such a theory T. We can extend T to S by extending its language with the unary function symbol $[\sqrt{\cdot}]$ intended to denote the whole part of square root. We can find a formula of $\mathcal{L}_T$ equivalent to the formula $[\sqrt{x}] = y$ applying the square root function in the context of an atomic formula. We can use the formula in the contextual definition:

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

This definition can be justified if we can prove in T that to every individual there will exactly one square root. This amounts to proving the following:

$$T \vdash \exists y(y^2 \leq x \land x < (y+1)^2)$$
$$T \vdash y_1^2 \leq x \land x < (y_1+1)^2 \land y_2^2 \leq x \land x < (y_2+1)^2 \rightarrow y_1 = y_2 \ .$$

We can then translate every formula of $\mathcal{L}_S$ into an equivalent formula of $\mathcal{L}_T$ where every application of $[\sqrt{\tau}]$ used in a context of an atomic formula $\phi[z]$ as $\phi[[\sqrt{\tau}]]$ is translated to the formula on the right for which S proves:

$$S \vdash \phi[[\sqrt{\tau}]] \leftrightarrow \exists z(z^2 \leq \tau \land \tau < (z+1)^2 \leftrightarrow \phi[z]) \ .$$

6.5.1 Extensions by contextually defined functions. Let T be a theory, $\phi[\vec{x}, y]$ a formula of $\mathcal{L}_T$ with the free variables among the indicated ones, and f a new n-ary function symbol ($n \geq 0$). If T proves the *existence*:

$$T \vdash \exists y\, \phi[\vec{x}, y] \qquad (1)$$

and the *uniqueness*

$$T \vdash \phi[\vec{x}, y_1] \land \phi[\vec{x}, y_2] \rightarrow y_1 = y_2 \qquad (2)$$

conditions then we can extend T to S by adding to $\mathcal{L}_T$ the function symbol f and to the axioms of T any closure of

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

as the *defining axiom* for f. Because $S = T, \forall(3)$ we trivially have $S \vdash T$ and so S extends T. We call this kind of extension *extension by contextually defined function*. The term 'contextual' refers to the fact that although we are not able to find a term ρ of $\mathcal{L}_T$ identical to $f(\vec{x})$, we have a formula $\phi[\vec{\tau}, \rho]$ of $\mathcal{L}_T$ equivalent to an application of f in the context of an atomic formula $f(\vec{\tau}) = \rho$ for all terms $\vec{\tau}$, ρ of $\mathcal{L}_T$. Note that f cannot be applied in ϕ which is of $\mathcal{L}$ and so contextual definitions are not recursive. We have

$$S \vdash f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y] \tag{4}$$

by Thm. 5.3.8 because S trivially proves its axiom $\forall(3)$.

6.5.2 Translation function. Let S be the extended theory $T, \forall 6.5.1(3)$. For every atomic formula ψ of $\mathcal{L}_S$ we define by induction on the number k of applications of f in ψ a formula $\psi^\star$ of $\mathcal{L}_T$ called a *translation* of ψ. If $k = 0$ then we set $\psi^\star \equiv \psi$. Otherwise we have $\psi \equiv \psi_1[f(\vec{\tau})]$ for some terms $\vec{\tau}$ of $\mathcal{L}_T$ and a formula $\psi_1[z]$ of $\mathcal{L}_S$ with $< k$ applications of f. We then set

$$\psi^* \equiv \exists z(\phi'[\vec{\tau}, z] \wedge \psi_1[z]^\star) \tag{1}$$

where ϕ' is a variant of ϕ with the bound variables different from z and those free in $\vec{\tau}$. Note that $\psi_1[z]^\star$ is a formula of $\mathcal{L}_T$ by IH and so is the formula $\psi^\star$. We extend the translation function to all formulas ψ of $\mathcal{L}_S$ by designating by $\psi^\star$ any formula obtained from ψ by replacing any of its atomic formulas ψ_1 by formulas $\psi_1^\star$.

6.5.3 Translation lemma. *If S is the extended theory $T, \forall 6.5.1(3)$ and ψ a formula of $\mathcal{L}_S$ then we have*

$$S \vdash \psi \leftrightarrow \psi^* \ . \tag{1}$$

Proof. We first prove (1) for atomic ψ by metainduction on the number of applications of f in ψ. If there are none then the property is a tautology because $\psi^\star \equiv \psi$. Otherwise, using the notation of Par. 6.5.2 we have $\psi \equiv \psi_1[f(\vec{\tau})]$ and $\psi^\star \equiv 6.5.2(1)$ for some terms $\vec{\tau}$ of $\mathcal{L}_T$ and a formula $\psi_1[z]$ of $\mathcal{L}_S$ with $< k$ applications of f. Working in S we obtain $f(\vec{\tau}) = z \leftrightarrow \phi'[\vec{\tau}, z]$ by the Variant theorem (6.1.6) from an instantiation of the defining axiom for f. Hence $\exists z(f(\vec{\tau}) = z \wedge \psi_1[z]^\star) \leftrightarrow \psi^\star$ by the Equivalence theorem (6.1.5). We have $\psi[z] \leftrightarrow \psi_1[z]^\star$ by IH and so $\exists z(f(\vec{\tau}) = z \wedge \psi_1[z]) \leftrightarrow \psi^\star$. Thus $\psi_1[f(\vec{\tau})] \leftrightarrow \psi^\star$ by Identity theorem (6.1.4). □

6.5.4 Theorem. *The extension $S = T, \forall(f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y])$ of T by the contextually defined function f is conservative and for any formula ψ of $\mathcal{L}_T$ we have*

$$S \vdash \psi \Leftrightarrow T \vdash \psi^\star \ . \tag{1}$$

Proof. Designate by T_2 the Skolem extension of T with a closure of

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

T_2 is conservative over T by Thm. 6.4.5 and $\mathcal{L}_S = \mathcal{L}_{T_2}$. If we succeed in proving

$$S \equiv T_2 \tag{3}$$

then for any ψ of $\mathcal{L}_T$ such that $S \vdash \psi$ we will have $T_2 \vdash \psi$ by (3) and $T \vdash \psi$ because T_2 is conservative over T.

(3): In order to prove that S extends T_2 it suffices to derive (2) in S. So working in S we instantiate its defining axiom for f with $y := f(\vec{x})$ and obtain $\phi[\vec{x}, f(\vec{x})]$. from which (2) trivially follows.

That T_2 extends S follows from a proof in T_2 of $f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$. In the direction ($\rightarrow$) it suffices to prove $\phi[\vec{x}, f(\vec{x})]$. Since T_2 proves the existence condition 6.5.1(1) because it extends T and also the instance (2) of its Skolem axiom, we obtain $\phi[\vec{x}, f(\vec{x})]$. In the direction ($\leftarrow$) and working in T_2 we assume $\phi[\vec{x}, y]$ from which we get $\exists y \phi[\vec{x}, y]$ and then $\phi[\vec{x}, f(\vec{x})]$ from (2). We then get $f(\vec{x}) = y$ from the uniqueness condition 6.5.1(2) which holds in T_2.

(1): In the direction ($\rightarrow$) assume $S \vdash \psi$. We have $S \vdash \psi^\star$ by Lemma 6.5.3 and, since $\phi^\star$ is of $\mathcal{L}_T$, we obtain $T \vdash \psi^\star$ because S is conservative over T. In the direction ($\leftarrow$) assume $T \vdash \psi^\star$. Since S extends T, we have $S \vdash \psi^\star$ and so $S \vdash \psi$ by Lemma 6.5.3. □

6.6 Extensions by Definitions

Extensions of theories T to S by explicit definitions of predicates and by contextual definition of functions have the important property that any model of T is uniquely expanded to a model of S. This, and the property that the theorems of S can be translated to the theorems of T and back, are crucial to our treatment of formal arithmetic in the Part II of this text.

6.6.1 Extensions by definitions. Let T be a theory and T_1 its conservative extension either by explicit definition of a predicate P or by contextual definition of a function f. Let S be a theory in the same language as $\mathcal{L}_{T_1}$. We say that S is an *extension by definition* of T if S and T_1 are equivalent. This can be visualized as

$$S \vdash \psi_1 \Leftrightarrow T, \psi \vdash \psi_1 \qquad \text{for any formula } \psi_1 \text{ of } \mathcal{L}_S \tag{1}$$

where ψ is the defining axiom of T_1, i.e a universal closure of either $P(\vec{x}) \leftrightarrow \phi[\vec{x}]$ or $f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$ for a suitable formula ϕ of $\mathcal{L}_T$.

A theory S is an *extension by definitions* of a theory T if S is obtained by a finite number of extensions by definition of T, i.e. if there is a number n and theories T_0, T_1, $\ldots$, T_n such that $T = T_0$, T_{i+1} is an extension by definition of T_i for every $i < n$, and $S = T_n$.

6.6.2 Theorem (Extensions by definitions). *If the theory S is an extension by definitions of a theory T then S is conservative over T, every model of T has a unique expansion to the model of S, and there is an effective translation function taking formulas ψ of $\mathcal{L}_S$ to formulas $\psi^\star$ of $\mathcal{L}_T$ such that for every formula ψ of $\mathcal{L}_S$ we have*

$$S \vdash \psi \leftrightarrow \psi^\star \tag{1}$$

$$S \vdash \psi \Leftrightarrow T \vdash \psi^\star \ . \tag{2}$$

Proof. By induction on the number k of extensions by definition of T to obtain S. If $k = 0$ then $S = T$ and there is nothing to prove. If $k > 0$ then S is an extension by definition of a theory T_1 which is obtained by $k-1$ extensions by definition from T. By IH there is an effective translation function taking formulas ψ of $\mathcal{L}_{T_1}$ to formulas $\psi^{\star_1}$ of $\mathcal{L}_T$ such that

$$T_1 \vdash \psi \leftrightarrow \psi^{\star_1} \tag{3}$$

$$T_1 \vdash \psi \Leftrightarrow T \vdash \psi^{\star_1} \ . \tag{4}$$

Since S is an extension by definition of T_1, there is a defining axiom ψ_1 of $\mathcal{L}_S$ such that $S \equiv T_1, \forall\psi_1$. Depending on whether the new symbol of S is a predicate symbol P or a function symbol f, there is by Translation Lemma 6.3.3 or 6.5.3 an effective translation function, which takes formulas ψ of $\mathcal{L}_S$ to formulas $\psi^{\star_2}$ of $\mathcal{L}_{T_1}$, and such that

$$T_1, \forall\psi_1 \vdash \psi \leftrightarrow \psi^{\star_2} \ . \tag{5}$$

We also have

$$T_1, \forall\psi_1 \vdash \psi \Leftrightarrow T_1 \vdash \psi^{\star_2} \tag{6}$$

by Theorem 6.3.4 or 6.5.4.

We define the translation function taking formulas ψ of $\mathcal{L}_S$ to formulas $\psi^\star$ of $\mathcal{L}_T$ as $\psi^\star \equiv (\psi^{\star_2})^{\star_1}$. The translation function is clearly effective.

(1): We work in the theory $T_1, \forall\psi_1$ and take any formula ψ of $\mathcal{L}_S$. We have ψ iff $\psi^{\star_2}$ by (5) iff, since $T_1, \forall\psi_1$ is an extension of T_1, $(\psi^{\star_2})^{\star_1}$, i.e. $\psi^\star$, by (3). We have just proved $T_1, \forall\psi_1 \vdash \psi \leftrightarrow \psi^\star$. Thus also (1) because S is equivalent to $T_1, \forall\psi_1$.

(2): Take any formula ψ of $\mathcal{L}_S$. We have $S \vdash \psi$ iff, by the equivalence, $T_1, \forall\psi_1 \vdash \psi$ iff, by (6), $T_1 \vdash \psi^{\star_2}$ iff, by (4), $T \vdash (\psi^{\star_2})^{\star_1}$ iff $T \vdash \psi^\star$.

We now prove that S is conservative over T. So we take any formula ψ of $\mathcal{L}_T$ and assume $S \vdash \psi$. We have $T \vdash \psi^\star$ by (2) and, since $\psi^\star \equiv \psi$, also $T \vdash \psi$.

Now suppose that a structure $\mathcal{M}$ for $\mathcal{L}_T$ with a domain D is a model of T: $\mathcal{M} \models T$. There is a unique expansion $\mathcal{M}_1$ of $\mathcal{M}$ which is a model of

T_1 by IH. We expand $\mathcal{M}_1$ to the structure $\mathcal{N}$ for $\mathcal{L}_S$ by choosing a suitable interpretation of the new symbol P or f. For any such interpretation we will have $\mathcal{N} \vDash S$ iff, by the completeness and by $S \equiv T_1, \forall\psi_1$, $\mathcal{N} \vDash T_1, \forall\psi_1$ iff, since $\mathcal{N} \vDash T_1$ by Thm. 5.2.7, $\mathcal{N} \vDash \forall\psi$.

We now consider two cases. If $\psi \equiv P(\vec{x}) \leftrightarrow \phi[\vec{x}]$ then $\mathcal{N} \vDash \forall\psi$ iff $\mathcal{N} \vDash \forall\vec{x}(P(\vec{x}) \leftrightarrow \phi[\vec{x}])$ iff for all $\vec{d} \in D$

$$P^{\mathcal{N}}(\vec{d}) \Leftrightarrow \mathcal{N} \vDash P(\vec{x})[\vec{d}] \Leftrightarrow \mathcal{N} \vDash \phi[\vec{d}] \overset{5.2.7}{\Leftrightarrow} \mathcal{M}_1 \vDash \phi[\vec{d}]$$

which means that the unique interpretation $P^{\mathcal{N}}(\vec{d}) \Leftrightarrow \mathcal{M}_1 \vDash \phi[\vec{d}]$ is both sufficient and necessary for $\mathcal{N}$ to be a model of S.

If $\psi \equiv f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$ then for all $\vec{d} \in D$ there is a $k \in d$ such that $\mathcal{M}_1 \vDash \phi[\vec{d}, k]$ by the existence condition and this k is unique because for any k_1 we have

$$\mathcal{M}_1 \vDash \phi[\vec{d}, k] \text{ and } \mathcal{M}_1 \vDash \phi[\vec{d}, k_1] \Rightarrow k = k_1$$

by the uniqueness condition. We have $\mathcal{N} \vDash \forall\psi$ iff $\mathcal{N} \vDash \forall\vec{x}\forall y(f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y])$ iff for all $\vec{d}, k \in D$

$$f^{\mathcal{N}}(\vec{d}) = k \Leftrightarrow \mathcal{N} \vDash (f(\vec{x}) = y)[\vec{d}, k] \Leftrightarrow \mathcal{N} \vDash \phi[\vec{d}, k] \overset{5.2.7}{\Leftrightarrow} \mathcal{M}_1 \vDash \phi[\vec{d}, k] \ .$$

But this means that interpreting $f^{\mathcal{N}}$ for $\vec{d} \in D$ as the unique $k \in D$ such that $\mathcal{M}_1 \vDash \phi[\vec{d}, k]$ is both sufficient and necessary for $\mathcal{N}$ to be a model of S. □

6.6.3 Theorem (Implicit definition of functions). *For any theory T and any formula $\phi[\vec{x}, y]$ of $\mathcal{L}_T$ with the free variables among the $n+1$ indicated ones and satisfying the existence and uniqueness conditions the extension of $\mathcal{L}_T$ with a new n-ary function symbol f and of T with a new axiom:*

$$\forall\vec{x}\phi[\vec{x}, f(\vec{x})]$$

is an extension by definition.

Proof. Designate by S the extended theory. It is sufficient to prove that S is equivalent to the extension T_1 of T by the contextual definition $\forall\vec{x}\forall y(f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y])$ because T_1 is an extension by definition of T. By instantiating the defining axiom of T_1 with $y := f(\vec{x})$ and generalizing we obtain $T_1 \vdash \forall\vec{x}\phi[\vec{x}, f(\vec{x})]$ and so T_1 extends S.

In order to demonstrate that S extends T_1 it suffices to show $S \vdash f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$. Working in S we assume in the direction $(\rightarrow)$ $f(\vec{x}) = y$. We then obtain $\phi[\vec{x}, y]$ because $\phi[\vec{x}, f(\vec{x})]$ follows from the new axiom of S. In the direction $(\leftarrow)$ we assume $\phi[\vec{x}, y]$ and, since $\phi[\vec{x}, f(\vec{x})]$, we obtain $f(\vec{x}) = y$ by the uniqueness condition which is provable in S because S extends T. □

6.6.4 Theorem (Explicit definition of functions). *For any theory T and any term $\tau[\vec{x}]$ of $\mathcal{L}_T$ with the free variables among the n indicated ones the extension of $\mathcal{L}_T$ with a new n-ary function symbol f and of T with a new axiom:*

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

is an extension by definition.

Proof. Designate by S the extended theory. We first show that T proves the existence and uniqueness conditions for the formula $\phi[\vec{x}, y] \equiv f(\vec{x}) = \tau[\vec{x}]$ The existence condition $\exists y\, y = \tau[\vec{x}]$ follows from $\tau[\vec{x}] = \tau[\vec{x}]$. For the uniqueness condition assume $y_1 = \tau[\vec{x}]$ and $y_2 = \tau[\vec{x}]$ and obtain $y_1 = y_2$ by the properties of identity. We now use Thm. 6.6.3 to extend by definition T to S with the new axiom $\forall \vec{x}\, f(\vec{x}) = \tau[\vec{x}]$. □

7. Peano Arithmetic

7.1 Basic Theorems in PA

In this section we introduce and prove basic theorems of the formal system of arithmetic called Peano arithmetic.

7.1.1 Language of Peano arithmetic. The language $\mathcal{L}_{\mathrm{PA}}$ of Peano arithmetic consists of the constant 0, the unary function symbol x', and of two binary function symbols $x+y$ and $x{\cdot}y$. Both $+$ and $\cdot$ associate to the left and $\cdot$ has greater precedence than $+$.

We will abbreviate $0'$ as 1 but only in this section.

7.1.2 Axioms of Peano arithmetic. The axioms of Peano arithmetic consist of universal closures of the following six formulas:

$$\vdash_{\mathrm{PA}} 0 \neq x' \tag{1}$$
$$\vdash_{\mathrm{PA}} x' = y' \rightarrow x = y \tag{2}$$
$$\vdash_{\mathrm{PA}} 0 + y = y \tag{3}$$
$$\vdash_{\mathrm{PA}} x' + y = (x+y)' \tag{4}$$
$$\vdash_{\mathrm{PA}} 0{\cdot}y = y \tag{5}$$
$$\vdash_{\mathrm{PA}} x'{\cdot}y = x{\cdot}y + y \tag{6}$$

and for every formula $\phi[x]$ of $\mathcal{L}_{\mathrm{PA}}$ and an indicated variable x a universal closure of the *induction axiom* $I_x\phi[x]$:

$$\vdash_{\mathrm{PA}} \phi[0] \land \forall x(\phi[x] \rightarrow \phi[x']) \rightarrow \phi[x] \ . \tag{7}$$

The *induction formula* $\phi[x]$ can contain, in addition to the *induction* variable x, zero or more free variables as *parameters.*

We use the symbol $\vdash_{\mathrm{PA}} \phi$ of *provability in* PA as an abbreviation for PA $\vdash \phi$.

7.1.3 The standard model $\mathcal{N}$ of PA. The *standard model* $\mathcal{N}$ of Peano arithmetic is the structure for $\mathcal{L}_{\mathrm{PA}}$ whose domain is the set of natural numbers $\mathbb{N}$ and the interpretations $0^{\mathcal{N}}$, $'^{\mathcal{N}}$, $+^{\mathcal{N}}$, and $\cdot^{\mathcal{N}}$ of the function symbols of

$\mathcal{L}_{\mathrm{PA}}$ are in that order the number 0, the *successor* function $S(x) = x+1$, the addition, and multiplication functions. We leave to the reader the obvious demonstration that $\mathcal{N}$ satisfies the six axioms 7.1.2(1) through 7.1.2(6).

We now prove that also the induction axioms 7.1.2(7) are satisfied in $\mathcal{N}$. So take any formula $\phi[x, \vec{y}]$ of $\mathcal{L}_{\mathrm{PA}}$ with all its free variables among the indicated ones. We wish to show $\mathcal{N} \vDash \forall x \forall \vec{y} I_x \phi[x, \vec{y}]$. For that we take any $\vec{d} \equiv d_1, \ldots, d_n \in \mathbb{N}$ and it clearly suffices to show $\mathcal{N} \vDash \forall x \phi[x, \vec{d}]$. So assume on the contrary that $\mathcal{N} \nvDash [m, \vec{d}]$ for some $m \in \mathbb{N}$. Furthermore, assume that m is the least such number. We thus have the base case assumption $\mathcal{N} \vDash \phi[0, \vec{d}]$, the inductive assumption: $\mathcal{N} \vDash \forall x(\phi[x, \vec{d}] \to \phi[x, \vec{d}])$, and $\mathcal{N} \nvDash \phi[m, \vec{d}]$. Consider now two cases. If $m = 0$ then we get a contradiction with the base case assumption. If $m > 0$ then we have $\mathcal{N} \vDash \phi[m-1, \vec{d}]$ by the minimality of m and we get a contradiction $\mathcal{N} \vDash \phi[(m-1)+1, \vec{d}]$ from the inductive assumption.

7.1.4 Informal reasoning by induction. Induction axioms of PA can be used anywhere in the proofs in PA. The typical situation is that we use an axiom $I_x\phi[x]$ in an instantiation $x := \tau$:

$$\phi[0] \land \forall x(\phi[x] \to \phi[x']) \to \phi[\tau]$$

under certain assumptions $\psi_1, \ldots, \psi_k$. The informal use of this axiom is to derive the formula $\phi[\tau]$ by considering two cases.

In the *base* case we prove $\phi[0]$ under the above assumptions.

In the *inductive* case we prove $\phi[x']$ for a new eigen-variable x under the same assumptions $\psi_1, \ldots, \psi_k$ to which we add *inductive hypothesis* $\phi[x]$, shortly IH, as an additional assumption.

Both cases taken together then prove $\phi[\tau]$ from $I_x\phi[\tau]$ by modus ponens.

7.1.5 Case analysis on 0 and positive numbers. The base case analysis is on 0 and on positive numbers:

$$\vdash_{\mathrm{PA}} x = 0 \lor \exists y\, x = y' \tag{1}$$

which is proved by induction on x. In the base case there is nothing to prove. In the inductive case we get $\exists y\, x' = y'$ from $x' = x'$.

7.1.6 Successor versus addition. We have

$$\vdash_{\mathrm{PA}} 1 + x = x' \tag{1}$$

because

$$1 + x = 0' + x \overset{7.1.2(4)}{=} (0+x)' \overset{7.1.2(3)}{=} x' \;.$$

7.1.7 Nullpoints of addition. We have the following property of addition:

$$\vdash_{\mathrm{PA}} x + y = 0 \leftrightarrow x = 0 \land y = 0 \tag{1}$$

In the direction ($\rightarrow$) assume $x + y = 0$ and consider two cases by 7.1.5(1). If $x = 0$ then we have $0 = 0 + y \overset{7.1.2(3)}{=} 0$. The case $x = x_1'$ for some x_1 cannot hold because it leads to a contradiction: $0 = x_1' + y \overset{7.1.2(4)}{=} (x+y)'$ by 7.1.2(1). In the direction ($\leftarrow$) the property is a direct consequence of 7.1.2(3).

7.1.8 Addition is commutative. In order to prove that $+$ commutes

$$\vdash_{\mathrm{PA}} x + y = y + x \tag{1}$$

we need two lemmas

$$\vdash_{\mathrm{PA}} x + 0 = x \tag{2}$$

$$\vdash_{\mathrm{PA}} x + y' = x' + y. \tag{3}$$

(2) is proved by induction on x. In the base case we have $0 + 0 \overset{7.1.2(3)}{=} 0$ and in the inductive case:

$$x' + 0 \overset{7.1.2(4)}{=} (x + 0)' \overset{IH}{=} x' \ .$$

(3) is proved by induction on x. In the base case we have

$$0 + y' \overset{7.1.2(3)}{=} y' \overset{7.1.2(3)}{=} (0 + y)' \overset{7.1.2(4)}{=} 0' + y \ .$$

In the inductive case we have

$$x' + y' \overset{7.1.2(4)}{=} (x + y')' \overset{IH}{=} (x' + y)' \overset{7.1.2(4)}{=} x'' + y \ .$$

We now prove (1) by induction on x. In the base case we have

$$0 + y \overset{7.1.2(3)}{=} y \overset{(2)}{=} y + 0 \ .$$

In the inductive case we have

$$x' + y \overset{7.1.2(4)}{=} (x + y)' \overset{IH}{=} (y + x)' \overset{7.1.2(4)}{=} y' + x \overset{(3)}{=} y + x' \ .$$

From now on we will not explicitly indicate the uses of the two axioms of addition 7.1.2(3)(4).

7.1.9 Addition is associative. That the addition is associative

$$\vdash_{\mathrm{PA}} (x + y) + z = x + (y + z) \tag{1}$$

is proved by induction on x. In the base case we have:

$$(0 + y) + z = y + z = 0 + (y + z) \ .$$

In the inductive case we have:

$$(x' + y) + z = (x + y)' + z = ((x + y) + z)' \overset{IH}{=} (x + (y + z))' = x' + (y + z) \ .$$

7.1.10 Cancellation rules for addition. *Cancellation* rules for the addition are:

$$\vdash_{\mathrm{PA}}\ z+x=z+y\to x=y \tag{1}$$
$$\vdash_{\mathrm{PA}}\ x+z=y+z\to x=y\ . \tag{2}$$

(1) is proved by induction on z. In the base case we have

$$0+x=0+y\Rightarrow x=y\ .$$

In the inductive case we have

$$z'+x=z'+y\Rightarrow(z+x)'=(z+y)'\overset{7.1.2(2)}{\Rightarrow}z+x=z+y\overset{IH}{\Rightarrow}x=y\ .$$

(2) is proved as follows:

$$x+z=y+z\overset{7.1.8(1)}{\Rightarrow}z+x=z+y\overset{(1)}{\Rightarrow}x=y\ .$$

From now on we will not explicitly indicate the properties of addition proved until now.

7.1.11 Multiplication by 0 and 1. We have

$$\vdash_{\mathrm{PA}}\ x\cdot 0=0 \tag{1}$$
$$\vdash_{\mathrm{PA}}\ x\cdot 1=1\ . \tag{2}$$

(1) is proved by induction on x. In the base case we have $0\cdot 0\overset{7.1.2(5)}{=}0$. In the inductive case we have:

$$x'\cdot 0\overset{7.1.2(6)}{=}x\cdot 0+0=x\cdot 0\overset{IH}{=}0\ .$$

(2) is proved by induction on x. In the base case we have $0\cdot 1\overset{7.1.2(5)}{=}0$. In the inductive case we have

$$x'\cdot 1\overset{7.1.2(6)}{=}x\cdot 1+1=1+x\cdot 1\overset{IH}{=}1+x=x+1=x'\ .$$

7.1.12 Units of multiplication. Multiplication has the following property

$$\vdash_{\mathrm{PA}}\ x\cdot y=1\leftrightarrow x=1\land y=1\ . \tag{1}$$

Indeed, in the direction $(\to)$ we assume $x\cdot y=1$ and consider three cases for x. The case $x=0$ leads to the contradiction $0'=1=0\cdot y\overset{7.1.2(5)}{=}0$. If $x=1$ then we have

$$1=1\cdot y=0'\cdot y\overset{7.1.2(6)}{=}0\cdot y+y\overset{7.1.2(5)}{=}0+y=y\ .$$

The case $x = x_1''$ for some x_1 cannot happen, This is shown by considering two cases for y. The case $y = 0$ leads to a contradiction

$$0' = 1 = x_1''{\cdot}0 \overset{7.1.11(1)}{=} 0 \ .$$

Also the second case $y = y_1'$ for some y_1 leads to a contradiction:

$$0' = 1 = x_1''{\cdot}y_1' \overset{7.1.2(6)}{=} x_1'{\cdot}y_1' + y_1' \overset{7.1.2(6)}{=} x_1{\cdot}y_1' + y_1' + y_1' = (x_1{\cdot}y_1' + y_1 + y_1)'' \ .$$

The direction ($\leftarrow$) follows from the following

$$1{\cdot}1 = 0'{\cdot}1 \overset{7.1.2(6)}{=} 0{\cdot}1 + 1 \overset{7.1.2(5)}{=} 0 + 1 = 1 \ .$$

7.1.13 Multiplication distributes over addition. The *distributive* property of the multiplication:

$$\vdash_{\mathrm{PA}} z{\cdot}(x + y) = z{\cdot}x + z{\cdot}y \qquad (1)$$

is proved by induction on z. In the base case we have

$$0{\cdot}(x + y) \overset{7.1.2(5)}{=} 0 = 0 + 0 \overset{7.1.2(5)}{=} 0{\cdot}x + 0{\cdot}y \ .$$

In the inductive case we have

$$z'{\cdot}(x + y) \overset{7.1.2(6)}{=} z{\cdot}(x + y) + (x + y) \overset{IH}{=} (z{\cdot}x + z{\cdot}y) + (x + y) =$$

$$z{\cdot}x + (z{\cdot}y + (x + y)) = z{\cdot}x + (z{\cdot}y + (y + x)) = z{\cdot}x + ((z{\cdot}y + y) + x) \overset{7.1.2(6)}{=}$$

$$z{\cdot}x + (z'{\cdot}y + x) = z{\cdot}x + (x + z'{\cdot}y) = (z{\cdot}x + x) + z'{\cdot}y \overset{7.1.2(6)}{=} z'{\cdot}x + z'{\cdot}y \ .$$

From now on we will not explicitly indicate the uses of the two axioms of multiplication 7.1.2(5)(6).

7.1.14 Multiplication is commutative. That the multiplication commutes:

$$\vdash_{\mathrm{PA}} x{\cdot}y = y{\cdot}x \qquad (1)$$

is proved by induction on x. In the base case we have

$$0{\cdot}y = 0 \overset{7.1.11(1)}{=} y{\cdot}0 \ .$$

In the inductive case we have

$$x'{\cdot}y = x{\cdot}y + y \overset{7.1.11(2)}{=} x{\cdot}y + y{\cdot}1 \overset{IH}{=} y{\cdot}x + y{\cdot}1 \overset{7.1.11(1)}{=} y{\cdot}(x + 1) = y{\cdot}x' \ .$$

7.1.15 Multiplication is associative. The proof that the multiplication is associative:

$$\vdash_{\mathrm{PA}} (x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z) \tag{1}$$

is by induction on x. In the base case we have

$$(0{\cdot}y){\cdot}z = 0{\cdot}z = 0 = 0{\cdot}(y{\cdot}z) \ .$$

In the inductive case we have

$$(x'{\cdot}y){\cdot}z = (x{\cdot}y+y){\cdot}z \overset{7.1.14(1)}{=} z{\cdot}(x{\cdot}y+y) \overset{7.1.11(1)}{=} z{\cdot}(x{\cdot}y)+z{\cdot}y \overset{7.1.14(1)}{=}$$
$$(x{\cdot}y){\cdot}z+y{\cdot}z \overset{IH}{=} x{\cdot}(y{\cdot}z)+y{\cdot}z = x'{\cdot}(y{\cdot}z) \ .$$

7.1.16 Cancellation rules for multiplication. *Cancellation* rules for the multiplication are:

$$\vdash_{\mathrm{PA}} z \neq 0 \land z{\cdot}x = z{\cdot}y \to x = y \tag{1}$$
$$\vdash_{\mathrm{PA}} z \neq 0 \land x{\cdot}z = y{\cdot}z \to x = y \ . \tag{2}$$

(1) follows by the commutativity of multiplication from (2) which is proved by assuming $z = z_1'$ for some z_1 and continuing by induction on x with the induction formula $\forall y(x{\cdot}z_1' = y{\cdot}z_1' \to x = y)$. In the base case we take any y, assume $0{\cdot}z_1' = y{\cdot}z_1'$, and consider two cases. If $y = 0$ then we have $x = 0 = y$ trivially. The case $y = y_1'$ for some y_1 leads to a contradiction:

$$0 = 0{\cdot}z_1' = y_1'{\cdot}z_1' = y{\cdot}z_1' + z_1' = (y_1{\cdot}z_1' + z_1)' \ .$$

In the inductive case we take any y, assume $x'{\cdot}z_1' = y{\cdot}z_1'$, and consider two cases again. If $y = 0$ then the assumption is shown contradictory similarly as above. If $y = y_1'$ then we have

$$x{\cdot}z_1' + z_1' = x'{\cdot}z_1' = y_1'{\cdot}z_1' = y_1{\cdot}z_1' + z_1'$$

and so $x{\cdot}z_1' = y_1{\cdot}z_1'$. We obtain $x = y_1$ by IH and so we get $x' = y_1'$.

From now on we will not explicitly refer to the properties of multiplication proved until now.

7.2 Extensions of PA

We study in this section the effect of extensions by definitions of PA on the axioms of induction.

7.2.1 Proper extensions of PA. We are interested in *proper* extensions T of PA which prove all induction axioms of T, i.e. $T \vdash I_x\phi[x]$ for all fromulas ϕ of $\mathcal{L}_T$.

Clearly, the basic theory PA is proper. The following theorem asserts that extensions by definitions yield proper theories from proper ones.

7.2.2 Theorem. *If S is an extension by definitions of a proper extension T of PA then also S is proper.*

Proof. Take the induction axiom $I_x\phi[x]$:

$$\phi[0] \wedge \forall x(\phi[x] \rightarrow \phi[x']) \rightarrow \phi[x]$$

for an arbitrary formula ϕ of $\mathcal{L}_S$. We use the Theorem on Extensions by definitions 6.6.2 and translate away the predicate or function symbols introduced into S. The translation $(I_x\phi[x])^\star$ of $I_x\phi[x]$ is the following formula of $\mathcal{L}_T$:

$$\phi^\star[0] \wedge \forall x(\phi^\star[x] \rightarrow \phi^\star[x']) \rightarrow \phi^\star[x] \ .$$

We thus have $(I_x\phi[x])^\star \equiv I_x\phi^\star[x]$ and, since ϕ^* is a formula of the proper extension T, we have $T \vdash (I_x\phi[x])^\star$. Hence $S \vdash I_x\phi[x]$ by 6.6.2(2). □

7.2.3 Peano arithmetic in wider sense. In order to escape the irritating references to extensions of extensions of PA we will relativize our terminology. We will designate by PA not only the basic theory of Peano arithmetic, i.e. the six axioms for the function symbols of PA and infinitely many induction axioms, but also the current extension of Peano arithmetic. We will also designate by $\mathcal{L}_{\mathrm{PA}}$ the language of the current extension of PA. Thus both the language and the axioms of PA will be relative notions depending on the context where the symbols $\mathcal{L}_{\mathrm{PA}}$ and PA are used. It will be always possible to determine the meaning of both symbols.

Only in situations where we will be introducing new schemas of extension of PA such as minimalization (see Par. 7.4.3), primitive recursion (see Sect. 8.2), or course of values recursion with measure (see Sect. 8.4) we will temporary revert to designating the extensions of PA by symbols T, S, etc.

We will also have to be specific about concrete extensions of PA when we will be introducing new induction schemas such as complete induction (see Thm. 7.3.8), the least number principle (see Par. 7.3.10), or measure induction (see Par. 8.4.5). The new induction schemas will be reduced to the induction axioms $I_x\phi$ of PA. All extensions of PA in this text will be extensions by definitions, which are proper extensions. As a consequence, the new induction principles will be always provable in the extensions.

We will also use the expression *standard model* of PA in the relativized sense to designate the unique expansion of the standard model $\mathcal{N}$ of PA to the model of the current extension of PA. The uniqueness of expansion is guaranteed by Thm. 6.6.2.

We will be using the symbol of provability $\vdash_{\mathrm{PA}} \phi$ in the relativized sense as $T \vdash \phi$ where T is the current extension of PA. We will also use the symbol $\vdash_{\mathrm{PAx}} \phi$ with the meaning of $\vdash_{\mathrm{PA}} \phi$ when we will wish to emphasize that the formula ϕ is the defining axiom of the new extension of PA.

7.3 Introduction of Basic Predicates into PA

We define in PA the four relations of comparison and prove their basic properties.

7.3.1 Comparison predicates. We introduce into PA the binary *comparison* predicates $<$, $\leq$, $>$, and $\geq$ by explicit definitions:

$$\vdash_{\text{PAx}} x < y \leftrightarrow \exists z\, x + z' = y \tag{1}$$
$$\vdash_{\text{PAx}} x \leq y \leftrightarrow x < y \lor x = y \tag{2}$$
$$\vdash_{\text{PAx}} x > y \leftrightarrow y < x \tag{3}$$
$$\vdash_{\text{PAx}} x \geq y \leftrightarrow y \leq x\ . \tag{4}$$

The relation $\leq$ has the following property:

$$\vdash_{\text{PA}} x \leq y \leftrightarrow \exists z\, x + z = y\ . \tag{5}$$

Indeed, in the direction $(\rightarrow)$ assume $x \leq y$ and consider two cases by the definition (2). If $x < y$ then we have $x + z' = y$ from the definition and so $\exists z\, x + z = y$ holds. If $x = y$ then we have $x + 0 = y$ and $\exists z\, x + z = y$ holds again.

In the direction $(\leftarrow)$ assume $x + z = y$ for some z and consider two cases for z. If $z = 0$ we have $x = x + 0 = y$. If $z = z_1'$ for some z_1 then we have $x + z_1' = y$ and so $x < y$ holds from the definition. In either case we have $x \leq y$ from the definition.

7.3.2 The relation $<$ is a linear order. The relation $<$ (and also $>$) is a linear order because we have

$$\vdash_{\text{PA}} \neg x < x \tag{1}$$
$$\vdash_{\text{PA}} x < y \land y < z \rightarrow x < z \tag{2}$$
$$\vdash_{\text{PA}} x < y \lor x = y \lor y < x\ . \tag{3}$$

The properties are called in that order *irreflexivity*, *transitivity*, and *linearity*.

Irreflexivity is proved by induction on x. In the base case we have: $0+z' = (0+z)' \neq 0$ and so $\neg\exists z\, 0+z' = 0$ from which we get $\neg 0 < 0$ from definition. In the inductive case we derive a contradiction by assuming $x' < x'$ as follows. We have $x' + z' = x'$ for some z from the definition and from $(x + z')' = x' + z' = x'$ we obtain $x + z' = x$ from which we get $x < x$ contradicting IH.

Transitivity is proved by assuming $x < y$ and $y < z$ from which we get $x + a' = y$ and $y + b' = z$ for some a and b from the definitions. Hence, $x + (a + b')' = x + a' + b' = y + b' = z$, i.e. $\exists c\, x + c' = z$ and so we have $x < z$.

For the linearity we need an auxiliary property

$$\vdash_{\text{PA}} 0 < x' \tag{4}$$

which follows from $0 + x' = x'$ by existential instantiation and the definition.

Linearity is proved by induction on x. In the base case we wish to prove $0 < y \lor 0 = y \lor y < 0$ for which we consider two cases. If $y = 0$ the property holds trivially. If $y = y_1'$ for some y_1 then we have $0 \overset{(4)}{<} y_1' = y$. In the inductive case we wish to prove $x' < y \lor x' = y \lor y < x'$. From the inductive hypothesis (3) we consider three cases. If $x < y$ then we have $x' + z = x + z' = y$ for some z from the definition and so $x' \leq y$ by 7.3.1(5). From this we get $x' < y$ or $x' = y$ from the definition.

If $x = y$ we have $y + 0' = x + 0' = x' + 0 = x'$ and so $y < x'$ from the definition.

Finally, if $y < x$ we have $y + z' = x$ for some z from the definition and so $y + z'' = (y + z')' = x'$ from which we get $y < x'$ from the definition.

7.3.3 Trichotomy and dichotomy laws. The laws of trichotomy and dichotomy are in that order the following formulas:

$$\vdash_{\text{PA}} x < y \lor x = y \lor x > y \tag{1}$$

$$\vdash_{\text{PA}} x \leq y \lor x > y \; . \tag{2}$$

The laws are typically used for case analysis and they directly follow from the linearity 7.3.2(3) of $<$ by the definitions.

7.3.4 Ordering properties of relation $\leq$. The predicate $\leq$ constitutes a (total) *ordering relation* which is reflexive, transitive, *antisymmetric*, and linear. This is expressed in that order as follows:

$$\vdash_{\text{PA}} x \leq x \tag{1}$$

$$\vdash_{\text{PA}} x \leq y \land y \leq z \to x \leq z \tag{2}$$

$$\vdash_{\text{PA}} x \leq y \land y \leq x \to x = y \tag{3}$$

$$\vdash_{\text{PA}} x \leq y \lor y \leq y \; . \tag{4}$$

Property (1) follows directly from the definition. Property (2) follows from the transitivity of $<$.

(3): If $x \leq y$, $y \leq x$, and $x \neq y$ hold then we obtain $x < y$ and $y < x$ from the definitions. From this we get $x < x$ by transitivity which contradicts the irreflexivity of $<$.

Property (4) is a direct consequence of linearity 7.3.2(3) of $<$.

7.3.5 Additional properties of comparisons. We have the following additional properties of the comparison relations:

$$\vdash_{\text{PA}} x \not< 0 \tag{1}$$

$$\vdash_{\text{PA}} 0 \leq x \tag{2}$$

$$\vdash_{\text{PA}} x < x' \tag{3}$$

$$\vdash_{\text{PA}} x < y \leftrightarrow x' \leq y \; . \tag{4}$$

(1) Assume on the contrary $x < 0$. We then have $x + z' = 0$ for some z from the definition and we get the contradiction $z' = 0$ by 7.1.7(1).

(2) is a direct consequence of (1) and 7.3.3(2).

(3): We have $x + 0' = (x + 0)' = x'$ and so $\exists z\, x + z = x'$ holds. We now get $x < x'$ from the definition.

For (4) we have $x < y$ iff $x + z' = y$ for some z iff $x' + z = y$ for some z iff $x' \leq y$ by 7.3.1(5).

7.3.6 Monotonicity of addition and multiplication. Addition and multiplication are monotone:

$$\vdash_{\text{PA}} x < y \leftrightarrow z + x < z + y \tag{1}$$
$$\vdash_{\text{PA}} x < y \leftrightarrow x + z < y + z \tag{2}$$
$$\vdash_{\text{PA}} z > 0 \to x < y \leftrightarrow z \cdot x < z \cdot y \tag{3}$$
$$\vdash_{\text{PA}} z > 0 \to x < y \leftrightarrow x \cdot z < y \cdot z\ . \tag{4}$$

(1): We have $x < y$ iff $x + u' = y$ for some u iff, by 7.1.10(1) and properties of identity, $z + (x + u') = z + y$ for some u iff $(z + x) + u' = z + y$ for some u iff $z + x < z + y$.

Property (2) follows from (1) by commutativity of addition.

(3): In the direction ($\to$) assume $z = z_1'$ and $x + u' = y$ for some z_1 and u. We have

$$z_1' \cdot y = z_1' \cdot (x + u') = z_1' \cdot x + z_1' \cdot u' = z_1' \cdot x + (z_1 \cdot u' + u') = z_1' \cdot x + (z_1 \cdot u' + u)'$$

and so $z_1' \cdot x < z_1' \cdot y$ holds by definition.

In the direction ($\leftarrow$) assume $z = z_1'$ and prove

$$\forall y (z_1' \cdot x < z_1' \cdot y \to x < y)$$

by induction on x. In the base case take any y and assume $z_1' \cdot 0 < z_1' \cdot y$. We then have $0 < z_1' \cdot y$. If it were the case that $y = 0$ we would get a contradiction $0 < z_1' \cdot 0 = 0$ with 7.3.2(1). Hence $y = y_1'$ for some y_1 and we have $0 < y$ by 7.3.2(4).

In the inductive case take any y and assume $z_1' \cdot x' < z_1' \cdot y$. If it were the case that $y = 0$ we would get a contradiction $z_1' \cdot x' < z_1' \cdot 0 = 0$ with 7.3.5(1). Hence $y = y_1'$ for some y_1 and we have

$$x \cdot z_1' + z_1' = x' \cdot z_1' = z_1' \cdot x' < z_1' \cdot y_1' = y_1' \cdot z_1' = y_1 \cdot z_1' + z_1'\ .$$

We now obtain $x \cdot z_1' < y_1 \cdot z_1'$ by (2) and $x < y_1$ by IH. Hence $x' = x + 1 \overset{(2)}{<} y_1 + 1 = y_1'$.

Property (4) follows from (3) by the commutativity of multiplication.

7.3.7 Complete induction. Let T be a proper extension of PA containing the predicate $<$, $\phi[x]$ a formula of $\mathcal{L}_T$ with the indicated variable x free and with possibly additional parameters, and y a new variable. The formula of *complete induction on x for $\phi[x]$* is the following one:

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

7.3.8 Theorem. *Every proper extension T of PA containing the predicate $<$ proves the schema of complete induction 7.3.7(1).*

Proof. We prove 7.3.7(1) in T from an auxiliary property

$$T \vdash \forall x(\forall y(y < x \to \phi[y]) \to \phi[x]) \to \forall y(y < x \to \phi[y])$$

which is proved by assuming the formula expressing that ϕ *is progressive*:

$$\forall x(\forall y(y < x \to \phi[y]) \to \phi[x]) \tag{1}$$

and proving

$$\forall y(y < x \to \phi[y]) \tag{2}$$

by induction on x. In the base case there is nothing to prove. In the inductive case we take any y s.t. $y < x'$ and consider two cases by dichotomy. If $y < x$ we obtain $\phi[y]$ from IH: (2). If $y \geq x$ then we have $y = x$ and note that IH, i.e. (2), is the antecedent of (1) from which we obtain $\phi[x]$, i.e. $\phi[y]$.

We obtain $T \vdash$ 7.3.7(1) from the auxiliary property by instantiating its consequent with $x := x'$ and $y := x$. □

7.3.9 The least number principle. Let T be a proper extension of PA containing the predicate $<$, $\phi[x]$ a formula of $\mathcal{L}_T$ with the indicated variable x free and with possibly additional parameters, and y a new variable. The formula of the *least number principle for ϕ* is the following one:

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

The least number principle says that if the property $\phi[x]$ holds for some x then it holds for the least such x.

7.3.10 Theorem. *Every proper extension T of PA containing the predicate $<$ proves the schema of the least number principle 7.3.9(1).*

Proof. We prove 7.3.9(1) in T from the complete induction for $\neg\phi$:

$$T \vdash \forall x(\forall y(y < x \to \neg\phi[y]) \to \neg\phi[x]) \to \neg\phi[x]$$

which is a theorem of T by Thm. 7.3.8. Its converse is

$$\phi[x] \to \exists x(\forall y(y < x \to \neg\phi[y]) \land \phi[x])$$

and 7.3.9(1) logically follows by quantifier operations. □

7.4 Introduction of Basic Functions into PA

We will extend PA by some basic functions such as division, introduce extensions by minimalization, and prove that they are extensions by definition.

7.4.1 Small constants. We have used 1 as abbreviation for the term $0'$ in Sect. 7.1. We now introduce the symbols 1, 2, 3, and 4 into PA as constants by explicit definitions:

$$\vdash_{\mathrm{PA}} 1 = 0' \tag{1}$$
$$\vdash_{\mathrm{PA}} 2 = 1' \tag{2}$$
$$\vdash_{\mathrm{PA}} 3 = 2' \tag{3}$$
$$\vdash_{\mathrm{PA}} 4 = 3' \; . \tag{4}$$

7.4.2 Extensions by minimalization. Let T be a proper extension of PA containing the predicate $<$ and $\phi[\vec{x}, y]$ a formula of $\mathcal{L}_T$ with all free variables among the indicated ones where $\vec{x}$ contains $n \geq 0$ variables. If T proves the existence condition:

$$T \vdash \exists y \phi[\vec{x}, y] \tag{1}$$

then the extension of T to S with the n-ary function symbol f and the defining axiom a universal closure of

$$\phi[\vec{x}, f(\vec{x})] \land \forall y (y < f(\vec{x}) \to \neg\phi[\vec{x}, y]) \; . \tag{2}$$

is called *extension by minimalization*.

We will use a more suggestive notation as an abbreviation for the above defining axiom:

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

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 because on accord of the existence condition $\exists y \phi[\vec{x}, y]$ there is at least one such y.

The defining axiom clearly implies $S \vdash \phi[\vec{x}, f(\vec{x})]$ and

$$S \vdash y < f(\vec{x}) \to \neg\phi[\vec{x}, y] \; ,$$

which is equivalent to

$$S \vdash \phi[\vec{x}, y] \to f(\vec{x}) \leq y$$

whenever T contains the predicate $\leq$.

7.4.3 Theorem. *If T is a proper extension of PA containing the predicate $<$ then an extension of T by minimalization is an extension by definition.*

Proof. Let S be the extension of T by minimalization as in Par. 7.4.2 and S_1 an extension of T with f implicitly defined by $\psi[\vec{x}, f(\vec{x})]$ for the formula

$$\psi[\vec{x}, y] \equiv \phi[\vec{x}, y] \land \forall z(z < y \to \neg\phi[\vec{x}, z]) \ .$$

Since $\psi[\vec{x}, f(\vec{x})]$ and 7.4.2(2) are variants, S will be an extension by definition of T by Thm. 6.6.3 provided T proves the existence and uniqueness conditions for ψ. We note that the existence condition $\exists y\psi[\vec{x}, y]$ is the consequent of the instance of the least number principle

$$T \vdash \exists y\phi[\vec{x}, y] \to \exists y(\phi[\vec{x}, y] \land \forall z(z < y \to \neg\phi[\vec{x}, z])) \ ,$$

which is provable in T by Thm. 7.3.10. We thus get $\exists y\psi[\vec{x}, y]$ in T from 7.4.2(1).

For the proof of the uniqueness condition we work in T, assume $\psi[\vec{x}, y_1]$, $\psi[\vec{x}, y_2]$, and consider three cases. If $y_1 < y_2$ then we obtain $\neg\phi[\vec{x}, y_1]$ from $\psi[\vec{x}, y_2]$ which contradicts $\phi[\vec{x}, y_1]$ implied by $\psi[\vec{x}, y_1]$. If $y_1 > y_2$ we derive a contradiction similarly. Thus it must be the case that $y_1 = y_2$. □

7.4.4 Modified subtraction. We wish to extend Peano arithmetic with a binary *modified subtraction* function $x \mathbin{\dot{-}} y$ with the basic properties

$$\vdash_{\text{PA}} \ y \leq x \to x = y + (x \mathbin{\dot{-}} y) \ . \qquad (1)$$

$$\vdash_{\text{PA}} \ y > x \to x \mathbin{\dot{-}} y = 0 \ . \qquad (2)$$

The modified subtraction function is introduced by minimalization

$$\vdash_{\text{PAx}} \ x \mathbin{\dot{-}} y = \mu_z[y \leq x \to x = y + z] \qquad (3)$$

with the existence condition

$$\vdash_{\text{PA}} \ \exists z(y \leq x \to x = y + z) \ ,$$

which is equivalent to $y \leq x \to \exists z \ x = y + z$, holding by 7.3.1(5).

The defining axiom for $\dot{-}$ directly implies Property (1) and

$$\vdash_{\text{PA}} \ z < x \mathbin{\dot{-}} y \to \neg(y \leq x \to x = y + z) \ . \qquad (4)$$

From this we obtain $0 < x \mathbin{\dot{-}} y \to y \leq x$, which is equivalent to (2) by dichotomy and 7.3.5(1).

7.4.5 Maximum. The *maximum* function $\max(x, y)$ with the basic properties

$$\vdash_{\mathrm{PA}} x \leq \max(x,y) \tag{1}$$
$$\vdash_{\mathrm{PA}} y \leq \max(x,y) \tag{2}$$
$$\vdash_{\mathrm{PA}} x = \max(x,y) \lor y = \max(x,y) \tag{3}$$

is introduced into PA by minimalization

$$\vdash_{\mathrm{PAx}} \max(x,y) = \mu_z[x \leq z \land y \leq z] \tag{4}$$

whose existence condition

$$\vdash_{\mathrm{PA}} \exists z(x \leq z \land y \leq z)$$

is proved by case analysis. If $x \leq y$ then we take $z := y$. If $x > y$ then we take $z := x$.

The defining axiom for the maximum function directly implies Properties (1) and (2) as well as

$$\vdash_{\mathrm{PA}} x \leq z \land y \leq z \rightarrow \max(x,y) \leq z \ .$$

Property (3) is proved from the last by case analysis. If $x \leq y$ then for $z := y$ we get $\max(x,y) \leq y$ and we obtain $\max(x,y) = y$ from (2) by antisymmetry. If $x > y$ then for $z := x$ we get $\max(x,y) \leq x$ and we obtain $\max(x,y) \leq y$ similarly.

7.4.6 The square function. We introduce the unary function x^2 yielding the square of x into PA by explicit definition:

$$\vdash_{\mathrm{PAx}} x^2 = x{\cdot}x \ . \tag{1}$$

7.4.7 Whole part of square root. We wish to introduce into PA the function $[\sqrt{x}]$ yielding the whole part of the square root of x which satisfies

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

The function is defined by minimalization:

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

whose existence condition

$$\vdash_{\mathrm{PA}} \exists y\, x < (y+1)^2$$

holds for $y := x$ because

$$x = x{\cdot}1 \leq x{\cdot}(x+1) < (x+1){\cdot}(x+1) = (x+1)^2 \ .$$

The defining axiom for $[\sqrt{x}]$ implies

$$\vdash_{\text{PA}} x < ([\sqrt{x}]+1)^2 \tag{3}$$
$$\vdash_{\text{PA}} y < [\sqrt{x}] \to (y+1)^2 \leq x \ . \tag{4}$$

From (3) we can see that for the proof of (1) it suffices to show $[\sqrt{x}]^2 \leq x$. For that we consider two cases. If $[\sqrt{x}] = 0$ then we have $0^2 = 0 \leq x$. If $[\sqrt{x}] = y'$ for some y then, since $y < [\sqrt{x}]$, we obtain $[\sqrt{x}]^2 = (y+1)^2 \leq x$ from (4).

7.4.8 Integer division and remainder. We wish to introduce into PA *integer division* $x \div y$ and *remainder* $x \bmod y$ functions satisfying the following

$$\vdash_{\text{PA}} y > 0 \to x = x \div y \cdot y + x \bmod y \land x \bmod y < y \tag{1}$$
$$\vdash_{\text{PA}} y > 0 \land x = q \cdot y + r \land r < y \to q = x \div y \land r = x \bmod y \tag{2}$$
$$\vdash_{\text{PA}} x \div 0 = 0 \tag{3}$$
$$\vdash_{\text{PA}} x \bmod 0 = 0 \ . \tag{4}$$

We claim that the functions can be introduced by minimalization:

$$\vdash_{\text{PAx}} x \div y = \mu_q[y > 0 \to x < (q+1) \cdot y] \tag{5}$$
$$\vdash_{\text{PAx}} x \bmod y = \mu_z[y > 0 \to z = x \dot{-} x \div y \cdot y] \tag{6}$$

The existence condition for the division function (5) is

$$\vdash_{\text{PA}} \exists q (0 < y \to x < (q+1) \cdot y) \ .$$

This holds because if $y = 0$ then set $q := 0$ and if $y > 0$ then set $q := x$ because we have

$$x = x \cdot 1 \leq x \cdot y < (x+1) \cdot y \ .$$

The defining axiom for $\div$ implies:

$$\vdash_{\text{PA}} y > 0 \to x < (x \div y + 1) \cdot y \tag{7}$$
$$\vdash_{\text{PA}} q < x \div y \to (q+1) \cdot y \leq x \tag{8}$$
$$\vdash_{\text{PA}} y = 0 \to x \div y \leq q \tag{9}$$

The existence condition for the remainder function (6) is equivalent to $\vdash_{\text{PA}} y > 0 \to \exists z\, z = x \dot{-} x \div y \cdot y$ and it is proved trivially. The defining axiom for mod implies

$$\vdash_{\text{PA}} y > 0 \to x \bmod y = x \dot{-} x \div y \cdot y \tag{10}$$
$$\vdash_{\text{PA}} z = x \dot{-} x \div y \cdot y \to x \bmod y \leq z \tag{11}$$
$$\vdash_{\text{PA}} y = 0 \to x \bmod y \leq z \ . \tag{12}$$

For the proof of Property (1) we assume $y > 0$ and prove an auxiliary property

$$\vdash_{\mathrm{PA}} x \div y \cdot y \leq x \tag{13}$$

by considering two cases. If $x \div y = 0$ then we have

$$x \div y \cdot y = 0 \cdot y = 0 \leq x\ .$$

If $x \div y = q'$ for some q then, since $q < x \div y$, we obtain

$$x \div y \cdot y = (q+1) \cdot y \overset{(8)}{\leq} x\ .$$

We then have

$$x \div y \cdot y + x \bmod y \overset{(10)}{=} x \div y \cdot y + (x \dot{-} x \div y \cdot y) \overset{(13),7.4.4(1)}{=} x \overset{(7)}{<} (x \div y + 1) \cdot y = x \div y \cdot y + y\ .$$

Note that $x \bmod y < y$ holds by 7.3.6(1).

Property (2) is proved by assuming $y > 0$, $x = q \cdot y + r$ with $r < y$, and considering three cases by trichotomy. If $x \div y < q$ then we have $q = x \div y + z'$ for some z. We then obtain

$$x \div y \cdot y + x \bmod y \overset{(1)}{=} x = q \cdot y + r = x \div y \cdot y + z' \cdot y + r$$

and hence $x \bmod y = z' \cdot y + r$ by 7.3.6(1). Thus $x \bmod y = y + z \cdot y + r$ from which we have $x \bmod y \geq y$ which contradicts $x \bmod y < y$.

If $q < x \div y$ then we have $x \div y = q + z'$ for some z. We then similarly obtain

$$q \cdot y + r = x \overset{(1)}{=} x \div y \cdot y + x \bmod y = q \cdot y + z' \cdot y + x \bmod y$$

from which we get $r = z' \cdot y + x \bmod y = y + z \cdot y + x \bmod y$ and thus $r \geq y$ which contradicts $r < y$.

Thus the third case $x \div y = q$ must obtain and, since then

$$q \cdot y + r = x \overset{(1)}{=} x \div y \cdot y + x \bmod y = q \cdot y + x \bmod y\ ,$$

we get $r = x \bmod y$ by 7.3.6(1).

(3): Take $y := 0$ and $q := 0$ in (9) and obtain $x \div 0 \leq 0$, i.e. $x \div 0 = 0$.

(4): Take $y := 0$ and $z := 0$ in (12) and obtain $x \bmod 0 \leq 0$, i.e. $x \bmod 0 = 0$.

7.5 The Lattice of Divisibility

7.5.1 Divisibility predicate. The binary *divisibility* predicate $x \mid y$, read as x *divides* y is defined in PA by an explicit definition:

$$\vdash_{\mathrm{PA}\times} x \mid y \leftrightarrow \exists z\, y = z{\cdot}x \ . \tag{1}$$

The predicate of divisibility is a relation of *partial order* (similar to $\leq$ but without the linearity 7.3.4(4)) which satisfies the reflexivity, transitivity, and antisymmetry. The partial order is with the least element 1 and the greatest element 0:

$$\vdash_{\mathrm{PA}} x \mid x \tag{2}$$
$$\vdash_{\mathrm{PA}} x \mid y \land y \mid z \to x \mid z \tag{3}$$
$$\vdash_{\mathrm{PA}} x \mid y \land y \mid x \to x = y \tag{4}$$
$$\vdash_{\mathrm{PA}} 1 \mid x \tag{5}$$
$$\vdash_{\mathrm{PA}} x \mid 0 \ . \tag{6}$$

(2): We have $x = 0 + x = 0{\cdot}x + x = 0'{\cdot}x$ and so $\exists z\, x = z{\cdot}x$.

(3): Assume $x \mid y$ and $y \mid z$, i.e. $y = a{\cdot}x$ and $z = b{\cdot}y$ for some a and b. Then $z = b{\cdot}y = b{\cdot}a{\cdot}x$ and so for $u := b{\cdot}a$ we have $\exists u\, z = u{\cdot}x$.

(4): Assume $x \mid y$ and $y \mid x$, i.e. $y = a{\cdot}x$ and $x = b{\cdot}y$ for some a and b. Then $x = b{\cdot}y = b{\cdot}a{\cdot}x$. We consider two cases. If $x = 0$ then also $y = a{\cdot}0 = 0$. If $x > 0$ then, since $x = b{\cdot}a{\cdot}x + 0$ and $x = 1{\cdot}x + 0$, we obtain $b{\cdot}a = x \div x = 1$ by 7.4.8(2). But then $a = 1$ and $b = 1$ by 7.1.12(1) and so $x = y$.

(5): We have $x = x{\cdot}1$ and so $\exists z\, x = z{\cdot}1$.

(6): We have $0 = 0{\cdot}x$ and so $\exists z\, 0 = z{\cdot}x$.

7.5.2 Additional properties of divisibility. The relation between the divisibility predicate and the remainder function is given by the following property:

$$\vdash_{\mathrm{PA}} y \neq 0 \to y \mid x \leftrightarrow x \bmod y = 0 \ . \tag{1}$$

Indeed, assume $y > 0$. In the direction ($\to$) also assume $y \mid x$, i.e. $x = z{\cdot}y = z{\cdot}y + 0$ for some z. We get $x \bmod y = 0$ by 7.4.8(2). In the direction ($\leftarrow$) also assume $x \bmod y = 0$. We then have

$$x \overset{7.4.8(1)}{=} x \div y{\cdot}y + x \bmod y = x \div y{\cdot}y$$

and for $z := x \div y$ we have $\exists z\, x = z{\cdot}y$. We also have

$$\vdash_{\mathrm{PA}} x \neq 0 \land y \mid x \to y \leq x \tag{2}$$

because if $x > 0$ and $x = z{\cdot}y$ for some z then it must be the case that $z > 0$. If it were the case that $x < y$ then we would have $z{\cdot}x < z{\cdot}y = x = 1{\cdot}x$ and hence $z < 1$ by the monotonicity of multiplication.

We will need the following property:

$$\vdash_{\mathrm{PA}} a + b = c \land x \mid a \land x \mid c \to x \mid b \tag{3}$$

which is proved by assuming its antecedent and considering two cases. If $x = 0$ then from $x \mid a$ and $x \mid c$ we get $a = b = 0$ and hence $b = 0$ from which we have $x \mid b$ by 7.5.1(6).

If $x > 0$ then for some a_1 and c_1 we have $a_1 \cdot x = a \le c = c_1 \cdot x$ and so $a_1 \le c_1$, i.e. $a_1 + u = c_1$ for some u. We then get

$$a + u \cdot x = a_1 \cdot x + u \cdot x = (a_1 + u) \cdot x = c_1 \cdot x = c \ .$$

Thus $b = u \cdot x$, i.e. $x \mid b$.

7.5.3 The lattice of divisibility. A set with a partial order where for every two elements x and y exists their least upper bound $x \cup y$ (called the *join* of x and y) and their greatest lower bound $x \cap y$ (called the *meet* of x and y) is a *lattice*. The partial order on natural numbers given by the relation of divisibility $x \mid y$ forms a lattice where the join $x \cup y$ is the *least common multiple of x and y* and the meet $x \cap y$ is the *greatest common divisor of x and y*. We wish to introduce the operations into PA to satisfy:

$$\vdash_{\text{PA}} x \mid x \cup y \land y \mid x \cup y \tag{1}$$
$$\vdash_{\text{PA}} x \mid z \land y \mid z \to x \cup y \mid z \tag{2}$$
$$\vdash_{\text{PA}} x \cap y \mid x \land x \cap y \mid y \tag{3}$$
$$\vdash_{\text{PA}} z \mid x \land z \mid y \to z \mid x \cap y \ . \tag{4}$$

We claim that the least common multiple $x \cup y$ can be introduced by minimalization:

$$\vdash_{\text{PAx}} x \cup y = \mu_z[x = 0 \lor y = 0 \lor z > 0 \land x \mid z \land y \mid z] \tag{5}$$

whose existence condition

$$\vdash_{\text{PA}} \exists z(x = 0 \lor y = 0 \lor z > 0 \land x \mid z \land y \mid z)$$

is proved by taking any x, y and considering two cases. If $x = 0$ or $y = 0$ then it suffices to take $z := 0$. If $x > 0$ and $y > 0$ then also $x \cdot y > 0$ and, since $x \mid x \cdot y$ and $y \mid x \cdot y$, it suffices to take $z := x \cdot y$. The defining axiom for $\cup$ implies

$$\vdash_{\text{PA}} x > 0 \land y > 0 \to x \cup y > 0 \land x \mid x \cup y \land y \mid x \cup y \tag{6}$$
$$\vdash_{\text{PA}} x = 0 \lor y = 0 \to x \cup y \le z \tag{7}$$
$$\vdash_{\text{PA}} z > 0 \land x \mid z \land y \mid z \to x \cup y \le z \ . \tag{8}$$

We prove (1) by taking any x, y and considering two cases. If $x = 0$ or $y = 0$ then we have $x \cup y = 0$ from (7) by taking $z := 0$ and the property holds on account of 7.5.1(6). If $x > 0$ and $y > 0$ then (1) follows from (6).

We prove (2) by taking any x, y and considering two cases. If $x = 0$ or $y = 0$ then if $x \mid z$ and $y \mid z$ we obtain $z = 0$ and thus $x \cup y \mid z$ by 7.5.1(6).

If $x > 0$ and $y > 0$ then we prove (2) by complete induction on z. Assume $x \mid z$, $y \mid z$ and consider two cases. If $z = 0$ then also $x \cup y \mid z$ by 7.5.1(6). If $z > 0$ then we have

$$0 \overset{(6)}{<} x \cup y \overset{(8)}{\leq} z$$

and so $(x \cup y) + u = z$ for some u. Since $x \mid x \cup y$, we obtain $x \mid u$ by 7.5.2(3). We get $y \mid u$ similarly and, since $u < z$, we obtain $x \cup y \mid u$ by IH, i.e $u = a \cdot (x \cup y)$ for some a. From this, since

$$z = (x \cup y) + u = (x \cup y) + a \cdot (x \cup y) = (1 + a) \cdot (x \cup y) \ ,$$

we get $x \cup y \mid z$.

We claim that the greatest common divisor $x \cap y$ can be introduced by minimalization:

$$\vdash_{\text{PAx}} x \cap y = \mu_z[z \mid x \land z \mid y \land \forall u(u \mid x \land u \mid y \to u \mid z)] \tag{9}$$

whose existence condition is implied by

$$\vdash_{\text{PA}} \forall y \exists z(z \mid x \land z \mid y \land \forall u(u \mid x \land u \mid y \to u \mid z))$$

which is proved by complete induction on x. We take any y and consider two cases. If $x = 0$ then it clearly suffices to take $z := y$. If $x > 0$ then, since $x > y \bmod x$, we use IH with $y := x$ to obtain a z s.t. $z \mid y \bmod x$, $z \mid x$, and

$$\forall u(u \mid y \bmod x \land u \mid x \to u \mid z) \ . \tag{10}$$

We have $x = a \cdot z$ and $y \bmod x = b \cdot z$ for some a and b. We claim that

$$z \mid x \land z \mid y \land \forall u(u \mid x \land u \mid y \to u \mid z)$$

holds for the same z. For that we note

$$y \overset{7.4.8(1)}{=} y \div x \cdot x + y \bmod x = y \div x \cdot a \cdot z + b \cdot z = (y \div x \cdot a + b) \cdot z$$

and so $z \mid y$. Now take any u and assume $u \mid x$ and $u \mid y$. Since $y = y \div x \cdot x + y \bmod x$, we see that $u \mid y \bmod x$ by 7.5.2(3). We now get $u \mid z$ from (10).

Properties (3) and (4) are a direct consequence of the defining axiom for $x \cap y$.

7.5.4 Joins and meets versus the partial order. In every lattice we have $x \cup y = x$ iff $y \leq x$ iff $x \cap y = y$. This has the following form in the lattice of divisibility:

$$\vdash_{\text{PA}} x \cup y = x \leftrightarrow y \mid x \tag{1}$$

$$\vdash_{\text{PA}} x \cap y = y \leftrightarrow y \mid x \tag{2}$$

and the proof is solely by using the four properties 7.5.3(3) through 7.5.3(4) of $\cup$ and $\cap$ as well as the fact that $\mid$ is a partial order satisfying 7.5.1(2) through 7.5.1(4).

(1): In the direction ($\rightarrow$) assume $x \cup y = x$ and get $y \mid x \cup y = x$ from 7.5.3(1). In the direction ($\leftarrow$) assume $y \mid x$. Since $x \mid x$ by 7.5.1(2) we get $x \cup y \mid x$ by 7.5.3(2). We have $x \mid x \cup y$ by 7.5.3(1) and so $x \cup y = x$ by 7.5.1(4).

Property (2) is proved similarly.

7.5.5 The lattice of divisibility is atomic. An element p of a lattice is an *atom* if the only element strictly less than p is the least element of the lattice. The divisibility lattice has 1 as the least element and so the predicate $Prime(p)$ is explicitly introduced to hold exactly of its atoms:

$$\vdash_{\mathrm{PAx}} \; Prime(p) \leftrightarrow p \neq 1 \wedge \forall x(x \mid p \rightarrow x = 1 \vee x = p) \tag{1}$$

Thus p is an atom of the divisibility lattice iff it is a prime number. For $2 = 1' = 0''$ we have

$$\vdash_{\mathrm{PA}} \; \neg Prime(0) \tag{2}$$
$$\vdash_{\mathrm{PA}} \; Prime(2)\ . \tag{3}$$

(2): We have $2 \mid 0$ by 7.5.1(6) and $1 \neq 2 \neq 0$.

(3): We have $2 \neq 1$ and if $x \mid 2$ then $x \leq 2$ by 7.5.2(2). Since $0 \nmid 2$ we can have at most $1 \mid 2$ or $2 \mid 2$.

A lattice with the least element is *atomic* if to every non-minimal element x which is not an atom there is an atom p less than x. The divisibility lattice is atomic because we have:

$$\vdash_{\mathrm{PA}} \; x \neq 1 \rightarrow \exists p(Prime(p) \wedge p \mid x)\ . \tag{4}$$

This is proved by complete induction on x. We consider three cases for x. If $x = 0$ then it suffices to take $p := 2$ by (3) and 7.5.1(6). If $x = 1$ there is nothing to prove. Finally, if $x > 1$ then we consider two cases again. If $Prime(x)$ then it suffices to take $p := x$ because of 7.5.1(2). If $\neg Prime(x)$ then there is a number $q \mid x$ s.t. $x \neq 1$ and $q \neq x$ from the definition. We have $q \leq x$ by 7.5.2(2) and, since $q < x$, there is a prime p s.t. $p \mid q$ by IH for which we get $p \mid x$ by 7.5.1(3).

7.5.6 The lattice of divisibility is distributive. In every lattice we have $(x \cap y) \cup (x \cap z) \leq x \cap (y \cup z)$. We prove this and its dual for the lattice of divisibility solely from the seven properties given in Par. 7.5.4:

$$\vdash_{\mathrm{PA}} \; y \mid z \rightarrow x \cap y \mid x \cap z \tag{1}$$
$$\vdash_{\mathrm{PA}} \; (x \cap y) \cup (x \cap z) \mid x \cap (y \cup z)\ . \tag{2}$$

(1): Assume $y \mid z$. We have $x \cap y \mid x$ and $x \cap y \mid y$ and so $x \cap y \mid z$ by transitivity. Hence $x \cap y \mid x \cap z$ by 7.5.3(4).

(2): Since $y \mid y \cup z$ and $z \mid y \cup z$, we obtain $x \cap y \mid x \cap (y \cup z)$ and $x \cap z \mid x \cap (y \cup z)$ by (1). The property now follows by 7.5.3(2).

UNFINISHED

$$\vdash_{\mathrm{PA}} x \cap (y \cup z) = (x \cap y) \cup (x \cap z) \tag{3}$$

$$\vdash_{\mathrm{PA}} x \cup (y \cap z) = (x \cup y) \cap (x \cup z)\ . \tag{4}$$

7.5.7 A property of prime divisors. We will need Property (2) which is a simple consequence of distributivity of the lattice of divisibility.

$$\vdash_{\mathrm{PA}} x \cup x = x \tag{1}$$

$$\vdash_{\mathrm{PA}} \mathit{Prime}(p) \land p \mid x \cup y \to p \mid x \lor p \mid y\ . \tag{2}$$

(1): This follows from $x \mid x$ by 7.5.4(1).

(2): Assume that p is a prime s.t. $p \mid x \cup y$. If it were the case that $p \nmid x$ and $p \nmid y$ then we would get $p \cap x \neq p$, $p \cap y \neq p$ by 7.5.4(2). But $p \cap x \mid p$ and $p \cap y \mid p$ by 7.5.3(3) and so $p \cap x = 1 = p \cap y$ by the definition of primes. But then we could derive a contradiction:

$$p \overset{7.5.4(2)}{=} p \cap (x \cup y) \overset{7.5.6(3)}{=} (p \cap x) \cup (p \cap y) = 1 \cup 1 \overset{(1)}{=} 1$$

with the definition of primes.

7.5.8 Coprime numbers. Two numbers x and y are *coprime* (relatively prime) if $x \cap y = 1$. We will need the following property which says that every two successive numbers are coprime:

$$\vdash_{\mathrm{PA}} x \cap (x+1) = 1\ . \tag{1}$$

We claim that (1) follows from

$$\vdash_{\mathrm{PA}} z \mid x \land z \mid x+1 \to z = 1\ . \tag{2}$$

Indeed, take $z := x \cap x+1$ and use 7.5.3(3). For the proof of (2) assume $z \mid x$, $z \mid x+1$, and consider three cases. If $z = 0$ then, since $x + 1 = b \cdot z$ for some b, we obtain a contradiction $x + 1 = b \cdot z = 0$. If $z > 1$ then, since $x = a \cdot z$ for some a, we get $x + 1 = a \cdot z + 1$ and so $(x + 1) \bmod z = 1$ by 7.4.8(2). We now obtain contradiction $z \nmid x+1$ by 7.5.2(1). Thus it must be the case that $z = 1$.

7.5.9 Least common multiple of interval $[1..x]$. We will need a function $\bigcup_{i=1}^{x} i$ yielding finitely many least common multiples of numbers in the (possibly empty) interval $[1..x]$. The function should satisfy:

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

$$\vdash_{\mathrm{PA}} \bigcup_{i=1}^{x'} i = (\bigcup_{i=1}^{x} i) \cup x' \ . \tag{2}$$

We claim that the function can be defined by minimalization

$$\vdash_{\mathrm{PAx}} \bigcup_{i=1}^{x} i = \mu_m[x = 0 \land m = 1 \lor$$
$$x \neq 0 \land \forall y(0 < y \leq x \to y \mid m) \land$$
$$\forall z(\forall y(0 < y \leq x \to y \mid z) \to m \mid z)]$$

whose existence condition

$$\vdash_{\mathrm{PA}} \exists m(x = 0 \land m = 1 \lor$$
$$x > 0 \to \forall y(0 < y \leq x \to y \mid m) \land \forall z(\forall y(0 < y \leq x \to y \mid z) \to m \mid z))$$

is proved by iduction on x. In the base case we take $m := 1$. In the inductive case we obtain an m_1 s.t.

$$x = 0 \land m_1 = 1 \lor$$
$$x \neq 0 \land \forall y(0 < y \land y \leq x \to y \mid m_1) \land \forall z(\forall y(0 < y \leq x \to y \mid z) \to m_1 \mid z) \tag{3}$$

by IH. We claim that it suffices to take $m := m_1 \cup x'$. We consider two cases for x. If $x = 0$ then $m_1 = 1$ and $m = 1 \cup 1 \overset{7.5.7(1)}{=} 1$. We now take any y s.t. $0 < y \leq x' = 1$, i.e. $y = 1$, and we have $y \mid m$. We also have $m = 1 \mid z$ for any z.

If $x \neq 0$ then for any y s.t. $0 < y \leq x'$ we consider two cases agian. If $y < x'$ then $y \mid m_1$ by (3) and, since $m_1 \mid m$, we get $y \mid m$ by transitivity. If $y \geq x'$ then $y = x'$ and we have $y = x' \mid m$. If for any z we have $\forall y(0 < y \leq x' \to y \mid z)$ then we have $x' \mid z$ directly and $m_1 \mid z$ by (3). Hence $m = m_1 \cup x' \mid z$ by 7.5.3(2).

The defining axiom for $\bigcup_{i=1}^{x} i$ directly implies Property (1) and also

$$\vdash_{\mathrm{PA}} 0 < y \leq x \to y \mid \bigcup_{i=1}^{x} i \tag{4}$$

$$\vdash_{\mathrm{PA}} x \neq 0 \land \forall y(0 < y \leq x \to y \mid u) \to \bigcup_{i=1}^{x} i \mid u \ . \tag{5}$$

We prove Property (2) by showing first

$$\vdash_{\mathrm{PA}} \bigcup_{i=1}^{x'} i \mid (\bigcup_{i=1}^{x} i) \cup x' \tag{6}$$

by taking any y s.t. $0 < y \leq x'$ and considering two cases. If $y \leq x$ then $y \mid \bigcup_{i=1}^{x} i$ by (4) and, since $\bigcup_{i=1}^{x} i \mid (\bigcup_{i=1}^{x} i) \cup x'$, we obtain $y \mid (\bigcup_{i=1}^{x} i) \cup x'$ by transitivity. If $y = x'$ then we have $x' \mid (\bigcup_{i=1}^{x'} i) \cup x'$. (6) now follows by (5).

We then prove

$$\vdash_{\mathrm{PA}} (\bigcup_{i=1}^{x} i) \cup x' \mid \bigcup_{i=1}^{x'} i \tag{7}$$

by proving first $\bigcup_{i=1}^{x} i \mid \bigcup_{i=1}^{x'} i$ by considering two cases. If $x = 0$ then $\bigcup_{i=1}^{x} i \overset{(1)}{=} 1 \mid \bigcup_{i=1}^{x'} i$ by 7.5.1(5). If $x \neq 0$ we take any y s.t. $0 < y \leq x$. We have $y \mid \bigcup_{i=1}^{x'} i$ by (4) and hence $\bigcup_{i=1}^{x} i \mid \bigcup_{i=1}^{x'} i$ by (5). From $\bigcup_{i=1}^{x} i \mid \bigcup_{i=1}^{x'} i$ and from $x' \mid \bigcup_{i=1}^{x'} i$ which holds by (4) we get (7) by 7.5.3(2).

Property (2) now follows from (6) and (7) by antisymmetry.

Surprising as it is, we need a proof by induction on x of the following property:

$$\vdash_{\mathrm{PA}} \bigcup_{i=1}^{x} i \neq 0 \;. \tag{8}$$

In the base case the property follows from (1). In the inductive case we have $\bigcup_{i=1}^{x} i \neq 0$ by IH and

$$0 \overset{7.5.3(6)}{\neq} (\bigcup_{i=1}^{x} i) \cup x' \overset{(2)}{=} \bigcup_{i=1}^{x'} i \;.$$

We will need the following lower bound on our function:

$$\vdash_{\mathrm{PA}} x \leq \bigcup_{i=1}^{x} i \tag{9}$$

which is proved by considering two cases. If $x = 0$ then (9) holds by (1). If $x \neq 0$ then $x \mid \bigcup_{i=1}^{x} i$ by (4). Since $\bigcup_{i=1}^{x} i \neq 0$ by (8), the property follows from 7.5.2(2).

7.5.10 Euclid's theorem on prime numbers. The lattice of divisibility has infinitely many atoms by the famous second theorem of Euclid which asserts that there are infinitely many prime numbers:

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

This is proved by taking any x and considering the number $q = (\bigcup_{i=1}^{x} i) + 1$. For every $y > 1$ s.t. $y \leq x$ we have $y \mid \bigcup_{i=1}^{x} i$ by 7.5.9(4). If it were the case that $y \mid q$ then by 7.5.3(4) we would have a contradiction $y = 1$ by 7.5.8(2). We have just proved:

$$\forall y(y > 1 \land y \leq x \to y \nmid q) \tag{2}$$

Since $q \neq 1$ by 7.5.9(8) there is a prime p s.t. $p \mid q$ by 7.5.5(4). By (2) we have $p = 1$, $p = 0$, or $p > x$. But $p \neq 1$ by the definition of primes, $p \neq 0$ by 7.5.5(2), and so it must be the case that $p > x$.

8. Recursive Bootstrapping of PA

8.1 Exponentiation Function

8.1.1 Binary successors. We introduce two *binary successor* functions by explicit definitions:

$$\vdash_{\mathrm{PAx}} x\mathbf{0} = 2{\cdot}x \tag{1}$$
$$\vdash_{\mathrm{PAx}} x\mathbf{1} = 2{\cdot}x + 1\ . \tag{2}$$

Binary successors are interesting because of *binary discrimination*:

$$\vdash_{\mathrm{PA}} \exists y(x = y\mathbf{0} \vee x = y\mathbf{1}) \tag{3}$$

which directly follows from $x = x \div 2{\cdot}2 + x \bmod 2$ with $x \bmod 2 < 2$ holding by 7.4.8(1).

8.1.2 The plan for the introduction of 2^x into PA. We wish to introduce into PA the *exponentiation* function 2^x satisfying the natural recurrences:

$$\vdash_{\mathrm{PA}} 2^0 = 1 \tag{1}$$
$$\vdash_{\mathrm{PA}} 2^{x'} = 2{\cdot}2^x\ . \tag{2}$$

We will not be able to express the computation of 2^x by these recurrences but we will succeed in encoding the computation of 2^x by the following exponentially faster recurrences by *recursion on binary notation*:

$$\begin{aligned} 2^0 &= 1 \\ 2^{x\mathbf{0}} &= (2^x)^2 \quad \text{if } x > 0 \\ 2^{x\mathbf{1}} &= 2{\cdot}(2^x)^2\ . \end{aligned}$$

We will do this by introducing in Par. 8.1.3 the predicate $Pow_2(p)$ of being a power of two satisfying

$$\vdash_{\mathrm{PA}} Pow_2(p) \leftrightarrow \exists x\, 2^x = p$$

and with its help to define in Par. 8.1.10 the log function as the inverse of 2^x by encoding its computation from the following recurrences:

$$\begin{gathered}
Pow_2(p) \land p = 1 \rightarrow \log(p) = 0 \\
Pow_2(p) \land p = q^2 \land p > 1 \rightarrow \log(p) = \log(q)\mathbf{0} \\
Pow_2(p) \land p = 2{\cdot}q^2 \rightarrow \log(p) = \log(q)\mathbf{1} \ .
\end{gathered}$$

That the recurrences cover all cases for p and that they are exclusive will be proved in Par. 8.1.5.

Only then we will be able to introduce in Par. 8.1.11 the exponentiation function satisfying

$$\vdash_{\mathrm{PA}} 2^x = p \leftrightarrow Pow_2(p) \land \log(p) = x \ .$$

8.1.3 The predicate of being a power of two. The predicate $Pow_2(p)$ of p *being a power of two*, i.e. such that $\exists x\, 2^x = p$, can be presented in a clausal form with binary recurrences:

$Pow_2(x\mathbf{0}) \leftarrow x > 0 \land Pow_2(x)$
$Pow_2(x\mathbf{1}) \leftarrow x = 1$.

We claim that we can introduce into PA the predicate Pow_2 to satisfy the clauses properties by the following explicit definition:

$$\vdash_{\mathrm{PAx}} Pow_2(p) \leftrightarrow p \neq 0 \land \forall x(x \mid p \rightarrow x = 1 \lor 2 \mid x) \ . \tag{1}$$

Since $Pow_2(p) \rightarrow p > 0$ holds directly from the definition, the above clauses are equivalent to the following properties:

$$\vdash_{\mathrm{PA}} Pow_2(x\mathbf{0}) \leftrightarrow Pow_2(x) \tag{2}$$
$$\vdash_{\mathrm{PA}} Pow_2(x\mathbf{1}) \leftrightarrow x = 0 \ . \tag{3}$$

(2): In the direction ($\rightarrow$) assume $Pow_2(x\mathbf{0})$. We have $x\mathbf{0} > 0$ and thus $x > 0$ from the definition. Now take any z s.t. $z \mid x$. Since then $z \mid 2{\cdot}x$ we get $z = 1$ or $2 \mid z$ from the assumption $Pow_2(x\mathbf{0})$ and so $Pow_2(x)$ holds. In the direction ($\leftarrow$) assume $Pow_2(x)$. Since $x > 0$, we have $2{\cdot}x > 0$ and we take any z s.t. $z \mid 2{\cdot}x$. We wish to prove $z = 1$ or $2 \mid z$. We consider two cases according to 8.1.1(3). If $z = 2{\cdot}z_1$ for some z_1 then $2 \mid z$. If $z = 2{\cdot}z_1 + 1$ for some z_1 then, since $2{\cdot}x = a{\cdot}z$ for some a, we have $2{\cdot}x = 2{\cdot}a{\cdot}z_1 + a$ and we get $a = a_1{\cdot}2$ for some a_1 by 7.5.2(3). Thus $x = a_1{\cdot}2{\cdot}z_1 + a_1 = a_1{\cdot}(2{\cdot}z_1 + 1)$ and so $z \mid x$. We now get $z = 1$ or $2 \mid x$ from the assumption $Pow_2(x)$.

(3): In the direction ($\rightarrow$) assume $Pow_2(2{\cdot}x+1)$. Since $2{\cdot}x+1 \mid 2{\cdot}x+1$, we get from the assumption $2{\cdot}x + 1 = 1$ or $2 \mid 2{\cdot}x + 1$. The latter cannot be the case by 7.5.2(1) and so $2{\cdot}x + 1 = 1$ must hold. Hence $x = 0$. In the direction ($\leftarrow$) it suffice to prove $Pow_2(1)$. We have $1 \neq 0$ and we take any z s.t. $z \mid 1$. Since $1 \mid z$ by 7.5.1(5), we obtain $z = 1$ by antisymmetry.

We have the following property of powers of two:

$$\vdash_{\text{PA}} Pow_2(p) \rightarrow (Pow_2(q) \leftrightarrow Pow_2(p{\cdot}q)) \tag{4}$$

which is proved by complete induction on p. Assume $Pow_2(p)$ and consider the two cases implied by 8.1.11(3). If $p = 2{\cdot}u + 1$ for some u then $p = 1$ by (3) and the consequent of the property holds trivially. If $p = 2{\cdot}u$ for some u then $Pow_2(u)$ by (2) and we have $0 < u < p$. Hence

$$Pow_2(q) \overset{\text{IH}}{\Leftrightarrow} Pow_2(u{\cdot}q) \overset{(2)}{\Leftrightarrow} Pow_2(2{\cdot}u{\cdot}q) \ .$$

8.1.4 Order of powers. We will need the following ordering properties of powers of two:

$$\vdash_{\text{PA}} Pow_2(p) \land Pow_2(q) \rightarrow (p < q \leftrightarrow \exists u\, q = 2{\cdot}u{\cdot}p) \tag{1}$$
$$\vdash_{\text{PA}} Pow_2(p) \land Pow_2(q) \rightarrow (p \le q \leftrightarrow \exists u\, q = u{\cdot}p) \tag{2}$$
$$\vdash_{\text{PA}} Pow_2(p) \land Pow_2(q) \rightarrow ((p < q \leftrightarrow 2{\cdot}p \le q) \land (p < 2{\cdot}q \leftrightarrow p \le q)) \ . \tag{3}$$

(1): In the direction $(\rightarrow)$ we prove

$$\vdash_{\text{PA}} \forall p\,(Pow_2(p) \land Pow_2(q) \land p < q \rightarrow \exists u\, q = 2{\cdot}u{\cdot}p)$$

by complete induction on q. We take any p, assume $Pow_2(p)$, $Pow_2(q)$, $p < q$, and consider two cases for q implied by 8.1.11(3). The case $q = 2{\cdot}q_1 + 1$ for some q_1 leads to a contradiction because then $p < q = 1$ by 8.1.3(3). If $q = 2{\cdot}q_1$ for some q_1 then we consider two similar cases for p. If $p = 2{\cdot}p_1 + 1$ for some p_1 then $p = 1$ by 8.1.3(3) and it suffices to take $u := q_1$. If $p = 2{\cdot}p_1$ for some p_1 then we have $Pow_2(p_1)$ and $Pow_2(q_1)$ by 8.1.3(2) and, since $0 < p_1 < q_1 < q$, we have $q_1 = 2{\cdot}u{\cdot}p_1$ for some u by IH. It now suffices to take the same u because $q = 2{\cdot}q_1 = 2{\cdot}2{\cdot}u{\cdot}p_1 = 2{\cdot}u{\cdot}p$.

In the direction $(\leftarrow)$ assume $Pow_2(p)$, $Pow_2(q)$, and $q = 2{\cdot}u{\cdot}p$. Since $q \neq 0$ and $p \mid q$, we get $p \le q$ by 7.5.2(2) and it cannot be the case $p = q$ because then $p = q = a = 0$.

(2): In the direction $(\rightarrow)$ assume $Pow_2(p)$, $Pow_2(q)$, $p \le q$, and consider two cases. If $p < q$ then the conclusion of the property holds by (1). If $p = q$ then it suffices to take $u := 1$. In the direction $(\leftarrow)$ assume $Pow_2(p)$, $Pow_2(q)$, and $q = u{\cdot}p$. We now consider two cases for u implied by 8.1.11(3). If $u = 2{\cdot}u_1$ for some u_1 then $p < q$ by (1). If $u = 2{\cdot}u_1 + 1$ for some u_1 then, since $Pow_2(p{\cdot}u)$, we have $Pow_2(u)$ by (4). Hence $u = 1$ by 8.1.3(3) and we have $p = q$.

(3): Assume $Pow_2(p)$ and $Pow_2(q)$. For the first conjunct we have $p < q$ iff, by (1), $q = 2{\cdot}u{\cdot}p$ for some u, since $Pow_2(2{\cdot}p)$ by 8.1.3(2), iff, by (2) $2{\cdot}p \le q$.

For the second conjunct we have $p \le q$ iff not $q < p$ iff, by the first conjunct, not $2{\cdot}q \le p$ iff $p < 2{\cdot}q$.

We will also often use the following easy to prove property of powers of two:

$$\vdash_{\mathrm{PA}} Pow_2(u) \land Pow_2(p) \land u \geq p \land u > s \land p > t \rightarrow \\ (\exists b(a{\cdot}u + s = b{\cdot}p + t) \leftrightarrow \exists c(s = c{\cdot}p + t)) \tag{4}$$

which is proved by assuming its antecedent and noting that $u = d{\cdot}p$ for some d by (2). In the direction ($\rightarrow$) assume also $a{\cdot}u + s = b{\cdot}p + t$ for some b and, since $p > 0$, we get

$$b{\cdot}p + t = a{\cdot}u + s = a{\cdot}d{\cdot}p + s \div p{\cdot}p + s \bmod p = (a{\cdot}d + s \div p){\cdot}p + s \bmod p \ .$$

Because $t < p$ we have $t = s \bmod p$ by 7.4.8(2) and, since then

$$s = s \div p{\cdot}p + s \bmod p = s \div p{\cdot}p + t \ ,$$

it suffices to take $c := s \div p$.

In the direction ($\leftarrow$) assume also $s = c{\cdot}p + t$ for some c and, since

$$a{\cdot}u + s = a{\cdot}d{\cdot}p + c{\cdot}p + t = (a{\cdot}d + c){\cdot}p + t \ ,$$

it suffices to take $b := a{\cdot}d + c$.

8.1.5 Fast computation of Pow_2. The clausal form for the predicate Pow_2 given in Par. 8.1.3 has a 'slow' recursion. We need the recursion exponentially speeded up to that for the log function discussed in Par. 8.1.2. The predicate Pow_2 with fast recursion has the following clausal form:

$$\begin{aligned}
&Pow_2(p) \leftarrow p = 1 \\
&Pow_2(p) \leftarrow p > 1 \land p = [\sqrt{p}]^2 \land Pow_2([\sqrt{p}]) \\
&Pow_2(p) \leftarrow p > 1 \land p = 2{\cdot}[\sqrt{p \div 2}]^2 \land Pow_2([\sqrt{p \div 2}]) \ .
\end{aligned}$$

We wish to prove that the three clauses cover all cases and that they are mutually exclusive. In other words we wish to prove that for every power of two p we have either $p = 1$ or else there is a unique power of two $q < p$ such that either $p = q^2$ or $p = 2{\cdot}q^2$ and that at most one of the cases applies.

The existence part of the discrimination is expressed by

$$\vdash_{\mathrm{PA}} Pow_2(p) \rightarrow p = 1 \lor p > 1 \land \exists q(q < p \land Pow_2(q) \land (p = q^2 \lor p = 2{\cdot}q^2)) \tag{1}$$

which is proved by complete induction on p. Thus assume $Pow_2(p)$ and consider two cases for p implied by 8.1.11(3). If $p = 2{\cdot}p_1 + 1$ for some p_1 we have $p = 1$ by 8.1.3(3). If $p = 2{\cdot}p_1$ for some p_1 then we have $Pow_2(p_1)$ by 8.1.3(2). Note that $p > p_1 \geq 1$. Since $p_1 < p$, one of the three cases obtains by IH. If $p_1 = 1$ then we have $p = 2{\cdot}p_1 = 2{\cdot}1 = 2{\cdot}1^2$. It now suffices to take $q := 1$ because we also have $1 < p$ and $Pow_2(1)$ by 8.1.3(3).

If $p_1 > 0$, $p_1 = q_1^2$, $q_1 < p_1$, and $Pow_2(q_1)$ for some q_1 then it suffices to take $q := q_1$ because we have $q = q_1 < p_1 < p$, $Pow_2(q)$, and $p = 2{\cdot}p_1 = 2{\cdot}q_1^2 = 2{\cdot}q^2$.

If $p_1 > 0$, $p_1 = 2{\cdot}q_1^2$, $q_1 < p_1$, and $Pow_2(q_1)$ for some q_1 then it suffices to take $q := 2{\cdot}q_1$ because we have $Pow_2(q)$ by 8.1.3(2), $q = 2{\cdot}q_1 < 2{\cdot}p_1 = p$, and $p = 2{\cdot}p_1 = 2{\cdot}2{\cdot}q_1^2 = (2{\cdot}q_1)^2 = q^2$.

The uniqueness part of the discrimination follows from

$$\vdash_{\mathrm{PA}} Pow_2(p) \land (p = q_1^2 \lor p = 2{\cdot}q_1^2) \land (p = q_2^2 \lor p = 2{\cdot}q_2^2) \to q_1 = q_2 \ . \qquad (2)$$

This property follows from two auxiliary properties

$$\vdash_{\mathrm{PA}} x^2 = y^2 \to x = y \qquad (3)$$

$$\vdash_{\mathrm{PA}} x^2 = 2{\cdot}y^2 \to x = 0 \land y = 0 \qquad (4)$$

because when we assume the antecedent of (2) there are four cases to consider. For the cases $p = q_1^2$ and $p = q_2^2$ or $p = 2{\cdot}q_1^2$ and $p = 2{\cdot}q_2^2$ the conclusion $q_1 = q_2$ follows by (3).

The cases $p = q_1^2$ and $p = 2{\cdot}q_2^2$ or $p = 2{\cdot}q_1^2$ and $p = q_2^2$ cannot happen because we obtain $p = q_1 = q_2 = 0$ by (4) which contradicts the assumption $Pow_2(p)$.

We now prove the two auxiliary properties: (3): This follows by trichotomy from

$$x < y \to x^2 < y^2$$

proved by assuming $x < y$ and considering two cases. If $x = 0$ then, $0^2 = 0{\cdot}y < y{\cdot}y = y^2$. If $x > 0$ then, since also $y > 0$, we get

$$x^2 = x{\cdot}x < y{\cdot}x < y{\cdot}y = y^2 \ .$$

(4): This is proved as $\forall y(4)$ by complete induction on x. Thus assume $x^2 = 2{\cdot}y$ and consider two cases for x implied by 8.1.11(3). If $x = 2{\cdot}x_1 + 1$ for some x_1 then we obtain a contradiction from $x^2 = 4{\cdot}x_1^2 + 4{\cdot}x_1 + 1 = 2{\cdot}y_2$ because the left-hand side is odd.

If $x = 2{\cdot}x_1$ some x_1 then we get $2{\cdot}x_1^2 = y^2$ and we consider two cases for y implied by 8.1.11(3). The case $y = 2{\cdot}y_1 + 1$ leads to a contradiction similarly as above and when $y = 2{\cdot}y_1$ for some y_1 we obtain $x_1^2 = 2{\cdot}y_1^2$. If it were the case that $x_1 > 0$ then we would have $x_1 < x$ and we would obtain a contradiction $x_1 = y_1 = 0$ by IH. Thus it must be the case that $x_1 = 0$ and then from $0 = 2{\cdot}y_1^2$ we get $y_1 = 0$. Hence $x = y = 0$.

We will also need the following property:

$$\vdash_{\mathrm{PA}} Pow_2(p) \land (p > 1 \land p = q^2 \lor p = 2{\cdot}q^2) \to p > 1 \land q < p \land Pow_2(q) \ . \quad (5)$$

whis is proved by assuming the antecedent. In either case we have $p > 1$ and from (1) we get

$$q_1 < p \land Pow_2(q_1) \land (p = q_1^2 \lor p = 2{\cdot}q_1^2)$$

for some q_1. But then $q = q_1$ by (2).

8.1.6 Sequences of powers. We wish to encode the sequence of powers needed in the computation of $Pow_2(p)$ by the clauses given in Par. 8.1.5. If $p = 2^x$ then the sequence is given by $2^{x_k}, 2^{x_{k-1}}, \ldots, 2^{x_0}$ for some $k \geq 0$ such that $x_k = x$, $x_i = x_{i+1} \div 2$ for all $i < k$, and $x_0 = 0$ where $x_1 > 0$ if $k > 0$. We encode the sequence of powers by its sum $s = \sum_{i \leq k} 2^{x_i}$. The reader can convince himself that the following explicitly defined predicate:

$$\vdash_{\text{PAx}} Ps(s) \leftrightarrow \forall p \forall a \forall t (Pow_2(p) \land s = a{\cdot}2{\cdot}p + p + t \land p > t \rightarrow p = 1 \lor \exists q \exists t_1 (t = q + t_1 \land q > t_1 \land (p{>}1 \land p = q^2 \lor p = 2{\cdot}q^2))) \tag{1}$$

is true in the standard interpretation of PA whenever s is a sequence of powers determined as above. The main idea is that when p and q are two adjacent powers in s then either $p = q^2$ or $p = 2{\cdot}q^2$ holds. We will need the following properties of sequences of powers:

$$\vdash_{\text{PA}} Ps(0) \tag{2}$$
$$\vdash_{\text{PA}} Ps(1) \tag{3}$$
$$\vdash_{\text{PA}} Pow_2(u) \land u > 1 \land u > s \land \vdash_{\text{PA}} Ps(u + s) \rightarrow Ps(u^2 + u + s) \tag{4}$$
$$\vdash_{\text{PA}} Pow_2(u) \land u > s \land \vdash_{\text{PA}} Ps(u + s) \rightarrow Ps(2{\cdot}u^2 + u + s) \tag{5}$$
$$\vdash_{\text{PA}} Pow_2(u) \land u > s \land Ps(u + s) \rightarrow Ps(u) \ . \tag{6}$$

(2): This holds trivially because $0 = a{\cdot}2{\cdot}p + p + t$ for no p such that $Pow_2(p)$.

(3): Take any p, a, t such that $Pow_2(p)$, $1 = a{\cdot}2{\cdot}p + p + t$ and $p > t$ holds. We then clearly have $p = 1$ and $a = t = 0$.

(4): Assume the antecedent of the property and note that $Pow_2(u^2)$ by 8.1.3(4) and $Pow_2(2{\cdot}u^2)$ by 8.1.3(2). For the proof of $Ps(u^2 + u + s)$ take any p, a, t such that $Pow_2(p)$, $u^2 + u + s = a{\cdot}2{\cdot}p + p + t$, $p > t$, and note that $Pow_2(2{\cdot}p)$ by 8.1.3(2). We consider two cases by dichotomy. If $u^2 \geq 2{\cdot}p$ then, since $u^2 > u$, we have $u^2 \geq 2{\cdot}u > u + s$ by 8.1.4(3). We also have $2{\cdot}p > p + t$ and so $u + s = a_1{\cdot}2{\cdot}p + p + t$ for some a_1 by 8.1.4(4). We now obtain from $Ps(u + s)$

$$p = 1 \lor \exists q \exists t_1 (t = q + t_1 \land q > t_1 \land (p{>}1 \land p = q^2 \lor p = 2{\cdot}q^2))$$

which is the consequent of $Ps(u^2 + u + s)$.

If $u^2 < 2{\cdot}p$ then $u^2 \leq p$ by 8.1.4(3). If it were the case that $a > 0$ then we would obtain contradiction

$$u^2 + u + s < 2{\cdot}u^2 \leq 2{\cdot}p < a{\cdot}2{\cdot}p + p + t \ .$$

We thus have $u^2 + u + s = p + t$ and if it were the case that $u^2 < p$ then we would obtain contradiction

$$u^2 + u + s < 2{\cdot}u^2 \overset{8.1.4(3)}{\leq} p \leq p + t \ .$$

This means that $p = u^2$ and so $u + s = t$, $p > 1$. Thus it suffices to take $q := u$ and $t_1 := s$ to satisfy the consequent of $Ps(u^2 + u + s)$:

$$\exists q \exists t_1 (t = q + t_1 \land q > t_1 \land (p{>}1 \land p = q^2 \lor p = 2{\cdot}q^2)) \; .$$

Property (5) is proved similarly.

(6): Assume $Pow_2(u)$, $u > s$, $Ps(u + s)$, and for the proof of $Ps(s)$ take any p, a, t such that $Pow_2(p)$, $s = a{\cdot}2{\cdot}p + p + t$, $p > t$, and note that $Pow_2(2{\cdot}p)$ by 8.1.3(2). We consider two cases by dichotomy. If $u \geq 2{\cdot}p$ then $u + s = a_1{\cdot}2{\cdot}p + p + t$ for some a_1 by 8.1.4(4). The consequent of $Ps(s)$ now follows from that of $Ps(u + s)$. The case $u < 2{\cdot}p$ cannot happen because it would lead to contradiction by

$$s < u \overset{8.1.4(3)}{\leq} p \leq a{\cdot}2{\cdot}p + p + t \; .$$

8.1.7 The function $ps(p)$ yielding the power sequence for p. The function $ps(p)$ yielding the sequence of powers starting from $p = 2^k$, i.e. such that $ps(p) = \sum_{i \leq k} 2^{x_i}$ (see Par. 8.1.6), is introduced into PA by minimalization:

$$\vdash_{\text{PAx}} ps(p) = \mu_s[Pow_2(p) \to p \leq s \land Ps(s)]$$

whose existence condition follows from the stronger property

$$\vdash_{\text{PA}} Pow_2(p) \to \exists s(Ps(p + s) \land p > s) \tag{1}$$

proved by complete induction on p where we assume $Pow_2(p)$ and continue by the case analysis implied by 8.1.5(1). If $p = 1$ then it suffices to take $s := 0$ by 8.1.6(3). If $p > 1$ then there is a q s.t. $q < p$, $Pow_2(q)$, and either $p = q^2$ or $p = 2{\cdot}q^2$. In either case we have a t s.t. $Ps(q + t)$ and $q > t$ by IH. If $q > 1$ then we have $2{\cdot}q \leq q^2 \leq p$ and if $q = 1$ then $p = 2$ and we have $2{\cdot}q = p$. In either case it suffices to take $s := q + t < 2{\cdot}q \leq p$ because we have $Ps(p + s)$ by 8.1.6(4) when $p = q^2$ and by 8.1.6(5) when $p = 2{\cdot}q^2$.

The defining axiom for ps implies:

$$\vdash_{\text{PA}} Pow_2(p) \to p \leq ps(p) \land Ps(ps(p)) \tag{2}$$

$$\vdash_{\text{PA}} p \leq s \land Ps(s) \to ps(p) \leq s \; . \tag{3}$$

The basic properties of the power sequence function are:

$$\vdash_{\text{PA}} Pow_2(p) \to ps(p) < 2{\cdot}p \tag{4}$$

$$\vdash_{\text{PA}} ps(1) = 1 \tag{5}$$

$$\vdash_{\text{PA}} \forall s(Pow_2(p) \land Ps(p + s) \land p > s \to ps(p) = p + s) \tag{6}$$

$$\vdash_{\text{PA}} Pow_2(p) \land (p > 1 \land p = q^2 \lor p = 2{\cdot}q^2) \to ps(p) = p + ps(q) \; . \tag{7}$$

(4): Assume $Pow_2(p)$ and obtain an s s.t. $Ps(p + s)$ and $p > s$ by (1). Thus

$$ps(p) \overset{(3)}{\leq} p + s < 2{\cdot}p \ .$$

(5): We have $Pow_2(1)$ by 8.1.3(3) and so $1 \leq ps(1)$ by (2). We have $ps(1) < 2$ by (4) and so $ps(1) = 1$.

(6): By complete induction on p. Take any s, assume $Pow_2(p)$, $Ps(p+s)$, $p > s$, and consider two cases. If $p = 1$ then we have $s = 0$ and $ps(1) \overset{(5)}{=} 1+0$. If $p > 1$ then for $s := p + s = 0{\cdot}2{\cdot}p + p + s$ we obtain from $Ps(p+s)$

$$s = q + t_1 \land q > t_1 \land (p > 1 \land p = q^2 \lor p = 2{\cdot}q^2)$$

for some q and t_1. We have $Pow_2(q)$ and $q < p$ by 8.1.5(5), $Ps(s)$ by 8.1.6(6), and so $ps(q) = q + t_1 = s$ by IH.

We also have $p \leq ps(p)$ and $Ps(ps(p))$ by (2) and $ps(p) < 2{\cdot}p$ by (4). Hence $ps(p) = p + t$ for some t s.t. $p > t$. For $s := p + t = 0{\cdot}2{\cdot}p + p + t$ we obtain from $Ps(p+t)$

$$t = q_1 + t_2 \land q_1 > t_2 \land (p > 1 \land p = q_1^2 \lor p = 2{\cdot}q_1^2)$$

for some q_1 and t_2. We have $q = q_1$ by 8.1.5(2) and, since $Ps(t)$ holds by 8.1.6(6), we get $ps(q) = q+t_2 = t$ by IH. Thus $ps(p) = p+t = p+ps(q) = p+s$.

(7): Assume the antecedent of the property. We have $Pow_2(q)$, $q < p$, and $p > 1$ by 8.1.5(5). From (1) there is an s s.t. $Ps(p+s)$ and $p > s$. Hence $ps(p) = p + s$ by (6). For $s := p + s = 0{\cdot}2{\cdot}p + p + s$ we obtain from $Ps(p+s)$

$$s = q_1 + t_1 \land q_1 > t_1 \land (p > 1 \land p = q_1^2 \lor p = 2{\cdot}q_1^2)$$

for some q_1 and t_1. We get $q = q_1$ by 8.1.5(2), and, since $Ps(s)$ holds by 8.1.6(6), we get $ps(q) = q + t_1 = s$ by (6).

8.1.8 Course of values sequences for the log **function.** We will encode the computation of the log function such that for $p = 2^x$ we have $\log(p) = x$ by the fast recurrences discussed in Par. 8.1.2. To that end we take the power sequence $ps(p) = \sum_{i \leq k} 2^{x_i}$, use the fact that each $x_i < 2^{x_i}$, that there are at least x_i bits in $ps(p)$ between the bit postions $2^{x_{i+1}}$ and 2^{x_i}, and encode the *course of values* sequence $x_k, x_{k-1}, \ldots, x_0$ for the successive values of $\log(2^{x_i})$ at the corresponding bits of the power sequence. We will define in Par. 8.1.9 a *course of values* function $\overline{\log}(p)$ to yield the course of values sequence such that $\overline{\log}(p) = \sum_{i \leq k} x_i{\cdot}2^{x_i}$.

We recall that two neighboring powers p and q in the power sequence $ps(u)$ are such that either $p = q^2$ or $p = 2{\cdot}q^2$. The relation between the values x and y stored in the course of values sequence $\overline{\log}(u)$ at the powers p and q respectively, i.e. when $\overline{\log}(u) = a{\cdot}p^2 + x{\cdot}p + y{\cdot}q + u_1$ for $x < p$, $y < q$, and $q > u_1$, is expressed by $Nbs(\overline{\log}(u), p, q)$ where the predicate is explicitly introduced into PA as follows:

$$\vdash_{\mathrm{PAx}} Nbs(s,p,q) \leftrightarrow \exists b \exists x \exists y \exists s_1 (s = b{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1 \land$$
$$x < p \land y < q \land q > s_1 \land (p > 1 \land p = q^2 \land x = y\mathbf{0} \lor p = 2{\cdot}q^2 \land x = y\mathbf{1})) \ . \quad (1)$$

The reader can convince himself that the following explicitly defined predicate:

$$\vdash_{\mathrm{PAx}} Cs(u,s) \leftrightarrow \forall p \forall a \forall t(Pow_2(p) \wedge ps(u) = a{\cdot}2{\cdot}p + p + t \wedge p > t \rightarrow p = 1 \wedge 2 \mid s \vee \exists q\, Nbs(s,p,q)) \tag{2}$$

is true in the standard interpretation of PA whenever s is the course of values sequence for u, i.e. $\overline{\log}(u) = s$. We will need the following properties of Cs:

$$\vdash_{\mathrm{PA}} Cs(1,0) \tag{3}$$

$$\vdash_{\mathrm{PA}} Pow_2(u) \wedge u > 1 \wedge Cs(u, z{\cdot}u + s) \wedge z < u \wedge u > s \rightarrow Cs(u^2, z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s) \tag{4}$$

$$\vdash_{\mathrm{PA}} Pow_2(u) \wedge Cs(u, z{\cdot}u + s) \wedge z < u \wedge u > s \rightarrow Cs(2{\cdot}u^2, z\mathbf{1}{\cdot}2{\cdot}u^2 + z{\cdot}u + s) \tag{5}$$

$$\vdash_{\mathrm{PA}} Pow_2(u) \wedge Cs(u, z{\cdot}u + s) \wedge u > s \wedge (u > 1 \wedge u = u_1^2 \vee u = 2{\cdot}u_1^2) \rightarrow Cs(u_1, s)\ . \tag{6}$$

(3): Take any p, a, t such that $Pow_2(p)$, $ps(1) = a{\cdot}2{\cdot}p + p + t$ and $p > t$ holds. We have $ps(1) = 1$ by 8.1.7(5) and thus we clearly must have $p = 1$ and $a = t = 0$. Since $2 \mid 0$, we get $Cs(1,0)$.

(4): Assume the antecedent of the property and note that $Pow_2(2{\cdot}u)$ by 8.1.3(2) and $Pow_2(u^2)$ by 8.1.3(4). Since $u < u^2$, we also have

$$u < 2{\cdot}u \overset{8.1.4(3)}{\leq} u^2\ . \tag{7}$$

For the proof of $Cs(u^2, z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s)$ we take any p, a, t such that $Pow_2(p)$, $ps(u^2) = a{\cdot}2{\cdot}p + p + t$, $p > t$. We thus have

$$u^2 + ps(u) \overset{8.1.7(7)}{=} ps(u^2) = a{\cdot}2{\cdot}p + p + t\ . \tag{8}$$

We also have $Pow_2(2{\cdot}p)$ by 8.1.3(2) and

$$u^2 = u{\cdot}u \geq (z+1){\cdot}u = z{\cdot}u + z > z{\cdot}u + s\ . \tag{9}$$

We now consider two cases by dichotomy. If $u^2 > p$ then, since $u^2 \geq 2{\cdot}p$ by 8.1.4(3), we have $u^2 \geq 2{\cdot}u > ps(u)$ by 8.1.7(4). Since also $2{\cdot}p > p + t$, we obtain

$$ps(u) = a_1{\cdot}2{\cdot}p + p + t \tag{10}$$

for some a_1 by 8.1.4(4). From $Cs(u, z{\cdot}u + s)$ we now get two cases. If $p = 1$ and $2 \mid z{\cdot}u + s$ then, since $1 < u < u^2$, we have $u^2 = 2{\cdot}k{\cdot}u$ for some k by 8.1.4(1) and so $2 \mid z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s$. This proves $Cs(u^2, z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s)$.

If $Nbs(z{\cdot}u + s, p, q)$ for some q then we have

$$z{\cdot}u + s = b{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1 \wedge x < p \wedge y < q \wedge q > s_1 \wedge$$
$$(p > 1 \wedge p = q^2 \wedge x = y\mathbf{0} \vee p = 2{\cdot}q^2 \wedge x = y\mathbf{1}))\ .$$

for some b, x, y, and s_1. We then get

$$p^2 = p{\cdot}p \geq (x+1){\cdot}p = x{\cdot}p + p \geq x{\cdot}p + q^2 \geq x{\cdot}p + (y+1){\cdot}q = x{\cdot}p + y{\cdot}q + q > x{\cdot}p + y{\cdot}q + s_1\ . \tag{11}$$

We have $2{\cdot}u \overset{8.1.7(4)}{>} ps(u) \overset{(10)}{\geq} p$ and so $u \geq p$ by 8.1.4(3). From $u^2 \geq p^2$, (9), and (11) we obtain

$$z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s = b_1{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1$$

for some b_1 by 8.1.4(4). But then $Nbs(z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s, p, q)$ which proves $Cs(u^2, z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s)$.

The other dichotomy case is $u^2 \leq p$. If $a > 0$ then we contradict (8) by

$$ps(u^2) \overset{8.1.7(4)}{<} 2{\cdot}u^2 \leq 2{\cdot}p \leq a{\cdot}2{\cdot}p + p + t\ .$$

We thus have $a = 0$ and by (8) we get $u^2 + ps(u) = p + t$. If it were the case that $u^2 < p$ then we would contradict (8) again:

$$u^2 + ps(u) = ps(u^2) \overset{8.1.7(4)}{<} 2{\cdot}u^2 \overset{8.1.4(3)}{\leq} p \leq p + t\ .$$

Thus $u^2 = p$ and for $q := u$, $b := 0$, $x := z\mathbf{0}$, $y := z$, and $s_1 := s$ we have $Nbs(z\mathbf{0}{\cdot}u^2 + z{\cdot}u + s, p, q)$ because then $y = z < u = q$, $q = u > s = s_1$, and $x = z\mathbf{0} < 2{\cdot}u \overset{(7)}{\leq} u^2 = p$.

(5): This is proved similarly as (4).

(6): Assume the antecedent of the property. We have $Pow_2(u_1)$ and $Pow_2(2{\cdot}u_1)$ by 8.1.3(2) and 8.1.3(4). If $u = u_1^2$ and $u_1 > 1$ then, since $u > u_1$, we get $u \geq 2{\cdot}u_1$ by 8.1.4(3). If $u = 2{\cdot}u_1^2$ then, since $u_1^2 \geq u_1$, we get $u \geq 2{\cdot}u_1$ again. For the proof of $Cs(u_1, s)$ we take any p, a, t such that $Pow_2(p)$, $ps(u_1) = a{\cdot}2{\cdot}p + p + t$, and $p > t$. Since $Pow_2(2{\cdot}p)$ by 8.1.3(2), $2{\cdot}p > p + t$, and

$$u \geq 2{\cdot}u_1 \overset{8.1.7(4)}{>} ps(u_1) \geq p\ , \tag{12}$$

i.e. $u \geq 2{\cdot}p$ by 8.1.4(3), we obtain

$$ps(u) \overset{8.1.7(7)}{=} u + ps(u_1) = a_1{\cdot}2{\cdot}p + p + t$$

for some a_1 by 8.1.4(4). From $Cs(u, z{\cdot}u + s)$ we now get two cases. If $p = 1$ and $2 \mid z{\cdot}u + s$ then from $u \geq 2{\cdot}u_1$ we obtain $2 \mid z{\cdot}u$ because $u = k{\cdot}2{\cdot}u_1$ for some k by 8.1.4(2). Thus $2 \mid s$ by 7.5.2(3) which proves $Cs(u_1, s)$.

If $Nbs(z{\cdot}u + s, p, q)$ for some q then we have

$$z{\cdot}u + s = b{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1 \land x < p \land y < q \land q > s_1 \land$$
$$(p > 1 \land p = q^2 \land x = y\mathbf{0} \lor p = 2{\cdot}q^2 \land x = y\mathbf{1}))\ .$$

for some b, x, y, and s_1. We have $Pow_2(p^2)$ by 8.1.3(4) and $p^2 > x{\cdot}p+y{\cdot}q+s_1$ similarly as in (11). We have $2{\cdot}u_1 > p$ by (12) and so $2{\cdot}u_1 \geq 2{\cdot}p$, i.e. $u_1 \geq p$, by 8.1.4(3). Hence $u \geq u_1^2 \geq p^2$ and, since $u > s$, we obtain $s = b_1{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1$ for some b_1 by 8.1.4(4). But then $Nbs(s, p, q)$ which proves $Cs(u_1, s)$.

8.1.9 The course of values function $\overline{\log}(p)$ for $\log(p)$. The function $\overline{\log}(p)$ yielding the course of values sequence from $p = 2^k$ for log, i.e. such that $\overline{\log}(p) = \sum_{i \leq k} x_i{\cdot}2^{x_i}$ (see Par. 8.1.8), is introduced into PA by minimalization:

$$\vdash_{\text{PAx}} \overline{\log}(p) = \mu_s[Pow_2(p) \rightarrow Cs(p, s)] \tag{1}$$

whose existence condition follows from the stronger property

$$\vdash_{\text{PA}} Pow_2(p) \rightarrow \exists s(Cs(p, s) \land p^2 > s) \tag{2}$$

proved by complete induction on p where we assume $Pow_2(p)$ and continue by the case analysis of p implied by 8.1.5(1). If $p = 1$ then it suffices to take $s := 0$ by 8.1.8(3). If $p > 1$ then there is a q s.t. $q < p$, $Pow_2(q)$, and either $p = q^2$ or $p = 2{\cdot}q^2$. We obtain a $t < q^2$ such that $Cs(q, t)$ by IH. We have $t = z{\cdot}q + s_1$ for $z = t \div q$ and $s_1 = t \bmod q < q$. Also $z < q$ because if $z \geq q$ we would get a contradiction $q^2 > t = z{\cdot}q + s_1 \geq q^2$. But then

$$z\mathbf{1} = 2{\cdot}z + 1 = z + z + 1 < q + z + 1 \leq q + q = 2{\cdot}q \overset{8.1.4(3)}{\leq} p\ . \tag{3}$$

Now, if $p = q^2$ then we have $Cs(p, s)$ for $s = z\mathbf{0}{\cdot}p + t$ by 8.1.8(4), and if $p = 2{\cdot}q^2$ then we have $Cs(p, s)$ for $s = z\mathbf{1}{\cdot}p + t$ by 8.1.8(5). In either case we have

$$s \leq z\mathbf{1}{\cdot}p + t < z\mathbf{1}{\cdot}p + q^2 \leq z\mathbf{1}{\cdot}p + p = (z\mathbf{1} + 1){\cdot}p \overset{(3)}{\leq} p^2\ .$$

The defining axiom for $\overline{\log}$ implies:

$$\vdash_{\text{PA}} Pow_2(p) \rightarrow Cs(p, \overline{\log}(p)) \tag{4}$$
$$\vdash_{\text{PA}} Cs(p, s) \rightarrow \overline{\log}(p) \leq s\ . \tag{5}$$

We will need the following basic properties of the function $\overline{\log}$:

$$\vdash_{\text{PA}} Pow_2(p) \rightarrow \overline{\log}(p) < p^2 \tag{6}$$
$$\vdash_{\text{PA}} \overline{\log}(1) = 0 \tag{7}$$
$$\vdash_{\text{PA}} \forall z \forall s(Pow_2(p) \land Cs(p, z{\cdot}p + s) \land z < p \land p > s \rightarrow \overline{\log}(p) = z{\cdot}p + s) \tag{8}$$
$$\vdash_{\text{PA}} Pow_2(p) \land p > 1 \land p = q^2 \rightarrow \overline{\log}(p) = (\overline{\log}(q) \div q)\mathbf{0}{\cdot}p + \overline{\log}(q) \tag{9}$$
$$\vdash_{\text{PA}} Pow_2(p) \land p = 2{\cdot}q^2 \rightarrow \overline{\log}(p) = (\overline{\log}(q) \div q)\mathbf{1}{\cdot}p + \overline{\log}(q)\ . \tag{10}$$

(6): Assume $Pow_2(p)$. We have $Cs(p,s)$ for some $s < p^2$ by (2) and so $\overline{\log}(p) \le s < p^2$ by (5).

(7): We have $Cs(1,0)$ by 8.1.8(3) and so $\overline{\log}(1) \le 0$ by (5).

(8): By complete induction on p. Take any z, s, assume $Pow_2(p)$, $Cs(p, z{\cdot}p + s)$, $z < p$, and $p > s$. We have

$$ps(p) = p + t = 0{\cdot}2{\cdot}p + p + t \tag{11}$$

for some $t < p$ by 8.1.7(2) and 8.1.7(4). From $Cs(p, z{\cdot}p + s)$ we then obtain

$$p = 1 \wedge 2 \mid z{\cdot}p + s \vee \exists q Nbs(z{\cdot}p + s, p, q) \ .$$

If $p = 1$ then, since $z = s = 0$, we have

$$\overline{\log}(p) = \overline{\log}(1) \overset{(7)}{=} 0 = z{\cdot}p + s \ .$$

If $Nbs(z{\cdot}p + s, p, q)$ for some q then we have

$$\begin{aligned}&z{\cdot}p + s = b{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s_1 \wedge x < p \wedge y < q \wedge q > s_1 \wedge\\&(p > 1 \wedge p = q^2 \wedge x = y\mathbf{0} \vee p = 2{\cdot}q^2 \wedge x = y\mathbf{1}))\end{aligned}$$

for some b, x, y, and s_1. We obtain $p > 1$, $q < p$, $Pow_2(q)$ by 8.1.5(5). Because $z < p$ we have $b = 0$ and, since $y{\cdot}q + s_1 < y{\cdot}q + q = (y + 1){\cdot}q \le q^2 \le p$, we obtain $z = x$, and $s = y{\cdot}q + s_1$ by 7.4.8(1)(2). We have $Cs(q, s)$ by 8.1.8(6) and so $\overline{\log}(q) = y{\cdot}q + s_1 = s$ by IH.

We have $Cs(p, \overline{\log}(p))$ by (4) and from (11), since $p > 1$, we obtain $Nbs(\overline{\log}(p), p, q_1)$ for some q_1 for which we have $q = q_1$ by 8.1.5(2). Thus

$$\begin{aligned}&\overline{\log}(p) = b_1{\cdot}p^2 + x_1{\cdot}p + y_1{\cdot}q + s_2 \wedge x_1 < p \wedge y_1 < q \wedge q > s_2 \wedge\\&(p > 1 \wedge p = q^2 \wedge x_1 = y_1\mathbf{0} \vee p = 2{\cdot}q^2 \wedge x_1 = y_1\mathbf{1}))\end{aligned}$$

for some b_1, x_1, y_1, and s_2. We have $b_1 = 0$ by (6) and, since $y_1{\cdot}q + s_2 < p$, we obtain $Cs(q, y_1{\cdot}q + s_2)$ by 8.1.8(6) and $\overline{\log}(q) = y_1{\cdot}q + s_2$ by IH. Since $\overline{\log}(q) = y{\cdot}q + s_1$, we obtain $y = y_1$ and $s_1 = s_2$ by 7.4.8(1)(2). Because of 8.1.5(4) we then have $x = x_1$ and so

$$\overline{\log}(p) = x_1{\cdot}p + y_1{\cdot}q + s_2 = x{\cdot}p + \overline{\log}(q) = z{\cdot}p + y{\cdot}q + s_1 = z{\cdot}p + s \ .$$

(9): Assume $Pow_2(p)$, $p > 1$, and $p = q^2$. We have $Pow_2(q)$ and $p > q$ by 8.1.5(5), $Pow_2(2{\cdot}q)$ by 8.1.3(2), and also

$$ps(p) \overset{8.1.7(7)}{=} p + ps(q) = 0{\cdot}2{\cdot}p + p + ps(q)$$

where

$$ps(q) \overset{8.1.7(4)}{<} 2{\cdot}q \overset{8.1.4(3)}{\le} p \ .$$

From (4) we thus obtain $Nbs(\overline{\log}(p), p, q_1)$ for some q_1 for which we have $q = q_1$ by 8.1.5(2). We thus have

$$\overline{\log}(p) = b{\cdot}p^2 + x{\cdot}p + y{\cdot}q + s \wedge x < p \wedge y < q \wedge q > s \wedge x = y\mathbf{0}$$

for some b, x, y, and s, since $p \neq 2{\cdot}q^2$ by 8.1.5(4). It must be the case that $b = 0$ by (6) and, since $p > y{\cdot}q + s$, we obtain $Cs(q, y{\cdot}q + s)$ from (4) by 8.1.8(6). Hence, $\overline{\log}(q) = y{\cdot}q + s$ by (8), from which we get $y = \overline{\log}(q) \div q$ and so

$$\overline{\log}(p) = x{\cdot}p + \overline{\log}(q) = y\mathbf{0}{\cdot}p + \overline{\log}(q) = (\overline{\log}(q) \div q)\mathbf{0}{\cdot}p + \overline{\log}(q)\ .$$

(10): This is proved similarly as (9).

8.1.10 Introduction of log **into PA.** We explicitly introduce into PA the logarithm function log by

$$\vdash_{\text{PAx}} \log(p) = \overline{\log}(p) \div p\ . \tag{1}$$

The function satisfies the following:

$$\vdash_{\text{PA}} \log(1) = 0 \tag{2}$$
$$\vdash_{\text{PA}} Pow_2(p) \wedge p = q^2 \wedge p > 1 \rightarrow \log(p) = \log(q)\mathbf{0} \tag{3}$$
$$\vdash_{\text{PA}} Pow_2(p) \wedge p = 2{\cdot}q^2 \rightarrow \log(p) = \log(q)\mathbf{1} \tag{4}$$
$$\vdash_{\text{PA}} Pow_2(p) \rightarrow \log(2{\cdot}p) = \log(p) + 1 \tag{5}$$
$$\vdash_{\text{PA}} Pow_2(p) \wedge Pow_2(q) \wedge p < q \rightarrow \log(p) < \log(q)\ . \tag{6}$$

(2): We have $\log(1) = \overline{\log}(1) \div 1 \overset{8.1.9(7)}{=} 0 \div 1 = 0$.

(3): Assume $Pow_2(p)$, $p = q^2$, and $p > 1$. We have $Pow_2(q)$ by 8.1.5(5) and by 8.1.9(9) we get $\overline{\log}(p) = (\overline{\log}(q) \div q)\mathbf{0}{\cdot}p + \overline{\log}(q)$. Since $\overline{\log}(q) < q^2 = p$ by 8.1.9(6) we have

$$\log(p) = \overline{\log}(p) \div p \overset{7.4.8(2)}{=} \log(q)\mathbf{0}\ .$$

(4): Assume $Pow_2(p)$ and $p = 2{\cdot}q^2$. We have $Pow_2(q)$ by 8.1.5(5) and by 8.1.9(10) we get $\overline{\log}(p) = (\overline{\log}(q) \div q)\mathbf{1}{\cdot}p + \overline{\log}(q)$. Since $\overline{\log}(q) < q^2 < p$ by 8.1.9(6) we have

$$\log(p) = \overline{\log}(p) \div p \overset{7.4.8(2)}{=} \log(q)\mathbf{1}\ .$$

(5): By complete induction on p. Assume $Pow_2(p)$ and obtain $Pow_2(2{\cdot}p)$ by 8.1.3(2). We now consider three cases implied by 8.1.5(1). If $p = 1$ then we have

$$\log(2{\cdot}p) = \log(2{\cdot}1) = \log(2{\cdot}1^2) \overset{(4)}{=} 2{\cdot}\log(1) + 1 \overset{(2)}{=} \log(1) + 1 = \log(p) + 1\ .$$

If $p > 1$ and $p = q^2$ for some q such that $Pow_2(q)$ then we have

$$\log(2{\cdot}p) = \log(2{\cdot}q^2) \overset{(4)}{=} 2{\cdot}\log(q) + 1 \overset{(3)}{=} \log(q^2) + 1 = \log(p) + 1 \ .$$

If $p = 2{\cdot}q^2$ for some $q < p$ such that $Pow_2(q)$ then $p > 1$ and we have

$$\log(2{\cdot}p) = \log(2{\cdot}(2{\cdot}q^2)) = \log((2{\cdot}q)^2) \overset{(3)}{=} 2{\cdot}\log(2{\cdot}q) \overset{\text{IH}}{=} 2{\cdot}(\log(q) + 1) =$$
$$(2{\cdot}\log(q) + 1) + 1 \overset{(4)}{=} \log(2{\cdot}q^2) + 1 = \log(p) + 1 \ .$$

(6): By complete induction on q. Assume $Pow_2(p)$, $Pow_2(q)$, and $p < q$. We have $q = 2{\cdot}u{\cdot}p$ for some u by 8.1.4(1) and, since $Pow_2(u{\cdot}p)$ by 8.1.3(2), we obtain $p \le u{\cdot}p < q$ by 8.1.4(2). Thus either $p < u{\cdot}p$ and then $\log(p) < \log(u{\cdot}p)$ by IH, or else $p = u{\cdot}p$ and then $\log(p) = \log(u{\cdot}p)$. In either case we have

$$\log(p) \le \log(u{\cdot}p) < \log(u{\cdot}p) + 1 \overset{(5)}{=} \log(2{\cdot}u{\cdot}p) = \log(q) \ .$$

8.1.11 Introduction of 2^x into PA. We are now ready to introduce the exponentiation function by minimalization:

$$\vdash_{\text{PAx}} 2^x = \mu_p[Pow_2(p) \land \log(p) = x]$$

whose existence condition:

$$\vdash_{\text{PA}} \exists p(Pow_2(p) \land \log(p) = x)$$

is proved by induction on x. In the base case it suffices to take $p := 1$ because of 8.1.3(3) and 8.1.10(2). In the inductive case we obtain a p s.t. $Pow_2(p)$ and $\log(p) = x$ by IH. We then get $Pow_2(2{\cdot}p)$ by 8.1.3(2), $\log(2{\cdot}p) = \log(p) + 1 = x + 1$ by 8.1.10(5), and so it suffices to take $p := 2{\cdot}p$.

The following property asserts that the functions 2^x and log are inverse:

$$\vdash_{\text{PA}} 2^x = p \leftrightarrow Pow_2(p) \land \log(p) = x \ . \tag{1}$$

In the direction ($\rightarrow$) assume $2^x = p$ and from the first conjunct of the defining axiom for 2^x: $Pow_2(2^x) \land \log(2^x) = x$ we immediately obtain $Pow_2(p)$ and $\log(p) = x$.

In the direction ($\leftarrow$) assume $Pow_2(p)$ and $\log(p) = x$ and obtain $Pow_2(2^x)$ as well as $\log(2^x) = x$ from the defining axiom for 2^x. Consider three cases by trichotomy. If $2^x < p$ we obtain a contradiction $x = \log(2^x) < \log(p) = x$ by 8.1.10(6). We similarly derive a contradiction for the case $2^x > p$. This means that the third case $2^x = p$ must hold.

We are now ready to prove the two recurrences for the exponentiation function from Par. 8.1.2. Property 8.1.2(1) follows from (1) by 8.1.3(3) and 8.1.10(2).

Property 8.1.2(2) is proved as follows. From $2^x = 2^x$ we get $Pow_2(2^x)$ and $\log(2^x) = x$ by (1). We get $\log(2{\cdot}2^x) = \log(2^x) + 1 = x + 1$ by 8.1.10(5) and, since $Pow_2(2{\cdot}2^x)$ by 8.1.3(2), we get $2^{x+1} = 2{\cdot}2^x$ by (1).

8.1.12 Some properties of 2^x. We list some of the properties of the exponential function:

$$\vdash_{\text{PA}} 2^0 = 1 \tag{1}$$
$$\vdash_{\text{PA}} 2^{x+1} = 2{\cdot}2^x \tag{2}$$
$$\vdash_{\text{PA}} Pow_2(p) \leftrightarrow \exists x\, 2^x = p \tag{3}$$
$$\vdash_{\text{PA}} x < 2^x \tag{4}$$
$$\vdash_{\text{PA}} x < y \to 2^x < 2^y \tag{5}$$
$$\vdash_{\text{PA}} 2^{x+y} = 2^x{\cdot}2^y\ . \tag{6}$$

Properties (1) and (2) are just the properties 8.1.2(1)(2) proved in Par. 8.1.11.

(3): Follows directly from 8.1.11(1) because $\exists x \log(p) = x$ holds trivially.

(4): By induction on x. In the base case we have $0 < 1 = 2^0$. In the inductive case we have $x < 2^x$ by IH from which we get $x+1 \leq 2^x < 2^{x+1}$.

(5): Assume $x < y$. By substituting $p := 2^x$ in 8.1.11(1) we obtain $Pow_2(2^x)$ and $\log(2^x) = x$. We get $Pow_2(2^y)$ and $\log(2^y) = y$ similarly. We thus have $\neg \log(2^y) < \log(2^x)$ and we obtain $\neg 2^y < 2^x$, i.e. $2^x \leq 2^y$, by 8.1.10(6). If it were the case that $2^x = 2^y$ then we would het a contradiction $x = \log(2^x) = \log(2^y) = y$.

(6): By induction on x. In the base case we have $2^{0+y} = 2^y = 1{\cdot}2^y = 2^0{\cdot}2^y$. In the inductive case we have

$$2^{x'+y} = 2^{(x+y)'} = 2{\cdot}2^{x+y} \overset{\text{IH}}{=} 2{\cdot}2^x{\cdot}2^y = 2^{x'}{\cdot}2^y\ .$$

8.2 Primitive Recursion

8.2.1 Primitive recursive functions. The class of *primitive recursive* functions can be characterized as the smallest class obtained by explicit definitions:

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

and by *primitive recursion*:

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

where the term τ is formed from the constant 0, variables among the n ($n \geq 0$) variables of $\vec{x}$, applications of the successor function ρ', and of applications of previously introduced functions. In the scheme of primitive recursion the functions g and h are previously introduced functions, g is n-ary ($n \geq 0$) and h is $(n+2)$-ary.

We already know that we can introduce into PA explicitly defined functions. In this section we show that PA is *closed* under primitive recursion, i.e. if functions g and h have been introduced into PA then we can introduce into PA by minimalization a function f to satisfy the above recurrences. The reader will note that by showing the closure of PA under primitive recursion will have proved more than that the primitive recursive functions can be introduced into PA. This is because we will be able to introduce into PA the function f defined by primitive recursion from any two functions g and h already in PA even if one or both of them will not be primitive recursive.

8.2.2 Bounded indexing function. For the introduction of primitive recursive functions into PA we will need to recover digits of numbers represented in the notation with the base 2^k for for a given $k \geq 1$. As is well-known, any number x can be uniquely written in such a representation as $x = \sum_i d_i{\cdot}2^{k\cdot i}$. We can recover the i-th digit d_i of x by a ternary *bounded indexing function* $d_i = (x)_i^{[k]}$ which can be explicitly introduced into PA by:

$$\vdash_{\mathrm{PAx}} (s)_i^{[k]} = s \div 2^{k\cdot i} \bmod 2^k \ . \tag{1}$$

We will need the following properties of the bounded indexing function:

$$\vdash_{\mathrm{PA}} \ b < 2^k \land t < 2^{k\cdot i} \to (a{\cdot}2^{k\cdot i'} + b{\cdot}2^{k\cdot i} + t)_i^{[k]} = b \tag{2}$$

$$\vdash_{\mathrm{PA}} \ i < j \land s < 2^{k\cdot j} \to (a{\cdot}2^{k\cdot j}{+}s)_i^{[k]} = (s)_i^{[k]} \ . \tag{3}$$

(2): Assume the antecedent of the property. Since

$$a{\cdot}2^{k\cdot i'} + b{\cdot}2^{k\cdot i} + t \overset{8.1.12(6)}{=} (a{\cdot}2^k + b){\cdot}2^{k\cdot i} + t \ ,$$

we have

$$(a{\cdot}2^{k\cdot i'}{+}b{\cdot}2^{k\cdot i}{+}t)_i^{[k]} = (a{\cdot}2^{k\cdot i'}{+}b{\cdot}2^{k\cdot i}{+}t){\div}2^{k\cdot i} \bmod 2^k = (a{\cdot}2^k{+}b) \bmod 2^k = b \ .$$

(3): Assume $s < 2^{k\cdot j}$ and $i < j$, i.e. $j = i + n'$ for some n. We have $s = b{\cdot}2^{k\cdot i} + t$ for some $b < 2^k$ and $t < 2^{k\cdot i}$. We then obtain

$$(a{\cdot}2^{k\cdot j}{+}s)_i^{[k]} \overset{8.1.12(6)}{=} ((a{\cdot}2^{k\cdot n}){\cdot}2^{k\cdot i'} + b{\cdot}2^{k\cdot i} + t)_i^{[k]} \overset{(2)}{=} b \overset{(2)}{=}$$
$$(0{\cdot}2^{k\cdot i'} + b{\cdot}2^{k\cdot i} + t)_i^{[k]} = (s)_i^{[k]} \ .$$

8.2.3 Extensions by primitive recursion. Let T be a proper extension of PA and $\tau_1[\vec{y}]$, $\tau_2[x,\vec{y},v]$ terms of $\mathcal{L}_T$ with free variables among the indicated ones and where $\vec{y}$ is an n-tuple of variables ($n \geq 0$). The extension of the theory T into S with a new $(n+1)$-ary function symbol f and with the axioms universal closures of

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

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

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

is called *extension by primitive recursion.* The formula $\phi[x, \vec{y}, z]$, which is of $\mathcal{L}_T$ and with all free variables indicated, will be effectively determined in Par. 8.2.6 as a graph of the function f. The formula is used in the induction axiom (3) which is the sole induction axiom of S containing the function symbol f.

We keep the notation introduced in this paragraph fixed until the end of the section where we prove in Thm. 8.2.7 that S is an extension by definition of T.

8.2.4 The outline of extensions. Pursuing our plan of introducing the function f into the theory S we first extend T to T_1 by definitions to contain the exponentiation function 2^x and the bounded indexing function $(s)_i^{[k]}$ if not already in T.

We plan to extend T_1 to T_2 by explicitly defining a a predicate Fs such that when f will be introduced into S we will have

$$Fs(k, s, x, \vec{y}) \leftrightarrow \exists k(s \bmod 2^{k \cdot x'} = \sum_{i \leq x} f(i, \vec{y}) \cdot 2^{k \cdot i} \land \forall (i \leq x \rightarrow f(i, \vec{y}) < 2^k))$$

in the standard model of S. The third argument s of Fs is a course of values sequence for the computation of f coded in the base 2^k representation.

The function f can be then introduced into S_1 by implicit definition which is equivalent to

$$S_1 \vdash \exists k \exists s(Fs(k, s, x, \vec{y}) \land (s)_x^{[k]} = f(x, \vec{y})) \ . \qquad (1)$$

8.2.5 Course of values sequences for f. We extend T_1 to T_2 by an explicit definition of an $(n+3)$-ary predicate Fs:

$$T_2 \vdash Fs(k, s, x, \vec{y}) \leftrightarrow (s)_0^{[k]} = \tau_1[\vec{y}] \land \forall i(i < x \rightarrow (s)_{i+1}^{[k]} = \tau_2[i, \vec{y}, (s)_i^{[k]}]) \ . \qquad (1)$$

Clearly, $Fs(k, s, x, \vec{y})$ holds in the standard model of T_2 iff s codes in the 2^k-representation the course of values sequence for $f(x, \vec{y})$ provided $f(i, \vec{y}) < 2^k$ holds for all $i \leq x$.

The following properties of Fs express the fact that course of values sequences can be constructed for all arguments x, $\vec{y}$:

$$T_2 \vdash \tau_1[\vec{y}] < 2^k \rightarrow Fs(k, \tau_1[\vec{y}], 0, \vec{y}) \wedge (\tau_1[\vec{y}])_0^{[k]} = \tau_1[\vec{y}] \tag{2}$$

$$T_2 \vdash Fs(k, s, x, \vec{y}) \wedge s{<}2^{k\cdot x'} \wedge a = \tau_2[x, \vec{y}, (s)_x^{[k]}]{<}2^k \rightarrow Fs(k, a{\cdot}2^{k\cdot x'}{+}s, x', \vec{y}) \tag{3}$$

$$T_2 \vdash Fs(k, a{\cdot}2^{k\cdot x'}{+}s, x', \vec{y}) \wedge s < 2^{k\cdot x'} \rightarrow Fs(k, s, x, \vec{y}) \ . \tag{4}$$

$$T_2 \vdash \forall s_1 \forall s_2 \forall i (Fs(k_1, s_1, x, \vec{y}) \wedge Fs(k_2, s_2, x, \vec{y}) \wedge i \leq x \rightarrow (s_1)_i^{[k_1]} = (s_2)_i^{[k_2]}) \tag{5}$$

$$T_2 \vdash \forall k_1 \forall k_2 (\exists s Fs(k_1, s, x, \vec{y}) \wedge k_1{\leq}k_2 \rightarrow \exists s(s{<}2^{k_2\cdot x'} \wedge Fs(k_2, s, x, \vec{y}))) \ . \tag{6}$$

(2): If $\tau_1[\vec{y}] < 2^k$ then, since $\tau_1[\vec{y}] = 0{\cdot}2^{k\cdot 1} + \tau_1[\vec{y}]{\cdot}2^{k\cdot 0} + 0$, we have $(\tau_1[\vec{y}])_0^{[k]} = \tau_1[\vec{y}]$ by 8.2.2(2) and so $Fs(k, \tau_1[\vec{y}], 0, \vec{y})$ holds by (1).

(3): Assume the antecedent of the property and set $t = a{\cdot}2^{k\cdot x'}{+}s$. Since, $0 < x'$, we have

$$(t)_0^{[k]} \overset{8.2.2(3)}{=} (s)_0^{[k]} \overset{Fs(k,s,x,\vec{y})}{=} \tau_1[\vec{y}] \ .$$

For $i = x < x'$ we have

$$(t)_{i+1}^{[k]} \overset{8.2.2(2)}{=} a = \tau_2[i, \vec{y}, (s)_i^{[k]}] \overset{8.2.2(3)}{=} \tau_2[i, \vec{y}, (t)_i^{[k]}]$$

and for any $i < x < x'$ we have

$$(t)_{i+1}^{[k]} \overset{8.2.2(3)}{=} (s)_{i+1}^{[k]} \overset{Fs(k,s,x,\vec{y})}{=} \tau_2[i, \vec{y}, (s)_i^{[k]}] \overset{8.2.2(3)}{=} \tau_2[i, \vec{y}, (t)_i^{[k]}] \ .$$

(4): For $t = a{\cdot}2^{k\cdot x'}{+}s$ and $s < 2^{k\cdot x'}$ assume $Fs(k, t, x', \vec{y})$. Since $0 < x'$, we have

$$(s)_0^{[k]} \overset{8.2.2(3)}{=} (t)_0^{[k]} \overset{Fs(k,t,x',\vec{y})}{=} \tau_1[\vec{y}]$$

and for any $i < x < x'$ we have

$$(s)_{i+1}^{[k]} \overset{8.2.2(3)}{=} (t)_{i+1}^{[k]} \overset{Fs(k,t,x',\vec{y})}{=} \tau_2[i, \vec{y}, (t)_i^{[k]}] \overset{8.2.2(3)}{=} \tau_2[i, \vec{y}, (s)_i^{[k]}] \ .$$

But this means that $Fs(k, s, x, \vec{y})$ holds.

(5): By induction on x. In the base case take any s_1, s_2, and $i \leq 0$, i.e. $i = 0$, s.t. $Fs(k_1, s_1, 0, \vec{y})$, and $Fs(k_2, s_2, 0, \vec{y})$ holds. We then have

$$(s_1)_0^{[k_1]} \overset{Fs(k_1,s_1,0,\vec{y})}{=} \tau_1[\vec{y}] \overset{Fs(k_2,s_2,0,\vec{y})}{=} (s_2)_0^{[k_2]} \ .$$

In the inductive case take any s_1, s_2, and $i < x'$ s.t. $Fs(k_1, s_1, x', \vec{y})$, and $Fs(k_2, s_2, x', \vec{y})$. We have $s_1 = a_1{\cdot}2^{k_1\cdot x'} + t_1$, $t_1 < 2^{k_1\cdot x'}$ and $s_2 = a_2{\cdot}2^{k_2\cdot x'} + t_2$, $t_2 < 2^{k_2\cdot x'}$ for some a_1, a_2, t_1 and t_2. We then get $Fs(k_1, t_1, x, \vec{y})$ and $Fs(k_2, t_2, x, \vec{y})$ by (4). We now consider two cases for i. If $i \leq x < x'$ then

$$(s_1)_i^{[k_1]} \overset{8.2.2(3)}{=} (t_1)_i^{[k_1]} \overset{\text{IH}}{=} (t_2)_i^{[k_2]} \overset{8.2.2(3)}{=} (s_2)_i^{[k_2]} \ .$$

If $i = x'$ then we have

$$(s_1)^{[k_1]}_{x+1} \overset{Fs(k_1,s_1,x',\vec{y})}{=} \tau_2[x,\vec{y},(s_1)^{[k_1]}_x] \overset{\text{IH}}{=} \tau_2[x,\vec{y},(s_2)^{[k_2]}_x] \overset{Fs(k_2,s_2,x',\vec{y})}{=} (s_2)^{[k_2]}_i \ .$$

(6): By induction on x. In the base case take any k_1, k_2 s.t. $k_1 \le k_2$ and $Fs(k_1,s_1,0,\vec{y})$ for some s_1. We then have

$$2^k_2 \overset{8.1.12(5)}{\ge} 2^k_1 > (s_1)^{[k_1]}_0 = \tau_1[\vec{y}]$$

and for $s_2 = \tau_1[\vec{y}]$ we obtain $Fs(k_2,s_2,0,\vec{y})$ by (2).

In the inductive case take any k_1, k_2 s.t. $k_1 \le k_2$ and $Fs(k_1,s_1,x',\vec{y})$ for some s_1. We have $s_1 = a \cdot 2^{k_1 \cdot x'} + t_1$ for some a and $t_1 < 2^{k_1 \cdot x'}$. Thus $Fs(k_1,t_1,x,\vec{y})$ by (4) and $Fs(k_2,t,x,\vec{y})$ for some $t < 2^{k_2 \cdot x'}$ by IH. We have

$$2^{k_2} \overset{8.1.12(5)}{\ge} 2^k_1 > (s_1)^{[k_1]}_{x+1} \overset{Fs(k_1,s_1,x',\vec{y})}{=} \tau_2[x,\vec{y},(s_1)^{[k_1]}_x] \overset{8.2.2(3)}{=}$$
$$\tau_2[x,\vec{y},(t_1)^{[k_1]}_x] \overset{(5)}{=} \tau_2[x,\vec{y},(t)^{[k_2]}_x] \ .$$

We set $s := \tau_2[x,\vec{y},(t)^{[k_2]}_x] \cdot 2^{k_2 \cdot x'} + t$ and obtain $Fs(k_2,s,x',\vec{y})$ by (3) together with:

$$s = \tau_2[x,\vec{y},(t)^{[k_2]}_x] \cdot 2^{k_2 \cdot x'} + t < \tau_2[x,\vec{y},(t)^{[k_2]}_x] \cdot 2^{k_2 \cdot x'} + 2^{k_2 \cdot x'} =$$
$$(\tau_2[x,\vec{y},(t)^{[k_2]}_x] + 1) \cdot 2^{k_2 \cdot x'} \le 2^{k_2} \cdot 2^{k_2 \cdot x'} \overset{8.1.12(6)}{=} 2^{k_2 \cdot x''} \ .$$

8.2.6 The graph ϕ of the function f. We could now extend T_2 to S_1 by the implicit definition of f:

$$S_1 \vdash \exists k \exists s (Fs(k,s,x,\vec{y}) \land (s)^{[k]}_x = f(x,\vec{y})) \tag{1}$$

but we will instead equivalently extend T in Thm. 8.2.7 with the help of a formula $\phi[x,\vec{y},z]$ of $\mathcal{L}_T$ which is a graph of f. Since T_2 is an extension by definitions of T, the formula ϕ is effectively obtained by translation from the formula

$$\exists k \exists s (Fs(k,s,x,\vec{y}) \land (s)^{[k]}_x = z)$$

of T_2 in such a way that we have

$$T_2 \vdash \phi[x,\vec{y},z] \leftrightarrow \exists k \exists s (Fs(k,s,x,\vec{y}) \land (s)^{[k]}_x = z) \ . \tag{2}$$

by the Theorem on Extensions by definitions 6.6.2. We will need in the proof of Thm. 8.2.7 the following properties of the formula:

$$T \vdash \phi[0,\vec{y},\tau_1[\vec{y}]] \tag{3}$$
$$T \vdash \phi[x,\vec{y},z] \to \phi[x',\vec{y},\tau_2[x,\vec{y},z]] \tag{4}$$
$$T \vdash \exists z \phi[x,\vec{y},z] \tag{5}$$
$$T \vdash \phi[x,\vec{y},z_1] \land \phi[x,\vec{y},z_2] \to z_1 = z_2 \ . \tag{6}$$

In the following proofs we work in T_2 and use the equivalence (2) without explicitly referring to it. Properties (3) through (6) are thus derived in T_2 but, since they all are in the language $\mathcal{L}_T$, they are also theorems of T because T_2 is conservative over T.

(3): We set $k = \tau_1[\vec{y}]$. Since $\tau_1[\vec{y}] < 2^k$ by 8.1.12(4), we get $Fs(k, \tau_1[\vec{y}], 0, \vec{y})$ and $(\tau_1[\vec{y}])_0^{[k]} = \tau_1[\vec{y}]$ by 8.2.5(2). We thus have $\phi[0, \vec{y}, \tau_1[\vec{y}]]$.

(4): We assume $\phi[x, \vec{y}, z]$, i.e. $Fs(k_1, t_1, x, \vec{y})$ and $(t_1)_x^{[k_1]} = z$ for some k_1 and t_1. We set $k := \max(k_1, \tau_2[x, \vec{y}, z])$ and, since $k_1 \leq k$, we obtain a $t < 2^{k \cdot x'}$ s.t. $Fs(k, t, x, \vec{y})$ by 8.2.5(6). We have $z = (t_1)_x^{[k_1]} = (t)_x^{[k]}$ by 8.2.5(5) and so

$$2^k \overset{8.1.12(5)}{\geq} 2^{\tau_2[x,\vec{y},z]} \overset{8.1.12(4)}{>} \tau_2[x, \vec{y}, z] = \tau_2[x, \vec{y}, (t)_x^{[k]}] \ .$$

Thus by setting $s := \tau_2[x, \vec{y}, (t)_x^{[k]}] \cdot 2^{k \cdot x'} + t$, we obtain $Fs(k, s, x', \vec{y})$ by 8.2.5(3) and $(s)_{x'}^{[k]} = \tau_2[x, \vec{y}, (t)_x^{[k]}] = \tau_2[x, \vec{y}, z]$ by 8.2.2(2). Hence $\phi[x', \vec{y}, \tau_2[x, \vec{y}, z]]$.

(5): By induction on x. The base case is implied by (3). In the inductive case we have $\phi[x, \vec{y}, z]$ for some z by IH, we get $\phi[x', \vec{y}, \tau_2[x, \vec{y}, z]]$ by (4), and it suffices to set $z := \tau_2[x, \vec{y}, z]$.

(6): We assume $\phi[x, \vec{y}, z_1]$ and $\phi[x, \vec{y}, z_2]$, i.e. $Fs(k_1, s_1, x, \vec{y})$, $(s_1)_x^{[k_1]} = z_1$, $Fs(k_2, s_2, x, \vec{y})$, and $(s_2)_x^{[k_2]} = z_2$ for some k_1, k_2, s_1, s_2. We then obtain

$$z_1 = (s_1)_x^{[k_1]} \overset{8.2.5(5)}{=} (s_2)_x^{[k_2]} = z_2 \ .$$

8.2.7 Theorem (Extensions by primitive recursion). *If T is a proper extension of PA then an extension of T by primitive recursion is an extension by definition.*

Proof. Let S be an extension of T by primitive recursion as in Par. 8.2.3 and S_1 an extension of T by implicit definition with the defining axiom an universal closure of $\phi[x, \vec{y}, f(x, \vec{x})]$. We have $\mathcal{L}_{S_1} = \mathcal{L}_S$ and S_1 is an extension by definition of T by Thm. 6.6.3 because T proves the existence 8.2.6(5) and uniqueness 8.2.6(6) conditions for f. Clearly

$$S_1 \vdash \phi[x, \vec{y}, f(x, \vec{y})] \ . \tag{1}$$

In order to prove the theorem it suffices to prove that the theories S and S_1 are equivalent.

We prove first $S_1 \vdash S$. First of all, S_1 proves 8.2.3(1), i.e. $S_1 \vdash f(0, \vec{y}) = \tau_1[\vec{y}]$, because we have $\phi[0, \vec{y}, f(0, \vec{y})]$ by (1) and $\phi[0, \vec{y}, \tau_1[\vec{y}]]$ by 8.2.6(3). Thus $f(0, \vec{y}) = \tau_1[\vec{y}]$ by 8.2.6(6).

Secondly, S_1 proves 8.2.3(2), i.e. $S_1 \vdash f(x', \vec{y}) = \tau_2[x, \vec{y}, f(x, \vec{y})]$, because we have $\phi[x, \vec{y}, f(x, \vec{y})]$ and $\phi[x', \vec{y}, f(x', \vec{y})]$, by (1) and $\phi[x', \vec{y}, \tau[x, \vec{y}, f(x, \vec{y})]]$ by 8.2.6(4). Hence $f(x', \vec{y}) = \tau[x, \vec{y}, f(x, \vec{y})]$ by 8.2.6(6).

Finally, since S_1 is proper, it also proves the induction axiom 8.2.3(3) of S: $I_x \phi[x, \vec{y}, f(x, \vec{y})]$.

Vice versa, in order to prove $S \vdash S_1$ it suffices to show $S \vdash \phi[x, \vec{y}, f(x, \vec{y})]$. This is done by working in S and by using the induction axiom 8.2.3(3) of S. In the base case we have $\phi[0, \vec{y}, \tau_1[\vec{y}]]$ by 8.2.6(3) and hence $\phi[0, \vec{y}, f(0, \vec{y})]$ by 8.2.3(1). In the inductive case we have $\phi[x, \vec{y}, f(x, \vec{y})]$ by IH, from which we get $\phi[x', \vec{y}, \tau_2[x, \vec{y}, f(x, \vec{y})]]$ by 8.2.6(4), and hence $\phi[x', \vec{y}, f(x', \vec{y})]$ by 8.2.3(2). □

8.3 Suitable Pairing Function

Our main task in this section will be to introduce the suitable pairing function from Par. 1.3.11 into PA and prove its properties 1.3.7(1) through 1.3.7(3) as theorems.

8.3.1 Dyadic size function $|x|_d$. We will introduce our suitable pairing function by arithmetizing the binary trees from Fig. 1.2 in the dyading notation.

Toward that end we introduce into PA the dyadic size function (see Par. 1.3.4) by minimalization:

$$\vdash_{\mathrm{PAx}} |x|_d = \mu_n[x + 1 < 2^{n+1}] \tag{1}$$

whose existence condition $\vdash_{\mathrm{PA}} \exists n\, x + 1 < n + 1$ is proved by taking $n := x$ and using 8.1.12(4). The defining axiom for the dyadic size function implies:

$$\vdash_{\mathrm{PA}} x + 1 < 2^{|x|_d+1} \tag{2}$$

$$\vdash_{\mathrm{PA}} n < |x|_d \to 2^{n+1} \leq x + 1 \tag{3}$$

and the function satisfies:

$$\vdash_{\mathrm{PA}} |x|_d > 0 \leftrightarrow x > 0 \tag{4}$$

$$\vdash_{\mathrm{PA}} |x|_d = n \leftrightarrow 2^n \leq x + 1 < 2^{n+1} \; . \tag{5}$$

(4): If $0 < |x|_d$ then $2^1 \leq x{+}1$ by (3) and so $1 \leq x$. Vice versa, if $0 < x$ then we cannot have $|x|_d = 0$ because we would then get $2 \leq x + 1 \overset{(2)}{<} 2^{0+1} = 2$.

(5): In the direction ($\to$) assume $|x|_d = n$ and consider two cases. If $n = 0$ then $x = 0$ by (4) and we have $2^0 = 1 \leq 0 + 1 < 2 = 2^{0+1}$. If $n > 0$ then

$$2^n = 2^{n \dot- 1+1} \overset{(3)}{\leq} x + 1 \overset{(2)}{<} 2^{n+1} \; .$$

In the direction ($\to$) assume $2^n \leq x + 1 < 2^{n+1}$ and consider three cases. If $n < |x|_d$ then we get a contradiction

$$2^{n+1} \overset{(3)}{\leq} x + 1 < 2^{n+1} \; .$$

If $n = |x|_d$ there is nothing to prove. If $|x|_d < n$ then $|x|_d + 1 \leq n$ and we get a contradiction:

$$x + 1 \overset{(2)}{<} 2^{|x|_d+1} \overset{8.1.12(5)}{\leq} 2^n \leq x + 1 \; .$$

8.3.2 Dyadic concatenation function $x \star y$. We will also need the dyadic concatenation function $x \star y$ (see Par. 1.3.5) which is explicitly introduced into PA as follows:

$$\vdash_{\text{PAx}} x \star y = x{\cdot}2^{|y|_d} + y \tag{1}$$

The concatenation function satisfies the following:

$$\vdash_{\text{PA}} x = 0 \star x \land x = x \star 0 \tag{2}$$
$$\vdash_{\text{PA}} |x \star y|_d = |x|_d + |y|_d \tag{3}$$
$$\vdash_{\text{PA}} (x \star y) \star z = x \star (y \star z) \tag{4}$$
$$\vdash_{\text{PA}} n \le |x|_d \to \exists a \exists b (x = a \star b \land |b|_d = n) \tag{5}$$
$$\vdash_{\text{PA}} a \star b = c \star d \land |b|_d = |d|_d \to a = c \land b = d \;. \tag{6}$$

(2): We have $0 \star x = 0{\cdot}2^{|x|_d} + x = x$ and $x \star 0 = x{\cdot}2^{|0|_d} + 0 \overset{8.3.1(4)}{=} x{\cdot}2^0 + 0 = x$.

(3): We have $2^{|x|_d} \le x + 1$ by 8.3.1(5). Since $2^{|y|_d} > 0$, we get

$$2^{|x|_d + |x|_d} = 2^{|x|_d}{\cdot}2^{|y|_d} \le x{\cdot}2^{|y|_d} + 2^{|y|_d} \overset{8.3.1(5)}{\le} x{\cdot}2^{|y|_d} + y + 1 = (x \star y) + 1 \;.$$

We have $x + 1 < 2^{|x|_d + 1}$, i.e. $x + 2 \le 2{\cdot}2^{|x|_d}$, by 8.3.1(5). Since $2^{|y|_d} > 0$, we get

$$(x \star y) + 1 = x{\cdot}2^{|y|_d} + y + 1 \overset{8.3.1(5)}{<} x{\cdot}2^{|y|_d} + 2{\cdot}2^{|y|_d} \le 2{\cdot}2^{|x|_d}{\cdot}2^{|y|_d} = 2^{|x|_d + |x|_d + 1} \;.$$

Combining the two inequalities we obtain $|x \star y|_d = |x|_d + |x|_d$ by 8.3.1(5).

(4): We have

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

(5): Assume $n \le |x|_d$. We have

$$2^n \overset{8.1.12(5)}{\le} 2^{|x|_d} \overset{8.3.1(5)}{\le} x + 1$$

and for $a = (x + 1 \mathbin{\dot-} 2^n) \div 2^n$, $c = (x + 1 \mathbin{\dot-} 2^n) \bmod 2^n$ we get $x + 1 \mathbin{\dot-} 2^n = a{\cdot}2^n + c$, $c < 2^n$ by 7.4.8(1). For $b = 2^n \mathbin{\dot-} 1 + c$ we then get $x = a{\cdot}2^n + b$. We have $2^n \le 2^n + c < 2{\cdot}2^n$, i.e. $2^n \le b + 1 < 2^{n+1}$. Hence $|b|_d = n$ by 8.3.1(5) and thus $x = a \star b$.

(6): Assume $a \star b = c \star d$ and $|b|_d = |d|_d$. Thus

$$a{\cdot}2^{|d|_d} + b = a{\cdot}2^{|b|_d} + b = a \star b = c \star d = c{\cdot}2^{|d|_d} + d$$

and we get $a = c$, $b = d$ by 7.4.8(1)(2).

In the following we will use the associativity of the concatenation operation (4) without explicitly referring to it.

8.3.3 Counting function $\#(x)$. For the arithmetization of binary trees we will need a unary *counting* function $\#(x)$ yielding the number of digits **2** in the dyadic representation of x.

The counting function is introduced with the help of an auxiliary binary *dyadic indexing function* $[x]_i$ yielding the i-th dyadic digit in the dyadic representation of x decreased by one where the least significant digit is with the index 0. We intend to introduce the function into PA by contextual definition:

$$[x]_i = z \leftrightarrow \exists a \exists b(x = a \star (z+1) \star b \land z \leq 1 \land |b|_d = i) \lor i \geq |x|_d \land z = 0 \ . \tag{1}$$

Its existence condition follows from (2), (3) and its uniqueness condition from (4), (5):

$$\vdash_{\text{PA}} \ i < |x|_d \to \exists z \exists a \exists b(x = a \star (z+1) \star b \land z \leq 1 \land |b|_d = i) \tag{2}$$

$$\vdash_{\text{PA}} \ i \geq |x|_d \to \exists z\, z = 0 \tag{3}$$

$$\begin{aligned}\vdash_{\text{PA}} \ & x = a_1 \star (z_1+1) \star b_1 \land z_1 \leq 1 \land |b_1|_d = i \land \\ & x = a_2 \star (z_2+1) \star b_2 \land z_1 \leq 1 \land |b_2|_d = i \to i < |x|_d \land z_1 = z_2 \end{aligned} \tag{4}$$

$$\vdash_{\text{PA}} \ z_1 = 0 \land z_2 = 0 \to z_1 = z_2 \ . \tag{5}$$

(2): Assume $i < |x|_d$ and get $x = c \star b$ for some b, c such that $|b|_d = i$ by 8.3.2(5). We have $|x|_d = |c|_d + i$ by 8.3.2(3). Thus $1 \leq |c|_d$ and we get $c = a \star y$ for some a, y such that $|y|_d = 1$ by 8.3.2(5) again. We have $2^1 \leq y+1 < 2^2$ and so $1 \leq y \leq 2$ by 8.3.1(5). Hence $y = z+1$ for a z s.t. $z \leq 1$.

(3): This is trivial.

(4): Assume the antecedent. We get $b_1 = b_2$, and $a_1 \star (z_1+1) = a_2 \star (z_2+1)$ by 8.3.2(6). Since $|z_1+1|_d = 1 = |z_2+1|_d$ by 8.3.1(5), we get $a_1 = a_2$ and $z_1 = z_2$ by 8.3.2(6) again. We have $|x|_d = |a_1|_d + 1 + i$ by 8.3.2(3) and so $i < |x|_d$.

(5): This is trivial.

We will need the following simple properties of the dyadic indexing function:

$$\vdash_{\text{PA}} \ i < |b|_d \to [a \star b]_i = [b]_i \tag{6}$$

$$\vdash_{\text{PA}} \ [a \star b]_{i+|b|_d} = [a]_i \tag{7}$$

(6): Assume $i < |b|_d$ and obtain $b = b_1 \star (z+1) \star b_2$ for some b_1, b_2, $z \leq 1$ such that $|b_2|_d = i$ by (2). Since also $a \star b = (a \star b_1) \star (z+1) \star b_2$, we obtain $[b]_i = z = [a \star b]_i$ by (1).

(7): Consider two cases. If $i \geq |a|_i$ then also $i + |b|_d \geq |a|_i + |b|_d \overset{8.3.2(3)}{=} |a \star b|_s$ and we have $[a \star b]_{i+|b|_d} = 0 = [a]_i$ by (1). If $i < |a|_i$ then also $i + |b|_d < |a \star b|$ and we have $a = a_1 \star (z+1) \star a_2$ for some a_1, a_2, $z \leq 1$ such that $|a_2|_d = i$ by (2). Since also $a \star b = a_1 \star (z+1) \star (a_2 \star b)$ with $|a_2 \star b|_d = i + |b|_d$, we have $[a \star b]_{i+|b|_d} = z = [a]_i$ by (1) again.

We also need an auxiliary binary function f which is introduced into PA by primitive recursion:

$$\vdash_{\mathrm{PAx}} f(0,x) = 0 \tag{8}$$

$$\vdash_{\mathrm{PAx}} f(i',x) = [x]_i + f(i,x)\ . \tag{9}$$

We will need the following properties of f:

$$\vdash_{\mathrm{PA}} \forall a \forall b (i \leq |b|_d \to f(i, a \star b) = f(i,b)) \tag{10}$$

$$\vdash_{\mathrm{PA}} \forall a \forall b\, f(i + |b|_d, a \star b) = f(i,a) + f(|b|_d, b)\ . \tag{11}$$

(10): By induction on i. In the base case we have $f(0, a \star b) = 0 = f(0,b)$. In the inductive case we assume $i + 1 \leq |b|_d$ and, since $i < |b|_d$, we obtain:

$$f(i+1, a\star b) = [a \star b]_i + f(i, a \star b) \overset{\mathrm{IH}}{=}$$
$$[a \star b]_i + f(i,b) \overset{(6)}{=} [b]_i + f(i,b) = f(i+1,b)\ .$$

(11): By induction on i. In the base case we have

$$f(0 + |b|_d, a \star b) = f(|b|_d, a \star b) \overset{(10)}{=} f(|b|_d, b) = f(0,a) + f(|b|_d, b)\ .$$

In the inductive case we have:

$$f(i + 1 + |b|_d, a \star b) = [a \star b]_{i+|b|_d} + f(i + |b|_d, a \star b) \overset{\mathrm{IH}}{=}$$
$$[a \star b]_{i+|b|_d} + f(i,a) + f(|b|_d, b) \overset{(7)}{=}$$
$$[a]_i + f(i,a) + f(|b|_d, b) = f(i+1, a) + f(|b|_d, b)\ .$$

We introduce the function $\#(x)$ into PA by explicit definition:

$$\vdash_{\mathrm{PAx}} \#(x) = f(|x|_d, x) \tag{12}$$

The counting function has the following properties which we will need below:

$$\vdash_{\mathrm{PA}} \#(0) = 0 \tag{13}$$

$$\vdash_{\mathrm{PA}} \#(1) = 0 \tag{14}$$

$$\vdash_{\mathrm{PA}} \#(2) = 1 \tag{15}$$

$$\vdash_{\mathrm{PA}} \#(a \star b) = \#(a) + \#(b)\ . \tag{16}$$

(13): $\#(0) = f(|0|_d, 0) \overset{8.3.1(4)}{=} f(0,0) = 0$.

(14): We have $\#(1) = f(|1|_d, 1) \overset{8.3.1(5)}{=} f(1,1) = [1]_0 + f(0,0) = [1]_0$ and, since $1 \overset{8.3.2(2)}{=} 0 \star (0+1) \star 0$, $|0|_d \overset{8.3.1(4)}{=} 0$, we get $[1]_0 = 1$ by (1).

(15): Similarly as (14).

(16): We have

$$\#(a \star b) = f(|a \star b|_d, a \star b) \overset{8.3.2(3)}{=} f(|a|_d + |b|_d, a \star b) \overset{(11)}{=}$$
$$f(|a|_d, a) + f(|b|_d, b) = \#(a) + \#(b)\ .$$

Prefix Codes

We will define a subset *Prf* of natural numbers which code in the dyadic notation the binary trees from Fig. 1.2 expressed in the prefix notation.

8.3.4 Prefix notation of pair numerals. Recall that every pair numeral is either 0 or it has a form (τ_1, τ_2) for pair numerals τ_1 and τ_2. The right parentheses and commas in pair numerals are superfluous in the sense that when we omit them we are still able to represent every natural number uniquely.

The sequences of words over the two element alphabet (and 0 obtained in this way from pair numerals represent the same numbers as pair numerals in the (Polish) *prefix* notation. Thus the number zero 0 is represented by the pair numeral 0 and the prefix notation 0. The number 1 is represented by the pair numeral $(0,0)$ and by the prefix notation (00. The number 2 is denoted by the pair numeral $(0,(0,0))$ and by the prefix notation (0(00. The number 3 is denoted by the pair numeral $((0,0),0)$ and by the prefix notation ((000, and so on.

The reader will note that every number x with the pair size $|x|_p = n$ is represented by the prefix word of length $2{\cdot}n + 1$ which contains exactly n left parentheses and $n+1$ zeroes. However, not every such word is a prefix notation. For instance out of $\binom{5}{2} = 10$ words of length 5 six words 0(00(, (00(0, 00(0(, 0(0(0 (000(, 000((, 00((0, and 0((00 are not prefix notations. It can be shown that there are

$$\frac{1}{2{\cdot}n+1}{\cdot}\binom{2{\cdot}n+1}{n} = \frac{1}{n+1}{\cdot}\binom{2{\cdot}n}{n}$$

prefix notations of numbers with the pair size n out of $\binom{2\cdot n+1}{n}$ possible words with n symbols (and $n+1$ symbols 0.

A moment of thought shows that the word w with n symbols (and $n+1$ symbols 0 is a prefix notation iff for all $w = w_1 w_2$ where w_2 is not empty the count i of symbols (in w_2 is less than the count j of 0. We clearly have $i < j$ iff the size of w_2 is greater than $2{\cdot}i$, i.e. iff $i + j > 2{\cdot}i$.

The plan for the introduction of the pairing function x, y is as follows. We will first arithmetize (code) the prefix notation in the dyadic representation in Par. 8.3.5. We will then define a pairing function over prefix codes in Par. 8.3.6. We will well-order the prefix codes by a relation $<_p$ defined in Par. 8.3.8. We will then enumerate the prefix codes by a function π in Par. 8.3.13 such that $<$ and $<_p$ will be isomorphic. The isomorphism then defines in Par. 8.3.14 the pairing function x, y as the isomorphic image of the pairing function on the prefix codes.

8.3.5 Prefix codes. We arithmetize the prefix notation by coding into natural numbers where in the dyadic representation the symbol (is coded by the digit 2 and the symbol 0 by the digit 1. The unary predicate $Prf(x)$ holding iff x codes the prefix code of length $|x|_d$ is introduced into PA by explicit definition:

$$\vdash_{\mathrm{PAx}} \ \mathit{Prf}(x) \leftrightarrow |x|_d = 2{\cdot}\#(x) + 1 \wedge \forall a \forall b(x = a \star b \wedge b > 0 \rightarrow |b|_d > 2{\cdot}\#(b)) \tag{1}$$

We have

$$\vdash_{\mathrm{PA}} \ |x|_d > 2{\cdot}\#(x) \rightarrow \exists v \exists b(x = v \star b \wedge \mathit{Prf}(v)) \tag{2}$$

$$\vdash_{\mathrm{PA}} \ \mathit{Prf}(a \star b) \wedge \mathit{Prf}(a) \rightarrow b = 0 \ . \tag{3}$$

$$\vdash_{\mathrm{PA}} \ \mathit{Prf}(x) \wedge \#(x) = 0 \leftrightarrow x = 1 \ . \tag{4}$$

(2): Assume $|x|_d > 2{\cdot}\#(x)$ and consider the formula

$$\phi[x, k] \equiv \exists v \exists b(|v|_d = k \wedge x = v \star b \wedge |v|_d > 2{\cdot}\#(v)) \ .$$

Since $x = x \star 0$ by 8.3.2(2), we have $\phi[x, |x|_d]$ for $v := x$ and $b := 0$. By the least number principle there is the smallest such k for which there are v and b such that $|v|_d = k$, $x = v \star b$,

$$|v|_d > 2{\cdot}\#(v) \ , \tag{5}$$

and for any $m < k$ we have $\neg\phi[x, m]$, i.e.

$$\forall v_1 \forall b_1(|v_1|_d < |v|_d \wedge x = v_1 \star b_1 \rightarrow |v_1|_d \leq 2{\cdot}\#(v_1)) \ . \tag{6}$$

In order to prove $\mathit{Prf}(v)$ we assume $v = v_1 \star v_2$ for some v_1, $v_2 > 0$. We have

$$|v_1|_d + |v_2|_d \overset{8.3.2(3)}{=} |v|_d \overset{(5)}{>} 2{\cdot}\#(v) \overset{8.3.3(16)}{=} 2{\cdot}\#(v_1) + 2{\cdot}\#(v_2) \tag{7}$$

and, since $x = v_1 \star (v_2 \star b)$, $|v_1|_d < |v|$, we obtain

$$|v_1|_d \leq 2{\cdot}\#(v_1) \tag{8}$$

by (6). This means that

$$|v_2|_d > 2{\cdot}\#(v_2) \ . \tag{9}$$

It remains to derive $|v|_d = 2{\cdot}\#(v) + 1$. We have $|v|_d > 0$ by (5) and so $v = v_1 \star v_2$ for some v_1, v_2 s.t. $|v_2|_d = 1$ by 8.3.2(5). We then get $\#(v_2) = 0$ by (9) and $|v_1|_d{+}1 > 2{\cdot}\#(v_1)$, i.e. $|v_1|_d \geq 2{\cdot}\#(v_1)$, by (7). Thus $|v_1|_d = 2{\cdot}\#(v_1)$ by (8) and hence $|v|_d = |v_1|_d + 1 = 2{\cdot}\#(v) + 1$.

(3): Assume $\mathit{Prf}(a \star b)$ and $\mathit{Prf}(a)$. If it were the case that $b > 0$ then we would have $|b|_d > 2{\cdot}\#(b)$ and we would obtain a contradiction

$$|a \star b|_d \overset{8.3.2(3)}{=} |a|_d + |b|_b = 2{\cdot}\#(a) + 1 + |b|_b > 2{\cdot}\#(a) + 1 + 2{\cdot}\#(b) =$$
$$2{\cdot}(\#(a) + \#(b)) + 1 \overset{8.3.3(16)}{=} 2{\cdot}\#(a \star b) + 1 \ .$$

(4): If $Prf(x)$ and $\#(x) = 0$ then $|x|_d = 1$ and we get $x = 1$ or $x = 2$ by 8.3.1(5) and, since $\#(2) = 1$ by 8.3.3(15), it must be the case that $x = 1$.

Vice versa, if $x = 1$ then we have

$$|1|_d \overset{8.3.1(5)}{=} 1 \overset{8.3.3(14)}{=} 2{\cdot}\#(1) + 1 \ .$$

Take any $b > 0$ and a such that $1 = x = a \star b$. We have $1 = |1|_d \overset{8.3.2(3)}{=} |a|_d + |b|_d$ and so $a = 0$ by 8.3.1(4). But then $b = 1$ by 8.3.2(2) and so

$$|b|_d = 1 > 0 \overset{8.3.3(14)}{=} 2{\cdot}\#(b) \ .$$

8.3.6 Pairing function over prefix codes. We define the binary *prefix code pairing* function $x \,,_p y$ by explicit definition:

$$\vdash_{\text{PAx}} \ x \,,_p y = 2 \star x \star y \ . \tag{1}$$

The basic properties of the function are that it is over prefix codes and that it satisfies there the pairing property:

$$\vdash_{\text{PA}} \ Prf(x) \land Prf(y) \to Prf(x \,,_p y) \land \#(x \,,_p y) = \#(x) + \#(y) + 1 \tag{2}$$

$$\vdash_{\text{PA}} \ Prf(x) \land Prf(v) \land x \,,_p y = v \,,_p w \to x = v \land y = w \ . \tag{3}$$

(2): Assume $Prf(x)$ and $Prf(y)$. We have

$$\#(x, \,,_p y) = \#(2 \star x \star y) = 1 + \#(x) + \#(y) \tag{4}$$

by the properties of the counting function and

$$|x \,,_p y|_d = |2 \star x \star y|_d = 1 + |x|_d + |y|_d = \\ 1 + 2{\cdot}\#(x) + 1 + 2{\cdot}\#(y) + 1 \overset{(4)}{=} 2{\cdot}\#(x \,,_p y) + 1 \tag{5}$$

by the properties of the dyadic size function. We now take any a and $b > 0$ such that $x \,,_p y = 2 \star x \star y = a \star b$ and, since $|b|_d \leq |x \,,_p y|_d$, we consider three cases for the dyadic size of b:

$|b|_d \leq |y|_d$: We have $y = y_1 \star y_2$ for some y_1, y_2 s.t. $|b|_d = |y_2|_d$ by 8.3.2(5), $y_2 = b$ by 8.3.2(6), and $|y_2|_d > 2{\cdot}\#(y_2)$, i.e. $|b|_d > 2{\cdot}\#(b)$, by $Prf(y)$. Hence $Prf(x \,,_p y)$.

$|y|_d < |b|_d \leq |x|_d + |y|_d$: We have $b = b_1 \star b_2$ for some b_1, b_2 s.t. $|y|_d = |b_2|_d$ by 8.3.2(5) and, since $2 \star x \star y = a \star b_1 \star b_2$, we get $2 \star x = a \star b_1$ and $y = b_2$ by 8.3.2(6). From $|y|_d < |b|_d = |b_1|_d + |b_2|_d \leq |x|_d + |y|_d$ we get $0 < |b_1|_d \leq |x|_d$ and so $b_1 > 0$, by 8.3.1(4). Thus $x = x_1 \star x_2$ for some x_1, x_2 s.t. $|x_2|_d = |b_1|_d$ by 8.3.2(5) and, since $2 \star x_1 \star x_2 = a \star b_1$, we get $x_2 = b_1$ by 8.3.2(6). Now, $|x_2|_d > 2{\cdot}\#(x_2)$ by $Prf(x)$. Hence

$$|b|_d = |b_1 \star b_2|_d = |b_1|_d + |b_2|_d = |x_2|_d + 2{\cdot}m + 1 > 2{\cdot}\#(x_2) + 2{\cdot}m = \\ 2{\cdot}(\#(b_1) + m) = 2{\cdot}(\#(b_1) + \#(b_2)) = 2{\cdot}\#(b)$$

and thus $\mathit{Prf}(x \,,_p\, y)$.

$|b|_d = 1+|x|_d+|y|_d$: We have $0 \star (x \,,_p\, y) \overset{8.3.2(2)}{=} x \,,_p\, y = a \star b$, $|x \,,_p\, y|_d = |b|_d$, and hence $x \,,_p\, y = b$ by 8.3.2(6). Thus

$$|b|_d = |x \,,_p\, y|_d \overset{(5)}{=} 2{\cdot}(1 + \#(x) + \#(y)) + 1 > 2{\cdot}(1 + \#(x) + \#(y)) \overset{(4)}{=} \\ 2{\cdot}\#(x \,,_p\, y) = 2{\cdot}\#(b)$$

and so $\mathit{Prf}(x \,,_p\, y)$ again.

(3): Assume $\mathit{Prf}(x)$, $\mathit{Prf}(v)$, and $x \,,_p\, y = v \,,_p\, w$. From $2 \star x \star y = 2 \star v \star w$ we get $|x \star y|_d = |v \star w|_d$ by 8.3.2(3) and then $x \star y = v \star w$ by 8.3.2(6). We now consider two cases. If $|y|_d \leq |w|_d$ then we have $w = w_1 \star w_2$ for some w_1, w_2 such that $|w_2|_d = |y|_d$ by 8.3.2(5). We obtain $x = v \star w_1$ and $w_2 = y$ by 8.3.2(6). But then $w_1 = 0$ by 8.3.5(3) and so $x = v$ as well as $w = w_2 = y$ by 8.3.2(2). The case $|y|_d \geq |w|_d$ is similar.

Additional property of the prefix code pairing function is that its range is the set of the prefix codes with positive counts of digits **2**:

$$\vdash_{\mathrm{PA}} 1 \neq x \,,_p\, y \tag{6}$$

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \wedge \#(x) > 0 \rightarrow \exists v \exists w (x = v \,,_p\, w \wedge \mathit{Prf}(v) \wedge \mathit{Prf}(w)) \; . \tag{7}$$

(6): This is because

$$\#(1) \overset{8.3.3(14)}{=} 0 < 1 + \#(x \star y) \overset{8.3.3(15)}{=} \#(2) + \#(x \star y) \overset{8.3.3(16)}{=} \\ \#(2 \star x \star y) = \#(x \,,_p\, y) \; .$$

(7): Assume $\mathit{Prf}(x)$ and $\#(x) > 0$. Since $2{\cdot}\#(x) + 1 = |x|_d$, we have $x = x_1 \star x_2$ for some x_1, x_2 s.t. $|x_2|_d = 2{\cdot}\#(x)$ by 8.3.2(5). Thus $|x_1|_d = 1$ by 8.3.2(3). Since $|x_2|_d > 0$ we have $x_2 > 0$ by 8.3.1(4) and from $\mathit{Prf}(x)$ we obtain

$$2{\cdot}\#(x) = |x_2|_d > 2{\cdot}\#(x_2) \; , \tag{8}$$

i.e. $\#(x_1) + \#(x_2) \overset{8.3.3(16)}{=} \#(x) > \#(x_2)$ and so $\#(x_1) > 0$. Since $x_1 = 1$ or $x_1 = 2$ by 8.3.1(5), it must be the case that $x_1 = 2$ by 8.3.3(14). We have $x_2 = v \star w$ for some v, w s.t. $\mathit{Prf}(v)$ by (8) and 8.3.5(2). We have $x = x_1 \star x_2 = 2 \star v \star w = v \,,_p\, w$ and it suffices to prove $\mathit{Prf}(w)$. For that we note that

$$2 + 2{\cdot}\#(v) + 2{\cdot}\#(w) + 1 = 2{\cdot}\#(2 \star v \star w) + 1 = 2{\cdot}\#(x) + 1 = |x|_d = \\ |2 \star v \star w|_d = 1 + |v|_d + |w|_d = 1 + 2{\cdot}\#(v) + 1 + |w|_d$$

by the properties of the dyadic size and counting functions. Hence $2{\cdot}\#(w) + 1 = |w|_d$. We now take any $a, b > 0$ such that $w = a \star b$. From $x = (2 \star v \star a) \star b$ and $\mathit{Prf}(x)$ we get the desired $|b|_d > 2{\cdot}\#(b)$.

Order on Prefix Codes

We will now define a well-ordering relation $<_p$ on the prefix codes which will impose the structure of $\mathbb{N}$ on the codes.

8.3.7 Indexing properties of prefix codes. Following properties of prefix codes play central role in the definition of the order $<_p$:

$$\vdash_{\text{PA}} \mathit{Prf}(x) \land i < |x|_d \to \exists a \exists v \exists b(x = a \star v \star b \land |a|_d = i \land \mathit{Prf}(v)) \tag{1}$$

$$\vdash_{\text{PA}} \mathit{Prf}(x) \land x = a_1 \star v_1 \star b_1 \land \mathit{Prf}(v_1) \land x = a_2 \star v_2 \star b_2 \land \mathit{Prf}(v_2) \land \\ |a_1|_d = |a_2|_d \to a_1 = a_2 \land v_1 = v_2 \land b_1 = b_2 \ . \tag{2}$$

Property (1) says that in every prefix code x it is possible to find a prefix code v at an arbitrary distance $i < |x|_d$ from the beginning of x. Property (2) asserts that this v is uniquely determined. We can thus view the number i as an index selecting the *prefix code v at the position i of x*.

(1): Assume $\mathit{Prf}(x)$ and $i < |x|_d$. Since $0 < |x|_d \dot- i < |x|_d$, we get $x = a \star c$ for some a, c such that $0 < |c|_d = |x|_d \dot- i$ by 8.3.2(5). We have $|a|_d = i$ by 8.3.2(3) and $|c| > \#(c)$ because $\mathit{Prf}(x)$. Thus $c = v \star b$ for some v, b s.t. $\mathit{Prf}(v)$ by 8.3.5(2).

(2): Assume the antecedent of the property. We have $|v_1 \star b_1|_d = |v_2 \star b_2|_d$ by 8.3.2(3) and $a_1 = a_2$, $v_1 \star b_1 = v_2 \star b_2$ by 8.3.2(6). We now consider two cases. If $|b_1|_d \leq |b_2|_d$ then $b_2 = c_1 \star c_2$ for some c_1, c_2 such that $|b_1|_d = |c_2|_d$ by 8.3.2(5), $v_1 = v_2 \star c_1$, $b_1 = c_2$ by 8.3.2(6), and $c_1 = 0$ by 8.3.5(3). Thus $v_1 = v_2$ and $b_1 = b_2$ by 8.3.2(2).

The case $|b_1|_d \geq |b_2|_d$ is similar.

8.3.8 Order on prefix codes. We will define $x <_p y$ to hold iff x and y code in the prefix notation binary trees t and s respectively and if in the order from left to right there is a subtree in t with a lesser number of inner nodes than the corresponding subtree in s while all corresponding subtrees to the left are identical and all corresponding ancestors subtrees have equal number of inner nodes.

This seemingly complicated property expresses the first-by-size-than-by-lexicographic-order enumeration of binary trees in Fig. 1.2. The property is arithmetized with the help of two auxiliary ternary predicates $x <_p^i y$ and $x =_p^i y$ introduced into PA by explicit definitions:

$$\vdash_{\text{PAx}} x <_p^i y \leftrightarrow \exists a \exists v_1 \exists b_1 \exists v_2 \exists b_2(x = a \star v_1 \star b_1 \land \mathit{Prf}(v_1) \land |a|_d = i \land \\ y = a \star v_2 \star b_2 \land \mathit{Prf}(v_2) \land \#(v_1) < \#(v_2)) \tag{1}$$

$$\vdash_{\text{PAx}} x =_p^i y \leftrightarrow \exists a \exists v_1 \exists b_1 \exists v_2 \exists b_2(x = a \star v_1 \star b_1 \land \mathit{Prf}(v_1) \land |a|_d = i \land \\ y = a \star v_2 \star b_2 \land \mathit{Prf}(v_2) \land \#(v_1) = \#(v_2)) \ . \tag{2}$$

For two prefix codes x and y we clearly have $x <_p^i y$ ($x =_p^i y$) iff the code at the position i of x has a lesser (equal) dyadic size than the code at the position i of y.

We now explicitly introduce into PA the binary predicate $x <_p y$:

$$\vdash_{\mathrm{PAx}} x <_p y \leftrightarrow \mathit{Prf}(x) \wedge \mathit{Prf}(y) \wedge \exists i(x <^i_p y \wedge \forall j(j < i \rightarrow x =^j_p y)) \tag{3}$$

and prove that it is a linear order over Prf:

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \rightarrow x \not<_p x \tag{4}$$

$$\vdash_{\mathrm{PA}} x <_p y \wedge y <_p z \rightarrow x <_p z \tag{5}$$

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \wedge \mathit{Prf}(y) \wedge x \neq y \rightarrow x <_p y \vee y <_p x \ . \tag{6}$$

(4): Assume $\mathit{Prf}(x)$ and take any $i < |x|_d$. By Par. 8.3.7 we have $x = a \star v \star b$ for the uniquely determined a, v, b s.t. $|a|_i = i$. We cannot have $\#(v) < \#(v)$, thus $\neg v <^i_p v$, and hence $x \not<_p x$.

(5): Suppose $x <_p y$ and $y <_p z$. We have $\mathit{Prf}(x)$, $\mathit{Prf}(y)$, $\mathit{Prf}(z)$, $x <^{i_1}_p y$, $y <^{i_2}_p z$ for some i_1, i_2 s.t. $i_1 < |x|_d$, $i_1 < |y|_d$, $i_2 < |y|_d$, $i_2 < |z|_d$. In the following we will use obvious transitivity properties of the auxiliary predicates which are direct consequences of the indexing of prefix codes (see Par. 8.3.7). We consider three cases. If $i_1 < i_2$ then for any $j < i_1$ we have $x =^j_p y$, $y =^j_p z$, and hence $x =^j_p z$. From $x <^{i_1}_p y$ and $y =^{i_1}_p z$ we get $x <^{i_1}_p z$. Thus $x <_p z$.

The case when $i_2 < i_1$ is similar, and if $i_1 = i_2$ then we clearly have $x <^{i_1}_p z$ and $x =^j_p z$ for any $j < i_1$. Hence $x <_p y$ again.

(6): Suppose $\mathit{Prf}(x)$, $\mathit{Prf}(y)$, $x \neq y$, and consider three cases. If $|x|_d < |y|_d$ then $x <^0_p y$ and so $x <_p y$. If $|y|_d < |x|_d$ then $y <^0_p x$ and so $y <_p x$. Finally, if $|y|_d = |x|_d = k$ then $\neg y =^i_p x$ for some $i < k$ and there is the least such i by the least number principle. We thus have $x =^j_p y$ for all $j < i$ and either $x <^i_p y$ or $y <^i_p x$. Hence $x <_p y$ or $y <_p x$.

8.3.9 Additional properties of $<_p$. We introduce the binary predicate $\leq_p$ by explicit definition:

$$\vdash_{\mathrm{PAx}} x \leq_p y \leftrightarrow x <_p y \vee x = y \wedge \mathit{Prf}(x) \ . \tag{1}$$

The reader will note that $x \leq x$ does not hold if x is not a prefix code.

We now prove the remaining properties of the order on Prf which we will need below:

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \wedge \mathit{Prf}(y) \rightarrow \#(x) < \#(y) \rightarrow x <_p y \tag{2}$$

$$\vdash_{\mathrm{PA}} x \leq_p y \rightarrow \#(x) \leq \#(y) \tag{3}$$

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \rightarrow 1 \leq_p x \tag{4}$$

$$\begin{aligned}\vdash_{\mathrm{PA}} \ & \mathit{Prf}(x) \wedge \mathit{Prf}(y) \wedge \mathit{Prf}(v) \wedge \mathit{Prf}(w) \wedge \#(x) + \#(y) = \#(v) + \#(w) \rightarrow \\ & (x \,,_p y <_p v \,,_p w \leftrightarrow x <_p v \vee x = v \wedge y <_p w) \ . \end{aligned} \tag{5}$$

(2): Suppose $\mathit{Prf}(x)$, $\mathit{Prf}(y)$, and $\#(x) < \#(y)$. We have $x <^0_p y$ and so $x <_p y$.

(3): Suppose $x \leq_p y$. We have $Prf(x)$, $Prf(y)$. If it were the case that $\#(x) > \#(y)$ we would have $y <_p x$ by (2) and $x <_p x$ by 8.3.8(5) thus contradicting 8.3.8(4).

(4): Suppose $Prf(x)$ and consider two cases. If $\#(x) = 0$ then $x = 1$ by 8.3.5(4) and we trivially have $1 \leq_p 1$. If $\#(x) > 0$ then, since $\#(1) = 0$, we have $1 <_p x$ by (2).

(5): Assume the antecedent of the property. We have $Prf(x \,,_p y)$, $Prf(v \,,_p w)$, and

$$\#(x \,,_p y) = \#(x) + \#(v) + 1 = \#(v) + \#(w) + 1 = \#(v \,,_p w) \qquad (6)$$

by 8.3.6(2). Set $k = 2{\cdot}\#(x \,,_p y) + 1 = |x \,,_p y|_d = |v \,,_p w|_d$ and in the direction ($\rightarrow$) assume $2 \star x \star y = x \,,_p y <_p v \,,_p w = 2 \star v \star w$. We now consider three cases. If $\#(x) < \#(v)$ then $x <_p v$ by (2) and we are done.

If $\#(x) > \#(v)$ then $v <_p x$ by (2) and we have $v <_p^i x$ for some $i < |v|_d = 2{\cdot}\#(v) + 1 < 2{\cdot}\#(x) + 1 = |x|_d$. Also $v =_p^j x$ for all $j < i$. But we then get a contradiction as we obtain $v \,,_p w = 2 \star v \star w <_p 2 \star x \star y = x \,,_p y$ from $2 \star v \star w <_p^{i+1} 2 \star x \star y$ and $2 \star v \star w <_p^j 2 \star x \star y$ for all $j < i + 1$.

The final case is $\#(x) = \#(v)$. Then $\#(y) = \#(w)$, $|x|_d = |v|_d$, and $|y|_d = |w|_d$. We cannot have $v <_p x$ by the same reasoning as in the preceding case. Thus $x \leq_p v$ by 8.3.8(6). If $x <_p v$ we are done. If $x = v$ then we have $2 \star x \star y <_p^i 2 \star v \star w$ for some $1 + |x|_d \leq i < k$ and $2 \star x \star y =_p^j 2 \star v \star w$ for all $j < i$. But then $y <_p^{i \dot{-} (1 + |x|_d)} w$ and $y =_p^j w$ for all $j < i \dot{-} (1 + |x|_d)$. Hence $y <_p w$.

In the direction ($\leftarrow$) consider two cases. If $x <_p v$ then, since $\#(x) \leq \#(v)$ and so $|x|_d \leq |v|_d$ by (3), we have $x <_p^i v$ for some $i < |x|_d$ and $x =_p^j v$ for all $j < i$. We then get $x \,,_p y <_p v \,,_p w$ because $2 \star x \star y <_p^{i+1} 2 \star v \star w$ and $2 \star x \star y =_p^j 2 \star v \star w$ for all $j < i + 1$.

The second case is when $x = v$ and $y <_p w$. We similarly as in the preceding case get $y <_p^i w$ for some $i < |y|_d$ and $y =_p^j w$ for all $j < i$. But then $x \,,_p y <_p v \,,_p w$ because $2 \star x \star y <_p^{i + |x|_d + 1} 2 \star v \star w$ and $2 \star x \star y =_p^j 2 \star v \star w$ for all $j < i + |x|_d + 1$.

8.3.10 The principle of the least prefix code. The ordering predicate of prefix codes $<_p$ is a well-order. This can be expressed in the first-order language of PA only as a theorem schema called the *principle of the least prefix code* with a formula $\phi[x]$ with one indicated variable x and y a variable not occurring in ϕ:

$$T \vdash \exists x \phi[x] \wedge \forall x(\phi[x] \rightarrow Prf(x)) \rightarrow \exists x(\phi[x] \wedge \forall y(y <_p x \rightarrow \neg\phi[y]))\ . \qquad (1)$$

Here T is any proper extension of PA which contains the functions and predicates involved with prefix codes.

Property (1) is proved by working in T. We assume $\forall x(\phi[x] \rightarrow Prf(x))$ and $\phi[z_0]$ for some z_0. For the formula $\psi_1[n, x] \equiv \phi[x] \wedge |x|_d = n$ we have

$\psi_1[|z_0|_d, z_0]$. Hence $\exists x \psi_1[m,x]$ for the least m by the least number principle. Thus $\phi[z]$ and $|z|_d = m$ for some z. Thus also $\mathit{Prf}(z)$ and $m = |z|_d = 2{\cdot}\#z + 1 > 0$. By the minimality of m we have $m \leq |y|_d$ for any y s.t. $\phi[y]$.

For the formula

$$\psi_2[n,x] \equiv \phi[x] \wedge |x|_d = m \wedge \\ \forall y \forall i(\phi[y] \wedge i < n \wedge \forall j(j < i \rightarrow x =_p^j y) \rightarrow \neg y <_p^i x)$$

we prove by induction on n

$$n \leq m \rightarrow \exists x \psi_2[n,x] \ . \tag{2}$$

Roughly speaking, the property says that for every $n \leq m$ there is an x satisfying $\phi[x]$ which is a minimal prefix code up to n.

The base case is trivially satisfied by $x := z$ because $i \not< 0$. In the inductive case assume $n{+}1 \leq m$ and there is an x_0 s.t. $\psi_2[n,x_0]$ by IH. We have $\mathit{Prf}(x_0)$ and $n < m = |x_0|_d$ and let v_0 be the code at the position n of x_0. We thus have $\psi_3[|v_0|_d, x_0]$ for the formula

$$\psi_3[k,x] \equiv \psi_2[n,x] \wedge \forall a \forall v \forall b(x = a \star v \star b \wedge \mathit{Prf}(v) \wedge |a|_d = n \rightarrow |v|_d = k) \ .$$

Hence $\exists k \exists x \psi_3[k,x]$, and by the least number principle there is a smallest such k for which $\exists x \psi_3[k,x]$. Thus $\psi_3[k,x]$ for some x and so $\psi_2[n,x]$ and for the code v at the position n of x we have $|v|_d = k$. We claim that $\phi_2[n+1,x]$ holds. So take any y, i s.t. $\phi[y]$, $i < n+1$, and $\forall j(j < i \rightarrow x =_p^j y)$. We have $\mathit{Prf}(y)$ and we wish to show $\neg y <_p^i x$. If $i \geq |y|_d$ then we have $\neg y <_p^i x$. If $i < |y|_d$ we consider two cases. If $i < n$ then we have $\neg y <_p^i x$ from $\psi_2[n,x]$. If $i = n$ then let w be the code at the position n of y. By the minimality of k we have $|v_0|_d = k \leq |w|_d$ and so we have $x <_p^n y$ or $x =_p^n y$ and hence $\neg y <_p^n x$. This ends the induction proof of (2).

From the just proved property we obtain $\psi_2[m,x]$ for some x and we claim that x is the least prefix code satisfying $\phi[x]$. So take any y s.t. $y <_p x$ and suppose on the contrary $\phi[y]$. We have $m \leq |y|_d$ and so there is an i such that $i < m$, $y <_p^i x$, and $y =_p^j x$ for all $j < i$. We now get the contradiction $\neg\phi[x]$ from $\psi_2[m,x]$.

8.3.11 Minima and maxima of prefix codes of count n. We call the number a the *minimum of codes with the count n* if

$$\mathit{Prf}(a) \wedge \#(a) = n \wedge \forall x(\mathit{Prf}(x) \wedge \#(x) = n \rightarrow a \leq_p x) \tag{1}$$

We call the number b the *maximum of codes with the count n* if

$$\mathit{Prf}(b) \wedge \#(b) = n \wedge \forall x(\mathit{Prf}(x) \wedge \#(x) = n \rightarrow x \leq_p b) \tag{2}$$

The reader will note that because $<_p$ is a linear order over Prf there is at most one minimum (maximum) code of a given count n. For each n the maxima and minima exist:

$$\vdash_{\mathrm{PA}} \exists b(Prf(b) \wedge \#(b) = n \wedge \forall x(Prf(x) \wedge \#(x) = n \rightarrow x \leq_p b)) \tag{3}$$
$$\vdash_{\mathrm{PA}} \exists a(Prf(a) \wedge \#(a) = n \wedge \forall x(Prf(x) \wedge \#(x) = n \rightarrow a \leq_p x)) \; . \tag{4}$$

(3): By induction on n. In the base case we set $b := 1$ because $\#(b) = 0$ and for any x s.t. $Prf(x)$, $\#(x) = 0$ we have $x = 1$ by 8.3.5(4). In the inductive case we obtain the maximum b_1 of the count n by IH and we set $b = b_1 ,_p 1$. Since $Prf(1)$, $\#(1) = 0$ by 8.3.5(4), we get $Prf(b)$, $\#(b) = n + 1$ by 8.3.6(2). Let x be any number such that $Prf(x)$ and $\#(x) = n + 1$. We have $x = x_1 ,_p x_2$ for some x_1, x_2 such that $Prf(x_1)$, $Prf(x_2)$ by 8.3.6(7) and $\#(x_1) + \#(x_2) = n$ by 8.3.6(2). For the proof of $x = x_1 ,_p x_2 \leq_p b_1 ,_p 1 = b$ in which we use 8.3.9(5) we note that $\#(x_1) \leq n = \#(b_1)$. If $\#(x_1) < \#(b_1)$ then $x_1 <_p b_1$ by 8.3.9(2) and so $x <_p b$. If $\#(x_1) = \#(b_1)$ then $x_1 \leq_p b_1$ by the maximality of b_1. If $x_1 <_p b_1$ then $x <_p b$ again. If $x_1 = b_1$ then $\#(x_2) = 0$ and we get $x_2 = 1$ by 8.3.5(4) and so $x <_p b$.

(4): We use the principle of the least prefix code with the formula $\phi[x, n] \equiv Prf(x) \wedge \#(x) = n$. Let b be the maximum code with the count n existing by (3). We have $\phi[b, n]$ and so the antecedent of 8.3.10(1) is satisfied. Thus there is a number a s.t. $\phi[a, n]$ and for any $y <_p a$ we have $\neg\phi[y, n]$. To see that a is the minimum take any number x s.t. $Prf(x)$ and $\#(x) = n$, i.e. $\phi[x, n]$. We must have $x \not<_p a$ and so $a \leq_p x$ by 8.3.8(6).

8.3.12 Successor function over prefix codes. We wish to introduce the unary function $s_p(x)$ yielding the least prefix code after x in the order $<_p$ by the following contextual definition:

$$\vdash_{\mathrm{PAx}} s_p(x) = y \leftrightarrow x <_p y \wedge \forall z(x <_p z \rightarrow y \leq_p z) \vee \neg Prf(x) \wedge y = 0 \tag{1}$$

Its existence condition

$$\vdash_{\mathrm{PA}} \exists y(x <_p y \wedge \forall z(x <_p z \rightarrow y \leq_p z) \vee \neg Prf(x) \wedge y = 0) \tag{2}$$

is proved by taking any x and considering two cases. If $\neg Prf(x)$ then it suffices to set $y := 0$. If $Prf(x)$ then also $Prf(x ,_p x)$ and $\#(x, ,_p x) = 2\#(x)+1 > \#(x)$ by 8.3.6(2). Hence $x <_p x ,_p x$ by 8.3.9(2) and thus $\exists y\, x <_p y$. The formula $\phi[x, y] \equiv x <_p y$ satisfies for y the antecedent of the principle of the least prefix code because we have $\exists y\, x <_p y$ and from $x <_p y$ we obtain $Prf(y)$. Hence there is a y s.t. $x <_p y$ and for any z such that $z <_p y$ we have $\neg x <_p z$. But then if $x <_p z$ we have $\neg z <_p y$, i.e. $y \leq_p z$ by 8.3.8(6).

The uniqueness condition for s_p follows from

$$\begin{aligned} \vdash_{\mathrm{PA}}\; & x <_p y_1 \wedge \forall z(x <_p z \rightarrow y_1 \leq_p z) \wedge \\ & \quad x <_p y_2 \wedge \forall z(x <_p z \rightarrow y_2 \leq_p z) \rightarrow y_1 = y_2 \end{aligned} \tag{3}$$
$$\vdash_{\mathrm{PA}} y_1 = 0 \wedge y_2 = 0 \rightarrow y_1 = y_2 \; . \tag{4}$$

(3): Assume the antecedent. We have $Prf(y_1)$ and $Prf(y_2)$. If it were the case that $y_1 \neq y_2$ then we would have $y_1 <_p y_2$ or $y_2 <_p y_1$ by 8.3.8(6). If

$y_1 <_p y_2$ then, since $x <_p y_1$, we instantiate the second quantifier formula in the antecedent with $z := y_1$ and get $y_2 \leq_p y_1$. Thus $y_2 <_p y_1$ and then $y_1 <_p y_1$ by 8.3.8(5) contradicting 8.3.8(4). The case $y_2 <_p y_1$ leads to a contradiction similarly.

(4): This is trivial.

The function s_p satisfies the following:

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \rightarrow x <_p s_p(x) \wedge \forall z(x <_p z \rightarrow s_p(x) \leq_p z) \tag{5}$$

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \leftrightarrow \mathit{Prf}(s_p(x)) \tag{6}$$

$$\vdash_{\mathrm{PA}} x <_p y \leftrightarrow s_p(x) <_p s_p(y) \tag{7}$$

$$\vdash_{\mathrm{PA}} \mathit{Prf}(x) \wedge \#(x) > 0 \rightarrow \exists y(\mathit{Prf}(y) \wedge s_p(y) = x)\ . \tag{8}$$

(5): If $\mathit{Prf}(x)$ then we get the consequent by instantiating (1) with $y := s_p(x)$.

(6): If $\mathit{Prf}(x)$ then $x <_p s_p(x)$ by (5) and we have $\mathit{Prf}(s_p(x))$. Vice versa, if $\mathit{Prf}(s_p(x))$ then if it were the case that $\neg\mathit{Prf}(x)$ we would get $s_p(x) = 0$ by instantiating (1) with $y := s_p(x)$. We would then get $\#(s_p(x)) = 0$ by 8.3.3(13) and a contradiction $s_p(x) = 1$ by 8.3.5(4).

(7): Assume $x <_p y$. We have $\mathit{Prf}(x)$ and $\mathit{Prf}(y)$ and we get $x <_p s_p(x) \leq_p y < s_p(y)$ by (5) and transitivity. Vice versa, assume $s_p(x) <_p s_p(y)$. We have $\mathit{Prf}(x)$ and $\mathit{Prf}(y)$ by 8.3.8(3) and (6). It cannot be the case that $x = y$ because then $s_p(x) = s_p(y)$ would contradict 8.3.8(4). Thus either $y <_p x$ or $x <_p y$ by 8.3.8(6). If the former then

$$s_p(y) \overset{(5)}{\leq_p} x \overset{(5)}{<_p} s_p(x) < s_p(y)$$

which by transitivity contradicts the irreflexivity of $<_p$.

(8): In the proof of this property we will repeatedly prove $s_p(x) = y$ for some $\mathit{Prf}(x)$ and $\mathit{Prf}(y)$ by using (1) where we will derive $x <_p y$ and then for any z such that $x <_p z$ we will prove $y \leq_p z$. The property follows from an auxiliary property

$$\vdash_{\mathrm{PA}} \forall x(\mathit{Prf}(x) \wedge k = \#(x) \wedge k > 0 \rightarrow \exists y(\mathit{Prf}(y) \wedge s_p(y) = x))$$

by instantiating $k := \#(x)$. The auxiliary property is proved by complete induction on k. So assume $\mathit{Prf}(x)$, $k = \#(x)$, and $k > 0$. We have $x = v\,,_p\, w$ for some v, w such that $\mathit{Prf}(v)$, $\mathit{Prf}(w)$ by 8.3.6(7) and $\#(x) = \#(v)+\#(w)+1$ by 8.3.6(2). We consider two cases.

The first case is when w is not the minimum code with the count $\#(w)$, i.e. $a <_p w$ and $\#(a) = \#(w)$ for some a. We then have $w \neq 1$ by 8.3.9(4) and $\#(w) > 0$ by 8.3.5(4). Since $\#(w) < k$, we have $\mathit{Prf}(w_1)$, $s_p(w_1) = w$ for some w_1 by IH. Thus $w_1 <_p w$ by (5) and $\#(w_1) \leq \#(w)$ by 8.3.9(3). If it were the case that $\#(w_1) < \#(w)$ then we would have $w_1 <_p a$ by 8.3.9(2) and we would get a contradiction $w = s_p(w_1) \leq_p a <_p w$ by (5). Thus $\#(w_1) = \#(w)$ and we get $v\,,_p\, w_1 <_p v\,,_p\, w$ by 8.3.9(5). Note that we have $\#(v\,,_p\, w_1) = k$.

We claim that $s_p(v \,{,_p}\, w_1) = v \,{,_p}\, w = x$. For that it remains to show that if $w \,{,_p}\, w_1 <_p z$ for any z then $x \leq_p z$. Thus take any z s.t. $w \,{,_p}\, w_1 <_p z$. We have $k = \#(v \,{,_p}\, w_1) \leq \#(z)$ by 8.3.9(3). If $k < \#(z)$ then $x <_p z$ by 8.3.9(2). If $0 < k = \#(z)$ then $z = z_1 \,{,_p}\, z_2$ for some z_1, z_2 such that $\mathit{Prf}(z_1)$, $\mathit{Prf}(z_2)$ by 8.3.6(7) and $\#(z) = \#(z_1) + \#(z_2) + 1$ by 8.3.6(2). Since $v \,{,_p}\, w_1 <_p z_1 \,{,_p}\, z_2$ and $\#(v) + \#(w) = \#(v) + \#(w_1) = \#(z_1) + \#(z_2)$, we have either $v <_p z_1$ or $v = z_1$ and $w_1 <_p z_2$, i.e. $w = s_p(w_1) \leq_p z_2$ by 8.3.9(5). But then $x = v \,{,_p}\, w \leq_p z_1 \,{,_p}\, z_2 = z$ by 8.3.9(5) again.

The second case is that w is the minimum code with the count $\#(w)$ and we consider three subcases.

The first subcase is when $v = 1$. We obtain from 8.3.11(3) the maximum code b with the count $k \dot{-} 1$. We claim that $s_p(b) = x = 1 \,{,_p}\, w$. For that we note that $b <_p x$ by 8.3.9(2). We now take any z s.t. $b <_p z$. We have $\mathit{Prf}(z)$ and it must be the case that $k \dot{-} 1 \leq \#(z)$ by 8.3.9(3). We cannot have $k \dot{-} 1 = \#(z)$ by the maximality of b. If $k < \#(z)$ then $x <_p z$ by 8.3.9(2). If $k = \#(z)$ then $z = z_1 \,{,_p}\, z_2$ for some z_1, z_2 such that $\mathit{Prf}(z_1)$, $\mathit{Prf}(z_2)$ by 8.3.6(7) and $\#(z_1) + \#(z_2) = \#(1) + \#(w) = k \dot{-} 1$ by 8.3.6(2). For the proof of $1 \,{,_p}\, w = x \leq_p z_1 \,{,_p}\, z_2 = z$ we note that $1 \leq_p z_1$ by 8.3.9(4). We use 8.3.9(5) in the two cases when either $1 <_p z_1$ and then $x <_p z$ or else $1 = z_1$ and then, since $\#(w) = \#(z_2)$, we get $w \leq_p z_2$ by the minimality of w and so $a \leq_p x$.

The second subcase is when $v \neq 1$ and v is not the minimum with the count $\#(v)$. By similar reasoning as in the first case above we obtain by IH a v_1 s.t. $\mathit{Prf}(v_1)$, $\#(v_1) = \#(v)$, $v_1 <_p v$, and $s_p(v_1) = v$. We then obtain from 8.3.11(3) the maximum code b with the count $\#(w)$. We have $\mathit{Prf}(b)$, $\#(b) = \#(w)$, and we claim that $s_p(v_1 \,{,_p}\, b) = v \,{,_p}\, w$. Since $\#(v_1)+\#(b) = \#(v)+\#(w)$, we get $v_1 \,{,_p}\, b <_p v \,{,_p}\, w$ by 8.3.9(5). Now we take any z such that $v_1 \,{,_p}\, b <_p z$. We have $k = \#(v_1 \,{,_p}\, b) \leq \#(z)$ by 8.3.9(3). If $k < \#(z)$ then $x <_p z$ by 8.3.9(2). If $0 < k = \#(z)$ then $z = z_1 \,{,_p}\, z_2$ for some z_1, z_2 such that $\mathit{Prf}(z_1)$, $\mathit{Prf}(z_2)$ by 8.3.6(7) and $\#(z) = \#(z_1) + \#(z_2) + 1$ by 8.3.6(2). Since $v_1 \,{,_p}\, b <_p z_1 \,{,_p}\, z_2$ and $\#(v) + \#(w) = \#(v_1) + \#(b) = \#(z_1) + \#(z_2)$, we have either $v_1 <_p z_1$ or $v_1 = z_1$ and $b <_p z_2$ by 8.3.9(5). The latter case cannot obtain because it would contradict the maximality of b because we would have $\#(b) = \#(z_2)$. Hence we have $v_1 < z_1$ and so $v = s_p(v_1) \leq_p z_1$ by (5). If $v <_p z_1$ then $x = v \,{,_p}\, w <_p z_1 \,{,_p}\, z_2 = z$ by 8.3.9(5) If $v = z_1$ then $\#(w) = \#(z_2)$ and we have $w \leq z_2$ by the minimality of w. Thus $x = v \,{,_p}\, w \leq_p z_1 \,{,_p}\, z_2 = z$ by 8.3.9(5) again.

The third subcase is when $v \neq 1$ and v is the minimum code with the count $\#(v)$. We must have $\#(v) > 0$ by 8.3.5(4). We use 8.3.11(3) to get the maxima b_1 with the count $\#(v) \dot{-} 1$ and b_2 with the count $\#(w) + 1$. Thus $\mathit{Prf}(b_1)$, $\#(b_1) = \#(v) \dot{-} 1$, $\mathit{Prf}(b_2)$, $\#(b_2) = \#(w)+1$, and $\#(b_1)+\#(b_2) = \#(v)+\#(w)$. We claim that $s_p(b_1 \,{,_p}\, b_2) = v \,{,_p}\, w$. First of all, we have $b_1 <_p v$ by 8.3.9(2) and so $b_1 \,{,_p}\, b_2 <_p v \,{,_p}\, w$ by 8.3.9(5). We now take any z s.t. $b_1 \,{,_p}\, b_2 <_p z$. We have $\mathit{Prf}(z)$ and $k = \#(b_1 \,{,_p}\, b_2) \leq \#(z)$ by 8.3.9(3). If $k < \#(z)$ we have $x <_p z$ by 8.3.9(2). If $0 < k = \#(z)$ then we have $z = z_1 \,{,_p}\, z_2$ for some z_1, z_2

such that $Prf(z_1)$, $Prf(z_2)$ by 8.3.6(7) and $\#(b_1) + \#(b_2) = \#(v) + \#(w) = \#(z_1) + \#(z_2)$ by 8.3.6(2). From $b_1 \,{,_p}\, b_2 <_p z_1 \,{,_p}\, z_2$ we get $b_1 <_p z_1$ or $b_1 = z_1$ and $b_2 <_p z_2$ by 8.3.9(5). The second case cannot happen because if $b_1 = z_1$ then $\#(b_2) = \#(z_2)$ and we get a contradiction $z_2 \leq_p b_2$ from the maximality of b_2. In the first case, when $b_1 <_p z_1$, we get $\#(b_1) \leq \#(z_1)$ by 8.3.9(3) and so $\#(b_1) < \#(z_1)$ by the maximality of b_1. Thus $\#(v) \leq \#(z_1)$. If $\#(v) < \#(z_1)$ then $x = v \,{,_p}\, w <_p z_1 \,{,_p}\, z_2 = z$ by 8.3.9(5). If $\#(v) = \#(z_1)$ then $v \leq_p z_1$ by the minimality of v and, since $\#(w) = \#(z_2)$, $w \leq_p z_2$ by the minimality of w. But then $x = v \,{,_p}\, w \leq_p z_1 \,{,_p}\, z_2 = z$ by 8.3.9(5).

Isomorphism of ℕ and *Prf* and a Pairing Function over ℕ

We will now establish an isomorphism between natural numbers and prefix codes. The isomorphism preserves orders and endows ℕ with pairing.

8.3.13 Enumeration of prefix codes. We define the unary function π by primitive recursion:

$$\vdash_{\text{PAx}} \pi(0) = 1 \tag{1}$$

$$\vdash_{\text{PAx}} \pi(x') = s_p\, \pi(x)\ . \tag{2}$$

The following properties of π assert that the function enumerates the prefix codes, i.e. it is an injection: (6), into Prf: (3), and onto Prf: (4). It is also an order isomorphism between ℕ and Prf: (5).

$$\vdash_{\text{PA}} Prf(\pi(x)) \tag{3}$$

$$\vdash_{\text{PA}} Prf(y) \rightarrow \exists x\, \pi(x) = y \tag{4}$$

$$\vdash_{\text{PA}} x < y \leftrightarrow \pi(y) <_p \pi(y) \tag{5}$$

$$\vdash_{\text{PA}} \pi(x) = \pi(y) \rightarrow x = y\ . \tag{6}$$

(3): By induction on x. In the base case we have $\pi(0) = 1$ and $Prf(1)$ by 8.3.5(4). In the inductive case we obtain $Prf(\pi(x))$ from IH and $\pi(x) <_p s_p\, \pi(x) = \pi(x')$ by 8.3.12(5). But then $Prf(\pi(x'))$.

(4): Assume $Prf(y)$ and $\forall x\, \pi(x) \neq y$. The formula

$$\phi[y] \equiv Prf(y) \wedge \forall x\, \pi(x) \neq y$$

thus satisfies the antecedent of the principle of least prefix code 8.3.10(1) and there is a z such that $\phi[z]$, i.e. $Prf(z)$, $\forall x\, \pi(x) \neq z$, and for all $u <_p z$ we have $\neg\phi[u]$. Since $\pi(0) = 1$, we have $z \neq 1$ and so $\#(z) > 0$ by 8.3.5(4). But then $s_p(u) = z$ for some u s.t. $Prf(u)$ by 8.3.12(8) and $u <_p z$ by 8.3.12(5). Hence $\pi(x) = u$ for some x from $\neg\phi[u]$ and we contradict $\phi[z]$ because:

$$\pi(x') = s_p\, \pi(x) = s_p(u) = z\ .$$

(5): We prove $\forall y(5)$ by induction on x. In the base case we assume in the direction $(\rightarrow)$ $0 < y$. Then $z' = y$ for some z and $\pi(y) = s_p\,\pi(z)$. We have $\mathit{Prf}(\pi(z))$, $\mathit{Prf}(\pi(y))$ by (3) and $\pi(x) <_p \pi(y)$ by 8.3.12(5). Since $1 \leq_p \pi(x)$ by 8.3.9(4), we get $\pi(0) = 1 <_p \pi(y)$ by 8.3.8(5). In the direction $(\leftarrow)$ we assume $\pi(0) <_p \pi(y)$. Thus $\mathit{Prf}(\pi(0))$, $\mathit{Prf}(\pi(y))$ and then $\pi(y) \neq 1 = \pi(0)$ by 8.3.8(4). Hence $y \neq 0$, i.e. $0 < y$.

In the inductive case we have $x' < y$ iff $x' < z'$ for some z s.t. $z' = y$ iff $x < z$ for some z s.t. $z' = y$ iff, by IH, $\pi(x) <_p \pi(z)$ for some z s.t. $z' = y$, iff by 8.3.12(7), $s_p\,\pi(x) <_p s_p\,\pi(z)$ for some z s.t. $z' = y$, iff $\pi(x') <_p \pi(z')$ for some z s.t. $z' = y$ iff $\pi(x') <_p \pi(y)$.

(6): if $\pi(x) = \pi(y)$ then, since $\mathit{Prf}(\pi(x))$ by (3), we have $\pi(x) \not<_p \pi(y)$ and $\pi(y) \not<_p \pi(x)$ by 8.3.8(4) and so $x \not< y$ and $y \not< x$ by (5).

Property (4) is the existence condition for the inverse of π introduced into PA by minimalization:

$$\pi^{-1}(x) = \mu_y[\mathit{Prf}(x) \rightarrow \pi(y) = x] \tag{7}$$

Direct consequence of the defining axiom for π^{-1} is

$$\vdash_{\text{PA}} \mathit{Prf}(x) \rightarrow \pi\,\pi^{-1}(x) = x\ . \tag{8}$$

8.3.14 Pairing function. We are now ready to introduce by explicit definition into PA the binary pairing function:

$$\vdash_{\text{PAx}} x, y = \pi^{-1}(\pi(x) ,_p \pi(y)) \tag{1}$$

The function extends the order isomorphism π between $\mathbb{N}$ and Prf also to pairing because we have:

$$\vdash_{\text{PA}} \pi(x, y) = \pi(x) ,_p \pi(y)\ . \tag{2}$$

Indeed, from the definition we obtain $\pi(x, y) = \pi\,\pi^{-1}(\pi(x) ,_p \pi(y))$ and, since $\mathit{Prf}(\pi(x, y))$ by 8.3.13(3), the property follows by 8.3.13(8).

The basic properties of the pairing function are the pairing property 1.3.7(1) which is proved as (3) and the property 1.3.7(3) proved as (4) which assert that every positive number is in the range of the pairing function:

$$\vdash_{\text{PA}} x, y = v, w \rightarrow x = v \land y = w \tag{3}$$
$$\vdash_{\text{PA}} x = 0 \lor \exists v \exists w\, x = v, w\ . \tag{4}$$

(3): Assume $x, y = v, w$. From $\pi(x, y) = \pi(v, w)$ we obtain $\pi(x) ,_p \pi(y) = \pi(v) ,_p \pi(w)$ by (2). Since $\mathit{Prf}(\pi(x))$, $\mathit{Prf}(\pi(v))$, $\mathit{Prf}(\pi(y))$ by 8.3.13(3), we obtain $\pi(x) = \pi(v)$; $\pi(y) = \pi(w)$ by 8.3.6(3), and $x = \pi^{-1}\pi(x) = \pi^{-1}\pi(v) = v$; $y = \pi^{-1}\pi(y) = \pi^{-1}\pi(w) = w$ by 8.3.13(8).

(4): Assume $0 < x$ and obtain $1 = \pi(x) <_p \pi(x)$ by 8.3.13(5). We have $\mathit{Prf}(1)$ and $\mathit{Prf}(\pi(x))$. Since $1 \not<_p 1$ by 8.3.8(4), we have $1 \neq \pi(x)$ and so

$\#(\pi(x)) > 0$ by 8.3.5(4). Thus $\pi(x) = y \,{,_p}\, z$ for some y, z such that $Prf(y)$, $Prf(z)$ by 8.3.6(7). We have $\pi(v) = y$ and $\pi(w) = z$ for some v and w by 8.3.13(4) and so

$$x \overset{8.3.13(8)}{=} \pi^{-1}\pi(x) = \pi^{-1}(y \,{,_p}\, z) = \pi^{-1}(\pi(v) \,{,_p}\, \pi(w)) \overset{8.3.14(1)}{=} v, w \ .$$

8.3.15 Pair size function. The function $\#(x)$ measures the complexity of prefix codes x as the number of inner nodes of binary trees expressed by x. The isomorphic image of $\#(x)$ is the pair size function $|x|_p$ (see Par. 1.3.9) which is introduced into PA by an explicit definition:

$$\vdash_{\mathrm{PAx}} |x|_p = \#(\pi(x)) \tag{1}$$

The following are the basic properties of the pair size function

$$\vdash_{\mathrm{PA}} |0|_p = 0 \tag{2}$$
$$\vdash_{\mathrm{PA}} |x, y|_p = |x|_p + |y|_p + 1 \ . \tag{3}$$

(2): We have $\pi(0) = 1$ and $\#(1) = 0$ by 8.3.3(14).
(3): We have $Prf(\pi(x))$ and $Prf(\pi(y))$ by 8.3.13(3) and so

$$|x, y|_p = \#(\pi(x, y)) \overset{8.3.14(2)}{=} \#(\pi(x) \,{,_p}\, \pi(y)) \overset{8.3.6(2)}{=}$$
$$\#(\pi(x)) + \#(\pi(y)) + 1 = |x|_p + |y|_p + 1 \ .$$

8.3.16 Ordering properties of pairs. The pairing property 8.3.14(3) and the property 8.3.14(4) which says that there is at most one atom 0 are concerned solely with the pairing function. We will now investigate properties connecting pairing to the order on $\mathbb{N}$. Property (1) says that 0 is an atom, (2) gives sufficent and necessary conditions for the comparison of two pairs, and (3) will be the basis for the induction principle on pairs:

$$\vdash_{\mathrm{PA}} 0 < x, y \tag{1}$$
$$\vdash_{\mathrm{PA}} x, y < v, w \leftrightarrow |x, y|_p < |v, w|_p \lor$$
$$|x, y|_p = |v, w|_p \land (x < v \lor x = v \land y < w) \tag{2}$$
$$\vdash_{\mathrm{PA}} x < x, y \land y < x, y \ . \tag{3}$$

(1): We have $Prf(\pi(x))$, $Prf(\pi(y))$ by 8.3.13(3) and $Prf(\pi(x) \,{,_p}\, \pi(y))$, $\#(\pi(x) \,{,_p}\, \pi(y)) > 0$ by 8.3.6(2). For $\pi(0) = 1$ we have $Prf(\pi(0))$ and $\#(\pi(0)) = 0$ by 8.3.5(4). We get $\pi(0) <_p \pi(x) \,{,_p}\, \pi(y)$ by 8.3.9(2) and, since and $\pi(x, y) = \pi(x) \,{,_p}\, \pi(y)$ by 8.3.14(2), we obtain $0 < x, y$ by 8.3.13(5).

(2): We have $x, y < v, w$ iff, by 8.3.13(5), $\pi(x, y) <_p \pi(v, w)$ iff, by 8.3.14(2), $\pi(x) \,{,_p}\, \pi(y) <_p \pi(v) \,{,_p}\, \pi(w)$.

We have $Prf(\pi(x))$, $Prf(\pi(y))$, $Prf(\pi(v))$, $Prf(\pi(w))$ by 8.3.13(3) and $Prf(\pi(x) \,{,_p}\, \pi(y))$, $Prf(\pi(v) \,{,_p}\, \pi(w))$ by 8.3.6(2).

We also have $|x,y|_p < |v,w|_p$ iff $\#(\pi(x,y)) < \#(\pi(v,w))$ iff, by 8.3.14(2), $\#(\pi(x)\,,_p\,\pi(y)) < \#(\pi(v)\,,_p\,\pi(w))$ iff, by 8.3.6(2), $\#(\pi(x)) + \#(\pi(y)) < \#(\pi(v)) + \#(\pi(w))$.

We similarly have $|x,y|_p = |v,w|_p$ iff $\#(\pi(x)) + \#(\pi(y)) = \#(\pi(v)) + \#(\pi(w))$.

Now, in the direction ($\rightarrow$) assume $x,y < v,w$, get $\pi(x,y) <_p \pi(v,w)$ by 8.3.13(5), and then $\#(\pi(x,y)) \leq \#(\pi(v,w))$ by 8.3.9(3). Thus $|x,y|_p \leq |v,w|_p$. If $|x,y|_p < |v,w|_p$ there is nothing to prove. If $|x,y|_p = |v,w|_p$ then $\#(\pi(x)) + \#(\pi(y)) = \#(\pi(v)) + \#(\pi(w))$ and we have

$$\pi(x) <_p \pi(v) \lor \pi(x) = \pi(v) \land \pi(y) <_p \pi(w) \tag{4}$$

by 8.3.9(5). Thus

$$x < v \lor x = v \land y < w \tag{5}$$

by 8.3.13(5)(6).

In the direction ($\leftarrow$) consider first the case $|x,y|_p < |v,w|_p$. We get $\#(\pi(x)\,,_p\,\pi(y)) < \#(\pi(v)\,,_p\,\pi(w))$ by 8.3.14(2) and $\pi(x)\,,_p\,\pi(y) <_p \pi(v)\,,_p\,\pi(w)$ by 8.3.9(2). Hence $x,y < v,w$ by the above.

In the second case assume $|x,y|_p = |v,w|_p$, i.e. $\#(\pi(x)) + \#(\pi(y)) = \#(\pi(v)) + \#(\pi(w))$, and (5). We obtain (4) by 8.3.13(5) and $\pi(x)\,,_p\,\pi(y) <_p \pi(v)\,,_p\,\pi(w)$ by 8.3.9(5). Thus $x,y < v,w$ again.

(3): If $x = 0$ we have $x < x,y$ by (1). If $x > 0$ we have $x = v,w$ for some v, w by 8.3.14(4), and since $|x|_d < |x,y|$ by 8.3.15(3), we get $x < x,y$ by (2). The proof of $y < x,y$ is similar.

8.3.17 Projection functions. The unary projection functions H and T (see Par. 1.3.13) are introduced into PA by minimalization whose existence conditions are direct consequences of 8.3.14(4):

$$\vdash_{\text{PAx}} H(x) = \mu_v[x > 0 \rightarrow \exists w\, v,w = x] \tag{1}$$

$$\vdash_{\text{PAx}} T(x) = \mu_w[x > 0 \rightarrow \exists v\, v,w = x]\ . \tag{2}$$

The projection functions satisfy the following:

$$\vdash_{\text{PA}} H(0) = 0 \tag{3}$$

$$\vdash_{\text{PA}} H(v,w) = v \tag{4}$$

$$\vdash_{\text{PA}} T(0) = 0 \tag{5}$$

$$\vdash_{\text{PA}} T(v,w) = w\ . \tag{6}$$

(3): The defining axiom for H implies $0 < H(0) \rightarrow \neg(0 > 0 \rightarrow \exists w\, 0,w = 0)$ from which we get $0 < H(0) \rightarrow 0 > 0$ and then $H(0) = 0$.

(4): The defining axiom for H implies $v,w > 0 \rightarrow \exists w_1\, H(v,w), w_1 = v,w$. We then get $H(v,w), w_1 = v,w$ for some w_1 by 8.3.16(1) and $H(v,w) = v$ by 8.3.14(3).

(5): Similarly as (3).

(6): Similarly as (4).

8.3.18 List indexing. We wish to introduce into PA the list indexing function $(x)_i$. The function was defined in Par. 1.3.18 by a clausal definition to satisfy:

$$\vdash_{\mathrm{PA}} (0)_x = 0 \tag{1}$$
$$\vdash_{\mathrm{PA}} (v,w)_0 = v \tag{2}$$
$$\vdash_{\mathrm{PA}} (v,w)_{i+1} = (w)_i \ . \tag{3}$$

The reader will note that this is not a definition by primitive recursion because the paramater x changes in the recursive application. We can introduce the list indexing function into PA with the help of the binary function $T^i(x)$ called the *iteration* of T. The iteration function is introduced into PA by primitive recursion:

$$\vdash_{\mathrm{PAx}} T^0(x) = x \tag{4}$$
$$\vdash_{\mathrm{PAx}} T^{i'}(x) = T\,T^i(x) \ . \tag{5}$$

The iteration function satisfies

$$\vdash_{\mathrm{PA}} T^i(0) = 0 \tag{6}$$
$$\vdash_{\mathrm{PA}} \forall x\, T^{i+1}(x) = T^i T(x) \ . \tag{7}$$

(6): By induction on i. In the base case we have $T^0(0) = 0$. In the inductive case we have $T^{i'}(0) = T\,T^i(0) \overset{\mathrm{IH}}{=} T(0) \overset{8.3.17(5)}{=} 0$.

(7): By induction on i. In the base case we have $T^{0+1}(x) = T\,T^0(x) = T(x) = T^0T(x)$. In the inductive case we get

$$T^{i'+1}(x) = T\,T^{i'}(x) \overset{\mathrm{IH}}{=} T\,T^iT(x) = T^{i+1}T(x) \ .$$

We can now introduce the indexing function into PA by explicit definition:

$$\vdash_{\mathrm{PAx}} (x)_i = H\,T^i(x) \tag{8}$$

and we can prove the above properties:

(1): We have $(0)_i = H\,T^i(0) \overset{(6)}{=} H(0) \overset{8.3.17(3)}{=} 0$.

(2): We have $(v,w)_0 = H\,T^0(v,w) = H(v,w) \overset{8.3.17(4)}{=} v$.

(3): We have $(v,w)_{i+1} = H\,T^{i+1}(v,w) \overset{(7)}{=} H\,T^iT(v,w) \overset{8.3.17(6)}{=} H\,T^i(w) = (w)_i$.

8.4 Course of Values Recursion with Measure

The most general form of introducing functions into PA by recursion is the *course of values recursion with measure* where we introduce into PA an n-ary function f such that $\vdash_{\mathrm{PA}} f(\vec{x}) = \tau[\vec{x}]$ for a term τ applying, in addition to the previously introduced functions, also the function symbol f provided a measure term $\mu[\vec{x}]$ 'goes down' in the recursion, i.e. we have $\mu[\vec{\rho}] < \mu[\vec{x}]$ for all recursive applications $f(\vec{\rho})$. There are no additional restrictions on the form of recursive applications $f(\vec{\rho})$ and the arguments $\vec{\rho}$ can again apply the function f in a *nested* form. The depth of nesting is not restricted.

The requirement that that the measure of arguments goes down is not met by all terms τ. Suitable terms must satisfy certain non-trivial contextual properties. We leave the specification of the properties until the third part **UNFINISHED** of this text where we introduce a *clausal language* for a 'comfortable' definition of functions such as is expected in computer programming.

In this section we introduce syntactical restrictions imposed on recursive applications in τ which will guarantee that the measure goes down regardless of the form of the term τ.

8.4.1 Case discrimination function D. We will need in PA the ternary *case discrimination* function D satisfying

$$\vdash_{\mathrm{PA}} D(x', y, z) = y \tag{1}$$

$$\vdash_{\mathrm{PA}} D(0, y, z) = z \; . \tag{2}$$

The function is introduced into PA by contextual definition:

$$\vdash_{\mathrm{PAx}} D(x, y, z) = v \leftrightarrow x > 0 \land v = y \lor x = 0 \land v = z \tag{3}$$

whose existence and uniqueness conditions are straightforward to prove. Properties (1), (2) directly follow from the defining axiom.

8.4.2 Characteristic functions of predicates. Let P be an n-ary predicate which has been introduced into PA. We denote by P_* its *characteristic function* which is an n-ary function introduced into PA by minimalization:

$$\vdash_{\mathrm{PAx}} P_*(\vec{x}) = \mu_y[P(\vec{x}) \rightarrow y = 1] \; . \tag{1}$$

This is legal because the existence condition $\vdash_{\mathrm{PA}} \exists y(P(\vec{x}) \rightarrow y = 1)$ is trivially provable. We have

$$\vdash_{\mathrm{PA}} P(\vec{x}) \rightarrow P_*(\vec{x}) = 1 \tag{2}$$

$$\vdash_{\mathrm{PA}} P(\vec{x}) \leftrightarrow P_*(\vec{x}) > 0 \tag{3}$$

because (2) directly follows from the defining axiom for P_*. Property (3) in the direction ($\rightarrow$) follows from (2). In the direction ($\leftarrow$) we note that the minimality condition in the defining axiom for P_* is $\vdash_{\mathrm{PA}} y < P_*(\vec{x}) \rightarrow \neg(P(\vec{x}) \rightarrow y = 1)$ from which we obtain ($\leftarrow$) of (3) by instantiation $y := 0$.

8.4.3 Numerals. We will need a notation for successor terms directly denoting natural numbers. The terms are called (monadic) *numerals* and they are defined in the meta-theory to satisfy the following recurrences:

$$\underline{0}_m \equiv 0 \tag{1}$$

$$\underline{n+1}_m \equiv \underline{n}'_m \ . \tag{2}$$

Note that for a number n the numeral term $\underline{n}_m$ is of the form

$$0\overbrace{'\ldots'}^{n}$$

and it denotes n in the standard model of PA.

8.4.4 Measure induction. Properties of functions introduced into PA by course of values recursion with measure μ are generally proved by induction with measure.

Let T be a proper extension of PA containing the predicate $<$, $\phi[\vec{x}]$ a formula, and $\mu[\vec{x}]$ a term of $\mathcal{L}_T$ with $n \geq 1$ indicated variables. Both ϕ and μ can have additional free variables as parameters. Furthermore, let the new variables $\vec{y}$ be pairwise distinct from the variables of $\vec{x}$. The formula of the *induction with measure* μ is

$$\forall \vec{x}(\forall \vec{y}(\mu[\vec{y}] < \mu[\vec{x}] \to \phi[\vec{y}]) \to \phi[\vec{x}]) \to \phi[\vec{x}] \tag{1}$$

The reader will note that for $\vec{x} \equiv x$ and $\mu[x] \equiv x$ the schema of measure induction is the schema of complete induction (see Par. 7.3.7).

8.4.5 Theorem. *Every proper extension T of PA containing the predicate $<$ proves the schema of induction with measure 8.4.4(1).*

Proof. We prove 8.4.4(1) from an auxiliary property

$$T \vdash \forall \vec{x}(\forall \vec{y}(\mu[\vec{y}] < \mu[\vec{x}] \to \phi[\vec{y}]) \to \phi[\vec{x}]) \to \forall \vec{v}(\mu[\vec{v}] < z \to \phi[\vec{v}])$$

with $\vec{v}$ and z new. We work in T, assume the formula expressing that ϕ *is μ-progressive*

$$\forall \vec{x}(\forall \vec{y}(\mu[\vec{y}] < \mu[\vec{x}] \to \phi[\vec{y}]) \to \phi[\vec{x}]) \ , \tag{1}$$

and prove

$$\forall \vec{v}(\mu[\vec{v}] < z \to \phi[\vec{v}]) \tag{2}$$

by induction on z. In the base case there is nothing to prove. In the inductive case we take any $\vec{v}$ s.t. $\mu[\vec{v}] < z'$ and consider two cases by dichotomy. If $\mu[\vec{v}] < z$ we obtain $\phi[\vec{v}]$ from IH: (2), If $\mu[\vec{v}] \geq z$ then we have $\mu[\vec{v}] = z$ and we use a variant $\forall \vec{y}(\mu[\vec{y}] < \mu[\vec{v}] \to \phi[\vec{y}])$ of IH: (2) in (1) instantiated with $\vec{x} := \vec{v}$ to obtain $\phi[\vec{v}]$.

The formula for the measure induction 8.4.4(1) follows from the auxiliary property by instantiating its consequent with $z := \mu[\vec{x}] + 1$ and $\vec{v} := \vec{x}$. □

8.4.6 Extension by course of values recursion with measure. Let T be a proper extension of PA. We assume that the pairing function (x, y), the case discrimination function $D(x, y, z)$, as well as the characteristic function $<_*$ of the predicate $<$ have been introduced into T.

We wish to extend T into a theory S whose language contains the n-ary function symbol f $(n \geq 1)$ in such a way that a *course of values* definition $f(\vec{x}) = \tau$ is a theorem of S. For that we assume that $\tau[f; \vec{x}]$ is a term of $\mathcal{L}_T + f$ with its free variables among the n indicated ones. Thus τ is built up from the variables among $\vec{x}$, numerals $\underline{n}_m$, and applications $g_1(\vec{\rho}), \ldots,$ $g_k(\vec{\rho})$ of functions where for $1 \leq j \leq k$ the function g_j has the arity $n_j \geq 1$. The term τ can also *recursively* apply the n-ary function symbol f which we indicate as $\tau[f; \vec{x}]$. The recursive applications $f(\vec{\rho})$ can be arbitrary, even *nested* when f is applied in terms $\vec{\rho}$. There is no restriction on the depth of nesting. The only condition, which is a semantic one, is that the recursion 'goes down' in a measure term $\mu[\vec{x}]$ of $\mathcal{L}_T$ with its free variables among the n indicated ones.

We keep the notation introduced in this paragraph fixed until the end of the section.

For an n-tuple of terms $\vec{\rho}$ and an n-ary function symbol g we will denote by $\tau[g; \vec{\rho}]$ the term obtained from τ by replacing in it all occurrences of variables from among $\vec{x}$ by the corresponding terms from among $\vec{\rho}$ and by the replacement of all recursive applications $f(\vec{\sigma})$ by applications $g(\vec{\sigma})$. We will also use the notation $\tau[[f]^{\mu}_{\vec{x}}; \vec{x}]$ as an abbreviation for the term

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

i.e. the term where we have replaced every application of $f(\vec{\sigma})$ in τ by the *guarded* application

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

The reader will note that the substitution of terms $\vec{\rho}$ for the corresponding variables $\vec{x}$ in the guarded term $\tau[[f]^{\mu}_{\vec{x}}; \vec{x}]$ is written as $\tau[[f]^{\mu]}_{\vec{\rho}}; \vec{\rho}]$ and it is an abbreviation for the term

$$\tau[\lambda\vec{y}.D((\mu[\vec{y}]<_*\mu[\vec{\rho}], f(\vec{y}), 0); \vec{\rho}] \;,$$

i.e. the term where we have replaced every application of $f(\vec{\sigma})$ in τ by the guarded application

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

We use the same notation $\rho[[f]^{\mu}_{\vec{x}}; \vec{x}]$ also for subterms ρ of τ.

The extension of T to S with the $(n+1)$-ary function symbol f and with the axioms universal closures of

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

$$\forall\vec{v}(\mu[\vec{v}] < \mu[\vec{x}] \to \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{v}, f(\vec{v})]) \to \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, f(\vec{x})] \;. \tag{2}$$

is called *extension by course of values recursion with measure*. The formula $\phi[c, \vec{x}, y]$, which is of $\mathcal{L}_T$ and it is used in the single induction (with measure) axiom of S containing the symbol f, will be determined in Par. 8.4.13.

We do not impose any semantic restrictions on the recursion in τ to guarantee that the measure goes down, We achieve the same effect by purely syntactic means where we surround recursive applications by *if-then-else* guards. Such a restriction is acceptable from the point of view of mathematics where it simplifies the proofs below but, by amounting to an additional test, it is not acceptable in computer programming. We will discuss the semantic conditions (which can be syntactically enforced by provability) in the part three **UNFINISHED** of this text.

8.4.7 Special terms. Pursuing our plan of introducing the function f into the theory S, we first extend the theory T to T_1 by introducing the list indexing function $(x)_i$ (see Par. 8.3.18) and, by explicit definitions, the functions h_j for every $1 \leq j \leq k$ as *special contractions* of functions g_j:

$$T_1 \vdash h_j(x) = g_j((x)_0, (x)_{\underline{1}_m}, \dots, (x)_{\underline{n_j - 1}_m}) \ . \tag{1}$$

It should be clear that we have for all $1 \leq j \leq k$:

$$T_1 \vdash h_j(x_1, \dots, x_{n_j}, 0) = g_j(x_1, \dots, x_{n_j}) \ . \tag{2}$$

The reader will recall our convention that the single argument of h_j is obtained by pairing.

For every subterm ρ of τ we define its *contraction* term $\langle\rho\rangle$ by induction on the structure of ρ to satisfy:

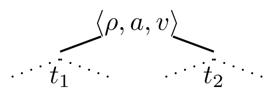
$$\begin{aligned}\langle x_i\rangle &\equiv (x)_{\underline{i-1}_m}\\ \langle \underline{n}_m\rangle &\equiv \underline{n}_m\\ \langle g_j(\rho_1, \dots, \rho_n)\rangle &\equiv h_j(\langle\rho_1\rangle, \dots, \langle\rho_n\rangle, 0)\\ \langle f(\rho_1, \dots, \rho_n)\rangle &\equiv r(\langle\rho_1\rangle, \dots, \langle\rho_n\rangle, 0)\end{aligned}$$

where r is a new unary function symbol. The reader will note that the term $\langle\tau\rangle$ is built up from the single variable x used only in applications $(x)_{\underline{i-1}_m}$ where $1 \leq i \leq n$ and from numerals $\underline{m}_m$ by applications of unary function symbols r and h_j where $1 \leq j \leq k$ whose arguments may contain also applications of the pairing function. Until the end of this section we call such terms *special*.

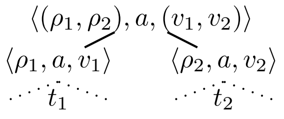
We designate by $\mu_1[x]$ the term obtained from the measure term $\mu[\vec{x}]$ by replacing for every $1 \leq i \leq n$ the occurrences of variables x_i by $(x)_{\underline{i-1}_m}$. Since $T_1 \vdash (x_1, \dots, x_n, 0)_{\underline{i-1}_m} = x_i$ for all $1 \leq i \leq n$, we van prove by a simple meta-theoretical induction on the structure of μ:

$$T_1 \vdash \mu_1[(x_1, \dots, x_n, 0)] = \mu[x_1, \dots, x_n] \ . \tag{3}$$

8.4.8 Computation trees. We will introduce the function f into S by the arithmetization of computations of terms $r(\underline{a}_m)$ using as *computation rule* the identity $r(x) = \langle\tau\rangle[[r]^{\mu_1}_x; x]$. Terms of the form $r(\underline{a}_m)$ are special cases of subterms of $\langle\tau\rangle[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$ and we will need to record the computation of all subterms ρ of $\langle\tau\rangle$. The computation of such a special term ρ is recorded by a binary labelled tree with the label consisting of a triple $\langle\rho, a, v\rangle$ where a is the value assigned to the variable x which may occur in ρ and v is the computed value, i.e. the denotation of the term $\rho[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$. The two sons t_1 and t_2 are the trees recording possible subcomputations:

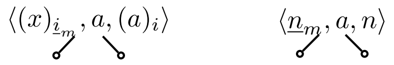


The form of the term ρ determines the shape of the sons t_1 and t_2 as follows. If $\rho \equiv \rho_1, \rho_2$ then the computation tree looks as follows:


$\langle(\rho_1, \rho_2), a, (v_1, v_2)\rangle$
$\langle\rho_1, a, v_1\rangle$
$\langle\rho_2, a, v_2\rangle$
t_1
t_2

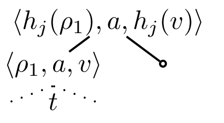
where the denotation v_1 of $\rho_1[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$ is computed in the left son and the denotation v_2 of $\rho_2[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$ is computed in the right son. The denotation of $(\rho_1, \rho_2)[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$ is then the pair (v_1, v_2).

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


$\langle(x)_{\underline{i}_m}, a, (a)_i\rangle$
$\langle\underline{n}_m, a, n\rangle$

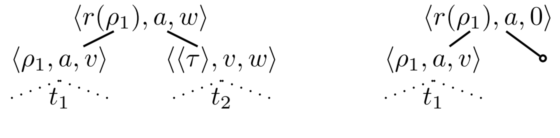
where there are no subcomputations because the denotations of $(x)_{\underline{i}_m}[[r]^{\mu}_{\underline{a}_m}; \underline{a}_m]$ and $\underline{n}_m[[r]^{\mu}_{\underline{a}_m}; \underline{a}_m]$ can be determined directly as $(a)_i$ and n respectively.

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


$\langle h_j(\rho_1), a, h_j(v)\rangle$
$\langle\rho_1, a, v\rangle$
t

where we record in the left son the computation of the argument ρ_1 into the value v and then the denotation of $h_j(\rho_1)[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]$ is $h_j(v)$. There is not need to record any computation in the right son.

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


$\langle r(\rho_1), a, w\rangle$
$\langle\rho_1, a, v\rangle$
$\langle\langle\tau\rangle, v, w\rangle$
t_1
t_2
$\langle r(\rho_1), a, 0\rangle$
$\langle\rho_1, a, v\rangle$
t_1

In both cases the denotation v of the argument ρ_1 is computed in the left son. The two cases are determined by the outcome of the test $\mathcal{N} \models \mu_1[\underline{v}_m] < \mu_1[\underline{a}_m]$. If the measure decreases, i.e. if the formula $\mu_1[\underline{v}_m] < \mu_1[\underline{a}_m]$ is true in the standard model, then the computation tree is shown above on the left. This is when the identity $r(x) = \langle\tau\rangle[[r]^{\mu_1}_x; x]$ is used as the computation rule in the form $r(\underline{v}_m) \mapsto \langle\tau\rangle[[r]^{\mu_1}_{\underline{v}_m}; \underline{v}_m]$. The denotation w of the term $\langle\tau\rangle[[r]^{\mu_1}_{\underline{v}_m}; \underline{v}_m]$ is computed in the right son and the denotation of the recursive application $r(\underline{v}_m)$ is determined as w.

If the outcome of the test is negative then the computation tree is shown above on the right where the denotation of $r(\underline{v}_m)$ is 0. The reader will note that the computation of $r(\rho)$ thus evaluates the guard

$$r(\rho_1)[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m] \equiv D(\mu_1[\rho_1[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]] <_* \mu_1[x[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]], r(\rho_1[[r]^{\mu_1}_{\underline{a}_m}; \underline{a}_m]), 0) \ .$$

It should be obvious that we can construct a computation tree for any subterm ρ of $\langle\tau\rangle$ 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, which is the denotation of $\mu_1[\underline{a}_m]$, can decrease only finitely many times.

The property of t being a computation tree can be expressed as follows:

> for all non-empty subtrees s of t the label of s is of a form $\langle\rho, a, v\rangle$ where ρ is a subterm of $\langle\tau\rangle$ and the relation between this label and the labels of the two sons of s is as shown in the above diagrams.

We will arithmetize the computation trees by extending in Par. 8.4.9 the theory T_1 to T_3 with the arithmetized subtree predicate $a \trianglelefteq b$ where a codes a subtree of the tree coded by b We will then extend in Par. 8.4.11 the theory T_3 to T_4 with a predicate $Ct(c)$ holding in the standard model iff c codes a computation tree t. The predicate Ct will be introduced with the help of the he predicate $Nd(a)$ expressing the local *node* relation between the label of the non-empty tree coded by a and the labels of its two sons.

8.4.9 Subtree predicate. We will now extend the theory T_1 into a theory T_3 by introducing a binary predicate $u \trianglelefteq t$ intended to hold whenever u codes a subtree of a binary tree coded by t. The empty binary tree is coded by the number 0 and the binary tree with the label b and two sons t_1, t_2 by the triple n, a_1, a_2 where a_1 and a_2 are the codes of the respective sons.

We wish the subtree predicate to satisfy the following:

$$T_3 \vdash u \trianglelefteq t \leftrightarrow u = 0 \lor u = t \lor \exists b \exists t_1 \exists t_2 (t = b, t_1, t_2 \land (u \trianglelefteq t_1 \lor u \trianglelefteq t_2)) \ . \qquad (1)$$

The reader will note that such a subtree predicate $u \trianglelefteq t$ does not fully express the relation that u codes a subtree of a binary tree coded by t. In particular, we have $0 \trianglelefteq t$ even if t is not a code. But if t codes a binary tree and $u \trianglelefteq t$ holds then u codes a subtree of the tree coded by t.

Property (1) is a recurrence where the subtree predicate is applied on the right to lesser second arguments $t_1 < t$ and $t_2 < t$ than on the left. Recursive predicates are usually defined from their characteristic functions. We cannot introduce $\trianglelefteq_*$ by primitive recursion because its natural recurrences

$$(0 \trianglelefteq_* t) = 1$$
$$(u \trianglelefteq_* u) = 1$$
$$(u \trianglelefteq_* t_1) = 1 \lor (u \trianglelefteq_* t_2) = 1 \to (u \trianglelefteq_* b, t_1, t_2) = 1$$

are of the form of so called *course of values recursion* where the recursive argument (the second one) does not decrease in the recursion to its immediate predecessor. Course of values recursion can be reduced to primitive recursion with the help of the *course of values function* h for $\trianglelefteq_*$ such that the following holds:

$$h(t, u) = (u \trianglelefteq_* t - 1), (u \trianglelefteq_* t - 2), \ldots, (u \trianglelefteq_* 1), (u \trianglelefteq_* 0), 0\ .$$

We clearly have

$$(h(t + i', u))_i \to u \trianglelefteq_* t$$

and so the values of applications $u \trianglelefteq_* t_1$ and $u \trianglelefteq_* t_2$ in the recurrences for $\trianglelefteq_*$ can be recovered from the course of values function h as $(h(t, u))_i$ where $t_1 + i' = t$ or $t_2 + i' = t$ respectively.

Preparatory to the introduction of the function h we explicitly extend T_1 into T_2 with an auxiliary ternary predicate R:

$$T_2 \vdash R(u, t, a) \leftrightarrow u = 0 \lor u = t \lor \exists b \exists t_1 \exists t_2 \exists i (t = b, t_1, t_2 \land (t_1 + i' = t \lor t_2 + i' = t) \land (a)_i > 0)\ , \tag{2}$$

we also introduce into T_1 its characteristic function R_* (see Par. 8.4.2), and finally the course of values function h which is defined by primitive recursion:

$$T_2 \vdash h(0, u) = 0 \tag{3}$$
$$T_2 \vdash h(t', u) = R_*(u, t, h(t, u)), h(t, u)\ . \tag{4}$$

The reader will note that the characteristic function $\trianglelefteq_*$ for $\trianglelefteq$ could be now introduced into T_2 (but we will not need to do this) by explicit definition $u \trianglelefteq_* t = R_*(u, t, h(t, u))$ because of the following property:

$$T_2 \vdash (h(s + i', u))_i = R_*(u, s, h(s, u)) \tag{5}$$

which is proved in T_2 by induction on i. In the base case we have

$$(h(s{+}0', u))_0 = (h(s', u))_0 = (R_*(u, s, h(s, u)), h(s, u))_0 \overset{8.3.18(2)}{=} R_*(u, s, h(s, u))\ .$$

In the inductive case we have

$$\begin{aligned}(h(s+i'',u))_{i'} = (h((s+i')',u))_{i'} = \\ (R_*(u,s+i',h(s+i',u)),h(s+i',u))_{i'} \overset{8.3.18(3)}{=} \\ (h(s+i',u))_i \overset{\mathrm{IH}}{=} R_*(u,s,h(s,u)) \ .\end{aligned}$$

We are now ready to extend T_2 into T_3 by introducing the subtree predicate by explicit definition:

$$T_3 \vdash u \trianglelefteq t \leftrightarrow R(u,t,h(t,u)) \ . \tag{6}$$

The predicate satisfies the following auxiliary property:

$$T_3 \vdash u \trianglelefteq s \land s < t \leftrightarrow \exists i(s+i' = t \land (h(t,u))_i > 0) \tag{7}$$

for which we have $u \trianglelefteq s$ and $s < t$ iff $R(u,s,h(s,u))$ and $s < t$ iff, by 8.4.2(3), $R_*(u,s,h(s,u)) > 0$ and $s+i' = t$ for some i iff, by (5), $(h(t,u))_i > 0$ and $s+i' = t$ for some i.

We are now ready to prove in T_3 the basic recurrence (1) for the subtree predicate. We have $u \trianglelefteq t$ iff $R(u,t,h(t,u))$ iff, by (2), $(u = 0 \lor u = t)$ or $t = b,t_1,t_2$, $(t_1+i' = t \lor t_2+i' = t)$, and $(h(t,u))_i > 0$ for some b, t_1, t_2, and i iff $(u = 0 \lor u = t)$ or $t = b,t_1,t_2$ and

$$\exists i(t_1+i' = t \land (h(t,u))_i > 0) \lor \exists i(t_2+i' = t \land (h(t,u))_i > 0)$$

for some b, t_1, t_2 iff, by (7), $(u = 0 \lor u = t)$ or $t = b,t_1,t_2$ and

$$t_1 < t \land u \trianglelefteq t_1 \lor t_2 < t \land u \trianglelefteq t_2$$

for some b, t_1, t_2 iff, by 8.3.16(3), $(u = 0 \lor u = t)$ or $t = b,t_1,t_2$ and $(u \trianglelefteq t_1 \lor u \trianglelefteq t_2)$ for some b, t_1, t_2.

8.4.10 Codes of special terms. Special terms are arithmetized by encoding into numbers. To that end we introduce the following *constructor* functions operating on terms:

$$\boldsymbol{X}_\rho \equiv 0,\rho \tag{1}$$
$$\boldsymbol{N}(\rho) \equiv 1,\rho \tag{2}$$
$$\boldsymbol{P}(\rho_1,\rho_2) \equiv 2,\rho_1,\rho_2 \tag{3}$$
$$\boldsymbol{H}_{\rho_1}(\rho_2) \equiv 3,\rho_1,\rho_2 \tag{4}$$
$$\boldsymbol{R}(\rho) \equiv 4,\rho \ . \tag{5}$$

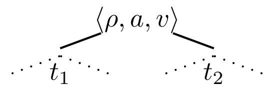
Constructor meta-functions are used to assign codes $\ulcorner\rho\urcorner$ (Gödel numbers) to special terms ρ to satisfy:

$$\ulcorner (x)_{\underline{i}_m} \urcorner \equiv \boldsymbol{X}_{\underline{i}_m} \tag{6}$$
$$\ulcorner \underline{m}_m \urcorner \equiv \boldsymbol{N}(\underline{m}_m) \tag{7}$$
$$\ulcorner \tau_1, \tau_2 \urcorner \equiv \boldsymbol{P}(\ulcorner \tau_1 \urcorner, \ulcorner \tau_2 \urcorner) \tag{8}$$
$$\ulcorner h_j(\tau) \urcorner \equiv \boldsymbol{H}_{\underline{j}_m}(\ulcorner \tau \urcorner) \tag{9}$$
$$\ulcorner r(\tau) \urcorner \equiv \boldsymbol{R}(\ulcorner \tau \urcorner) \ . \tag{10}$$

The reader will note that $\ulcorner \rho \urcorner$ stands for a closed term of $\mathcal{L}_{T_2}$ whose interpretation in the standard model of PA denotes the code of the special term ρ. Specifically, $\ulcorner \underline{3}_m \urcorner$ stands for the term $\boldsymbol{N}(0''')$ which denotes the same number as the term $1, 3$, i.e. the number 15.

The reader may ask why we have defined the constructor function as meta-functions yielding terms rather than as functions in PA? The reason for that will be seen in Par. 8.4.13 where it will be essential that the terms constructed with the help of constructors are of $\mathcal{L}_T$.

8.4.11 The predicate Ct**.** We will now extend the theory T_3 to T_4 by introducing the predicate $Ct(t)$ holding of codes t of computation trees. We arithmetize (encode) the empty computation tree as the number 0 and the tree



by the number denoted by the term

$$(\ulcorner \rho \urcorner, \underline{a}_m, \underline{v}_m), \ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner$$

where $\ulcorner t_1 \urcorner$ and $\ulcorner t_2 \urcorner$ encode the two sons respectively. The predicate is defined with the help of an auxiliary unary predicate $Nd(t)$ holding iff the labels of t and of its sons satisfy the local *node* conditions discussed in Par. 8.4.8.

Both predicates are introduced by explicit definitions:

$$\begin{aligned}
T_4 \vdash\ & Nd(t) \leftrightarrow \exists c \exists a \exists v \exists t_1 \exists t_2 (t = (c, a, v), t_1, t_2 \land (\\
& \quad \exists i (c = \boldsymbol{X}_i \land i < n \land v = (a)_i \land t_1 = 0 \land t_2 = 0) \lor \\
& \quad \exists m (c = \boldsymbol{N}(m) \land v = n \land t_1 = 0 \land t_2 = 0) \lor \\
& \quad \exists c_1 \exists c_2 \exists v_1 \exists v_2 \exists s_1 \exists s_2 (c = \boldsymbol{P}(c_1, c_2) \land v = v_1, v_2 \land \\
& \quad\quad t_1 = (c_1, a, v_1), s_1 \land t_2 = (c_2, a, v_2), s_2) \lor \\
& \quad \exists c_1 \exists j \exists w \exists s_1 (c = \boldsymbol{H}_j(c_1) \land t_1 = (c_1, a, w), s_1 \land t_2 = 0 \land \\
& \quad\quad (j = \underline{1}_m \land v = h_1(w) \lor \ldots \lor j = \underline{k}_m \land v = h_k(w))) \lor \\
& \quad \exists c_1 \exists w \exists s_1 \exists s_2 (c = \boldsymbol{R}(c_1) \land t_1 = (c_1, a, w), s_1 \land \\
& \quad\quad (\mu_1[w] < \mu_1[a] \land t_2 = (\ulcorner \tau \urcorner, w, v), s_2 \lor \\
& \quad\quad\ \mu_1[w] \geq \mu[a] \land v = 0 \land t_2 = 0))) && (1) \\
T_4 \vdash\ & Ct(t) \leftrightarrow \forall u (u \trianglelefteq t \land u > 0 \rightarrow Nd(u)) \ . && (2)
\end{aligned}$$

and Ct satisfies the following:

$$T_4 \vdash Ct(0) \tag{3}$$
$$T_4 \vdash i < n \to Ct((\boldsymbol{X}_i, a, (a)_i), 0, 0) \tag{4}$$
$$T_4 \vdash Ct((\boldsymbol{N}(m), a, m), 0, 0) \tag{5}$$
$$\begin{aligned} T_4 \vdash\ & Ct((c_1, a, v_1), s_1) \land Ct((c_2, a, v_2), s_2) \to \\ & \quad Ct((\boldsymbol{P}(c_1, c_2), a, v_1, v_2), ((c_1, a, v_1), s_1), ((c_2, a, v_2), s_2)) \end{aligned} \tag{6}$$
$$T_4 \vdash Ct((c, a, v), s) \to Ct((\boldsymbol{H}_{\underline{j}_m}(c), a, h_j(v)), ((c, a, v), s), 0) \tag{7}$$
$$\begin{aligned} T_4 \vdash\ & Ct((c, a, v), s_1) \land \mu_1[v] < \mu_1[a] \land Ct((\ulcorner\langle\tau\rangle\urcorner, v, w), s_2) \to \\ & \quad Ct((\boldsymbol{R}(c), a, w), ((c, a, v), s_1), ((\ulcorner\langle\tau\rangle\urcorner, v, w), s_2)) \end{aligned} \tag{8}$$
$$T_4 \vdash Ct((c, a, v), s_1) \land \mu_1[v] \geq \mu_1[a] \to Ct((\boldsymbol{R}(c), a, 0), ((c, a, v), s_1), 0) \tag{9}$$
$$T_4 \vdash Ct(b, t_1, t_2) \to Ct(t_1) \land Ct(t_2)\ . \tag{10}$$

Property (7) is a schema of k theorems one for each $1 \leq j \leq k$. Properties (3) through (9) can be called *sufficient* conditions for computation trees to be constructed out of smaller trees. The property (10) can be seen as stating a *necessary* condition for a triple b, t_1, t_2 to be a computation tree.

(3): This holds trivially.

(4): Assume $i < n$ and take any $u > 0$ such that $u \trianglelefteq (\boldsymbol{X}_i, a, (a)_i), 0, 0$. Since $u \trianglelefteq 0$ leads to contradiction $u = 0$, it must be the case that $u = (\boldsymbol{X}_i, a, (a)_i)), 0, 0$ by 8.4.9(1) and thus $Nd(u)$. This proves the consequent.

(5): Take any $u > 0$ such that $u \trianglelefteq (\boldsymbol{N}(m), a, m), 0, 0$. Since $u \trianglelefteq 0$ leads to contradiction $u = 0$, it must be the case that $u = (\boldsymbol{N}(m), a, m), 0, 0$ by 8.4.9(1) and thus $Nd(u)$. This proves the consequent.

(6): Assume $Ct((c_1, a, v_1), s_1)$, $Ct((c_2, a, v_2), s_2)$. For

$$t := (\boldsymbol{P}(c_1, c_2), a, v_1, v_2), ((c_1, a, v_1), s_1), ((c_2, a, v_2), s_2)$$

we wish to prove $Ct(t)$. We thus take any $u > 0$ such that $u \trianglelefteq t$. We consider the three cases implied by 8.4.9(1). If $u = t$ then we can see that $Nd(u)$ and hence $Ct(t)$ hold. If $u \trianglelefteq (c_1, a, v_1), s_1$ then we obtain $Nd(u)$ and hence $Ct(t)$ from the assumption $Ct((c_1, a, v_1), s_1)$. If $u \trianglelefteq (c_2, a, v_2), s_2$ then we obtain $Nd(u)$ and hence $Ct(t)$ from the assumption $Ct((c_2, a, v_2), s_2)$.

(7): Take any $1 \leq j \leq k$ and assume $Ct((c, a, v), s)$. For

$$t := (\boldsymbol{H}_{\underline{j}_m}(c), a, h_j(v)), ((c, a, v), s), 0$$

we wish to prove $Ct(t)$. We thus take any $u > 0$ such that $u \trianglelefteq t$. Since $u \trianglelefteq 0$ leads to contradiction $u = 0$, we consider the two cases implied by 8.4.9(1). If $u = t$ then we can see that $Nd(u)$ and hence $Ct(t)$ hold. If $u \trianglelefteq (c, a, v), s$ we obtain $Nd(u)$, and hence $Ct(t)$, from the assumption $Ct((c, a, v), s)$.

(8): Assume $Ct((c, a, v), s_1)$, $\mu_1[v] < \mu_1[a]$, $Ct((\ulcorner\langle\tau\rangle\urcorner, v, w), s_2)$, and for

$$t := (\boldsymbol{R}(c), a, w), ((c, a, v), s_1), ((\ulcorner\langle\tau\rangle\urcorner, v, w), s_2)$$

we wish to prove $Ct(t)$. We thus take any $u > 0$ such that $u \trianglelefteq t$. We consider the three cases implied by 8.4.9(1). If $u = t$ then we can see that $Nd(u)$ and hence $Ct(t)$ hold. If $u \trianglelefteq (c, a, v), s_1$ or $u \trianglelefteq (\ulcorner\langle\tau\rangle\urcorner, v, w), s_2$ then we obtain $Nd(u)$, and hence $Ct(t)$ from the corresponding assumptions on Ct.

(9): Assume $Ct((c, a, v), s_1)$, $\mu_1[v] \geq \mu_1[a]$, and for

$$t := (\boldsymbol{R}(c), a, 0), ((c, a, v), s_1), 0$$

we wish to prove $Ct(t)$. We thus take any $u > 0$ such that $u \trianglelefteq t$. Since $u \trianglelefteq 0$ leads to contradiction $u = 0$, we consider the two cases implied by 8.4.9(1). If $u = t$ then we can see that $Nd(u)$ and hence $Ct(t)$ hold. If $u \trianglelefteq (c, a, v), s_1$ then we obtain $Nd(u)$, and hence $Ct(t)$, from the assumption $Ct((c, a, v), s_1)$.

(10): Assume $Ct(b, t_1, t_2)$ and for the proof of $Ct(t_1)$ take any $u > 0$ such that $u \trianglelefteq t_1$. We have $u \trianglelefteq b, t_1, t_2$ by 8.4.9(1), $Nd(u)$ from the assumption by (2), and hence $Ct(t_1)$ by (2). We prove $Ct(t_2)$ similarly.

8.4.12 Existence and uniqueness properties for r. We could now introduce the function r by the implicit definition $\exists s\, Ct((\ulcorner\langle\tau\rangle\urcorner, x, r(x)), s)$ and prove that it solves the identity $r(x) = \langle\tau\rangle[[r]^{\mu_1}_x; x]$. In the proof of Theorem 8.4.14 we will introduce instead the n-ary function f directly by an implicit axiom equivalent to

$$\exists s\, Ct((\ulcorner\langle\tau\rangle\urcorner, (x_1, \ldots, x_n, 0), f(x_1, \ldots, x_n)), s)$$

and then solve the identity $f(\vec{x}) = \tau[[f]^{\mu}_{\vec{x}}; \vec{x}]$.

In any case we will need the following properties from which the existence uniqueness conditions for r and f will follow:

$$T_4 \vdash \exists y \exists s\, Ct((\ulcorner\langle\tau\rangle\urcorner, x, y), s) \tag{1}$$

$$T_4 \vdash Ct((c, x, y_1), s_1) \wedge Ct((c, x, y_2), s_2) \to y_1 = y_2 \wedge s_1 = s_2 \;. \tag{2}$$

(1): We first prove the following auxiliary properties:

$$T_4 \vdash \forall v(\mu_1[v] < \mu_1[x] \to \exists y \exists s\, Ct((\ulcorner\langle\tau\rangle\urcorner, v, y), s)) \to \exists y \exists s\, Ct((\ulcorner\rho\urcorner, x, y), s) \;. \tag{3}$$

for the finitely many subterms ρ of $\langle\tau\rangle$ in the order of their construction, i.e. by the meta-theoretical induction on the construction of the special term ρ. So assume the antecedent

$$\forall v(\mu_1[v] < \mu_1[x] \to \exists y \exists s\, Ct((\ulcorner\langle\tau\rangle\urcorner, v, y), s) \tag{4}$$

and continue by the case analysis of the special term ρ where we wish to find a y and s such that $Ct((\ulcorner\rho\urcorner, x, y), s)$.

If $\rho \equiv (x)_{\underline{i}_m}$ where $i < n = \underline{n}_m$ then $\ulcorner\rho\urcorner \equiv \boldsymbol{X}_i$, we use 8.4.11(4), and set $y := (x)_i$, $s := 0, 0$.

If $\rho \equiv \underline{m}_m$ then $\ulcorner\rho\urcorner \equiv \boldsymbol{N}(\underline{m}_m)$, we use 8.4.11(5), and set $y := \underline{m}_m$, $s := 0, 0$.

If $\rho \equiv \rho_1, \rho_2$ then $\ulcorner\rho\urcorner \equiv \boldsymbol{P}(\ulcorner\rho_1\urcorner, \ulcorner\rho_2\urcorner)$ and we have

$$T_4 \vdash (4) \to \exists y \exists s\, Ct((\ulcorner\rho_1\urcorner, x, y), s)$$
$$T_4 \vdash (4) \to \exists y \exists s\, Ct((\ulcorner\rho_2\urcorner, x, y), s)$$

by two inductive meta-hypotheses. We use the assumption (4) and obtain $Ct((\ulcorner\rho_1\urcorner, x, y_1), s_1)$, $Ct((\ulcorner\rho_2\urcorner, x, y_2), s_2)$ for some y_1, y_2, s_1, and s_2. We now use 8.4.11(6) and set $y := y_1, y_2$, $s := ((\ulcorner\rho_1\urcorner, x, y_1), s_1), ((\ulcorner\rho_2\urcorner, x, y_2), s_2)$.

If $\rho \equiv h_j(\rho_1)$ where $1 \leq j \leq k$ then $\ulcorner\rho\urcorner \equiv \boldsymbol{H}_{\underline{j}_m}(\ulcorner\rho_1\urcorner)$ and we have $Ct((\ulcorner\rho_1\urcorner, x, v), s_1)$ for some v and s_1 from the inductive meta-hypothesis. We now use 8.4.11(7) and set $y := h_j(v)$, $s := ((\ulcorner\rho_1\urcorner, x, v), s_1), 0$.

If $\rho \equiv r(\rho_1)$ then $\ulcorner\rho\urcorner \equiv \boldsymbol{R}(\ulcorner\rho_1\urcorner)$ and we have $Ct((\ulcorner\rho_1\urcorner, x, v), s_1)$ for some v and s_1 from the inductive meta-hypothesis. We now consider two cases. If $\mu_1[v] < \mu_1[x]$ we have $Ct((\ulcorner\langle\tau\rangle\urcorner, v, z), s_2)$ for some w and s_2 from (4) and it suffices now to use 8.4.11(8) and set $y := w$, $s := ((\ulcorner\rho_1\urcorner, x, v), s_1), ((\ulcorner\langle\tau\rangle\urcorner, v, w), s_2)$. If $\mu_1[v] \geq \mu_1[x]$ then it suffices to use 8.4.11(9) and set $y := 0$, $s := ((\ulcorner\rho_1\urcorner, x, v), s_1), 0$. This ends the proof of the auxiliary properties (3).

We now prove (1) by measure induction with $\mu_1[x]$. The induction hypothesis is (4) and the property follows from it by (3) with $\rho := \langle\tau\rangle$.

(2): We prove

$$T_4 \vdash \forall c \forall x \forall y_1 \forall y_2 \forall s_2 (Ct((c, x, y_1), s_1) \land Ct((c, x, y_2), s_2) \to y_1 = y_2 \land s_1 = s_2) \tag{5}$$

by complete induction on s_1. So we take any c, x, y_1, y_2, s_2, and assume $Ct((c, x, y_1), s_1)$, $Ct((c, x, y_2), s_2)$. We have $(c, x, y_1), s_1 \trianglelefteq (c, x, y_1), s_1$ by 8.4.9(1),

$$Nd((c, x, y_1), s_1) \tag{6}$$

by 8.4.11(2), $s_1 = t_1, t_2$ for some t_1 and t_2 by 8.4.11(1), and $Ct(t_1)$, $Ct(t_2)$ by 8.4.11(10). We similarly obtain

$$Nd((c, x, y_2), s_2)\ , \tag{7}$$

$s_2 = u_1, u_2$, $Ct(u_1)$, and $Ct(u_2)$ for some u_1 and u_2.

We now consider the cases implied by (6) and (7) where we wish to prove $y_1 = y_2$, $t_1 = u_1$, and $t_2 = u_2$.

If $c = \boldsymbol{X}_i$ for some i then we have $i < n$, $y_1 = (x)_i = y_2$, $t_1 = 0 = u_1$, and $t_2 = 0 = u_2$.

If $c = \boldsymbol{N}(m)$ for some m then we have $y_1 = m = y_2$, $t_1 = 0 = u_1$, and $t_2 = 0 = u_2$.

If $c = \boldsymbol{P}(c_1, c_2)$ for some c_1, c_2 then we have $y_1 = w_1, w_2$, $t_1 = (c_1, x, w_1), s_3$; $t_2 = (c_2, x, w_2), s_4$ for some w_1, w_2, s_3, s_4 by (6) and $y_2 =$

z_1, z_2; $u_1 = (c_1, x, z_1), s_5$; $u_2 = (c_2, x, z_2), s_6$ for some z_1, z_2, s_5, s_6 by (7). Since $s_3 < t_1 < s_1$, we get $w_1 = z_1$; $s_3 = s_5$ from $Ct(t_1)$, $Ct(u_1)$ by IH. Similarly, since $s_4 < t_2 < s_1$, we get $w_2 = z_2$; $s_4 = s_6$ from $Ct(t_2)$, $Ct(u_2)$ by IH. Hence $y_1 = y_2$; $t_1 = u_1$; and $t_2 = u_2$.

If $c = \boldsymbol{H}_i(c_1)$ for some i, c_1 then we have

$$i = \underline{1}_m \wedge y_1 = h_1(w) \vee \ldots \vee i = \underline{k}_m \wedge y_1 = h_k(w) \ ,$$

$t_1 = (c_1, x, w), s_3$; $t_2 = 0$ for some w and s_3 by (6) and

$$i = \underline{1}_m \wedge y_2 = h_1(z) \vee \ldots \vee i = \underline{k}_m \wedge y_2 = h_k(z) \ ,$$

$u_1 = (c_1, x, z), s_4$; $u_2 = 0$ for some z and s_4 by (7). Since $s_3 < t_1 < s_1$, we get $w = z$; $s_3 = s_4$ by IH from $Ct(t_1)$, $Ct(u_1)$. Hence $y_1 = y_2$; $t_1 = u_1$; and $t_2 = u_2$.

If $c = \boldsymbol{R}(c_1)$ for some c_1 then we have $t_1 = (c_1, x, w), s_3$ for some w and s_3 by (6) and $u_1 = (c_1, x, z), s_4$ for some z and s_4 by (7). Since $s_3 < t_1 < s_1$, we get $w = z$, $s_3 = s_4$ by IH from $Ct(t_1)$, $Ct(u_1)$. Hence $t_1 = u_1$. We now consider two cases. If $\mu_1[w] < \mu_1[x]$ then we have $t_2 = \ulcorner\tau\urcorner, w, y_1, s_5$ for some s_5 by (6) and $u_2 = \ulcorner\tau\urcorner, z, y_2, s_6$ for some s_6 by (7). Since $s_5 < t_2 < s_1$, $w = z$, we get $y_1 = y_2$, $s_5 = s_6$ by IH from $Ct(t_2)$, $Ct(u_2)$. Hence $t_2 = u_2$. If $\mu_1[w] \geq \mu_1[x]$ then we have $y_1 = 0 = y_2$ and $t_2 = 0 = u_2$ by (6), (7).

8.4.13 Graph of the arithmetized denotation function for subterms of τ. We will now effectively determine an $(n+2)$-ary formula $\phi[c, \vec{x}, y]$ of $\mathcal{L}_T$ with the free variables among the indicated ones, which is used in the induction axiom 8.4.6(2) of S. The formula can be viewed as the graph of the arithmetized denotation function for the subterms ρ of τ because we will have $\mathbb{N} \vDash \phi[\ulcorner\langle\rho\rangle\urcorner, \vec{x}, y]$ in the standard model of S iff y is the denotation of the term $\rho[[f]^{\mu}_{\vec{x}}, \vec{x}]$ in the assignment $\vec{x}$.

Since T_4 is an extension by definitions of T, the formula $\phi[c, x_1, \ldots, x_n, y]$ is effectively obtained from the formula

$$\exists s\, Ct((c, (x_1, \ldots, x_n, 0), y), s)$$

of $\mathcal{L}_{T_4}$ by translation and in such a way that we have

$$T_4 \vdash \phi[c, x_1, \ldots, x_n, y] \leftrightarrow \exists s\, Ct((c, (x_1, \ldots, x_n, 0), y), s) \tag{1}$$

by the Theorem on Extensions by definition 6.6.2. We will need the following properties of the formula ϕ:

$$T \vdash \exists y \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, y] \tag{2}$$
$$T \vdash \phi[c, \vec{x}, y_1] \wedge \phi[c, \vec{x}, y_2] \rightarrow y_1 = y_2 \tag{3}$$
$$T \vdash \phi[\boldsymbol{X}_{\underline{i-1}_m}, \vec{x}, x_i] \tag{4}$$
$$T \vdash \phi[\boldsymbol{N}(m), \vec{x}, m] \tag{5}$$
$$T \vdash \phi[c_1, \vec{x}, v_1] \wedge \phi[c_2, \vec{x}, v_2] \rightarrow \phi[\boldsymbol{P}(c_1, c_2), \vec{x}, (v_1, v_2)] \tag{6}$$
$$T \vdash \phi[c, \vec{x}, (v_1, \ldots, v_{n_j}, 0)] \rightarrow \phi[\boldsymbol{H}_{\underline{j}_m}(c), \vec{x}, g_j(v_1, \ldots, v_{n_j})] \tag{7}$$
$$\begin{aligned} T \vdash{} & \phi[c, \vec{x}, (v_1, \ldots, v_n, 0)] \wedge \mu[v_1, \ldots, v_n] < \mu[\vec{x}] \wedge \\ & \quad \phi[\ulcorner\langle\tau\rangle\urcorner, v_1, \ldots, v_n, w] \rightarrow \phi[\boldsymbol{R}(c), \vec{x}, w] \end{aligned} \tag{8}$$
$$\begin{aligned} T \vdash{} & \phi[c, \vec{x}, (v_1, \ldots, v_n, 0)] \wedge \mu[v_1, \ldots, v_n] \geq \mu[\vec{x}] \rightarrow \\ & \quad \phi[\boldsymbol{R}(c), \vec{x}, 0] \ . \end{aligned} \tag{9}$$

for every $1 \leq i \leq n$ and $1 \leq j \leq k$. In the following proofs we work in T_4 and use the equivalence (1) without explicitly referring to it. Properties (2) through (9) are thus derived in T_4 but, since they all are in the language $\mathcal{L}_T$, they are also theorems of T because T_4 is conservative over T.

(2): A direct consequence of 8.4.12(1).

(3): A direct consequence of 8.4.12(2).

(4): Take an i s.t. $1 \leq i \leq n$. Since T_4 proves $\underline{i-1}_m < n$ and

$$(x_1, \ldots, x_n, 0)_{\underline{i-1}_m} = x_i \ ,$$

we get $Ct((\boldsymbol{X}_{\underline{i-1}_m}, (x_1, \ldots, x_n, 0), x_i), 0, 0)$ by 8.4.11(4) and then

$$\exists s \, Ct((\boldsymbol{X}_{\underline{i-1}_m}, (x_1, \ldots, x_n, 0), x_i), s) \ .$$

(5): A direct consequence of 8.4.11(5).

(6): A direct consequence of 8.4.11(6).

(7): Take any j s.t. $1 \leq j \leq k$ and assume the antecedent. Then $Ct((c, (x_1, \ldots, x_n, 0), (v_1, \ldots, v_{n_j}, 0)), s_1)$ for some s_1 and we get

$$\exists s \, Ct((\boldsymbol{H}_{\underline{j}_m}(c), (x_1, \ldots, x_n, 0), h_j(v_1, \ldots, v_{n_j}, 0)), s)$$

by 8.4.11(7). We now use 8.4.7(2) to get

$$\exists s \, Ct((\boldsymbol{H}_{\underline{j}_m}(c), (x_1, \ldots, x_n, 0), g_j(v_1, \ldots, v_{n_j})), s) \ .$$

(8): Assume the antecedent. Then $Ct((c, (x_1, \ldots, x_n, 0), (v_1, \ldots, v_n, 0)), s_1)$, $\mu[v_1, \ldots, v_n] < \mu[x_1, \ldots, x_n]$, and $Ct((\ulcorner\langle c\rangle\urcorner, (v_1, \ldots, v_n, 0), w), s_2)$, for some s_1, s_2. We have $\mu_1[(v_1, \ldots, v_n, 0)] < \mu_1[(x_1, \ldots, x_n, 0)]$ by 8.4.7(3) and $\exists s \, Ct((\boldsymbol{R}(c), (x_1, \ldots, x_n, 0), w), s)$ by 8.4.11(8).

(9): This is similar to and somewhat simpler than (8).

8.4.14 Theorem. *If T is a proper extension of PA containing the pairing function (x, y), the case discrimination function $D(x, y, z)$, as well as the characteristic function $<_*$ of the predicate $<$ then an extension of T by course of values recursion with measure is an extension by definition.*

Proof. Let T be as in the theorem, S an extension of T by course of values recursion with measure as in Par. 8.4.6, and S_1 an extension of T by implicit definition with the defining axiom an universal closure of $\phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, f(\vec{x})]$. We have $\mathcal{L}_{S_1} = \mathcal{L}_S$ and S_1 is an extension by definition of T by Thm. 6.6.3 because T proves the existence 8.4.13(2) and uniqueness 8.4.13(3) conditions for f. Clearly

$$S_1 \vdash \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, f(\vec{x})] \ . \tag{1}$$

In order to prove the theorem it suffices to prove that the theories S and S_1 are equivalent.

For the proof $S_1 \vdash S$ we derive auxiliary properties

$$S_1 \vdash \phi[\ulcorner\langle\rho\rangle\urcorner, \vec{x}, \rho[[f]^{\mu}_{\vec{x}}; \vec{x}]] \tag{2}$$

for the finitely many subterms ρ of τ by the meta-induction on the construction of ρ.

If $\rho \equiv x_i$ where $1 \leq i \leq n$ then, since $\ulcorner\langle x_i\rangle\urcorner \equiv \ulcorner(x)_{\underline{i-1}_m}\urcorner \equiv \boldsymbol{X}_{\underline{i-1}_m}$, we have $\phi[\boldsymbol{X}_{\underline{i-1}_m}, \vec{x}, x_i]$ by 8.4.13(4) and we note that $x_i[[f]^{\mu}_{\vec{x}}; \vec{x}] \equiv x_i$.

If $\rho \equiv \underline{m}_m$ then, since $\ulcorner\langle\underline{m}_m\rangle\urcorner \equiv \ulcorner\underline{m}_m\urcorner \equiv \boldsymbol{N}(\underline{m}_m)$, we have $\phi[\boldsymbol{N}(\underline{m}_m), \vec{x}, \underline{m}_m]$ by 8.4.13(5) and we note that $\underline{m}_m[[f]^{\mu}_{\vec{x}}; \vec{x}] \equiv \underline{m}_m$.

If $\rho \equiv g_j(\rho_1, \dots, \rho_{n_j})$ where $1 \leq j \leq k$ we obtain $\phi[\ulcorner\langle\rho_1\rangle\urcorner, \vec{x}, \rho_1[[f]^{\mu}_{\vec{x}}; \vec{x}]]$, $\dots$, $\phi[\ulcorner\langle\rho_{n_j}\rangle\urcorner, \vec{x}, \rho_n[[f]^{\mu}_{\vec{x}}; \vec{x}]]$ by n_j inductive meta-hypotheses. Since for

$$c = \boldsymbol{P}(\ulcorner\langle\rho_1\rangle\urcorner, \dots, \boldsymbol{P}(\ulcorner\langle\rho_{n_j}\rangle\urcorner, \boldsymbol{N}(0)) \dots) \equiv \ulcorner\langle\rho_1\rangle, \dots, \langle\rho_{n_j}\rangle, 0\urcorner$$

we have

$$\ulcorner\langle g_j(\rho_1, \dots, \rho_{n_j})\rangle\urcorner \equiv \ulcorner h_j(\langle\rho_1\rangle, \dots, \langle\rho_{n_j}\rangle, 0)\urcorner \equiv \boldsymbol{H}_{\underline{j}_m}(c) \ ,$$

we get

$$\phi[c, \vec{x}, (\rho_1[[f]^{\mu}_{\vec{x}}; \vec{x}], \dots, \rho_{n_j}[[f]^{\mu}_{\vec{x}}; \vec{x}], 0)]$$

by n_j applications of 8.4.13(6) and then

$$\phi[\boldsymbol{H}_{\underline{j}_m}(c), \vec{x}, g_j(\rho_1[[f]^{\mu}_{\vec{x}}; \vec{x}], \dots, \rho_{n_j}[[f]^{\mu}_{\vec{x}}; \vec{x}])]$$

by 8.4.13(7). We are done since

$$g_j(\rho_1, \dots, \rho_{n_j})[[f]^{\mu}_{\vec{x}}; \vec{x}] \equiv g_j(\rho_1[[f]^{\mu}_{\vec{x}}; \vec{x}], \dots, \rho_{n_j}[[f]^{\mu}_{\vec{x}}; \vec{x}]) \ .$$

If $\rho \equiv f(\rho_1, \dots, \rho_n)$ we obtain $\phi[\ulcorner\langle\rho_1\rangle\urcorner, \vec{x}, \rho_1[[f]^{\mu}_{\vec{x}}; \vec{x}]], \dots, \phi[\ulcorner\langle\rho_n\rangle\urcorner, \vec{x}, \rho_n[[f]^{\mu}_{\vec{x}}; \vec{x}]]$ by n inductive meta-hypotheses. Since for

$$c = \boldsymbol{P}(\ulcorner\langle\rho_1\rangle\urcorner, \dots, \boldsymbol{P}(\ulcorner\langle\rho_n\rangle\urcorner, \boldsymbol{N}(0))\dots) \equiv \ulcorner\langle\rho_1\rangle, \dots, \langle\rho_n\rangle, 0\urcorner$$

we have

$$\ulcorner\langle f(\rho_1, \dots, \rho_n)\rangle\urcorner \equiv \ulcorner r(\langle\rho_1\rangle, \dots, \langle\rho_n\rangle, 0)\urcorner \equiv \boldsymbol{R}(c) \ ,$$

we get

$$\phi[c, \vec{x}, (\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}], 0)]$$

by n applications of 8.4.13(6). We now consider two cases. If

$$\mu[\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]] < \mu[\vec{x}] \tag{3}$$

then from (1) we obtain

$$\phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, f(\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}])] \tag{4}$$

and from 8.4.13(8) $\phi[\boldsymbol{R}(c), \vec{x}, f(\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}])]$. We are done because

$$\begin{aligned} f(\rho_1, \dots, \rho_n)[[f]^\mu_{\vec{x}}; \vec{x}] \equiv D((\mu[\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]] <_* \mu[\vec{x}]), \\ f(\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]), 0) = \\ f(\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]) \ . \end{aligned}$$

If not (3) then we get $\phi[\boldsymbol{R}(c), \vec{x}, 0]$ by 8.4.13(9) and we are done because

$$\begin{aligned} f(\rho_1, \dots, \rho_n)[[f]^\mu_{\vec{x}}; \vec{x}] \equiv D((\mu[\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]] <_* \mu[\vec{x}]), \\ f(\rho_1[[f]^\mu_{\vec{x}}; \vec{x}], \dots, \rho_n[[f]^\mu_{\vec{x}}; \vec{x}]), 0) = 0 \ . \end{aligned}$$

With (2) proved, we derive the defining axiom 8.4.6(1) of S for f: $S_1 \vdash f(\vec{x}) = \tau[[f]^\mu_{\vec{x}}; \vec{x}]$ by 8.4.13(3) from (1) and from (2) where we take the subterm τ of τ for ρ. Since S_1 is proper, it also proves the induction axiom 8.4.6(2) of S by Thm. 8.4.5.

Vice versa, for the proof $S \vdash S_1$ we derive auxiliary properties

$$S \vdash \forall\vec{v}(\mu[\vec{v}] < \mu[\vec{x}] \to \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{v}, f(\vec{y})]) \to \phi[\ulcorner\langle\rho\rangle\urcorner, \vec{x}, \rho[[f]^\mu_{\vec{x}}; \vec{x}]] \tag{5}$$

for the finitely many subterms ρ of τ by the meta-induction on the construction of ρ. So we assume

$$\forall\vec{v}(\mu[\vec{v}] < \mu[\vec{x}] \to \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{v}, f(\vec{y})]) \tag{6}$$

and continue by the case analysis of ρ exactly as in the proof of (2) where the only difference is in the case when $\rho \equiv f(\rho_1, \dots, \rho_n)$ and (3) holds. We obtain (4) from the assumption (6) rather than from (1) as was the case above.

With (5) proved, we derive the defining axiom (1) for f in S_1: $S \vdash \phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, f(\vec{x})]$ by the induction axiom with measure 8.4.6(2) where we assume (6) and use (5) with the subterm τ of τ for ρ to get $\phi[\ulcorner\langle\tau\rangle\urcorner, \vec{x}, \tau[[f]^\mu_{\vec{x}}; \vec{x}]]$. We now get (1) from the defining axiom 8.4.6(1) for f in S. □

9. Proof Theory of PA

UNFINISHED Gentzen was first. but we need the strength of induction schemas for the characterization purposes by means of provably recursive functions.

Part II

Computer Programming

10. Clausal Language

10.1 Clausal Definitions

10.1.1 Characteristic terms of formulas. UNFINISHED only boolean combinations of formulas

For a formula ϕ its characteristic term is denoted by ϕ_*

$$\vdash_{\mathrm{PA}} \phi \rightarrow \phi_*(\vec{x}) = 1 \tag{1}$$

$$\vdash_{\mathrm{PA}} \phi \leftrightarrow \phi_*(\vec{x}) > 0 \tag{2}$$

10.1.2 Generalized terms. *Generalized* terms extend the language of (an extension of) PA with new constructs which will be used purely in the meta-theory to describe the introduction of functions and predicates into PA by clausal definitions. Generalized terms will also be used in the description of computation of such functions and predicates.

Generalized terms will not be used for the presentation purposes. Clausal definitions are presented to humans in the form of clauses which are formulas derived from generalized terms, and which are in the language of PA.

We do not intend to extend the language of first-order theories to contain the generalized terms because some of the terms, by being *variable binding* constructs similar to the μ-construct used for the introduction of functions into PA by minimalization, go beyond the usual syntax of terms in mathematical logic. The reader will recall that the μ term in a definition by minimalization $f(\vec{x}) = \mu_y[\phi[\vec{x}, y]]$ does not belong to the language of first-order terms either. The sole purpose of this definition is graphically illustrate the defining axiom for f:

$$\phi[\vec{x}, f(\vec{x})] \land \forall z(z < f(\vec{x}) \rightarrow \phi[\vec{x}, z]) \ .$$

Generalized terms are similar to the μ-terms in that some of them bind variables and contain formulas , they also offer a succint notation in which to describe the semantics and computation (syntax) of clausal definitions, but they are definitely much less readable than the clauses derived from them.

We now describe the syntax of generalized terms which, unlike the complex syntax which seems to be de rigueur in the definitions of programming languages, is in the usual simple style of logic. The reader should bear on mind that this syntax is not meant for presentation to humans. We use, in addition to the usual possibly subscripted meta-variables, x, y, $\ldots$ ranging over variables, τ, ρ, $\ldots$ ranging over terms of PA, c, d, $\ldots$ ranging over constant symbols of PA, ϕ, ψ, $\ldots$ ranging over formulas of PA, the meta-variables α, β, $\ldots$ to range over generalized terms and p, m to range over meta-theoretic constants for natural numbers. The class of generalized terms is defined relatively to the current extension of PA as the minimal class containing the kinds of terms listed in Fig. 10.1.

$$\begin{gathered}
\rho \\
Neg(\phi, \alpha_1, \alpha_2) \\
Dich(\tau_1, \tau_2, \alpha_1, \alpha_2) \\
Trich(\tau_1, \tau_2, \alpha_1, \alpha_2, \alpha_3) \\
Const^m_{c_1,\ldots,c_m}(\tau, \alpha_1, \ldots, \alpha_m, \alpha_{m+1}) \\
Let_y(\tau, \alpha) \\
Pair_{y,z}(\tau, \alpha_1, \alpha_2) \\
Cart^m_{x_1,\ldots,x_m}(\tau, \alpha_1, \alpha_2) \\
Mon^m_y(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\
Padica^{p,m}_{y,z}(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\
Padicb^{p,m}_{y,z}(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\
\textbf{UNFINISHED}\ \text{constructor discr}
\end{gathered}$$

Fig. 10.1. Syntax of generalized terms

It would not be too difficult to assign denotations in the standard model of PA to the generalized terms but, since they will play role only in the meta-theory, we will not do it. We will instead translate the generalized terms to the terms of PA by defining a meta-theoretic function $\alpha^\star$ yielding a term of PA whose interpretation can be understood as the intended interpretation of α. The translation function is defined by induction on the construction of generalized terms α to satisfy the term identities listed in Fig. 10.2.

Generalized terms are never used in the formulas of theorems and proofs of PA except in the translated form $\alpha^\star$ which stands for a term of PA. We do not discuss the syntax and translation of generalized terms any further here because we will study them in detail one kind at a time below.

The logician readers can be now expected to ask a question which is quite natural to him. Why so many kinds of generalized terms are needed? Do we not, when studying a formal system, choose as simple syntax as possible in order to manage the study of the formal system? The answer is quite obvious to computer scientist readers. We need many kinds of terms for pragmatic reasons of good expressive power expected by computer programmers and

$$
\begin{gathered}
\mathit{Neg}(\psi, \alpha_1, \alpha_2)^\star \equiv D(\psi_*, \alpha_1^\star, \alpha_2^\star) \\
\mathit{Dich}(\tau_1, \tau_2, \alpha_1, \alpha_2)^\star \equiv D((\tau_1 \leq_* \tau_2), \alpha_1^\star, \alpha_2^\star) \\
\mathit{Trich}(\tau_1, \tau_2, \alpha_1, \alpha_2, \alpha_3)^\star \equiv D((\tau_1 <_* \tau_2), \alpha_1^\star, D((\tau_2 <_* \tau_1), \alpha_3^\star, \alpha_2^\star)) \\
\mathit{Const}^m_{c_1,\ldots,c_m}(\tau, \alpha_1, \ldots, \alpha_m, \alpha_{m+1})^\star \equiv \\
D((\tau =_* c_1), \alpha_1^\star, \ldots, D((\tau =_* c_m), \alpha_m^\star, \alpha_{m+1}^\star)) \\
\mathit{Let}_y(\tau, \alpha)^\star \equiv \alpha^\star{}_y[\tau] \\
\mathit{Pair}_{y,z}(\tau, \alpha_1, \alpha_2)^\star \equiv D(\tau, \alpha^\star_1{}_y[H(\tau)]_z[T(\tau)], \alpha_2^\star) \\
\mathit{Cart}^m_{x_1,\ldots,x_{n-1},x_m}(\tau, \alpha_1, \alpha_2)^\star \equiv \\
D(T^{m \dot- 2}(\tau), \alpha^\star_1{}_{x_1}[H\,T^0(\tau)]\ldots{}_{x_{m-1}}[H\,T^{m \dot- 2}(\tau)]_{x_m}[T^{m \dot- 1}(\tau)], \alpha_2^\star)
\end{gathered}
$$

UNFINISHED monadic translation

UNFINISHED p-adic I translation

UNFINISHED p-adic II translation

UNFINISHED constructor pattern translation.

Fig. 10.2. Translation of generalized terms to the terms of PA

also because different kinds of terms yield **efficient** reduction sequences, i.e. efficient computations. If we were only after the **effectivity** as studied in the theory of computability then we would not need to extend the class of PA terms at all.

10.1.3 Outline of clausal definitions. When we say that an n-ary function symbol f is introduced into PA by a *clausal definition* (see Par. 10.1.18 for the precise definition) then we mean a recursive extension with f of PA with axioms the universal closures of formulas from a finite set

$$S = \{\phi_1, \ldots, \phi_k\} \tag{1}$$

which are in the language of PA and are called *clauses* (cf Par. 10.1.4). The set S is obtained from a measure function m and a formula

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

by an effective meta-theoretic process called *unfolding* (cf Par. 10.1.17). The unfolding process satisfies the following:

$$\vdash_{\mathrm{PA}} \forall\vec{x}\, f(\vec{x}) = \alpha^\star[[f]^m_{\vec{x}}; \vec{x}] \leftrightarrow \forall\phi_1 \land \ldots \land \forall\phi_k \tag{3}$$

where $\forall\phi_i$ stands for an universal closure of the clause $\phi_i \in S$. This means that the function f can be also introduced into PA by course of values recursion with measure:

$$f(\vec{x}) = \alpha^\star[[f]^m_{\vec{x}}; \vec{x}]\ . \tag{4}$$

and then the formulas from S are provable as theorems.

If the definition (4) is recursive, i.e. if applications of f occur in the generalized term α and α is *regular* in m (cf Par. 10.2.1) then we additionally have:

$$\vdash_{\mathrm{PA}} f(\vec{x}) = \alpha^{\star}[f;\vec{x}] \tag{5}$$

which means that the restriction tests $m(\vec{\rho}) < m(\vec{x})$ are always satisfied and so they are superfluous in the recursive applications $f(\vec{\rho})$ in α.

10.1.4 Clauses. *Clauses* are *Horn* formulas, i.e. implications with atomic formulas in the consequent. Every clause can be presented in a form $\psi_1 \wedge \ldots \wedge \psi_k \rightarrow f(\vec{\tau}) = \alpha$. Clauses used in definitions are in logic programming customarily written with converse implications:

$$f(\vec{x}) = \alpha \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \; . \tag{1}$$

We adopt this custom and treat such a formula only as a notational variant of $\psi_1 \wedge \ldots \wedge \psi_k \rightarrow f(\vec{x}) = \alpha$.

We do not exclude the case when $k = 0$ when the *body* $\psi_1 \wedge \ldots \wedge \psi_k$ of the clause is empty and then the clause is written as $f(\vec{x}) = \alpha$.

A clause ϕ of a form (1) with α a generalized term can be used in a proof only with the generalized term translated away, i.e. as the formula

$$f(\vec{x}) = \alpha^{\star} \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

denoted by $\phi^{\star}$ and which is in the language of PA. We denote by $\forall\phi^{\star}$ any of the universal closures of the formula $\phi^{\star}$.

10.1.5 Unfolding invariant. We will discuss in detail the syntax and translation of all kinds of generalized terms in the following paragraps. We will also discuss with each kind of α the *unfolding step* which leads from a clause ϕ of a form 10.1.4(1) to a finite set of clauses $\phi_1, \ldots, \phi_l$ satisfying the following *unfolding invariant*

$$\vdash_{\mathrm{PA}} \forall\phi^{\star} \leftrightarrow \forall\phi_1^{\star} \wedge \ldots \wedge \forall\phi_l^{\star} \; . \tag{1}$$

The unfolding process is started from an initial clause of a form 10.1.3(2) and eventually leads to the set of clauses 10.1.3(1). The invariant is proved under the assumption that all function and predicate symbols applied in (1) have been introduced into PA. In particular we assume that the function symbol f has been introduced, for instance, by 10.1.3(4).

10.1.6 Negation discrimination terms. *Negation* (discrimination) terms have the following syntax:

$$Neg(\psi, \alpha_1, \alpha_2)$$

and their translation is:

$$Neg(\psi, \alpha_1, \alpha_2)^\star \equiv D(\psi_*, \alpha_1^\star, \alpha_2^\star) \ . \tag{1}$$

The clause

$$f(\vec{x}) = Neg(\psi, \alpha_1, \alpha_2) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

unfolds to

$$f(\vec{x}) = \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \psi \tag{3}$$
$$f(\vec{x}) = \alpha_2 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \neg\psi \tag{4}$$

and the following property clearly implies the unfolding invariant 10.1.5(1):

$$\vdash_{\text{PA}} (2)^\star \leftrightarrow (3)^\star \wedge (4)^\star \ . \tag{5}$$

In the direction ($\rightarrow$) we assume $(2)^\star$, $\psi_1, \ldots, \psi_k$, and consider two cases. If ψ then $(4)^\star$ holds trivially and, since $\psi_* > 1$ by 10.1.1(2), we have

$$f(\vec{x}) = Neg(\psi, \alpha_1, \alpha_2)^\star \equiv D(\psi_*, \alpha_1^\star, \alpha_2^\star) \overset{8.4.1(1)}{=} \alpha_1^\star \ .$$

Hence $(3)^\star$. If $\neg\psi$ then then $(3)^\star$ holds trivially and, since $\psi_* = 0$ by 10.1.1(2), we have

$$f(\vec{x}) = Neg(\psi, \alpha_1, \alpha_2)^\star \equiv D(\psi_*, \alpha_1^\star, \alpha_2^\star) \overset{8.4.1(2)}{=} \alpha_2^\star \ .$$

and hence $(4)^\star$.

In the direction ($\leftarrow$) we assume $(3)^\star$, $(4)^\star$, $\psi_1, \ldots, \psi_k$, and consider two cases. If ϕ then

$$f(\vec{x}) \overset{(3)^\star}{=} \alpha_1^\star \overset{10.1.1(2),8.4.1(1)}{=} D(\psi_*, \alpha_1^\star, \alpha_2^\star) \equiv Neg(\psi, \alpha_1, \alpha_2)^\star \ .$$

The case $\neg\phi$ is similar and uses the assumption $(4)^\star$.

Proofs of unfolding invariants $(2)^\star$ for the remaining forms of generalized terms as discussed below are quite similar to the one just done. They rely mostly on Property 10.1.1(2) expressing the relation between the characteristic terms and their formulas and on Properties 8.4.1(1)(2) of the case discrimination function D. For this reason we will mostly just sketch the proofs and leave the details to the interested reader.

UNFINISHED Current restrictions in CL: ϕ only $\tau_1 = \tau_2$, $R(\vec{\rho})$.

10.1.7 Dichotomy discrimination terms. *Dichotomy* (discrimination) terms have the following syntax:

$$Dich(\tau_1, \tau_2, \alpha_1, \alpha_2)$$

and their translation is

$$\mathit{Dich}(\tau_1,\tau_2,\alpha_1,\alpha_2)^\star \equiv D((\tau_1 {\le_*} \tau_2),\alpha_1^\star,\alpha_2^\star)\ . \tag{1}$$

The clause

$$f(\vec{x}) = \mathit{Dich}(\tau_1,\tau_2,\alpha_1,\alpha_2) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

unfolds to

$$f(\vec{x}) = \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau_1 \le \tau_2 \tag{3}$$
$$f(\vec{x}) = \alpha_2 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau_1 > \tau_2\ . \tag{4}$$

The following easy to prove property implies the unfolding invariant 10.1.5(1):

$$\vdash_{\mathrm{PA}} f(\vec{x}) = D((\tau_1 {\le_*} \tau_2),\alpha_1^\star,\alpha_2^\star) \leftrightarrow$$
$$(\tau_1 \le \tau_2 \to f(\vec{x}) = \alpha_1) \wedge (\tau_1 > \tau_2 \to f(\vec{x}) = \alpha_2)\ . \tag{5}$$

10.1.8 Trichotomy discrimination terms. UNFINISHED *Trichotomy* (discrimination) terms have the following syntax:

$$\mathit{Trich}(\tau_1,\tau_2,\alpha_1,\alpha_2,\alpha_3)$$

Translation:

$$\mathit{Trich}(\tau_1,\tau_2,\alpha_1,\alpha_2,\alpha_3)^\star \equiv D((\tau_1 {<_*} \tau_2),\alpha_1^\star, D((\tau_2 {<_*} \tau_1),\alpha_3^\star,\alpha_2^\star)) \tag{1}$$

In clausal contexts (2) they abbreviate clauses (3):

$$f(\vec{x}) = \mathit{Trich}(\tau_1,\tau_2,\alpha_1,\alpha_2,\alpha_3) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

$$\begin{aligned} f(\vec{x}) &= \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau_1 < \tau_2 \\ f(\vec{x}) &= \alpha_2 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau_1 = \tau_2 \\ f(\vec{x}) &= \alpha_3 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau_1 > \tau_2 \end{aligned} \tag{3}$$

10.1.9 Discrimination on constants. UNFINISHED *Constant discrimination* terms have the following syntax:

$$\mathit{Const}^m_{c_1,\ldots,c_m}(\tau,\alpha_1,\ldots,\alpha_m,\alpha_{m+1}) \tag{1}$$

where c_1, ..., c_m are symbols for constants.

Translation:

$$\mathit{Const}^m_{c_1,\ldots,c_m}(\tau,\alpha_1,\ldots,\alpha_m,\alpha_{m+1})^\star \equiv$$
$$D((\tau {=_*} c_1),\alpha_1^\star,\ldots,D((\tau {=_*} c_m),\alpha_m^\star,\alpha_{m+1}^\star)) \tag{2}$$

In clausal contexts (3) they abbreviate clauses (4):

$$f(\vec{x}) = \mathit{Const}^m_{c_1,\ldots,c_m}(\tau,\alpha_1,\ldots,\alpha_m,\alpha_{m+1}) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{3}$$

$$\begin{aligned} f(\vec{x}) &= \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = c_1 \\ &\ \vdots \\ f(\vec{x}) &= \alpha_m \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = c_m \\ f(\vec{x}) &= \alpha_{m+1} \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau \ne c_1 \wedge \ldots \wedge \tau \ne c_m \end{aligned} \tag{4}$$

10.1.10 Assignment terms. *Assignment* (let) terms have the following syntax:

$$Let_y(\tau, \alpha)$$

where the variable y is bound in α and may not occur in τ. The translation is

$$Let_y(\tau, \alpha)^\star \equiv \alpha^\star{}_y[\tau] \ . \tag{1}$$

The clause

$$f(\vec{x}) = Let_y(\tau, \alpha) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

unfolds to

$$f(\vec{x}) = \alpha \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = y \tag{3}$$

and the following property which follows from **UNFINISHED** implies the unfolding invariant 10.1.5(1):

$$\vdash_{\text{PA}} \ f(\vec{x}) = \alpha^\star{}_y[\tau] \leftrightarrow \forall y(\tau = y \rightarrow f(\vec{x}) = \alpha^\star) \ . \tag{4}$$

10.1.11 Discrimination on pair patterns. *Pair discrimination* terms have the following syntax:

$$Pair_{y,z}(\tau, \alpha_1, \alpha_2)$$

where the variables y, z are bound in α_1 and may not occur in τ. The translation is

$$Pair_{y,z}(\tau, \alpha_1, \alpha_2)^\star \equiv D(\tau, \alpha^\star_{1y}[H(\tau)]_z[T(\tau)], \alpha^\star_2) \tag{1}$$

The clause

$$f(\vec{x}) = Pair_{x,y}(\tau, \alpha_1, \alpha_2) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

unfolds to the clauses

$$f(\vec{x}) = \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = y, z \tag{3}$$
$$f(\vec{x}) = \alpha_2 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = 0 \tag{4}$$

and the following property implies the unfolding invariant 10.1.5(1):

$$\begin{aligned}\vdash_{\text{PA}} \ & f(\vec{x}) = D(\tau, \alpha^\star_{1y}[H(\tau)]_z[T(\tau)], \alpha^\star_2) \leftrightarrow \\ & \qquad \forall y \forall z(\tau = y, z \rightarrow f(\vec{x}) = \alpha^\star_1) \wedge (\tau = 0 \rightarrow f(\vec{x}) = \alpha^\star_2) \ .\end{aligned} \tag{5}$$

The property is proved in the direction $(\rightarrow)$ by assuming

$$f(\vec{x}) = D(\tau, \alpha^\star_{1y}[H(\tau)]_z[T(\tau)], \alpha^\star_2) \ . \tag{6}$$

The first conjuct is proved by taking any y, z and assuming $\tau = y, z$. We then have $H(\tau) = y$ and $T(\tau) = z$ and so

$$f(\vec{x}) \overset{(6)}{=} \alpha^\star_{1\,y}[H(\tau)]_z[T(\tau)] \overset{6.1.3(1)}{=} \alpha^\star_1 \,.$$

The second conjunct is proved by assuming $\tau = 0$ and deriving $f(\vec{x}) = \alpha^\star_2$ from (6).

In the direction $(\leftarrow)$ we assume the two conjucts and consider the two cases implied by 8.3.14(4). If $\tau = 0$ then we obtain

$$f(\vec{x}) = \alpha^\star_2 = D(\tau, \alpha^\star_{1\,y}[H(\tau)]_z[T(\tau)], \alpha^\star_2)$$

from the second assumption. If $\tau = y, z$ for some y, z then, since $y = H(\tau)$ and $z = T(\tau)$, we have

$$f(\vec{x}) = \alpha^\star_1 \overset{6.1.3(1)}{=} \alpha^\star_{1\,y}[H(\tau)]_z[T(\tau)] = D(\tau, \alpha^\star_{1\,y}[H(\tau)]_z[T(\tau)], \alpha^\star_2)$$

from the first assumption.

10.1.12 Discrimination on cartesian patterns. *Cartesian* (discrimination) terms have the following syntax:

$$Cart^m_{x_1,\ldots,x_m}(\tau, \alpha_1, \alpha_2)$$

where the variables $x_1, \ldots, x_m$ ($n \geq 3$) are bound in α_1 and may not occur in τ.

Translation:

$$\begin{aligned} &Cart^m_{x_1,\ldots,x_{n-1},x_m}(\tau, \alpha_1, \alpha_2)^\star \equiv \\ &\quad D(T^{m \dot{-} 2}(\tau), \alpha^\star_{1\,x_1}[H\,T^0(\tau)]\ldots_{x_{m-1}}[H\,T^{m \dot{-} 2}(\tau)]_{x_m}[T^{m \dot{-} 1}(\tau)], \alpha^\star_2) \end{aligned} \tag{1}$$

In clausal contexts (2) they abbreviate clauses (3):

$$f(\vec{x}) = Cart^m_{x_1,\ldots,x_m}(\tau, \alpha_1, \alpha_2) \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \tag{2}$$

$$\begin{aligned} f(\vec{x}) &= \alpha_1 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \tau = x_1, \ldots, x_m \\ f(\vec{x}) &= \alpha_2 \leftarrow \psi_1 \wedge \ldots \wedge \psi_k \wedge \neg\exists x_1 \ldots \exists x_m\, \tau = x_1, \ldots, x_m \end{aligned} \tag{3}$$

Restriction: not available in the current version of CL.

10.1.13 Discrimination on monadic patterns. UNFINISHED

10.1.14 Discrimination on p-adic patterns I. UNFINISHED

10.1.15 Discrimination on p-adic patterns I. UNFINISHED

10.1.16 Discrimination on constructor patterns. UNFINISHED

10.1.17 Unfolding. We now define a meta-theoretic *unfolding* function Cl taking a finite set T of clauses for f of a form 10.1.4(1) and yielding a finite set of clauses without generalized terms. The *size* of a generalized term α is the total number of its subterms which are not in the language of PA. If α is in the language of PA then its size is 0. The size of a clause ϕ of a form 10.1.4(1) is the size of α.

The function is defined by recursion on the maximum of sizes of clauses in T. If the maximum is 0 then the clauses of T are in the language of PA and we define $Cl(T) = T$. Otherwise there is a finite number of clauses in T with the maximum size $n > 0$. We form a finite set of clauses T_1 by including in it all clauses of T with the size $< n$ and adding to it for each clause $\phi \in T$ with the size n the clauses $\phi_1, \ldots, \phi_k$ obtained from ϕ by one unfolding step. As these clauses have sizes $< n$, the set T_1 has the maximum of sizes of its clauses $< n$ and we define $Cl(T) = Cl(T_1)$.

By (meta-theoretical) induction on the maximum of sizes of clauses in T we easily prove from the unfolding invariant 10.1.5(1) the following basic property of the unfolding function:

$$\vdash_{\mathrm{PA}} \bigwedge_{\phi \in T} \forall \phi^\star \leftrightarrow \bigwedge_{\phi \in Cl(T)} \forall \phi^\star \; . \tag{1}$$

The reader will note that the unfolding process works with concretely presented objects and that the function Cl is effective.

10.1.18 Clausal definitions of functions. Let f be an n-ary function symbol, $\vec{x}$ an n-tuple of variables, and m an n-ary function which has been introduced into PA. The generalized term $\alpha[f, \vec{x}]$ is *suitable* for the clausal definition of f if

- all function and predicate symbols other than f applied in $\alpha^\star$ have been introduced into PA,
- the function symbol f is not in the current language of PA,
- if α applies f then $n \geq 1$ otherwise $n \geq 0$,
- the free variables of α are among $\vec{x}$,
- for every subterm β of α no free variable of β is bound in it.

The last condition means that the variables $\vec{x}$ are not used as bound variables in α and no bound variable in α is bound again in its scope. The reader will note that the first condition guarantees not only that all function symbols other than f applied in the generalized term α are in PA but also that the auxiliary function symbols introduced by the translation into the term $\alpha^\star$ are in PA. The conditions on α guarantee that $\alpha^\star[f; \vec{x}]$, which is a term of PA, is suitable for the definition of f by course of values recursion with measure m.

The extension of PA with the new function symbols f and with the universal closures of the clauses

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

as defining axioms is called a *clausal extension* of PA. We also say that f has been introduced into PA by a *clausal definition*.

10.1.19 Theorem. UNFINISHED *clausal definitions of functions are recursive extensions.*

10.1.20 Clausal Definitions of Predicates. Clausal definitions can be used to introduce predicates into PA. This is done by modifying clausal definitions of their characteristic functions.

Let R be an n-ary predicate symbol, f an n-ary function symbol, $\vec{x}$ an n-tuple of variables, and m an n-ary function which has been introduced into PA. The generalized term $\alpha[f, \vec{x}]$ is *suitable* for the clausal definition of R if

- α is suitable for the clausal definition of f,
- α does not apply R,
- the predicate symbol R is not in the current language of PA,
- all recursive applications of f in α (if any) are as atomic subformulas $f(\vec{\tau}) = 1$ ($\vec{\tau}$ does not apply f) of formulas ϕ which are parts of of subterms $Neg(\phi, \alpha_1, \alpha_2)$ of α,
- the generalized term α is built up from the constituents listed in Fig. 10.1 where the terms ρ are restricted to the terms 0 or 1.

The extension of PA with the new predicate symbols R and with the universal closures of the formulas from the finite set S_1 as defining axioms is called a *clausal extension* of PA if the set S_1 is constructed from the set

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

by replacing in every $\phi \in S$

- the atomic formula in its head which, on accord of the last condition above, must be either $f(\vec{x}) = 0$ or $f(\vec{x}) = 1$ by the application $R(\vec{x})$ or $\neg R(\vec{x})$ respectively,
- by replacing every atomic formula with applications of f in the body of ϕ which, on accord of the fourth condition above, is of a form

  $$D((m(\vec{\tau}) {<_*} m(\vec{x})), f(\vec{\tau}), 0) = 1$$

  by the formula

  $$m(\vec{\tau}) < m(\vec{x}) \land R(\vec{\tau}) \ .$$

We also say that R has been introduced into PA by a *clausal definition*.

The reader worrying that our conditions on clausal definitions of predicates are too strict in the sense that they might exclude a clausal definition of a predicate R from the clausal definition $f(\vec{x}) = \alpha[[f]^m_{\vec{x}}; \vec{x}]$ of its characteristic function where α is suitable for f but not for R should note that the function

f is 0-1 valued and so the term α can be transformed to a suitable term β such that $\vdash_{\mathrm{PA}} \alpha^\star = \beta^\star$. The tranformation is simple because, if for instance, we have for a term $\tau_1[z]$ the generalized term $\mathit{Dich}(\tau_1[f(\vec{\sigma})], \tau_2, \alpha_1, \alpha_2)$ as a subterm of α then we can replace it in β by the generalized term

$$\mathit{Neg}((f(\vec{\sigma}) = 1), \mathit{Dich}(\tau_1[1], \tau_2, \alpha_1, \alpha_2), \mathit{Dich}(\tau_1[0], \tau_2, \alpha_1, \alpha_2)) \ .$$

In this way we can deal with every 'misplaced' recursive application of f in α except when it is in a subterm ρ of α where ρ should be either 0 or 1. The situation is remedied in the same way when ρ does not apply f by replacing ρ in β by the generalized term:

$$\mathit{Neg}(\rho, 1, 0) \ .$$

10.1.21 Theorem. UNFINISHED *clausal definitions of predicates are recursive extensions.*

10.1.22 Presentation of clauses. UNFINISHED Back substitution
UNFINISHED Order of clauses
UNFINISHED defaults
UNFINISHED renaming of bound variables
UNFINISHED alignment to the left as alert that default axioms may be missing.

10.1.23 Refinement versus syntax analysis.

10.2 Regular Clausal Definitions

10.2.1 Outline of conditions of regularity. UNFINISHED we need a stronger definition because of computability. Show with an example.

10.2.2 Conditions of regularity for clauses. Suppose that f is an n-ary function symbol which we wish to introduce into PA by a clausal definition with a measure m. We assign to every clause of a form 10.1.4(1):

$$f(\vec{x}) = \alpha \leftarrow \psi_1 \land \ldots \land \psi_k \tag{1}$$

its *condition of regularity with respect to* m which is the formula $\top$ or a formula of the language of PA of the form

$$\psi_1 \land \ldots \land \psi_k \to m(\vec{\rho}_1) < m(\vec{x}) \land \ldots \land m(\vec{\rho}_l) < m(\vec{x}) \ . \tag{2}$$

If $k = 0$ then (2) stands just for

$$m(\vec{\rho}_1) < m(\vec{x}) \land \ldots \land m(\vec{\rho}_l) < m(\vec{x}) \ .$$

The form of the condition of regularity depends on the form of the generalized term α according to Fig. 10.1. We will assign below a formula ϕ_0 of PA to each α. If τ does not apply f then the condition of regularity is $\top$. Otherwise it is the formula (2) where the n-tuples of terms $\vec{\rho}_1, \ldots, \vec{\rho}_l$ are the arguments of all applications of $f(\vec{\rho}_1), \ldots, f(\vec{\rho}_l)$, occurring in the order left to right and inside out in ϕ_0.

If $\alpha \equiv \rho$ with ρ a term of PA then we set $\phi_0 :\equiv \rho = 0$. If $\alpha \equiv \mathit{Neg}(\phi, \alpha_1, \alpha_2)$ then we set $\phi_0 :\equiv \phi$. If $\alpha \equiv \mathit{Dich}(\tau_1, \tau_2, \alpha_1, \alpha_2)$ or $\alpha \equiv \mathit{Trich}(\tau_1, \tau_2, \alpha_1, \alpha_2, \alpha_3)$ then we set $\phi_0 :\equiv \tau_1 = \tau_2$. If α is one of

$$\begin{gathered} \mathit{Const}^m_{c_1,\ldots,c_m}(\tau, \alpha_1, \ldots, \alpha_m, \alpha_{m+1}) \\ \mathit{Let}_y(\tau, \alpha_1) \\ \mathit{Pair}_{y,z}(\tau, \alpha_1, \alpha_2) \\ \mathit{Cart}^m_{x_1,\ldots,x_m}(\tau, \alpha_1, \alpha_2) \\ \mathit{Mon}^m_y(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\ \mathit{Padica}^{p,m}_{y,z}(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\ \mathit{Padicb}^{p,m}_{y,z}(\tau, \alpha_0, \alpha_1, \ldots, \alpha_m) \\ \textbf{UNFINISHED} \text{ constructor discr} \end{gathered}$$

then we set $\phi_0 :\equiv \tau = 0$.

10.2.3 Regularity function. We now define a meta-theoretic *regularity* function $\mathit{Reg}(m, T)$ where m is an n-ary function symbol, T a finite set of clauses for the n-ary function symbol f of a form 10.1.4(1). The function yields a finite set of formulas of PA. and it is defined by recursion on the maximum of sizes of clauses in T. There is a finite number of clauses in T with the maximum size $n \geq 0$. We form a finite set of clauses T_1 by including in it all clauses of T with the size $< n$. Let S be the finite set of conditions of regularity w.r.t. m assigned to each of the clauses $\phi \in T$ with the size n. If $n = 0$ we define $\mathit{Reg}(m, T) = S$ and $\mathit{Reg}(m, T) = S \cup \mathit{Reg}(m, T_1)$ otherwise. It should be clear that the regularity function Reg is effective.

10.2.4 Conditions of regularity for clausal definitions of functions. UNFINISHED

10.2.5 Theorem. UNFINISHED *If the definition is regular then* $\vdash_{PA} f(\vec{x}) = \alpha^\star[f; \vec{x}]$.

10.2.6 Regular Clausal definitions. UNFINISHED we need some theorem on regularity.

UNFINISHED special cases when cond of regularity is in the language of PA.

10.3 The Strength of Clausal Definitions

11. Computation of Clausal Programs

Until now we were interested only in the definability of functions and predicates over natural numbers. We will now investigate questions of their effective computability.

11.1 Computation over Monadic Numerals

11.1.1 Monadic numerals. We recall the notation

$$\underline{n}_m \equiv 0\overbrace{{}'\ldots{}'}^{n}$$

introduced in Par. 8.4.3 for monadic numerals. Monadic numerals are the least class of terms containing the constant 0 and with every term ρ containing also the term ρ'.

11.1.2 Generalized terms for monadic numerals. The basic generalized term for the definition of clausal programs operating over monadic numerals is

$$Mon_x(\tau, \alpha_1[x], \alpha_2) = z \leftrightarrow \exists x(\tau = x' \land \alpha_1[x] = z) \lor \tau = 0 \land \alpha_2 = z\ .$$

11.1.3 Monadic clausal definitions. *Monadic* functions and predicates are defined by monadic clausal definitions. We have two classes of functions, primitive recursive and μ-recursive.

All primitive recursive functions are provably recursive in PA but not all μ-recursive functions are such.

11.1.4 Reductions over monadic numerals.

$$\begin{aligned} Mon_x(0, \alpha_1[x], \alpha_2) &\blacktriangleright \alpha_2 \\ Mon_x(\rho', \alpha_1[x], \alpha_2) &\blacktriangleright \alpha_1[\rho]\ . \end{aligned}$$

11.1.5 Computability of functions defined by monadic clausal definitions. Both primitive recursive and μ-recursive functions can be effectively computed by reductions.

11.2 Computation over Binary Numerals

If we were interested only in the effective computability of functions and predicates then we would not need more than monadic functions and predicates. Since we are discussing computer programming, we are also interested in the efficiency of computed programs. Monadic computation is exponentially slower than it should be. Computationally optimal is the so-called recursion on notation of which binary notation is the most well-known.

11.2.1 Binary numerals. *Binary numerals* are the least set of terms containing the constant 0, with every term ρ also the term $\rho\mathbf{1}$, and with every term $\rho \not\equiv 0$ also the term $\rho\mathbf{0}$. Thus the term $0\mathbf{0}$ is not a binary numeral.

11.2.2 Generalized terms for binary numerals. The basic generalized terms for clausal programs operating over binary numerals are

$$\begin{gathered} \mathit{Bin}([\tau = x\mathbf{0}; \alpha_1]_x, [\tau = x\mathbf{1}; \alpha_2]_x) \\ \mathcal{D}([\tau > 0; \alpha_1], [\tau = 0; \alpha_2]) \ . \end{gathered}$$

11.2.3 Binary clausal definitions. *Binary* functions and predicates are defined by binary clausal definitions.

11.2.4 Fast binary arithmetic. Basic operations and comparisons.

11.2.5 Binary pairing function. $P_b(x, y)$

11.2.6 Binary projections. $H_b(x)$, $T_b(x)$

11.2.7 Characterization of binary functions and predicates. Binary functions and predicates are exactly the monadic ones.

11.2.8 Reductions over binary numerals.

$$\begin{aligned} \mathit{Bin}([0 = x\mathbf{0}; \alpha_1]_x, [0 = x\mathbf{1}; \alpha_2]_x) &\blacktriangleright \alpha_{1x}[0] \\ \mathit{Bin}([\rho\mathbf{0} = x\mathbf{0}; \alpha_1]_x, [\rho\mathbf{0} = x\mathbf{1}; \alpha_2]_x) &\blacktriangleright \alpha_{1x}[\rho] \\ \mathit{Bin}([\rho\mathbf{1} = x\mathbf{0}; \alpha_1]_x, [\rho\mathbf{1} = x\mathbf{1}; \alpha_2]_x) &\blacktriangleright \alpha_{2x}[\rho] \\ \mathcal{D}([0 > 0; \alpha_1], [0 = 0; \alpha_2]) &\blacktriangleright \alpha_2 \\ \mathcal{D}([\rho\mathbf{0} > 0; \alpha_1], [\rho\mathbf{0} = 0; \alpha_2]) &\blacktriangleright \alpha_1 \\ \mathcal{D}([\rho\mathbf{1} > 0; \alpha_1], [\rho\mathbf{1} = 0; \alpha_2]) &\blacktriangleright \alpha_1 \ . \end{aligned}$$

11.2.9 Computability of binary functions and predicates. Binary functions and predicates are effectively computable by reductions.

11.2.10 Memory models for binary numerals. Computation is a syntactic process which proceeds by the manipulation of concrete objects. In the above discussion the concrete objects were monadic and binary numerals. In any practical implementation of computations over binary numerals on an electronic computer the d numerals will have to be mapped into the memory structures of the computer.

UNFINISHED Bignums

11.3 Computation over Pair Numerals

Although exponentially more efficient than monadic computation, the binary computation greatly suffers when computing with symbolic (coded) data.

11.3.1 Pair numerals. We recall the definition in Par. 1.3.8 of pair numerals as the least class of terms containing 0 and with every two terms ρ_1 and ρ_2 also the term (ρ_1, ρ_2).

11.3.2 Generalized term for pair numerals. The basic generalized term for clausal programs operating over pair numerals is

$$Pair([\tau = v, w; \alpha_1]_{v,w}, [\tau = 0; \alpha_2]) \; .$$

11.3.3 Pair clausal definitions. *Pair* functions and predicates are defined by pair clausal definitions.

11.3.4 Pair arithmetic. Basic operations and comparisons.

11.3.5 Characterization of pair functions and predicates. Pair functions and predicates are exactly the monadic ones.

11.3.6 Reductions over pair numerals.

$$Pair([0 = v, w; \alpha_1]_{v,w}, [0 = 0; \alpha_2]) \blacktriangleright \alpha_2$$
$$Pair([(\rho_1, \rho_2) = v, w; \alpha_1]_{v,w}, [(\rho_1, \rho_2) = 0; \alpha_2]) \blacktriangleright {\alpha_1}_v[\rho_1]_w[\rho_2] \; .$$

11.3.7 Computability of pair functions. Pair functions and predicates are effectively computable by reductions.

11.3.8 Memory models for pair numerals. The memory model of LISP is natural for the representation of pair numerals. The pair numeral 0 is represented in computer's memory with, say 32-bit words, by the word 0. The pair numeral (ρ_1, ρ_2) is represented by a non-zero word interpreted as a pointer to a *LISP-cell*, i.e. to two adjacent 32-bit words. The first word represents the first projection ρ_1 and the second word the second projection ρ_2. **UNFINISHED**

11.4 Computation over Mixed Numerals

We can combine the fast computation of numeric functions and predicates defined as binary functions with the gast computation of symbolic functions and predicates defined in pair notations by computing over mixed numerals.

11.4.1 Mixed numerals. *Mixed numerals* 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, and pair numerals, the mixed numerals do not enjoy the unique representation of natural numbers. For instance, the number three is denoted by four mixed numerals which are all different as terms:

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

11.4.2 Mixed clausal definitions. *Mixed* functions and predicates are defined by mixed clausal definitions which are constructed from the *Bin*, *Pair*, and $\mathcal{D}$ generalized terms.

Mixed clausal definitions clearly define all primitive recursive and μ-recursive functions.

11.4.3 Reductions over mixed numerals. Since binary and pair numerals are a subset of mixed ones, the reductions given in Paragrahps 11.2.8 and 11.3.6 apply also to mixed numerals. A pair (ρ_1, ρ_2) as an argument of a $\mathcal{D}$ generalized term has a natural reduction:

$$\mathcal{D}([(\rho_1,\rho_2) > 0; \alpha_1], [(\rho_1,\rho_2) = 0; \alpha_2]) \blacktriangleright \alpha_1 \ .$$

Reductions without conversion between representations are impossible in the following three cases:

$$\begin{aligned}
&\mathit{Bin}([(\rho_1,\rho_2) = x\mathbf{0}; \alpha_1]_x, [(\rho_1,\rho_2) = x\mathbf{1}; \alpha_2]_x) \blacktriangleright \\
&\qquad \mathit{Bin}([\boldsymbol{M_2B}(\rho_1,\rho_2) = x\mathbf{0}; \alpha_1]_x, [\boldsymbol{M_2B}(\rho_1,\rho_2) = x\mathbf{1}; \alpha_2]_x) \\
&\mathit{Pair}([\rho\mathbf{0} = v, w; \alpha_1]_{v,w}, [\rho\mathbf{0} = 0; \alpha_2]) \blacktriangleright {\alpha_1}_v[\boldsymbol{H}(\rho\mathbf{0})]_w[\boldsymbol{T}(\rho\mathbf{0})] \\
&\mathit{Pair}([\rho\mathbf{1} = v, w; \alpha_1]_{v,w}, [\rho\mathbf{1} = 0; \alpha_2]) \blacktriangleright {\alpha_1}_v[\boldsymbol{H}(\rho\mathbf{1})]_w[\boldsymbol{T}(\rho\mathbf{1})]
\end{aligned}$$

where we have denoted by $\boldsymbol{M_2B}$, $\boldsymbol{H}$, and $\boldsymbol{T}$ three conversion functions which are defined in the following paragraph to operate over mixed numerals.

11.4.4 Conversion of mixed to binary numerals. The conversion function $\boldsymbol{M_2B}$ takes a mixed numeral into a binary numeral with the same denotation. The function is defined by recursion on the structure of mixed numerals to satisfy:

$$\begin{aligned}
&\boldsymbol{M_2B}(0) \equiv 0\\
&\boldsymbol{M_2B}(\rho\mathbf{0}) \equiv \boldsymbol{M_2B}(\rho)\mathbf{0}\\
&\boldsymbol{M_2B}(\rho\mathbf{1}) \equiv \boldsymbol{M_2B}(\rho)\mathbf{1}\\
&\boldsymbol{M_2B}(\rho_1,\rho_2) \equiv \rho \Leftarrow P_b(\boldsymbol{M_2B}(\rho_1), \boldsymbol{M_2B}(\rho_2)) \blacktriangleright \rho\ .
\end{aligned}$$

The function is clearly effectively computable.

The conversion function $\boldsymbol{H}(\rho)$ takes a mixed numeral and yields a mixed numeral with the same denotation as $H(\rho)$. The function $\boldsymbol{T}(\rho)$ is similar. The functions satisfy the following:

$$\begin{aligned}
&\boldsymbol{H}(0) \equiv 0\\
&\boldsymbol{H}(\rho_1,\rho_2) \equiv \rho_1\\
&\boldsymbol{H}(\rho\mathbf{0}) \equiv \rho_1 \Leftarrow H_b(\boldsymbol{M_2B}(\rho)\mathbf{0}) \blacktriangleright \rho_1\\
&\boldsymbol{H}(\rho\mathbf{1}) \equiv \rho_1 \Leftarrow H_b(\boldsymbol{M_2B}(\rho)\mathbf{1}) \blacktriangleright \rho_1\\
&\boldsymbol{T}(0) \equiv 0\\
&\boldsymbol{T}(\rho_1,\rho_2) \equiv \rho_2\\
&\boldsymbol{T}(\rho\mathbf{0}) \equiv \rho_1 \Leftarrow T_b(\boldsymbol{M_2B}(\rho)\mathbf{0}) \blacktriangleright \rho_1\\
&\boldsymbol{T}(\rho\mathbf{1}) \equiv \rho_1 \Leftarrow T_b(\boldsymbol{M_2B}(\rho)\mathbf{1}) \blacktriangleright \rho_1
\end{aligned}$$

and are clearly effectively computable.

11.4.5 Computability of clausal definitions. A clausally defined function f of T is *reducible* if for all mixed numerals $\vec{\rho}$ we have $f(\vec{\rho}) \blacktriangleright \tau$ for a mixed numeral τ and $T \vdash f(\vec{\rho}) = \tau$.

Let T be an extension of S with a (guarded) clausal definition of f. Assume that all clausally defined functions of S are reducible.

The mixed functions are computable over mixed numerals.

11.4.6 A memory model for mixed pumerals. A mixed numeral can be represented in a memory of an electronic computer with a 32-bit words, by a word whose lowest two bits serve as four tags distinguishing the four kinds of mixed numerals.

We assign the tag 0 to represent the numerals which are pairs of the form (ρ_1,ρ_2). A 32-bit number n with the tag 0 (note that $n \bmod 4 = 0$) is interpreted as a pointer to a LISP-cell as in Par. 11.3.8. A word n with the tag 1 represents the mixed numeral $\rho\mathbf{0}$ by interpreting the number $n-1$ as a pointer to a word representing the mixed numeral ρ. Similarly, a word n with the tag 2 represents the mixed numeral $\rho\mathbf{1}$ by interpreting the number $n-2$ as a pointer to a word representing the mixed numeral ρ. Finally, a word n with the tag 3 represents the fourth kind of mixed numerals. Normally this would be the single numeral 0. It is, however, advantageous to utilize the remaining 30-bits of such words to store small natural numbers directly. This means that the word represents the binary numeral denoting $n \div 4$.

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

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

holds and such that with each two extended mixed numerals ρ_1 and ρ_2 also the terms $\rho_1\mathbf{0}$ (provided $\rho_1 \not\equiv 0$), $\rho_2\mathbf{1}$, and (ρ_1, ρ_2) are extended mixed numerals. Figure 11.1 shows the memory representation of the extended mixed numeral $(5, 1073741823\mathbf{1})\mathbf{01}$.

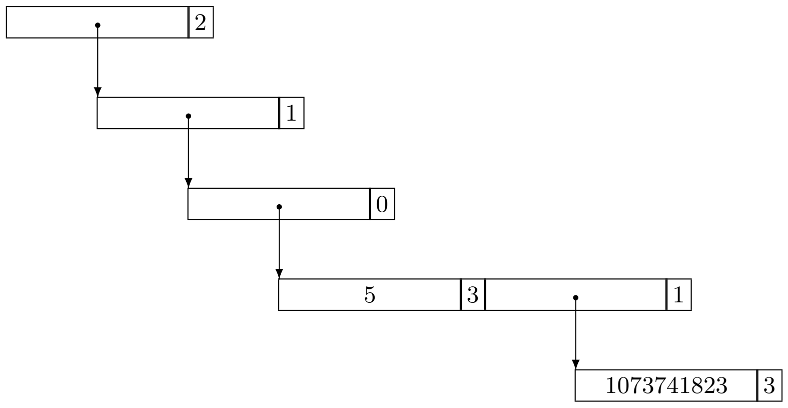


Fig. 11.1. Memory representation of the extended mixed numeral $(5, 1073741823\mathbf{1})\mathbf{21}$

12. Data Types

Typed programming languages, especially the functional ones, are popular for three reasons:

1. types naturally express properties of programs,
2. types permit good memory representation of data and the programs can be efficiently compiled,
3. types are indispensable for a natural semantics of higher-order functions (functionals).

The first reason for types is irrelevant for a programming language integrated with its own proof system (such as CL) because the formulas of the language permit a simpler formulation and proofs of much more complex properties than those expressible by typing. The remaining reasons for typing are exteremely important and we discuss the second reason in this chapter and the third one in Chapter 13.

However, there is a negative side to typing in a complication of syntax and semantics. The usual approach to types is via many-sorted languages and theories. The simple languages and semantics of LISP, PROLOG, and CL are lost. We now propose a type system which enables us both to have and eat the proverbial cake.

12.1 Pascal-Style Typing

Pascal was the first computer programming language whose data types supported good representation in memory.

12.1.1 Type predicates. Pascal-style types will be defined in **UNFINISHED** as codes of certain unary *type* predicates. By convention we write $x : T$ instead of $T(x)$ and read it as the code of T is the type of x Not every unary predicate is a type predicate. We will introduce in the following paragraphs several kinds of syntactic restrictions on the clauses of type predicates $x : T$ which will guarantee that the value x can be efficiently represented in computer memory.

Extension of a type predicate T is the set numbers x such that $x : T$ holds in the standard model of PA. *Intension* of T is the set of mixed numerals which are *canonical* terms of type T. This will be precisely defined in **UNFINISHED**

UNFINISHED Canonical terms of type T can be efficiently represented in computer memory. In the following we will see that although extensions of two type predicates T_1 and T_2 may overlap in that there is a mixed numeral ρ such that $\rho : T_1$ and $\rho : T_2$, the intesions are different in that ρ has two different memory representations.

UNFINISHED typing of generalized terms and of clausal definitions. Additionally, a well-typed function or predicate can be reduced over mixed numerals without the three reductions given in Par. 11.4.3 and which involve conversion of representation.

12.1.2 Basic type predicates. Predicates N and Ch with explicit definitions

$x : N$
$x : Ch \leftarrow x < 256$

are the basic type predicates. The code of $\mathbb{N}$ is the *type of natural numbers* and the code of Ch is the *type of characters*.

In programming practice there will be additional basic types inluding integers and floats.

12.1.3 Cartesian type predicates. Suppose that $T_1, \dots T_n$ are type predicates. The unary type predicate explicitly defined by

$(x_1, \dots, x_n) : T \leftarrow x : T_1 \wedge \dots \wedge x : T_n$

is the type predicate of *cartesian product* of codes of T_1, ..., T_n.

UNFINISHED The extension of T are exactly numbers $x_1, \dots, x_n$ for some x_1, ..., x_n. those mixed numerals which can be written as n-tuples $\rho_1, \dots, \rho_n$.

Memory representation: **UNFINISHED**

12.1.4 List type predicates. Suppose that S is a type predicate. The predicate

$0 : T$
$x, y : T \leftarrow x : S \wedge y : T$.

is the type predicate of *lists of codes of S*.

12.1.5 Fixed vector type predicates. Suppose that S is a type predicate and c a constant. The predicate

$0 : T$
$x, y : T \leftarrow x : S \wedge y : T$.

is the type predicate of *lists of codes of* S.

UNFINISHED typing of definitions

12.2 ML-Style Typing

ML-style *polymorphic* typing is a higher-order calculus where variables and arguments of predicates can range over types.

12.2.1 Type predicates with types as arguments. List types are best viewed as higher-order predicates with types as arguments Instead of viewing type predicates as unary we may permit additional argument. Consider as an example the binary type predicate of lists *List*:

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

We can view the binary predicate $x \mathbin{\mathbf{:}} t$ as a *universal typing* predicate where the second argument t ranges over type predicates T in such a way that

$$x : T \leftrightarrow x \mathbin{\mathbf{:}} T$$

holds.

For concrete type predicates t we can then consider $x : \mathit{List}(t)$ to be a unary predicate in x. For instance, $x : \mathit{List}(N)$ and $x : \mathit{List}(\mathit{Ch})$ can be viewed as type predicates which would have to be introduced in Pascal-style typing by two definitions:

$0 : \mathit{ListN}$
$x, y : \mathit{ListN} \leftarrow x : \mathit{ListN} \wedge y : \mathit{ListN}$
$0 : \mathit{ListCh}$
$x, y : \mathit{ListCh} \leftarrow x : \mathit{ListCh} \wedge y : \mathit{ListCh}$.

12.2.2 Example of polymorphic typing. UNFINISHED

12.2.3 Vectors. Higher-order type predicates such as $x : \mathit{List}(t)$ do need to have their arguments restricted to types. Consider for instance, the ternary predicate $x : V_1(n, t)$ of *fixed vectors* which for any number n and type t behaves as the Pascal-like type *array* $[0..n-1]$ *of* t.

$0 : V_1(0, t)$
$x, y : V_1(n+1, t) \leftarrow x \mathbin{\mathbf{:}} t \wedge y : V_1(n, t)$.

UNFINISHED discussion that they are lists but can be mapped into memory in a better way.

We can now introduce a binary typing predicate $x : V_2(t)$ of *flexible vectors* with the definition

$n, x : V_2(t) \leftarrow x : V_1(n, t)$.

UNFINISHED discussion and memory representation

UNFINISHED Without ontological commitment to types which would get us outside of first-order theories we can treat types as codes of type predicates. Universal functions in recursion theory. Assignment of indices, choices are endless but we will for illustration purposes stick to one type system with good mapping onto memory.

Higher-order typing even without functionals. We can deal with the problem by intensionality. The value of extensionality is diminished when types are used for intensional reasons such as computation.

13. Functional Programming

14. Modular Programming

14.1 Extraction of Programs from Proofs

14.1.1 Postfix machine. Our next task is the proof of correctness of a simple postfix machine evaluating numeric terms. The machine is represented by a binary function $\mathit{Run}(p, m)$ where p is a postfix program and m a list reresenting memory words. The memory is used as a stack with $(m)_0$ addressing its top.

The instructions of the machine are: .... as in the text including also the predicate *Program*...

The instruction $\mathit{LOAD}(i)$ pushes the contents of the i-th memory cell, i.e. of $(m)_i$ to the top of the stack thus modifying the memory from m to $(m)_i, m$. We can view the memory as composed of a stack part s and a variable part v, i.e. $m = s \oplus v$. The stack part s is used to hold the subterms of the term being evaluated by the program p. When the memory is addressed by $\mathit{LOAD}(L(s)+i)$ then the i-th element of the valuation v, i.e. $(v)_i$ is accessed which gives the value to the i-th variable $\mathbf{x_i}$.

A numeric term t is compiled into a program p by a binary function $\mathit{Comp}(j, t) = p$. The program p evaluating t expects the length of the stack part of the memory, i.e. $L(s)$, to be j and compiles the program p to access the values of variables of t in the memory with the *offset* j. The compiler is defined by the clauses:

$$\begin{aligned}
&\mathit{Comp}(j, \boldsymbol{x}_i) = \mathit{LOAD}(j+i), 0\\
&\mathit{Comp}(j, \bar{n}) = \mathit{PUSH}(n), 0\\
&\mathit{Comp}(j, t_1 + t_2) = \mathit{Comp}(j, t_1) \oplus \mathit{Comp}(j+1, t_2) \oplus (\mathit{ADD}, 0)\\
&\mathit{Comp}(j, t_1 \times t_2) = \mathit{Comp}(j, t_1) \oplus \mathit{Comp}(j+1, t_2) \oplus (\mathit{MULT}, 0)\ .
\end{aligned}$$

The reader will note that in the third clause the first argument t_1 of $t_1 + t_2$ is compiled with the offset j while the second argument t_2 is compiled with the offset $j+1$ because the top of the stack at the start of the execution of the program for t_2 will hold the denotation (value) of the term t_1.

.............Now come the clauses for the binary function $\mathit{Run}(p, s)$ (The function *Eval* is not needed at all because the correctness theorem is

$$\mathit{Term}(t) \rightarrow \mathit{Run}(\mathit{Comp}(0, t), v) = [\![t]\!]_v$$

which follows from the lemma

$$\mathit{Term}(t) \rightarrow \forall p \forall s\, \mathit{Run}(\mathit{Comp}(L(s), t) \oplus p, s \oplus v) = \mathit{Run}(p, [\![t]\!]_v, s \oplus v)\ .$$

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. L. Kirby, Paris J. *Accessible independence results for PA*. Bulletin. London Math. Soc. 14 (1982) p 225-285.
12. 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.
13. 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.
14. 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.
15. J. Komara, P. J. Voda. *Towards Provably Correct Programming*. www.fmph.uniba.sk/˜voda. April 2000.
16. J. Komara. *Specification and Verification of Computer Programs*. http://dent.ii.fmph.uniba.sk/˜voda/cl.html. May 2001.
17. R. Milner, *A Theory of Type Polymorphism in Programming*. J. Comput. System Sci. 17 (1978) 348-375.
18. J. C. Mitchell. *Type Systems for Programming Languages*. In Handbook of Theoretical Compute Science (vol B), J. van Leeuwen ed., Elsevier 1990.
19. Ch. Okasaki. *Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design* International Conference on Functional Programming, september 2000.
20. R. Péter. *Konstruktion nichtrekursiver Funktionen*. Math. Ann. vol 111 (1935), pages 42-60.

21. H. E. Rose. *Subrecursion: Functions and Hierarchies.* Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.
22. H. Schwichtenberg, *Eine Klassifikation der ϵ_0-rekursiven Funktionen*, Zeitschrift für mathematische Logik und Grundlagen der Mathematik, vol. 17, (1971), pp. 91-74.
23. J. R. Shoenfield. *Mathematical Logic*, Addison-Wesley, 1967.
24. R. M. Smullyan *First-Order Logic.* Springer, 1968.
25. W. W. Tait. *Intensional interpretation of functionals of finite type.* J. of Symbolic Logic, 32:198–212, 1967.
26. G. Takeuti. *Proof Theory.* North-Holland, 1975.
27. A. S. Troelstra. *Aspects of Constructive Mathematics.* In Handbook of Mathematical Logic, J. Barwise ed., North-Holland Publishing Co, 1977.
28. 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.
29. S. S. Wainer. *A classification of the ordinal recursive functions*, Archiv für mathematische Logik und Grundlagen Forschung, vol. 13, 1970, pp. 136-153.