Programming by Logic and Logic by Programming

(Extended Abstract)

Paul J. Voda, Ján Komara

Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 43 Bratislava, Slovakia.

Email: `voda@fmph.uniba.sk`, `komara@fmph.uniba.sk`

Abstract: This paper is an extended abstract of parts I and II of a monograph with the same title being prepared by the authors. Logic was developed before the arrival of computers and it can be presented without reference to them. The main idea of the monograph is to stress the mutual links between logic and computer programming by taking those parts of logic which deal with computability and finitary logic (proof theory) and presenting them by employing the techniques of computer programming. The techniques involve the use of a good programming language with good data structures and proving the properties of functions written in this language.

Keywords: computability, logic, computer programming

1 Introduction

Nobody disputes the importance of mathematical logic to computer science but the influence that computer science can have on mathematical logic may not be that obvious. This paper investigates the mutually beneficial links between both fields of study.

Mathematical logic was developed in order to give firm foundations to mathematics. As a part of this process the concept of computability was discovered. Computability is the central concept of computer science but not of mathematics. This puts the relation between mathematical logic and computer science into perspective.

An exposition of logic targeted at mathematicians is too general for the purposes of computer science. For this reason many presentations of mathematical logic were targeted at computer scientists. This process was accelerated in the recent years when *declarative* (i.e. mathematical) programming languages and *logic programming* (i.e. programming based in logic) have been accepted among computer theoreticians and even practitioners. By our count, there are at least twenty books (both texts and monographs) on logic for computer scientists. They can be roughly categorized into four groups. The first group emphasises the theory of computability. A typical representative is [18]. The second group centers on the problems of axiomatization of various data structures used in computer programming. A representative is [8]. The third group centers on automating the process of theorem deduction. A representative is [3]. Finally, the fourth group presents non-classical constructive logics. Constructive logics give foundations to mathematical theories by restricting the full power of classical logic to processes which can be viewed as 'effective'. A representative is [5].

We believe that there is an unexplored approach to the presentation of mathematical logic to computer scientists. All the approaches mentioned above present the methods of mathematical logic which are applicable to computer science. The flow of information is always from logic to computer science. As can be seen from the title of this abstract we intend the flow of information to be both ways.

We first use logic, specifically the theory of recursive functions, to construct a programming language called $L\mathcal{E}SP$ which can be used both to develop the theory of recursive functions and serve as a practical programming language efficiently implementable on computers. $L\mathcal{E}SP$ is not only a theoretical vehicle, in which capacity it is being used throughout the monograph, but it will eventually be also a practical programming language for the writing of real programs.

We then turn the flow of information around and we study formal systems of logic by means of programs written in $L\mathcal{E}SP$. We concentrate on the subjects of predicate calculus, first-order theories, and arithmetic, roughly to the extent as they are covered by the superb (but extremely difficult) text by J. Shoenfield [13].

Finitary methods of logic, first investigated by Hilbert and his school, known under the name *Proof theory* are essentially an application of programming methods and techniques to the study of formal systems presented as concrete objects. The latter are called data structures in computer programming. Standard theorems of proof theory such as the Herbrand and Cut elimination theorems can be obtained by first writing the appropriate programs and then proving their properties.

Computer programming has something to offer to logic here. Computer programmers are used to writing enormously large and complicated programs. Over the years they developed good programming techniques and came to appreciate the importance of a good programming language with good data structures, a good programming methodology and methods of proving the correctness of programs.

Our effort can be viewed as programming the finitary part of Shoenfield's text. By using the mutual links approach we cover the proof theory upto and including the formal arithmetic. This becomes a rich application area for computer programs to operate upon. Our experience with using this approach in two courses on mathematical logic and computability at the Comenius University were most encouraging. We had the impression that computer programmers actually did understand what logic was about.

The monograph consists of three parts constituting a Hegelian triad of thesis, antithesis, and synthesis.

The part I (thesis) is called 'Recursive Functions' with the undertitle 'Programming as Logic'. Here we develop a programming language $L\mathcal{E}SP$ operating on natural numbers as a language for the definition of recursive (computable) functions. The language which is necessary for the development of various classes of recursive functions, and thus it is a part of logic, can be at the same time used as a practical programming language, i.e. a language which can be efficiently implemented on computers. We show that the development of recursive functions uses the techniques such as stack and table operations which are widely known to computer programmers. The usual presentation of recursive functions obfuscates the programming techniques.

The message of the part I is that logic, i.e. the theory of recursive functions, is a basis for the design of computer programming languages.

The part II (antithesis) is called 'Proof Theory' with the undertitle 'Logic as Programming'. We are concerned predominantly with finitary methods. 'Finitary' means that we are working with concrete objects in a constructive fashion rather than with abstract objects (higher-order sets) and non-constructive methods (unrestricted use of quantifiers and negation). When the objects under the study are not too complicated - and we never go beyond the formal arithmetic - then 'finitary' means computable by primitive and ordinal recursive (for relatively small ordinals) functions.

This means that we can develop the proof theory by partially formalizing our meta-theory where we employ the language $\mathcal{LESP}$ developed in the part I. In the usual development of logic the language of the meta-theory is a natural language such as English and the concrete objects under the study of logic, i.e. terms, formulas and proofs, are finite sequences of signs. By using $\mathcal{LESP}$ in the meta-theory the concrete objects become *data structures*, i.e. natural numbers, which, of course, can be presented in a concrete way. The constructive processes over these objects investigated in logic become primitive and ordinal recursive functions.

The message of the part II is that the logic (at least the finitary part of it) can be viewed as writing of definitions of computable functions and proving their properties. This process is quintessentially what computer programming is about.

The part III (synthesis) is called 'Arithmetic' with the undertitle 'Programming Logic'. We single out for investigation one concrete theory of the sort discussed in the part II. It is the formal arithmetic, also known as Peano arithmetic. The power of Peano arithmetic is sufficient to *arithmetize* the concepts developed in the first two parts. In the part I we study primitive and ordinal recursive functions. We show that these functions can be formally introduced into Peano arithmetic by means of *recursive extensions*. Once we have them in a formal theory we can formulate and prove their properties in a formal way. This paves the way for a computer-assisted derivation of the properties. In the part II we study formal theories by means of primitive recursive functions. Once again we can arithmetize this process by reflecting it in an extension of Peano arithmetic. This gives us the means to reason about the reasoning process. Thus arithmetic serves as a *logic framework*.

The message of the part III is that in order to be able to reason about our programs in a computer-assisted way and to be able to reflect on the derivations of our automatic theorem-provers we do not need any stronger theory than Peano arithmetic.

This message should be contrasted with the usual approach to the formal semantics of programming languages and logic frameworks. For the former it has become almost standard to use theories with very powerful models which go deep into set or even category theory. For the latter constructive theories of logic are usually used. Constructive theories were developed by logicians in order to justify analysis, or at least parts of it. Those computer scientists who took over the theories have probably not realized that they were dealing with very small classes of primitive recursive functions, since only such can be computed in a feasible way on our computers. Peano arithmetic is more than sufficient for the analysis of such function classes.

Both recursion theory and arithmetic are presented by means of a pairing function rather than by the usual way of using the successor function $s(x) = x + 1$. The reason for this is simple. Recursive functions are over natural numbers and they cannot be directly implemented on computers in an efficient way because of rather inefficient coding known as *Gödel numbering*. When recursive functions are presented via a pairing function, the coding becomes very natural and efficiently implementable on computers. That a language based on pairing is usable was demonstrated by the design of LISP, by John McCarthy, thirty years ago. Today LISP is still the main programming tool in artificial intelligence. The first author has designed and implemented two practical computer programming languages Trilogy I and II [16, 2] based on pairing. The practical experience with them was extremely encouraging.

John McCarthy has analyzed the formal semantics of LISP by means of recursion theory already in 1963 [9]. Since then the formal semantics of programming languages was greatly developed. The reader should refer to [7]. He will find a detailed description of at least five different approaches to the formal semantics. He may wonder, just as we did, why the formal semantics of programming languages is not treated there by recursion-theoretic means. After all, our computers cannot possibly compute more than computable functions. Our monograph should be viewed as an attempt to revive the pioneering work of John McCarthy and to study the formal semantics of programming languages by recursion-theoretic means. The process is then reverted and logic is studied by computer-programming means.

Due to the limitations of time alloted to our presentation (two lectures) we have decided against covering the part III of the book in this abstract. We do not cover the arithmetization of the first two parts in the arithmetic. We hope that the application of the programming techniques to the finitary development of predicate calculus and formal theories are sufficient to illustrate our point of view.

The section 2 covers the development of primitive recursive and ordinal recursive functions by means of a pairing function. In the process of doing this we introduce $\mathcal{LESP}$ and characterize the class of functions it can compute. $\mathcal{LESP}$ has a flavor of LISP although it has a form of clauses as we know them from Prolog. The treatment in this section covers most of the part I of the monograph.

The section **??** covers predicate calculus and formal theories. The issues of semantics are underplayed by encapsulating all semantic assumptions needed into, so-called, reduction theorems. The rest of the section is strictly finitary and the development proceeds by designing data structures, writing of functions operating on them and of proving their properties. The treatment in this section covers the part II of the monograph.

In order to give the reader the flavor of subjects covered in the monograph we give a brief description of its contents. As already mentioned, only the first two parts are covered in this abstract.

Part I: Pairing functions, primitive recursion, $<\epsilon_0$-ordinal recursion, nested recursion, Tait's theorem, intensional functionals of finite types. Both the $<\epsilon_0$-recursive functions and intensional functionals (of type $0 \to 0$) form the same class. This gives us a programming language $\mathcal{LESP}$ with (the codes of) functions as arguments and results.

Part II: Tautologies, tautological consequence, quasi-tautological consequence, logical consequence. Provability in general and by a system based on tableaux dual to Smullyan's. Completeness in all three above categories. Cut-elimination, analytic tableaux, formal theories, extensions of theories, Herbrand and Skolem theorems.

Part III: Arithmetization of the first two parts. Every recursive function is representable in formal arithmetic (presented in the pair-based form) without induction. The first incompleteness theorem of Gödel. Pair induction, $<\epsilon_0$-recursive functions can be introduced into arithmetic by recursive extensions. The second incompleteness theorem. Cut elimination for arithmetic, provably recursive functions are just the $<\epsilon_0$-recursive functions. Derivation of transfinite induction in arithmetic and its limits. Gödel interpretation of arithmetic by means of intensional functionals of finite types into a quantifier-free fragment of itself ($<\epsilon_0$-arithmetic).

Part I: Recursion theory, Programming by logic

2 Recursive functions

Recursive functions are computable functions over the domain of natural numbers by the thesis of Church. We present the development of recursive functions by means of a pairing function P instead of the usual development via the successor function $s(x) = x+1$. The reason for this is twofold: (i) we have easier coding, (ii) we can define our computable functions within a language called $\mathcal{LESP}$ in the style of LISP. We believe that we can give an efficient implementation to $\mathcal{LESP}$ and thus turn a theoretical vehicle as used throughout this paper into a practical vehicle for the writing of real programs.

2.1 Pairing function

We study total functions and predicates over natural numbers. Unless we specifically mention the arity of a function or predicate we understand it unary. In a context where we will discuss only unary functions and predicates, we will write x, y as an abbreviation for $P(x, y)$ where P is a certain binary function determined below. The notation x, y, z is an abbreviation for $P(x, P(y, z))$.

Theorem 1 (Pairing function P) There is a unique binary function P such that the following sequence enumerates all natural numbers in the ascending order:

$$0 \quad 0,0 \quad 0,0,0 \quad (0,0),0 \quad 0,0,0,0 \quad 0,(0,0),0$$

$$(0,0),(0,0) \quad (0,0,0),0 \quad ((0,0),0),0 \quad \cdots$$

We have

$$\begin{aligned} x &< x, y \\ y &< x, y \\ 0 &\neq x, y \\ x > 0 &\rightarrow \exists v \exists w\, x = v, w \\ x_1, y_1 = x_2, y_2 &\rightarrow x_1 = x_2 \wedge y_1 = y_2. \end{aligned}$$

Sketch of the proof: The sequence is formed by writing down the terms consisting of 0 and applications of P ordered first by the number of applications of P (i.e. on the number of commas) and within the same number of applications lexicographically. Thus the sequence defines the function $P(x, y)$ by a double induction where $P(v, w)$ can be assumed defined for all numbers $v \leq x$ and $w \leq y$. The first two properties follow from the way the enumeration is formed and the remaining ones from the fact that P is a bijection from $\mathbf{N}^2$ onto $\mathbf{N} \setminus \{0\}$. ■

Theorem 2 (Pair Induction) In order to prove $\forall x\, R(x)$ for a predicate R it is sufficient to derive $R(0)$ and $\forall x \forall y (R(x) \wedge R(y) \rightarrow R(x, y)$.

Proof: By complete induction in a straightforward way. ■

Theorem 3 (Pair Representation) Every natural number has a unique *pair representation* as a term obtained from 0 by finitely many applications of the pairing function P.

Proof: By pair induction taking $R(x)$ to hold if *x has a unique representation.* ■

Theorems 2 and 3 guarantee that every number x is either 0 or it can be uniquely written in the form $x_1, x_2, \ldots, x_n, 0$ for some $n \geq 1$ and numbers x_i. Thus every number codes a single finite sequence of numbers (codes of finite sequences are called *lists* in the computer science), and vice versa. The *length* function Len with the properties $Len(0) = 0$ and $Len(x, y) = Len(y) + 1$ yields the length of the list x. The *pair size* function $|x|$ with the properties $|0| = 0$ and $|x, y| = |x| + |y| + 1$ yields the number of applications of P in the pair representation of x. We have $Len(x) \leq |x| \leq x$.

The reason why we have chosen this particular pairing function P is that it has the property $|x| = \Theta(log(x))$ which follows from $2^{|x|} \leq 4 \cdot x + 1$ and $x < 4^{|x|+1}$ (see [17]). The property permits a natural construction of Grzegorczyk-like hierarchies [6, 11] based on pairing. These hierarchies can be characterized both by the Grzegorczyk hierarchy and, in terms of computational complexity, by Turing machines. Discussion of these issues is beyond the scope of this paper and the reader is referred to [17].

2.2 Primitive recursive functions

The *identity* function is $I(x) = x$. The *zero* function is $Z(x) = 0$. The *first (H) and second (T) projection* functions are such that $H(0) = T(0) = 0$, $H(x, y) = x$, and $T(x, y) = y$. The *conditional* function D is such that $D(0, x, y) = y$, $D((v, w), x, y) = x$, and $D(0) = D(x, 0) = 0$. The function f is obtained from the functions g and h by *composition* if $f(x) = g\, h(x)$ and by *pairing* if $f(x) = g(x), h(x)$.

Functions *simple (in a class of functions $\mathcal{F}$)* are generated from $(\mathcal{F})$, I, Z, and D by composition and pairing.

Theorem 4 A class of functions simple in $\mathcal{F}$ is closed under *explicit definitions* of the form $f(x) = \boldsymbol{a}$ where $\boldsymbol{a}$ is a term constructed from the variable x and numerals by pairing $(\boldsymbol{b}, \boldsymbol{c})$ and applications $g(\boldsymbol{b})$ where the function g is a function simple in $\mathcal{F}$.

Proof: By induction on the structure of $\boldsymbol{a}$. We may assume that all numerals in $\boldsymbol{a}$ are in the pair representation. If $\boldsymbol{a}$ is 0 then take $f = Z$, If $\boldsymbol{a} = \boldsymbol{b}, \boldsymbol{c}$ then obtain $g(x) = \boldsymbol{b}$ and $h(x) = \boldsymbol{c}$ by induction hypotheses and define $f(x) = g(x), h(x)$ by pairing. If $\boldsymbol{a} = g(\boldsymbol{b})$ then obtain $h(x) = \boldsymbol{b}$ by induction hypothesis and define $f(x) = g\, h(x)$ by composition. ■

The projection functions are simple since they have explicit definitions $H(x) = D(1, x)$ and $T(x) = D(0, x)$. An example of a function f simple in g and h is

$$f(x) = D(x, D(H(x), 0, D(g\,T(x), \\ h(T\, g\, T(x), H\, g\, T(x)), 0)), c). \quad (1)$$

Definitions like (1) are made readable by a language which we call $\mathcal{LESP}$ where we define the same f by an *explicit definition* as

$f((v, w), x) = 0$
$f(0, x) = h(w, v) \leftarrow g(x) = v, w$
$f(0, x) = 0 \leftarrow g(x) = 0$
$f(0) = c.$

The clauses of $\mathcal{LESP}$ when read as inverted implications express properties of the defined function. The first and the third clause can be omitted as the defined function obtains the value 0 by *default* if no clause can be satisfied for a given argument. Clauses are mutually exclusive and may be given in any order (we have listed the clauses for f in the order obtained by removing the conditionals from the definition (1)). The clauses of $\mathcal{LESP}$ must be presented in such a way that the conversion from a clausal definition to an *initial form* as the above one should be always possible.

We do not have the space to go into the details of $\mathcal{LESP}$ and we hope that the $\mathcal{LESP}$ definitions given in this paper will be simple enough for the reader to reconstruct the initial forms of definitions. In the case of explicit definitions the initial form is always $f(x) = \boldsymbol{a}$. We permit the clausal language $\mathcal{LESP}$ to be used with recursive definitions where the initial form is a schema of recursive definition. The development of the $\mathcal{LESP}$ language should be viewed from the perspective of its eventual implementation on computers. We offer here no details on how to go about the automatic compilation.

We now present three schemas of pair-based inductive definitions which can be used as initial forms of $\mathcal{LESP}$ definitions. The function f is defined by the *unary iteration* of g if

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

We write $g^x(y)$ as an abbreviation for the application $f(x, y)$. The function f is defined by the *pair iteration* of g if

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

The function f is defined by the *pair recursion* from g and h if

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

The classes $\mathcal{PR}_1$, $\mathcal{PR}_2$, and $\mathcal{PR}_3$ are generated from I, H, and T by composition pairing and by pair recursion, pair iteration, and unary iteration respectively, We call the functions of $\mathcal{PR}_1$ *primitive pair recursive* functions.

Theorem 5 The classes $\mathcal{PR}_i$ are closed under explicit $\mathcal{LESP}$ definitions.

Proof: Clearly, it is sufficient to show that the classes $\mathcal{PR}_i$ are simple in themselves, i.e. that each of them contains the functions Z and D. The function Z is derived in $\mathcal{PR}_i$ by the iteration of T. In the class $\mathcal{PR}_1$, this is done by first defining f_1 by pair recursion from I and $h(x) = T\,T\,T\,T\,T(x)$ where h is obtained by a repeated composition. Note that we have $f_1((v, w), y) = T\, f_1(w, y)$. Then we define $f_2(x) = I(x), I(x)$ by pairing and, finally, $Z(x) = f_1\, f_2(x)$ by composition. The derivations of Z in $\mathcal{PR}_2$ and $\mathcal{PR}_3$ are similar but we define f_1 by pair iteration and unary iteration of T, respectively.

The derivation of D in $\mathcal{PR}_1$ is by pair recursion from T and $h(x) = H\,H\,T\,T(x)$. Note that we then have $D(0, y) = T(y)$ and $D((v, w), y) = H(y)$. The derivation of $D \in \mathcal{PR}_2$ is by composition $D(x) = T\, f(x)$ where f is defined by pair iteration of $g(x) = H(x), H(x)$ obtained by pairing. If we define f by unary iteration of g we get $D \in \mathcal{PR}_3$. ■

We can now use $\mathcal{LESP}$ with any of the three inductive schemas. The functions $\oplus$, $\otimes$, $\uparrow$, written in infix notation, for instance, $x \oplus y$ instead of $\oplus(x, y)$, are obtained in $\mathcal{PR}_1$ by pair recursion:

$$\begin{aligned}
&0 \oplus y = y\\
&(v, w) \oplus y = v, w \oplus y\\
&Rt(0) = 0\\
&Rt(v, w) = 0, Rt(v) \oplus Rt(w)\\
&x \otimes y = f_1(Rt(x), y)\\
&f_1(0, y) = 0\\
&f_1((v, w), y) = y \oplus f_1(w, y)\\
&x \uparrow y = f_2(Rt(y), x)\\
&f_2(0, x) = 1\\
&f_2((v, w), x) = x \otimes f_2(w, x).
\end{aligned}$$

The function $\oplus$ is the *list concatenation* function. The function Rt is obtained by *parameterless pair recursion* for which $\mathcal{PR}_1$ is easily seen closed. $Rt(x)$ yields a *right-leaning number* of the same pair size as x: $Rt(x) = \sigma\,|x|$ where

$$\sigma(n) = \overbrace{0, 0, \cdots, 0}^{n}, 0.$$

We have

$$|Rt(x)| = Len\; Rt(x) = |x|.$$

By pair induction we can derive

$$\begin{aligned} |x \oplus y| &= |x| + |y| \\ |x \otimes y| &= |x| \cdot |y| \\ |x \uparrow y| &= |x|^{|y|}. \end{aligned}$$

The function $Lt(x) = \sigma(|x|+1) - 1$ yielding a *left-leaning number* of the same pair size is obtained as primitive pair recursive by

$Lt(x) = f\, Rt(x)$
$f(0) = 0$
$f(v, w) = f(w), 0.$

We identify a computable predicate R with its *characteristic function*

$$R_*(x) = y \leftrightarrow R(x) \wedge y = 1 \vee \neg R(x) \wedge y = 0.$$

We say that the predicate R *is from* $\mathcal{F}$ if $R_* \in \mathcal{F}$. The predicate $Left(x)$ true of left-leaning numbers is defined primitive pair recursive as folows:

$Left(0)$
$Left(v, w) \leftarrow w = 0 \wedge Left(v).$

$\mathcal{LESP}$ definitions of predicates should be viewed as abbreviations for $\mathcal{LESP}$ definitions of their characteristic functions. If we replace $Left(x)$ by $Left_*(x) = 1$ above we get such a definition. Note that the clauses for $\neg Left(x)$, i.e. for $Left_*(x) = 0$, are obtained by default. We can similarly define the primitive pair recursive predicate *Right* holding of right-leaning numbers.

We now turn to arithmetic functions. The successor and predecessor functions are obtained by the case analysis of the enumeration of pairs from the theorem 1. We derive the successor function in $\mathcal{PR}_1$ by a parameterless pair recursion:

$s(0) = 1$
$s(v, w) = v, s(w) \leftarrow \neg Left(w)$
$s(v, w) = s(v), Rt(w) \leftarrow$
  $Left(w) \wedge \neg Left(v)$
$s(v, w) = 0, Rt(v, w) \leftarrow$
  $Left(w) \wedge Left(v) \wedge w = 0$
$s(v, w) = Rt(0, v), Rt(w_1) \leftarrow$
  $Left(w) \wedge Left(v) \wedge w = w_1, w_2.$

The *predecessor* function $p(0) = 0$ and $p\, s(x) = x$ is derived as primitive pair recursive in a similar way with with the help of Lt and *Right*. Before we can get additional arithmetic functions we need $\mathcal{PR}_1$ closed under unary iteration:

Theorem 6 The classes $\mathcal{PR}_i$ consist exactly of primitive pair recursive functions.

Proof: $\mathcal{PR}_1 \subset \mathcal{PR}_2$: It is clearly sufficient to show $f \in \mathcal{PR}_2$ for f derived by the pair recursion from g and h in $\mathcal{PR}_2$. Derive f in $\mathcal{PR}_2$ by pair iteration:

$f(x, y) = v \leftarrow$
  $h_1^{|x,x,x|}(y, 0, (0, x), 0) = y_1, ((v, x_1), s), i$
$h_1(y, s, (0, 0), i) = y, ((g(y), 0), s), i$
$h_1(y, s, (0, v, w), i) = y, s, (0, w), (0, v), 0, i$
$h_1(y, ((fv, v), (fw, w), s), 0, i) =$
  $y, ((h(v, w, y, fv, fw), v, w), s), i$
$h_1(y, s, 0) = y, s, 0.$

The 'second' argument s of h_1 is an output stack where the function values $f(z, y)$ of components z of x are assembled in the form $f(z, y), z$. The 'third' argument i is an input stack for breaking down the components z of x. The components are stored in the form $(0, z)$. The marker 0 is pushed on the input stack in the second clause for h_1 to indicate that when it eventually appears at the top of the stack in the third clause for h_1 the values $f(v, y), v$ and $f(w, y), w$ will be on the top of the output stack. The needed length of iteration of h_1 is $3 \cdot |x| + 1 < |x, x, x|$ as we need two iterations for each comma and one iteration for each 0 in the pair representation of x.

$\mathcal{PR}_2 \subset \mathcal{PR}_3$: It is sufficient to show $f \in \mathcal{PR}_3$ for f derived by pair iteration of $g \in \mathcal{PR}_3$. Derive f in $\mathcal{PR}_3$ by unary iteration:

$f(x, y) = a \leftarrow g_1^{x,x}(y, x, 0) = a, s$
$g_1(a, 0, s) = a, s$
$g_1(a, (v, w), s) = g(a), v, w, s$
$g_1(a, 0) = a, 0.$

The iteration of g_1 goes long enough $2 \cdot |x| + 1 = |x, x| \leq x, x$ to empty the 'second' argument s of g_1 which acts as an input stack for breaking the original argument x into its components. The 'first' argument a is an accumulator where g is applied each time we have a pair on the top of the stack.

$\mathcal{PR}_3 \subset \mathcal{PR}_1$: Assume f defined by the unary iteration of $g \in \mathcal{PR}_1$ and show $f \in \mathcal{PR}_1$. Derive f in $\mathcal{PR}_1$ by pair recursion:

$f(x, y) = y_1 \leftarrow$
  $f_1(Rt(9 \uparrow (0, x)), x, y) = x_1, y_1$
$f_1(0, x, y) = x, y$
$f_1((v, w), x, y) = 0, y_1 \leftarrow$

$$f_1(w,x,y) = 0, y_1$$
$$f_1((v,w),x,y) = p(v_1,w_1), g(y_1) \leftarrow$$
$$f_1(w,x,y) = (v_1,w_1), y_1.$$

The auxiliary function f_1 accepts as parameters and yields as results numbers of the form c,a where c is a counter of iterations and a an accumulator where g is repeatedly applied to. This works provided the length of recursion is at least x. We have

$$Len\ Rt(9 \uparrow (0,x)) = |9 \uparrow (0,x)| =$$
$$|9|^{|0,x|} = 4^{|x|+1} > x. \ \blacksquare$$

An *unary equivalent* of an n-ary function f ($n \geq 2$) is any function g such that

$$f(x_1, x_2, \cdots, x_{n-1}, x_n) =$$
$$g\, P(x_1, P(x_2, \cdots, P(x_{n_1}, x_n) \cdots)).$$

Unary equivalents of arithmetic functions $+$, $\dot{-}$ (modified subtraction), $\cdot$, $\div$ (integer division), and x^y can be derived in $\mathcal{PR}_i$ by unary iteration simulating the usual recursion.

Theorem 7 The classes of primitive pair recursive functions $\mathcal{PR}_i$ consist exactly of unary primitive recursive functions.

The proof is relatively straightforward where $\subset$ is proved by a primitive recursive derivation of the pairing function P and $\supset$ is proved by showing that the unary equivalents of primitive recursive functions are in $\mathcal{PR}_i$. $\blacksquare$

2.3 Beyond primitive recursion

The class of feasibly computable functions, i.e. the class of functions computable in polynomial time, was characterized in [17] as a very small subset of elementary functions which in turn is a very small subset of primitive recursive functions. We cannot possibly hope to compute on computers elementary and higher primitive recursive functions except for small arguments. In this section we will go beyond primitive recursive functions and make $\mathcal{LESP}$ a language which can define a huge class of functions. The characterization of $\mathcal{LESP}$ functions will be given by the theorems 9 and 10. There is no technical problem with extending $\mathcal{LESP}$ even further. We stop with the class of functions characterized by the theorem 10 because this is the smallest class which permits a proof-theoretic characterization of formal arithmetic. Thus the extended $\mathcal{LESP}$ will be a theoretical vehicle used in the development of logic. The extension of $\mathcal{LESP}$ beyond primitive recursive functions will give us also programming convenience where we will be able to program subelementary functions by using the apparatus of functional programming languages.

One way to go beyond primitive recursive functions is to lengthen the iteration sequences by means of ordinals. The function O with the properties $O(0) = 0$ and $O(a,b) = w^{O(a)} + O(b)$ maps the natural numbers onto the segment of ordinals less than ϵ_0. By setting $x \prec_1 y \leftrightarrow O(x) < O(y)$ we get a well-founded transitive relation $\prec_1$. We define the predicates of being an *ordinal code* $Oc(x)$ and a well-order $\prec$ which is a restriction of $\prec_1$ to Oc to be primitive recursive by simultaneous $\mathcal{LESP}$ definitions. The definitions use course of values recursion and recursion with substitution in parameters (see [11]):

$$Oc(0)$$
$$Oc(x,y) \leftarrow Oc(x) \wedge Oc(y) \wedge H(y) \preceq x$$
$$x \preceq y \leftarrow x \prec y \vee x = y$$
$$0 \prec x,y \leftarrow Oc(x,y)$$
$$x_1,y_1 \prec x_2,y_2 \leftarrow Oc(x_1,y_1) \wedge$$
$$Oc(x_2,y_2) \wedge x_1 = x_2 \wedge y_1 \prec y_2$$
$$x_1,y_1 \prec x_2,y_2 \leftarrow Oc(x_1,y_1) \wedge$$
$$Oc(x_2,y_2) \wedge x_1 \neq x_2 \wedge x_1 \prec x_2.$$

We can see that a number is an ordinal code iff it is 0 or else, if taken as a list, its elements are ordinal codes ordered by $\succeq$. Clearly, the map O restricted to Oc is a similarity map between Oc and ordinals less than ϵ_0. We note that the code of a finite ordinal n is $\sigma(n)$ and that $\omega_o = (1,0) = 3$ codes the ordinal ω.

Ordinal codes used for the proof-theoretic analysis of subsystems of analysis are for a much larger segment of ordinals than ϵ_0. Our coding of ordinals is sufficient for the proof-theoretic analysis of formal arithmetic and we will not go beyond it as we are interested only in the programming aspects of logic.

The functions $+_o$, $\cdot_o$, and ω_o^α performing the ordinal arithmetic on codes Oc are easily defined by primitive recursive $\mathcal{LESP}$ definitions.

For an ordinal code α we say that a function f is defined by the $O(\alpha)$-*iteration* of g, in writing $f(x) = g^{O(\alpha)}(x)$, if

$g^{O(\alpha)}(x) = g^{O(\alpha)}(x_1) \leftarrow$
$\quad H(x) \prec \alpha \wedge g(x) = x_1 \wedge H(x_1) \prec H(x)$
$g^{O(\alpha)}(x) = x \leftarrow$
$\quad H(x) \prec \alpha \wedge g(x) = x_1 \wedge H(x_1) \not\prec H(x)$
$g^{O(\alpha)}(x) = x \leftarrow H(x) \not\prec \alpha.$

The ϵ_0-*iteration* of g is defined similarly where we replace the application of $H(x) \prec \alpha$ by $Oc\,H(x)$ in all clauses for g^{ϵ_0}. The function g is iterated as long as the head of its argument descends in $\prec$.

For $\omega \leq \alpha \leq \epsilon_0$ the class $\mathcal{R}^\alpha$ of α-*recursive functions* is generated from the functions I, H, T, and Rt by composition, pairing and α-iteration. The class $\mathcal{R}^{<\epsilon_0}$ of *less than* ϵ_0-*recursive functions* is the union of $\mathcal{R}^\alpha$ for all $\omega \leq \alpha < \epsilon_0$. It is not too difficult to see that $\mathcal{R}^\omega = \mathcal{PR}_i$ as ω-iteration can be reduced to pair iteration and vice versa. We also have $\mathcal{R}^\alpha \subset \mathcal{R}^\beta$ if $\alpha < \beta$.

The well-known Péter's function ϕ [10] is an example of a non-primitive recursive function. It can be defined by *nested recursion with a measure* as follows:

$\phi(0, y) = s(y)$
$\phi(s(x), 0) = \phi(x, 1)$
$\phi(s(x), s(y)) = \phi(x, \phi(s(x), y)).$

It is easy to see that the *measure function*

$$m(x, y) = \omega_o \cdot_o \sigma(x) +_o \sigma(y) \prec \omega_o^{\sigma 2}$$

when applied to the arguments of ϕ goes down for all recursive applications. The nested recursion can be replaced by ω^ω-iteration so we get $\phi \in \mathcal{R}^{\omega^\omega}$:

$\phi(x, y) = v \leftarrow$
$\quad y, x, 0 = i \wedge g^{\omega^\omega}(m_1(i), i) = \alpha, v, s$
$g(\alpha, y, 0) = \alpha, y, 0$
$g(\alpha, y, 0, s) = m_1(v), v \leftarrow s(y), s = v$
$g(\alpha, 0, s(x), s) = m_1(v), v \leftarrow 1, x, s = v$
$g(\alpha, s(y), s(x), s) = m_1(v), v \leftarrow$
$\quad y, s(x), x, s = v.$

We can see that

$$g^{\omega^\omega}(\alpha, y, x_1, \cdots, x_{n-1}, x_n, 0) = \\ \alpha_1, \phi(x_n, \phi(x_{n-1}, \cdots, \phi(x_1, y) \cdots)), 0$$

for suitable ordinal codes α and α_1. The function m_1 is defined to satisfy

$$m_1(y, x_1, x_2, \cdots, x_n, 0) = \omega_o^{\sigma(x_n+1)} +_o \cdots \\ +_o \omega_o^{\sigma(x_2+1)} +_o \omega_o^{\sigma(x_1)} \cdot_o \sigma(y+1) +_o \sigma(y)$$

for $n \geq 1$ and be 0 otherwise. We have $m_1(x) \prec \omega_o^{\omega_o}$. It is left to the reader to convince himself that the iteration of g decreases the ordinal code in the head until the stack is exhausted.

The above example is an instance of the theorem of Tait [15, 11] which says that a nested α-recursion can be reduced to 2^α-iteration.

Another way of enlarging the class of primitive pair recursive functions, which turns out to be equivalent to ordinal recursion, is by defining ϵ_0-recursive functions $\bullet$, A, Val_s^k, $Code$ and consider a subset $\mathcal{T}$ of functions primitive recursive in these functions: $\mathcal{T} \subset \mathcal{R}^{\epsilon_0}$. We will use left associative infix notation for $\bullet$ where $x \bullet y \bullet z$ will stand for $\bullet(\bullet(x, y), z)$. The class $\mathcal{T}$ will consist of *intensional functionals* such that when we assign an *index* (code) $\ulcorner f \urcorner$ to a function $f \in \mathcal{T}$ we will get $\ulcorner f \urcorner \bullet y = f(y)$ for all y in a subset of $\mathbf{N}$ called the *domain* of f. Intensional functionals are functions which accept and/or yield indices of functions. In the following we will drop the qualifier 'intensional' and speak only of functionals.

The function $\bullet$ is a kind of a universal function so we have to be careful with the formation rules of $\mathcal{T}$ and render the self-application $\ulcorner f \urcorner \bullet \ulcorner f \urcorner$ harmless. Since $\mathcal{T}$ consists of total functions we cannot disallow self-application but we can prohibit $\ulcorner f \urcorner \bullet \ulcorner f \urcorner = f(\ulcorner f \urcorner)$ by placing $\ulcorner f \urcorner$ outside of the domain of f. This will destroy the standard diagonal argument by which it is shown that the universal function for a class $\mathcal{T}$ closed under composition cannot be in $\mathcal{T}$.

The formation rules for $\mathcal{T}$ are specified with the help of *types* which are taken as numbers satisfying the predicate $Tp(x)$. The definition of the predicate Tp uses a technique which will be employed throughout the book. The objects of logic, i.e. terms, formulas, and proofs which are usually linguistic objects (i.e. certain words over an alphabet) will be defined in the section 3 as subsets of natural numbers. We do the same with the types and with *typed lambda terms* specified by the predicate *Tlt* which will be used as indices of functions in $\mathcal{T}$. By identifying the numeric codes of linguistic objects with the objects themselves we can operate on the objects by $\mathcal{LESP}$ functions. The predicates holding of objects are defined with the help of what are known in programming languages as *union data types*. We illustrate the technique with the types of $\mathcal{T}$.

We first define two $\mathcal{LESP}$ functions as $[x, y] = 0, x, y$ and $x \mapsto y = 1, x, y$. The functions will act as *variant constructors* in defining the set Tp to satisfy the set equation

$$Tp = \{0\} \cup [Tp, Tp] \cup Tp \mapsto Tp.$$

The three sets on the right are clearly mutually disjoint. The disjointness of constructors can be used in $\mathcal{LESP}$ definitions. The predicate Tp is defined primitive recursive as

$Tp(0)$
$Tp([s, t]) \leftarrow Tp(s) \wedge Tp(t)$
$Tp(s \mapsto t) \leftarrow Tp(s) \wedge Tp(t).$

We let both constructors to associate to the right. This means that we will write $[s, t, u]$ as an abbreviation for $[s, [t, u]]$. The type 0 stands for numbers, the type $[s, t]$ for cartesian-products, and the type $s \mapsto t$ for function types. This will be made precise below.

We assume until the proof of the theorem 9 that the functionals $\bullet$, A, Val^t_s, and $Code$ have been defined and that they have the properties given below. The function classes $\mathcal{T}_w$ where w is a function type are then defined for all types s, t, and u and numbers k as follows:

1. $I \in \mathcal{T}_{s \mapsto s}$ and $H \in \mathcal{T}_{[s,t] \mapsto s}$,
2. $\bullet \in \mathcal{T}_{[(s \mapsto t), s] \mapsto t}$ and $A \in \mathcal{T}_{([s,t] \mapsto u) \mapsto s \mapsto t \mapsto u}$,
3. $Val^k_s \in \mathcal{T}_{0 \mapsto s}$ and $Code \in \mathcal{T}_{s \mapsto 0}$,
4. if $g \in \mathcal{T}_{u \mapsto t}$ and $h \in \mathcal{T}_{s \mapsto u}$ then for $f(x) = g\, h(x)$ we have $f \in \mathcal{T}_{s \mapsto t}$,
5. if $g \in \mathcal{T}_{s \mapsto t}$ and $h \in \mathcal{T}_{s \mapsto u}$ then for $f(x) = g(x), h(x)$ we have $f \in \mathcal{T}_{s \mapsto [t,u]}$,
6. if $g \in \mathcal{T}_{s \mapsto s}$ then for $f(x) = g^{|H(x)|}T(x)$ we have $f \in \mathcal{T}_{[0,s] \mapsto s}$.

We set $\mathcal{T} = \bigcup_{Tp(s \to t)} \mathcal{T}_{s \to t}$.

We now introduce the *typing predicate* $x : s$ holding if 'x is a value of type s.' The predicate ':' can be defined by a primitive recursive $\mathcal{LESP}$ definition which we cannot give here without going into details about the typed lambda terms used as indices. We give a pseudo-$\mathcal{LESP}$ definition instead:

$x : 0$
$x : [0, 0]$
$x : [s, t] \leftarrow$
$\quad (s \neq 0 \vee t \neq 0) \wedge x = v, w \wedge v : s \wedge w : t$
$\ulcorner f \urcorner : (s \mapsto t) \leftarrow f \in \mathcal{F}_{s \mapsto t}.$

The reader will note that the definition makes the types 0 and $[0, 0]$ identical as sets (not as codes). Thus we are obliged to add a final formation rule (7) for $\mathcal{T}$: $\mathcal{T}_s = \mathcal{T}_t$ if the types s and t differ by having 0 and $[0, 0]$ in one or more corresponding positions.

Our use of typed lambda terms is similar to their standard use in logic as primitive recursive functionals of finite types or as typed lambda calculus enriched by primitive recursion discussed, for instance, in [7]. We just mention that the variant constructors and the predicate Tlt are defined to satisfy the set equation:

$$\begin{aligned} Tlt \;=\; & Var(\mathbf{N}, Tp) \cup Num(\mathbf{N}) \cup Tlt \,\bar{\bullet}\, Tlt \,\cup \\ & Tlt \,\bar{;}\, Tlt \cup \lambda(\mathbf{N}, Tp, Tlt) \cup Rec(Tlt) \,\cup \\ & \overline{Val}(\mathbf{N}, Tp) \cup \overline{Code}(Tp) \end{aligned}$$

Functions in $\mathcal{T}$ are *polymorphic* in the sense that they can be in more classes $\mathcal{T}_{s \mapsto t}$ at the same time. The 'domain' of a function $f \in \mathcal{T}_{s \mapsto t}$ is s and its 'range' is t. We have

$$x : s \wedge \ulcorner f \urcorner : (s \mapsto t) \to f(x) : t.$$

Functions of $\mathcal{T}$ in the group (1) are the usual identity and first projection functions.

Functions in the group (2) are *combinators*. The *application functional* $\bullet$ has the property

$$x : s \wedge \ulcorner f \urcorner : s \mapsto t \to \ulcorner f \urcorner \bullet x = f(x).$$

The *abstraction* or *Currying functional* A has the property

$$\begin{gathered} f : [s, t] \mapsto u \wedge x : s \wedge y : t \to \\ A(f) \bullet x \bullet y = f \bullet (x, y). \end{gathered}$$

Functions in the group (3) play a similar role as *type coercion* functions as we know them from computer programming languages. The *valuation* functional Val^k_s converts a natural number x, such that $Tlt(x)$ and the typed lambda term x can be typed as s, into a value of that type: $Val_s(x) : s$. The value yielded is the typed lambda term x reduced to its *normal form* (see [7]). Valuation functionals permit a 'run-time' construction, for instance by compilation, of typed lambda terms which can be then converted into (indices of) functions and later applied to. The superscript k restricts the arguments of Val^k_s to the typed lambda terms of *type level* less than k. The *type level* $Tl(s)$ of a type s is defined to satisfy

$$Tl(0) = 0$$
$$Tl[s,t] = max(Tl(s), Tl(t))$$
$$Tl(s \mapsto t) = max(Tl(s)+1, Tl(t)).$$

The *type level* of a typed lambda term is defined as the maximal type level of functionals occurring in it. The functional $Code(a)$ retypes a value $a : s$ into the same number $a : 0$. It can be used to analyze the functions passed as arguments as typed lambda terms.

Functionals obtained by the rules in the groups (4), (5) and (6) are the typed counterparts of function operators used in the construction of primitive pair recursive functions.

The reader will note that if we use the formation rules for $\mathcal{T}$ only with the type 0 (taking into account that $[0,0]$ is 0) and never use the functionals in the groups (2) and (3) we obtain all primitive pair recursive functions as a subset of $\mathcal{T}_{0 \mapsto 0}$.

The language $L\mathcal{E}SP$ as developed in the section 2.2 naturally extends to accommodate the functionals in $\mathcal{T}$. If we take an explicit definition of the form $f(x) = \boldsymbol{a}$ of the type discussed in the theorem 5 where the functions applied in $\boldsymbol{a}$ are from $\mathcal{T}$ we get $f \in \mathcal{T}_{s \mapsto t}$ provided the term $\boldsymbol{a}$ is *well-typed*. What this means should be obvious from the proof of the theorem 5 if we assume that the argument x is of type s and derive that the term $\boldsymbol{a}$ is of type t. The term $\boldsymbol{a}$ is a meta-term, i.e. a string of symbols. If we code it into *Tlt* we get the typed lambda term c and then $\ulcorner f \urcorner = \lambda x^s(c)$.

The functional A is repeatedly used in the proof by induction on n of the following theorem:

Theorem 8 (Abstraction theorem) If $h \in \mathcal{T}_{[s_1,s_2,\cdots,s_n] \mapsto t}$ $(n \geq 2)$ then there is a function $f \in \mathcal{T}_{s_1 \mapsto s_2 \mapsto \cdots \mapsto s_n \mapsto t}$ such that

$$x_1 : s_1 \wedge x_2 : s_2 \wedge \cdots \wedge x_n : s_n \rightarrow$$
$$\ulcorner f \urcorner \bullet x_1 \bullet x_2 \bullet \cdots \bullet x_n = h(x_1, x_2, \cdots, x_n).$$

We will have

$$\ulcorner f \urcorner = \lambda x_1^{s_1} x_2^{s_2} \cdots x_n^{s_n}(h(x_1, x_2, \cdots, x_n)).$$

Because of the closure of $\mathcal{T}$ under the typed version of pair iteration (the formation rule (6)) all well-typed recursive schemas of $L\mathcal{E}SP$ which reduce to pair iteration are admissible in the definition of $\mathcal{T}$ functions. We illustrate this point with the derivation of the Péter's function in $\mathcal{T}_{0 \mapsto 0}$. We set $\phi(x,y) = B(x) \bullet y$ where

$$B(0) = \ulcorner s \urcorner$$
$$B(s(x)) = g\, B(x)$$

by primitive recursion,

$$\ulcorner g \urcorner \bullet b \bullet y = h(y,b)$$

by abstraction, and

$$h(0,b) = b \bullet 1$$
$$h(s(y),b) = b \bullet h(y,b)$$

by primitive recursion. We leave it to the reader to type the functionals and to convince himself that they indeed define ϕ.

Theorem 9 $\mathcal{T} \subset \mathcal{R}^{\epsilon_0}$.

Sketch of the proof: We first define a function $Valt(k,a)$ which takes a typed lambda term a and yields its normal form provided the type level of a is less than k. This is the most difficult part of the proof. The definition is made by the adaptation of techniques discussed in [12] to include the pairing and functionals from the group (3). We get *Valt* ϵ_0-recursive. For a fixed k the function $Valt_k(a) = Valt(k,a)$ is ω_k-recursive where $\omega_0 = 1$ and $\omega_{k+1} = \omega^{\omega_k}$. ■

We next define some auxiliary p.r. functions. The function $Type(a) = 0, s$ extracts the type s of the typed lambda term a or else yields 0. The function $Term(x,s)$ converts the value $x : s$ into a typed lambda term which denotes x and the function *Den* is an inverse of *Term*: $Den\ Term(x,s) = x$. We then define

$$Val(k,a,s) = Den\ Valt(k,a) \leftarrow$$
$$Type(a) = 0, s \wedge Tl(a) < k$$

yielding the value denoted by the normal form of the typed lambda term a of type s and type level less than k. For a fixed k the function is ω_k-recursive. The application functional $\bullet$ is defined ϵ_0-recursive as

$$x \bullet y = Val(Tl(x)+1, x \bar{\bullet} Term(y,s), t) \leftarrow$$
$$Type(x) = 0, (s \mapsto t) \wedge y : s$$

The valuation functionals are defined ω_k-recursive:

$$Val_s^k(a) = Val(k, a, s) \leftarrow Type(a) = 0, s.$$

The functional *Code* is just I. The combinator A is ϵ_0-recursive:

$$\begin{aligned} &A(f) = Val(Tl(f) + 1, a, w) \leftarrow \\ &\quad Type(f) = 0, [s,t] \mapsto u \wedge \\ &\quad a = \lambda x^s y^t (f \,\bar{\bullet}\, (x \,\bar{,}\, y)) \wedge \\ &\quad w = s \mapsto t \mapsto u. \end{aligned}$$

We can now prove the properties of functionals as postulated above. ■

Theorem 10 $\mathcal{T}_{0 \mapsto 0} = \mathcal{R}^{<\epsilon_0}$.

Sketch of the proof: $\subset$: A function $f \in F_{0 \mapsto 0}$ is $\omega_{Tl(\ulcorner f \urcorner)+1}$-recursive as

$$f(x) = Val(Tl(\ulcorner f \urcorner) + 1, \ulcorner f \urcorner \,\bar{\bullet}\, Term(x, 0), 0).$$

$\supset$: By induction on k we can show $\mathcal{T}_{0 \mapsto 0}$ closed under ω_k-iteration. The proof uses the inverse Tait's theorem [15, 11] saying that 2^α-iteration is reducible to nested α-recursion which is expressible in $\mathcal{T}$ similarly as we did with the Péter's function. ■

We can now reveal the rationale behind the name *LESP*. It is an acronym for *Less than* ϵ_0 *liSP*. Indeed, with the valuation functionals *Val* playing the role of *Eval* and with the application functional $\bullet$ playing the role of *Apply*, we can see that *LESP* is a scaled-down and tidied-up version of LISP with a single atom 0. With the benefit of hindsight it is now easy to claim that this is what LISP should have always been and probably would have been if John McCarthy has carried out his remarkable recursion-theoretic analysis of LISP [9] to its logical conclusions. The reader should take into account that the analysis was made at the dawn of the computing age in 1963. Since then the formal semantics of programming languages went through complicated high-powered set-theory-based models and even through categories. We hope that the design of *LESP* convinces some of the readers that the recursion-theoretic analysis of programming languages is more than sufficient.

The users of LISP has always claimed that theirs is an untyped language and that the use of types restricts the freedom of forming functions. This is indeed the case. Without the typing and without the restriction to certain legal schemas of recursion we cannot make LISP to compute anything less than the class of partial recursive functions.

Finally, we would like to correct the common misconception about LISP as a functional programming language. LISP, just as *LESP*, works only with indices of functions which are S-expressions. By working with intensional functionals both *LESP* and LISP enjoy tremendous advantage over functional languages in that both languages can directly manipulate indices of functionals. This is done in *LESP* by the functionals from the group (3).

Part II: Proof theory, Logic by programming

3 Object language

We will study predicate calculus and formal theories as *objects* by partially formalized English in the *meta-theory*. The partial formalization comes from our use of $\mathcal{LESP}$ functions. $\mathcal{LESP}$ functions operate on natural numbers, hence the objects of logic, i.e. terms, formulas, and proofs must be subsets of natural numbers.

The primitive recursive $\mathcal{LESP}$ predicates *Sterm* and *Term* (*Term* $\subset$ *Sterm*) holding of *semi-terms* and *terms* respectively are defined with the help of three variant constructors:

$$
\begin{aligned}
&Bvar(x) = 0, x\\
&Sc(x) = 1, x\\
&Applf(x) = 2, x.
\end{aligned}
$$

The predicates *Sterm* and *Sterms* are defined by simultaneous primitive recursion

$$
\begin{aligned}
&Sterm(Bvar(n))\\
&Sterm(Sc(n))\\
&Sterm(Applf(n, a)) \leftarrow Sterms(a)\\
&Sterms(0)\\
&Sterms(a, b) \leftarrow Sterm(a) \wedge Sterms(b).
\end{aligned}
$$

The predicate *Sterms*(a) holds iff a is a list of semi-terms. The pair of predicates *Term* and *Terms* has a similar definition except that the first clause for *Sterm* is omitted. We will use the meta-variables $\boldsymbol{a}$, $\boldsymbol{b}$, ... to range over both semi-terms and terms and $\boldsymbol{as}$, $\boldsymbol{bs}$, ... to range over (possibly empty) lists of these. We adopt the convention of designating meta-variables ranging over lists of a syntactic sort by the suffix $\boldsymbol{s}$ for all meta-variables ranging over the sort. A *function symbol* is a number of the form (a, n) where n is the *name* of the function symbol and $a \geq 0$ is its arity. We let $\boldsymbol{f}$, $\boldsymbol{g}$, ... to range over function symbols.

The semi-term of the form $Bvar(n)$ is the *n-th (bound) variable*. We will use the meta-variables $\boldsymbol{x}$, $\boldsymbol{y}$, ... to range over variables. The semi-term of the form $Sc(n)$ is a *special kind of constant*. Special kind of constants are intended to be semantically similar to the ordinary constants which are the function symbols $(0, n)$ of arity 0. The semi-term of the form $Applf(n, \boldsymbol{as})$ is the *application* of the function symbol $\boldsymbol{f} = Len(\boldsymbol{as}), n$ to the *arguments* $\boldsymbol{as}$. The above application will be abbreviated to $\boldsymbol{f}(\boldsymbol{as})$ with the understanding that the arity of $\boldsymbol{f}$ is $Len(\boldsymbol{as})$. We will often write $\boldsymbol{f}$ for $\boldsymbol{f}(0)$, i.e. for the application of the nullary function symbol $\boldsymbol{f}$ which is, of course, a constant symbol.

The primitive recursive $\mathcal{LESP}$ predicates *Sform* and *Form* (*Form* $\subset$ *Sform*) holding of *semi-formulas* and *formulas* respectively are defined with the help of the following variant constructors:

$$
\begin{aligned}
&Applp(x) = 0, x\\
&Eq(x) = 1, x\\
&\neg(x) = 2, x\\
&x \wedge y = 3, x, y\\
&x \vee y = 4, x, y\\
&x \rightarrow y = 5, x, y\\
&\exists(x) = 6, x\\
&\forall(x) = 7, x
\end{aligned}
$$

and constants

$$
\begin{aligned}
&\top = 8, 0\\
&\bot = 9, 0.
\end{aligned}
$$

We leave to the reader the extraction of primitive recursive $\mathcal{LESP}$ definitions of *Sform* and *Form* from the following description where the meta-variables $\boldsymbol{A}$, $\boldsymbol{B}$, ... range over semi-formulas and formulas. The semi-formula of the form $Applp(n, \boldsymbol{as})$ is the *application* of the *predicate symbol* $\boldsymbol{p} = Len(\boldsymbol{as}), n$ to the arguments $\boldsymbol{as}$. This will be abbreviated to $\boldsymbol{p}(\boldsymbol{as})$ with the understanding about the arity. The reader will notice that we treat predicate symbols similarly to function symbols and use the meta-variables $\boldsymbol{p}$, $\boldsymbol{q}$, ... to range over them. The semi-formula of the form $Eq(\boldsymbol{a}, \boldsymbol{b})$, which will be abbreviated to $\boldsymbol{a} = \boldsymbol{b}$ in contexts where it cannot be confused with the meta-logical symbol $=$ of identity, is an *identity* semi-formula. The semi-formula of the form $\neg(\boldsymbol{A})$, written as $\neg\boldsymbol{A}$, is a *negation* semi-formula. We abbreviate the formulas of the form $\neg\boldsymbol{a} = \boldsymbol{b}$ to $\boldsymbol{a} \neq \boldsymbol{b}$. The semi-formula of the form $\boldsymbol{A} \wedge \boldsymbol{B}$ is a *conjunctive* semi-formula, *disjunctive* ($\vee$) and *implicative* ($\rightarrow$) semi-formulas are similar. The semi-formula of the form $\exists(n, \boldsymbol{a})$, written as $\exists\boldsymbol{x}\boldsymbol{A}$ where $\boldsymbol{x} = Bvar(n)$, is an *existential quantifier* semi-formula. *Universal quantifier* semi-formulas are similar. The semi-formulas $\top$ and $\bot$ are intended to denote the *truth* and *falsehood* respectively.

We give the usual precedence to the binary propositional connectives where $\rightarrow$, $\vee$, $\wedge$ are listed in the order of increasing precedence. All three connectives associate to the right. $\boldsymbol{A} \leftrightarrow \boldsymbol{B}$ is an abbreviation for $(\boldsymbol{A} \rightarrow \boldsymbol{B}) \wedge (\boldsymbol{B} \rightarrow \boldsymbol{A})$.

The definition of *free* and *bound* occurrences of variables is the usual one. A semi-formula with all occurrences of variables bound is a *formula*. Note that this makes all formulas and terms *closed*, i.e. without *free* variables. Formulas and terms with no special constants are *pure*.

The notation $\boldsymbol{A}[\boldsymbol{x}]$ is an *indication* that the variable $\boldsymbol{x}$ may occur free in the semi-formula $\boldsymbol{A}$. Then $\boldsymbol{A}[\boldsymbol{a}]$ stands for the semi-formula $Sub(\boldsymbol{A}, \boldsymbol{x}, \boldsymbol{a})$ where the primitive recursive function *Sub* of *substitution* is the usual one.

For any existential quantifier formula $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$ the special kind of constant $Sc(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}])$ is called a *special constant for* the formula. In order to save space we will use the meta-variables $\boldsymbol{r}$, $\boldsymbol{s}$, $\ldots$ for special constants. To every term and formula we assign by primitive recursive functions $Sctl(\boldsymbol{a})$ and $Scfl(\boldsymbol{A})$ a *special constant level*. The terms and formulas which do not contain any special constants will obtain the level 0. Such terms and formulas are called *pure*. Terms and formulas which contain at most special constants for pure existential quantifier formulas and they contain at least one such will obtain the level 1. In general, terms and formulas which contain at most special constants for formulas of level i with at least one formula of level i will obtain the level $i + 1$. The role of special constants for existential formulas will be discussed in the chapter 5.

We allow lists of formulas in contexts $\boldsymbol{As} \rightarrow \boldsymbol{A}$, $\boldsymbol{As} \vee \boldsymbol{A}$ and $\boldsymbol{As} \wedge \boldsymbol{A}$ to stand for formulas obtained by *distributing* the connective over the list. The formula $\boldsymbol{As} \rightarrow \boldsymbol{B}$ is an abbreviation for the formula obtained by the function $\rightarrow_s$ as $\boldsymbol{As} \rightarrow_s \boldsymbol{B}$ where the primitive recursive function $\rightarrow_s$ is defined as follows:

$(0 \rightarrow_s \boldsymbol{B}) = \boldsymbol{B}$

$((\boldsymbol{A}, \boldsymbol{As}) \rightarrow_s \boldsymbol{B}) = (\boldsymbol{A} \rightarrow (\boldsymbol{As} \rightarrow_s \boldsymbol{A}))$.

The other two connectives with lists are defined similarly.

The quantified formula such as $\forall \boldsymbol{xs} \boldsymbol{A}[\boldsymbol{xs}]$ stands for the obvious formula with the distributed quantifier. The notation $\boldsymbol{as} = \boldsymbol{bs}$ where $Len(\boldsymbol{as}) = Len(\boldsymbol{bs})$ will be used as an abbreviation for the list of identities composed of the corresponding elements of both lists. Similarly for $\boldsymbol{as} \neq \boldsymbol{bs}$.

The notation $\boldsymbol{As} \subset \mathbf{T}$ where $\mathbf{T}$ is a set of formulas means $\{\boldsymbol{A} \mid \boldsymbol{A} \in \boldsymbol{As}\} \subset \mathbf{T}$.

4 Tautologies

A formula is a *propositional atom* if it is an atomic or a quantifier formula. The primitive recursive predicate *Patom* holds of propositional atoms. A *propositional valuation* σ is a list consisting of propostional atoms. The primitive recursive predicate $Patoms(\boldsymbol{A})$ yields a list consisting of all propositional atoms which are a part of the formula $\boldsymbol{A}$.

Sublists $\sigma \subset Patoms(\boldsymbol{A})$ play a role of rows in a boolean truth table. For a propositional atom $\boldsymbol{B} \in Patoms(\boldsymbol{A})$ we have $\boldsymbol{B} \in \sigma$ iff the propositional atom $\boldsymbol{B}$ is assigned the truth value *true*.

The primitive recursive predicate $True(\boldsymbol{A}, \sigma)$ holds whenever the formula $\boldsymbol{A}$ is *true in the propositional valuation* σ:

$$True(\boldsymbol{A}, \sigma) \leftarrow Patom(\boldsymbol{A}) \wedge \boldsymbol{A} \in \sigma$$
$$True(\neg\boldsymbol{A}, \sigma) \leftarrow \neg True(\boldsymbol{A}, \sigma)$$
$$True(\boldsymbol{A}\wedge\boldsymbol{B}, \sigma) \leftarrow True(\boldsymbol{A}, \sigma) \wedge True(\boldsymbol{B}, \sigma)$$
$$True(\boldsymbol{A}\vee\boldsymbol{B}, \sigma) \leftarrow True(\boldsymbol{A}, \sigma) \vee True(\boldsymbol{B}, \sigma)$$
$$True(\boldsymbol{A}\rightarrow\boldsymbol{B}, \sigma) \leftarrow \neg True(\boldsymbol{A}, \sigma) \vee True(\boldsymbol{B}, \sigma)$$
$$True(\top, \sigma).$$

The predicate $Taut(\boldsymbol{A})$ holds of formulas $\boldsymbol{A}$ which are *(propositional) tautologies*:

$$Taut(\boldsymbol{A}) \text{ iff } True(\boldsymbol{A}, \sigma) \text{ for all } \sigma \subset Patoms(\boldsymbol{A}).$$

There are only finitely many propositional valuations $\sigma \subset Patoms(\boldsymbol{A})$. Thus the predicate *Taut* is primitive recursive.

We can see that the property of being a tautology depends only on the properties of propositional connectives. Truth values of atomic and quantifier formulas do not matter as tautologies are true formulas regardless of the truth values of its propositional atoms. We now state some simple properties of tautologies:

Lemma 11 If the propositional valuations σ_1 and σ_2 *agree* on the list $Patoms(\boldsymbol{A})$, i.e. if for all $\boldsymbol{B} \in Patoms(\boldsymbol{A})$ we have $\boldsymbol{B} \in \sigma_1$ iff $\boldsymbol{B} \in \sigma_2$, then $True(\boldsymbol{A}, \sigma_1)$ iff $True(\boldsymbol{A}, \sigma_2)$.

Proof: By induction on formulas $\boldsymbol{A}$. The lemma is obvious for propositional atoms and propositional constants. If $\boldsymbol{A} = (\neg\boldsymbol{B})$ then $True(\neg\boldsymbol{B}, \sigma_1)$ iff not $True(\boldsymbol{B}, \sigma_1)$ iff by $Patoms(\boldsymbol{B}) \subset Patoms(\boldsymbol{A})$ and IH: not $True(\boldsymbol{B}, \sigma_2)$ iff $True(\neg\boldsymbol{B}, \sigma_2)$.

If $\boldsymbol{A} = (\boldsymbol{B} \wedge \boldsymbol{C})$ then $True(\boldsymbol{B}\wedge\boldsymbol{C}, \sigma_1)$ iff $True(\boldsymbol{B}, \sigma_1)$ and $True(\boldsymbol{C}, \sigma_1)$ iff by $Patoms(\boldsymbol{B}) \oplus Patoms(\boldsymbol{C}) \subset Patoms(\boldsymbol{A})$ and IH: $True(\boldsymbol{B}, \sigma_2)$ and $True(\boldsymbol{C}, \sigma_2)$ iff $True(\boldsymbol{B}\wedge\boldsymbol{C}, \sigma_2)$.

The other cases are similar. ■

Theorem 12

1. $Taut(\boldsymbol{A})$ iff $True(\boldsymbol{A}, \sigma)$ for all propositional valuations σ,
2. $True(\boldsymbol{As}\vee\bot, \sigma)$ iff $True(\boldsymbol{B}, \sigma)$ for some $\boldsymbol{B} \in \boldsymbol{As}$,
3. $True(\boldsymbol{As}\rightarrow\boldsymbol{A}, \sigma)$ iff $True(\boldsymbol{A}, \sigma)$ or not $True(\boldsymbol{B}, \sigma)$ for some $\boldsymbol{A} \in \boldsymbol{As}$,
4. if $Taut(\boldsymbol{A})$ and $Taut(\boldsymbol{A}\rightarrow\boldsymbol{B})$ then $Taut(\boldsymbol{B})$,
5. $Taut(\boldsymbol{A}\vee\bot)$ iff $Taut(\boldsymbol{A})$,
6. If $\boldsymbol{As} \subset \boldsymbol{Bs}$ then
   (a) if $Taut(\boldsymbol{As}\vee\bot)$ then $Taut(\boldsymbol{Bs}\vee\bot)$,
   (b) if $Taut(\boldsymbol{As}\rightarrow\boldsymbol{A})$ then $Taut(\boldsymbol{Bs}\rightarrow\boldsymbol{A})$.

Proof: (1) $\rightarrow$: Given a tautology $\boldsymbol{A}$ and a propositional valuation σ we construct a list σ_1 consisting of those elements of σ which are propositional atoms of $\boldsymbol{A}$. We have $True(\boldsymbol{A}, \sigma_1)$ and by the lemma 11 also $True(\boldsymbol{A}, \sigma)$. $\leftarrow$ is obvious.
(2)(3) by induction on $\boldsymbol{As}$,
(4) if for a propositional valuation σ we have $True(\boldsymbol{A}, \sigma)$ and also $True(\boldsymbol{A}\rightarrow\boldsymbol{B}, \sigma)$ then also $True(\boldsymbol{B}, \sigma)$. We now apply the lemma 11.
(5) follows from (1) and (2). ■
(6) The proof of (a) is similar to the proof of (b): if for a propositional valuation σ we have $True(\boldsymbol{As}\rightarrow\boldsymbol{A}, \sigma)$ then by (3) either $True(\boldsymbol{A}, \sigma)$ or for some $\boldsymbol{B} \in \boldsymbol{As}$ not $True(\boldsymbol{B}, \sigma)$. Hence also $True(\boldsymbol{Bs}\rightarrow\boldsymbol{A}, \sigma)$. We now use the lemma 11. ■

We now present another method for the recognition of tautologies: the method of *(propositional) tableaux*. Tableaux generally permit a faster derivation of tautologies than the method of truth tables which we used to define the notion of tautology. The main advantage of preferring tableaux over truth tables is that the former, unlike the latter, generalize to the cases where we consider the identity symbol and quantifiers.

Our tableaux are dual to the tableaux of Smullyan [14, 3]. Smullyan's tableaux are based on refutation ours (in the general case) on validity.

A *tableau* $\boldsymbol{X}$ *for* $\boldsymbol{As}$ can be visualized as a downward growing dyadic tree whose nodes are formulas. We start the tree with a branch where we write down the formulas from $\boldsymbol{As} = \boldsymbol{A}_1, \boldsymbol{A}_2, \cdots, \boldsymbol{A}_n, 0$ one after another and then append the tableau $\boldsymbol{X}$ to it. This is shown in the figure **??** (a).

A tableau $\boldsymbol{X}$ for $\boldsymbol{As}$ can be *expanded* at any *leaf* by applying a *unary* or *binary rule*. A rule consists of a *premise* which is a list of formulas (in the general case the premise can be an empty list) and of *conclusion(s)* which is a single formula in case of a unary rule and two formulas in case of a binary rule.

A *unary* rule has the form $\frac{\boldsymbol{Bs}}{\boldsymbol{B}}$ where we can extend a branch at a leaf by a formula $\boldsymbol{B}$ provided the formulas of $\boldsymbol{Bs}$ occur in the branch above the leaf. Such a unary expansion of the tableau (a) from the figure **??** is shown in (b).

A *binary* rule has the form $\frac{\boldsymbol{Bs}}{\boldsymbol{B}\ \ \boldsymbol{C}}$ where we can extend a branch at a leaf by two branches with formulas $\boldsymbol{B}$ and $\boldsymbol{C}$ provided the formulas of $\boldsymbol{Bs}$ occur in the branch above the leaf. Such a binary expansion of the tableau (a) from the figure **??** is shown in (c).

Tableaux are recursive objects as can be seen from the figure **??** where the tableau $\boldsymbol{X}$ for $\boldsymbol{As}$ contains a subtableau $\boldsymbol{Y}$ for $\boldsymbol{As} \oplus \boldsymbol{Bs}$ where the list $\boldsymbol{Bs}$ is formed by reading off all the formulas in the branch of $\boldsymbol{X}$ above the subtableau $\boldsymbol{Y}$. In other words any subtableau is a tableau for the entire branch above it.

The expansion rules for propositional tableaux, called the *propositional expansion rules*, will have the premises formed from lists of length one: $\boldsymbol{D}, 0$. We will abbreviate a unary rule with such a premise to $\frac{\boldsymbol{D}}{\boldsymbol{B}}$. For the propositional rules it will be always the case that $Taut(\boldsymbol{B} \rightarrow \boldsymbol{D})$. We will abbreviate a binary rule with such a premise to $\frac{\boldsymbol{D}}{\boldsymbol{B}\ \ \boldsymbol{C}}$. For the propositional rules it will be always the case that $Taut(\boldsymbol{B} \wedge \boldsymbol{C} \rightarrow \boldsymbol{D})$.

The *propositional expansion rules* consist of *negation* rules

$$\frac{\neg\neg \boldsymbol{B}}{\boldsymbol{B}},$$

disjunction rules:

$$\frac{\boldsymbol{B} \vee \boldsymbol{C}}{\boldsymbol{B}} \quad \frac{\boldsymbol{B} \vee \boldsymbol{C}}{\boldsymbol{C}} \quad \frac{\boldsymbol{B} \rightarrow \boldsymbol{C}}{\neg \boldsymbol{B}}$$

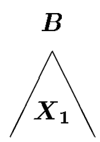
$$\frac{\boldsymbol{B} \rightarrow \boldsymbol{C}}{\boldsymbol{C}} \quad \frac{\neg(\boldsymbol{B} \wedge \boldsymbol{C})}{\neg \boldsymbol{B}} \quad \frac{\neg(\boldsymbol{B} \wedge \boldsymbol{C})}{\neg \boldsymbol{C}},$$

and *conjunction* rules:

$$\frac{\boldsymbol{B} \wedge \boldsymbol{C}}{\boldsymbol{B}\ \ \boldsymbol{C}} \quad \frac{\neg(\boldsymbol{B} \rightarrow \boldsymbol{C})}{\boldsymbol{B}\ \ \neg \boldsymbol{C}} \quad \frac{\neg(\boldsymbol{B} \vee \boldsymbol{C})}{\neg \boldsymbol{B}\ \ \neg \boldsymbol{C}}.$$

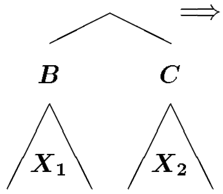
A tableau $\boldsymbol{X}$ for $\boldsymbol{As}$ is *closed* if each of its branches is *closed*. A branch is *closed* if it contains either two formulas $\boldsymbol{A}$ and $\neg \boldsymbol{A}$ or else at least one of the formulas $\top$ or $\neg\bot$.

We wish to operate on tableaux by *$\mathcal{LESP}$* functions. Hence we formally define tableaux as formulas. The leaf at every branch is an *empty tableau* $\bot$. The tableau starting with a unary rule:



$$\Longrightarrow \quad \boldsymbol{B} \vee \boldsymbol{X_1}$$

has the form $\boldsymbol{B} \vee \boldsymbol{X}_1$ where $\boldsymbol{X}_1$ is the tableau below the formula $\boldsymbol{B}$. The tableau starting with a binary rule:



$$\Longrightarrow \quad (\boldsymbol{B} \vee \boldsymbol{X_1}) \wedge (\boldsymbol{C} \vee \boldsymbol{X_2})$$

has the form $(\boldsymbol{B} \vee \boldsymbol{X}_1) \wedge (\boldsymbol{C} \vee \boldsymbol{X}_2)$ where $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are the tableaux under the fork. This definition of tableaux suggests that each branch is a disjunction of its formulas whereas each fork is a conjunction of branches. From the above mentioned properties of expansion rules we can see that if $\boldsymbol{X}$ is a tableau for $\boldsymbol{As}$ then

$$\boldsymbol{As} \vee \bot \leftrightarrow \boldsymbol{As} \vee \boldsymbol{X}$$

is a tautology. If $\boldsymbol{X}$ is a closed tableau for $\boldsymbol{As}$ then $\boldsymbol{As} \vee \boldsymbol{X}$ is a tautology. This suggests that the tableaux are sound. This fact will be proved in detail in the lemma 15.

The predicate $\boldsymbol{X}$ *is a closed propositional tableau for* $\boldsymbol{As}$ is written as $\boldsymbol{X} : \vdash_p \boldsymbol{As}$. It is obviously a primitive recursive predicate. We define the predicate $\boldsymbol{As}$ *is provable by a propositional tableau*, in writing $\vdash_p \boldsymbol{As}$, to hold if $\boldsymbol{X} : p \vdash^{\boldsymbol{As}}$ *f*or some $\boldsymbol{X}$.

The case when the list $\boldsymbol{As}$ consists of a single formula, i.e. $\boldsymbol{X} : \vdash_p \boldsymbol{A}, 0$ is abbreviated to $\boldsymbol{X} : \vdash_p \boldsymbol{A}$. For the predicate $\boldsymbol{A}$ *is provable by a propositional tableau* we similarly write $\vdash_p \boldsymbol{A}$.

Example 1 UNFINISHED:

The following lemma will be used often and usually without an explicit reference to it.

Lemma 13 (Weakening) If $\boldsymbol{As} \subset \boldsymbol{Bs}$ then

$$\frac{\boldsymbol{X} : \vdash_p \boldsymbol{As}}{\boldsymbol{X} : \vdash_p \boldsymbol{Bs}}.$$

Proof: By induction on $\boldsymbol{X}$. ∎

Theorem 14 (Soundness of propositional tableaux) If $\boldsymbol{X} : \vdash_p \boldsymbol{As}$ then $\mathit{Taut}(\boldsymbol{As} \vee \bot)$.

Proof: We will be referring directly to the points of the theorem 12 throughout this proof which goes by induction on tableaux $\boldsymbol{X}$.

If $\boldsymbol{X} = \bot$ then the branch $\boldsymbol{As}$ is closed and it is easy to see with the help of (2) that $\mathit{True}(\boldsymbol{As} \vee \bot, \sigma)$ for all propositional valuations σ. Hence $\mathit{Taut}(\boldsymbol{As} \vee \bot))$ by (1).

If $\boldsymbol{X} = \boldsymbol{A} \vee \boldsymbol{X}_1$ then $\boldsymbol{A}$ is the conclusion of a unary rule applied to a premise $\boldsymbol{B} \in \boldsymbol{As}$. By the remark about the property of unary rules $\boldsymbol{A} \to \boldsymbol{B}$ is a tautology. We also have $\boldsymbol{X}_1 : \vdash_p \boldsymbol{A}, \boldsymbol{As}$ and by induction hypothesis $(\boldsymbol{A}, \boldsymbol{As}) \vee \bot$ is a tautology. Let σ be a propositional valuation. By (1) we have $\mathit{True}((\boldsymbol{A}, \boldsymbol{As}) \vee \bot, \sigma)$. By (2) we get $\mathit{True}(\boldsymbol{A}, \sigma)$ or $\mathit{True}(\boldsymbol{As} \vee \bot, \sigma)$. If the former then by (4)(1) also $\mathit{True}(\boldsymbol{B}, \sigma)$ and by (2) $\mathit{True}(\boldsymbol{As} \vee \bot, \sigma)$. Hence in either case $\mathit{True}(\boldsymbol{As} \vee \bot, \sigma)$ and $\boldsymbol{As} \vee \bot$ is a tautology by (1).

If $\boldsymbol{X} = (\boldsymbol{A} \vee \boldsymbol{X}_1) \wedge (\boldsymbol{C} \vee \boldsymbol{X}_2)$ then $\boldsymbol{A}$ and $\boldsymbol{C}$ are in the conclusion of a binary rule with $\boldsymbol{B} \in \boldsymbol{As}$ as a premise. By the remark about the property of binary rules $\boldsymbol{A} \wedge \boldsymbol{C} \to \boldsymbol{B}$ is a tautology. We also have $\boldsymbol{X}_1 : \vdash_p \boldsymbol{A}, \boldsymbol{As}$ and $\boldsymbol{X}_2 : \vdash_p \boldsymbol{C}, \boldsymbol{As}$. By using the induction hypothesis twice we see that $(\boldsymbol{A}, \boldsymbol{As}) \vee \bot$ and $(\boldsymbol{C}, \boldsymbol{As}) \vee \bot$ are tautologies. By a similar reasoning as in the unary case we get that $\boldsymbol{As} \vee \bot$ is a tautology. ∎

Theorem 15 (Completeness of propositional tableaux) If $\mathit{Taut}(\boldsymbol{As} \vee \bot)$ then $\boldsymbol{X} : \vdash_p \boldsymbol{As}$ for some $\boldsymbol{X}$.

Proof: We temporarily define the *size* of a formula $\boldsymbol{A}$ as the total number of propositional connectives (but not of propositional constants) occurring in $\boldsymbol{A}$ where we do not count the connectives occuring within the quantifier semi-subformulas of $\boldsymbol{A}$. A propositional atom and the propositional constants $\bot$ and $\top$ have thus a size 0. The *rank* of a list of formulas $\boldsymbol{As}$ is the sum of sizes of all formulas $\boldsymbol{B} \in \boldsymbol{As}$ not counting the outermost negations. This means that if $(\neg \boldsymbol{B}) \in \boldsymbol{As}$ then the size contributed to the rank of $\boldsymbol{As}$ is the size of $\boldsymbol{B}$.

We will prove the theorem by induction on the rank of lists $\boldsymbol{As}$. Let us assume that $\mathit{Taut}(\boldsymbol{As} \vee \bot)$.

If the rank of $\boldsymbol{As}$ is 0 then $\boldsymbol{As}$ consists of propositional atoms, propositional constants, and their negations:

$$\boldsymbol{A},\ \top,\ \bot,\ \neg \boldsymbol{A},\ \neg \top,\ \neg \bot$$

where $\boldsymbol{A}$ is a propositional atom.

We claim that $\bot : \vdash_p \boldsymbol{As}$, i.e. that the branch $\boldsymbol{As}$ is closed. If this is not the case then the list $\boldsymbol{As}$ consists of

$$\boldsymbol{A},\ \bot,\ \neg \boldsymbol{A},\ \neg \top$$

and for no $\boldsymbol{A} \in \boldsymbol{As}$ we have both $\boldsymbol{A} \in \boldsymbol{As}$ and $(\neg \boldsymbol{A}) \in \boldsymbol{As}$. Let σ be a list of propositional atoms such that $\boldsymbol{A} \in \sigma$ iff $(\neg \boldsymbol{A}) \in \boldsymbol{As}$. For no propositional atom $\boldsymbol{A} \in \boldsymbol{As}$ we have $\boldsymbol{A} \in \sigma$. Hence for all $\boldsymbol{A} \in \boldsymbol{As}$ we have not $\mathit{True}(\boldsymbol{A}, \sigma)$ and by (2) of the theorem 12 not $\mathit{True}(\boldsymbol{As} \vee \bot, \sigma)$. We have a contradiction as by the lemma 11: $\boldsymbol{As} \vee \bot$ is not a tautology.

If the rank of $\boldsymbol{As}$ is greater than 0 then there is at least one $\boldsymbol{A} \in \boldsymbol{As}$ such that $\boldsymbol{A} = (\neg\neg \boldsymbol{B})$ or $\boldsymbol{A} = (\boldsymbol{B} \circ \boldsymbol{C})$ or $\boldsymbol{A} = \neg(\boldsymbol{B} \circ \boldsymbol{C})$ where $\circ$ is a propositional connective. Let $\boldsymbol{Bs}$ be a list like $\boldsymbol{As}$ but with the first such $\boldsymbol{A}$ removed. We continue by cases on the form of $\boldsymbol{A}$.

If $\boldsymbol{A} = (\neg\neg \boldsymbol{B})$ then $\mathit{Taut}((\boldsymbol{B}, \boldsymbol{Bs}) \vee \bot)$ by (6)(a), (2) of the theorem 12 and by the properties of *True*. The rank of $\boldsymbol{B}, \boldsymbol{Bs}$ is less than the rank of $\boldsymbol{As}$ and so by induction hypothesis we have $\boldsymbol{X}_1 : \vdash_p \boldsymbol{B}, \boldsymbol{Bs}$ for a tableau $\boldsymbol{X}_1$. Then $\boldsymbol{X} : \vdash_p \neg\neg \boldsymbol{B}, \boldsymbol{Bs}$ for the tableau $\boldsymbol{X} = \boldsymbol{B} \vee \boldsymbol{X}_1$

by a negation rule and the weakening lemma. Thus $\boldsymbol{X} : \vdash_p \boldsymbol{As}$ by the weakening lemma.

If $\boldsymbol{A} = (\boldsymbol{B} \vee \boldsymbol{C})$ then by a similar reasoning $Taut((\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Bs}) \vee \bot)$ and by induction hypothesis we have $\boldsymbol{X}_1 : \vdash_p \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Bs}$ for a tableau $\boldsymbol{X}_1$. Then $\boldsymbol{X} : \vdash_p (\boldsymbol{B} \vee \boldsymbol{C}), \boldsymbol{Bs}$ for the tableau $\boldsymbol{X} = \boldsymbol{B} \vee \boldsymbol{C} \vee \boldsymbol{X}_1$ by two disjunction rules. Thus $\boldsymbol{X} : \vdash_p \boldsymbol{As}$ by the weakening lemma.

If $\boldsymbol{A} = \neg(\boldsymbol{B} \vee \boldsymbol{C})$ then $Taut(\neg\boldsymbol{B}, \boldsymbol{Bs}) \vee \bot)$ and $Taut((\neg\boldsymbol{C}, \boldsymbol{Bs}) \vee \bot)$. Ranks of $\neg\boldsymbol{B}, \boldsymbol{Bs}$ and $\neg\boldsymbol{C}, \boldsymbol{Bs}$ are less than the rank of $\boldsymbol{As}$. Hence by two inductive hypotheses we get $\boldsymbol{X}_1 : \vdash_p \neg\boldsymbol{B}, \boldsymbol{Bs}$ and $\boldsymbol{X}_2 : \vdash_p \neg\boldsymbol{C}, \boldsymbol{Bs}$ for tableaux $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$. Then $\boldsymbol{X} : \vdash_p \neg(\boldsymbol{B} \vee \boldsymbol{C}), \boldsymbol{Bs}$ for the tableau $\boldsymbol{X} = (\neg\boldsymbol{B} \vee \boldsymbol{X}_1) \wedge (\neg\boldsymbol{C} \vee \boldsymbol{X}_2)$ by a conjuction rule. Thus $\boldsymbol{X} : \vdash_p \boldsymbol{As}$ by the weakening lemma.

The cases for the connectives $\wedge$ and $\rightarrow$ are similar. ■

Corollary 16 $Taut(\boldsymbol{A})$ iff $\vdash_p \boldsymbol{A}$.

Proof: Directly from the lemmas 14 and 15 with the help of (5) of the theorem 12. ■

5 Semantics of Predicate Logic

5.1 Consequence relations

For a set of formulas $\mathbf{T}$ and a formula $\boldsymbol{A}$ we will write $\mathbf{T} \models_p \boldsymbol{A}$, $\mathbf{T} \models_i \boldsymbol{A}$, $\mathbf{T} \models_h \boldsymbol{A}$, and $\mathbf{T} \models \boldsymbol{A}$, when the formula $\boldsymbol{A}$ is a *tautological*, *quasi-tautological*, *Henkin*, and *logical consequence* of $\mathbf{T}$ respectively. For all four qualifiers q we abbreviate $\emptyset \models_q \boldsymbol{A}$ to $\models_q \boldsymbol{A}$. The formula $\boldsymbol{A}$ is a *quasi-tautology* if $\models_i \boldsymbol{A}$ and *logically valid* if $\models \boldsymbol{A}$.

Intuitively, $\mathbf{T} \models_p \boldsymbol{A}$ holds when from the assumption that the formulas of $\mathbf{T}$ are true we can determine the truth of the formula $\boldsymbol{A}$ solely by the properties of propositional connectives. $\mathbf{T} \models_i \boldsymbol{A}$ holds when $\boldsymbol{A}$ follows from $\mathbf{T}$ by the properties of propositional connectives and of the identity symbol. $\mathbf{T} \models \boldsymbol{A}$ holds when $\boldsymbol{A}$ follows from $\mathbf{T}$ by the properties of all logical symbols, i.e. propositional, identity, and quantifiers. The relation $\mathbf{T} \models_h \boldsymbol{A}$ is used for reasons of symmetry and its properties will be seen below.

Example 2 A formula of the form $\boldsymbol{A} \vee \neg \boldsymbol{A}$ is a tautology. A formula of the form $\boldsymbol{a} = \boldsymbol{b} \vee \boldsymbol{b} \neq \boldsymbol{a}$ is a quasi-tautology but not a tautology. A formula of the form $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{a}] \vee \neg \boldsymbol{A}[\boldsymbol{a}]$ is a logically valid formula which is neither a quasi-tautology nor a tautology.

The reader interested only in the finitary development of predicate logic may decide to accept the definitions 1, 2, 3, and 4 as the definitions of the four consequence relations and then proceed directly to the chapter 6 skipping the remaining sections in this chapter. No further semantic assumptions will be needed in the part II.

The reader interested in a modern presentation of semantics of predicate logic should ignore the definitions 1, 2, 3, 4 given in this section and accept instead the definitions of consequence relations 5, 6, 8, and 7 given in the following sections. The theorems 20, 22, 30, and 32 corresponding to the definitions of consequence relations as given in this section are then derived by semantic, i.e. generally non-constructive, means.

Regardless of the reader's decision we need definitions of sets of formulas $\mathbf{Eq}$, $\mathbf{H}$, and $\mathbf{Q}$. We denote by $\mathbf{Eq}$ the *axioms of identity* which consist of *reflexive*, *symmetric*, *transitive* axioms listed in that order:

$$\begin{aligned} \boldsymbol{a} &= \boldsymbol{a} \\ \boldsymbol{a} = \boldsymbol{b} &\rightarrow \boldsymbol{b} = \boldsymbol{a} \\ \boldsymbol{a} = \boldsymbol{b} &\rightarrow \boldsymbol{b} = \boldsymbol{c} \rightarrow \boldsymbol{a} = \boldsymbol{c} \end{aligned}$$

and of the axioms of *function* and *predicate substitution* listed in that order:

$$\begin{aligned} \boldsymbol{as} = \boldsymbol{bs} &\rightarrow \boldsymbol{f}(\boldsymbol{as}) = \boldsymbol{f}(\boldsymbol{bs}) \\ \boldsymbol{as} = \boldsymbol{bs} &\rightarrow \boldsymbol{p}(\boldsymbol{as}) \rightarrow \boldsymbol{p}(\boldsymbol{bs}). \end{aligned}$$

The axioms of identity express the desired properties of the identity symbol.

We denote by $\mathbf{H}$ the *Henkin witnessing axioms*

$$\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \rightarrow \boldsymbol{A}[Sc(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}])]. \tag{2}$$

and by $\mathbf{Q}$ the *quantifier axioms* which consist of $\mathbf{H}$ and of *Henkin counterexample axioms*, *existential* and *universal substitution axioms* listed in that order:

$$\begin{aligned} \boldsymbol{A}[Sc(\exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}])] &\rightarrow \forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \qquad (3) \\ \boldsymbol{A}[\boldsymbol{a}] &\rightarrow \exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \\ \forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] &\rightarrow \boldsymbol{A}[\boldsymbol{a}]. \end{aligned}$$

When the Henkin witnessing axiom (2) holds then the special constant $\boldsymbol{r}$ for the formula $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$ denotes the *witness* which, when assigned to the variable $\boldsymbol{x}$, makes the semi-formula $\boldsymbol{A}[\boldsymbol{x}]$ true provided that $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$ is true.

If both the Henkin counterexample axiom (3) and the Henkin witnessing axiom

$$\exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}] \rightarrow \neg \boldsymbol{A}[\boldsymbol{r}] \tag{4}$$

where $\boldsymbol{r} = Sc(\exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}])$ hold then the formula $\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$ is true iff its *counterexample* denoted by the special constant $\boldsymbol{r}$ satisfies the semi-formula $\boldsymbol{A}[\boldsymbol{x}]$ when assigned to the variable $\boldsymbol{x}$ i.e. iff $\boldsymbol{A}[\boldsymbol{r}]$ is a true formula. The direction $\rightarrow$ is obvious. For the direction $\leftarrow$ we observe that if the formula $\boldsymbol{A}[\boldsymbol{r}]$ is true then by (4) we have $\neg \exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}]$ true and thus also $\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$ is true. We note that if the semi-formula $\boldsymbol{A}[\boldsymbol{x}]$ can be falsified at all then it will be falsified by its counterexample.

The semantic notions of consequence can be reduced to questions about sets of natural numbers either by accepting the definitions 1, 2, 3, and 4 or, alternatively, with the help of the corresponding theorems 20, 22, 30, and 32. In the latter case the set-theoretical concepts of structures needed for the definition of Henkin and logical consequence will be replaced by sets of natural numbers. Semantics can be formulated and answered in a relatively weak formal theory called the *Second-order arithmetic*.

The reader interested in the semantics of predicate calculus should ignore the rest of this section and proceed to the next one.

Definition 1 (Tautological reduction) $\mathbf{T} \models_p \boldsymbol{A}$ if there is a list of formulas $\boldsymbol{As}$ such that $\boldsymbol{As} \subset \mathbf{T}$ and $Taut(\boldsymbol{As} \to \boldsymbol{A})$.

Definition 2 (Quasi-tautological reduction) $\mathbf{T} \models_i \boldsymbol{A}$ if $\mathbf{T}, \mathbf{Eq} \models_p \boldsymbol{A}$.

Definition 3 (Henkin reduction) $\mathbf{T} \models_h \boldsymbol{A}$ if $\mathbf{T}, \mathbf{Q} \models_i \boldsymbol{A}$.

Definition 4 (Pure formula reduction) For a pure formula $\boldsymbol{A}$ and a set of pure formulas $\mathbf{T}$ $\mathbf{T} \models \boldsymbol{A}$ iff $\mathbf{T} \models_h \boldsymbol{A}$.

Theorem 17 *$Taut(\boldsymbol{A})$ iff $\models_p \boldsymbol{A}$.*

Proof: $Taut(\boldsymbol{A})$ iff $Taut(0 \to \boldsymbol{A})$ iff exists $\boldsymbol{As} \subset \emptyset$ such that $Taut(\boldsymbol{As} \to \boldsymbol{A})$ iff by the tautological reduction $\emptyset \models_p \boldsymbol{A}$. ∎

The reader interested only in the finitary aspects of predicate logic should proceed directly to the chapter 6.

5.2 Tautological reduction

This and the remaining sections in this chapter are a refinement of an excellent introduction to predicate logic by Jon Barwise in the Handbook of mathematical logic [1]. We have refined the reduction process by inserting the quasi-tautological and Henkin reductions. Our reasons for this will be appreciated in the next chapters.

We now extend the propositional valuations from the chapter 4 where they are lists of propositional atoms to possibly infinite sets where a *propositional set valuation* $\mathbf{S}$ is a subset of *Patom*. Propositional set valuations are closely connected to propositional truth sets:

A *propositional truth set* $\mathbf{S}$ is a subset of formulas such that

(1) $(\neg \boldsymbol{A}) \in \mathbf{S}$ iff $\boldsymbol{A} \notin \mathbf{S}$,

(2) $\top \in \mathbf{S}$ and $\perp \notin \mathbf{S}$,

(3) $(\boldsymbol{A} \wedge \boldsymbol{B}) \in \mathbf{S}$ iff $\boldsymbol{A} \in \mathbf{S}$ and $\boldsymbol{B} \in \mathbf{S}$,

(4) $(\boldsymbol{A} \vee \boldsymbol{B}) \in \mathbf{S}$ iff $\boldsymbol{A} \in \mathbf{S}$ or $\boldsymbol{B} \in \mathbf{S}$,

(5) $(\boldsymbol{A} \to \boldsymbol{B}) \in \mathbf{S}$ iff $(\neg \boldsymbol{A}) \in \mathbf{S}$ or $\boldsymbol{B} \in \mathbf{S}$.

We have the following straightforward lemma:

Lemma 18 To every propositional set valuation $\mathbf{S}_1$ there is a unique propositional truth set $\mathbf{S}$ such that $\mathbf{S}_1 = \mathbf{S} \cap Patom$.

The connection between the predicate *True* from the chapter 4 and propositional truth sets is given by the following lemma generalizing the lemma 11 and proved by induction on $\boldsymbol{A}$:

Lemma 19 If a propositional valuation σ and a propositional truth set $\mathbf{S}$ *agree* on the list $Patoms(\boldsymbol{A})$, i.e. if for all $\boldsymbol{B} \in Patoms(\boldsymbol{A})$ we have $\boldsymbol{B} \in \sigma$ iff $\boldsymbol{B} \in \mathbf{S}$, then $True(\boldsymbol{A}, \sigma)$ iff $\boldsymbol{A} \in \mathbf{S}$.

Definition 5 (Tautological consequence) A formula $\boldsymbol{A}$ is a *tautological consequence* of a set of formulas $\mathbf{T}$ if for every propositional truth set $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$.

We now prove the theorem 17: $Taut(\boldsymbol{A})$ iff $\models_p \boldsymbol{A}$.

Proof: $(\to)$: Assume that $\boldsymbol{A}$ is a tautology and that $\mathbf{S}$ is a propositional truth set. We wish to show $\boldsymbol{A} \in \mathbf{S}$. Let σ be any list of propositional atoms which has the same elements as the elements common to $\mathbf{S}$ and $Patoms(\boldsymbol{A})$. We have $True(\boldsymbol{A}, \sigma)$. The lemma 19 applies and we have also $\boldsymbol{A} \in \mathbf{S}$.

$(\leftarrow)$: Let σ be a propositional valuation and let $\mathbf{S}$ be the propositional truth set extending the propositional set valuation $\{\boldsymbol{A} \mid \boldsymbol{A} \in \sigma\}$ by the lemma 18. We have $\boldsymbol{A} \in \mathbf{S}$ from the assumption. The set $\mathbf{S}$ and the list σ agree on all propositional atoms so they agree also on propositional atoms of $\boldsymbol{A}$ and we get $True(\boldsymbol{A}, \sigma)$ by the lemma 19. ∎

The relation between tautologies and taulogical consequence is captured by the theorem 20 corresponding to the definition 1. The theorem is known as the *compactness theorem for propositional logic*:

Theorem 20 (Tautological reduction) $\mathbf{T} \models_p \boldsymbol{A}$ iff there is a list of formulas $\boldsymbol{As}$ such that $\boldsymbol{As} \subset \mathbf{T}$ and $Taut(\boldsymbol{As} \to \boldsymbol{A})$.

Proof: ($\leftarrow$): If $\boldsymbol{As} \subset \mathbf{T}$, $\mathit{Taut}(\boldsymbol{As} \to \boldsymbol{A})$, and if $\mathbf{S}$ is a propositional truth set containing $\mathbf{T}$ then $\boldsymbol{As} \subset \mathbf{S}$ and by the theorem 17: $(\boldsymbol{As} \to \boldsymbol{A}) \in \mathbf{S}$ so also $\boldsymbol{A} \in \mathbf{S}$ by repeated use of the property (5) of propositional truth sets.

($\rightarrow$): We assume that the formula $\boldsymbol{As} \to \boldsymbol{A}$ is not a tautology for any $\boldsymbol{As} \subset \mathbf{T}$ and show $\mathbf{T} \not\models_p \boldsymbol{A}$, i.e. that there is a propositional truth set $\mathbf{S}$ containing $\mathbf{T}$ such that $\boldsymbol{A} \notin \mathbf{S}$.

We first define a sequence $\mathbf{S}_i$ of sets of formulas such that $\mathbf{S}_0 = \mathbf{T}$ and

$$\mathbf{S}_{i+1} = \begin{cases} \mathbf{S}_i & \text{if } \neg \mathit{Form}(i) \text{ or} \\ & \mathit{Taut}((i, \boldsymbol{As}) \to \boldsymbol{A}) \\ & \text{for some } \boldsymbol{As} \subset \mathbf{S}_i \\ \mathbf{S}_i \cup \{i\} & \text{otherwise.} \end{cases}$$

Clearly, $\mathbf{T} \subset \mathbf{S}_i \subset \mathbf{S}_{i+1}$.

By induction on i we show that the formula $\boldsymbol{As} \to \boldsymbol{A}$ is not a tautology for any $\boldsymbol{As} \subset \mathbf{S}_i$. This holds for $\mathbf{S}_0$ from the assumption. For the inductive case $\mathbf{S}_{i+1}$ we consider two subcases. a) i is not a formula or for some $\boldsymbol{As} \subset \mathbf{S}_i$ the formula $(i, \boldsymbol{As}) \to \boldsymbol{A}$ is a tautology. Then the property holds from the inductive hypothesis because $\mathbf{S}_{i+1} = \mathbf{S}$. b) i is a formula and the formula $(i, \boldsymbol{As}) \to \boldsymbol{A}$ is not a tautology for any $\boldsymbol{As} \subset \mathbf{S}_i$. We have $\mathbf{S}_{i+1} = \mathbf{S}_i \cup \{i\}$ and consequently for any $\boldsymbol{Bs} \subset \mathbf{S}_{i+1}$ we have $\boldsymbol{BS} \subset (i, \boldsymbol{As})$ for some $\boldsymbol{As} \subset \mathbf{S}_i$. Hence $\boldsymbol{Bs} \to \boldsymbol{A}$ is not a tautology.

We next set $\mathbf{S} = \bigcup_i \mathbf{S}_i$ and immediately see both that $\mathbf{T} \subset \mathbf{S}$ and that the formula $\boldsymbol{As} \to \boldsymbol{A}$ is not a tautology for any $\boldsymbol{As} \subset \mathbf{S}$. Hence $\boldsymbol{A} \notin \mathbf{S}$ since $(\boldsymbol{A}, 0) \to \boldsymbol{A}$ is a tautology.

The proof will be finished when we show that $\mathbf{S}$ is a propositional truth set.

(1) holds because $(\boldsymbol{B}, \neg\boldsymbol{B}, 0) \to \boldsymbol{A}$ is a tautology and at most one of $\boldsymbol{B}$ and $\neg\boldsymbol{B}$ can be in $\mathbf{S}$. If neither of $\boldsymbol{B}$ and $\neg\boldsymbol{B}$ is in $\mathbf{S}$ then for some $\boldsymbol{As}_1 \subset \mathbf{S}_{\boldsymbol{B}} \subset \mathbf{S}$ and $\boldsymbol{As}_2 \subset \mathbf{S}_{\neg\boldsymbol{B}} \subset \mathbf{S}$ we have both $(\boldsymbol{B}, \boldsymbol{As}_1) \to \boldsymbol{A}$ and $(\neg\boldsymbol{B}, \boldsymbol{As}_2) \to \boldsymbol{A}$ tautologies. We get a contradiction as also $\boldsymbol{As}_1 \oplus \boldsymbol{As}_2 \to \boldsymbol{A}$ is then a tautology.

(2) holds because from the tautologies $(\neg\top, 0) \to \boldsymbol{A}$ and $(\bot, 0) \to \boldsymbol{A}$ we get $(\neg\top) \notin \mathbf{S}$ and $\bot \notin \mathbf{S}$ and then use (1).

(3) follows from (1) and from the tautologies $(\boldsymbol{B} \wedge \boldsymbol{C}, \neg\boldsymbol{B}, 0) \to \boldsymbol{A}$, $(\boldsymbol{B} \wedge \boldsymbol{C}, \neg\boldsymbol{C}, 0) \to \boldsymbol{A}$, and $(\neg(\boldsymbol{B} \wedge \boldsymbol{C}), \boldsymbol{B}, \boldsymbol{C}, 0) \to \boldsymbol{A}$.

(4) follows from (1) and from the tautologies $(\boldsymbol{B} \vee \boldsymbol{C}, \neg\boldsymbol{B}, \neg\boldsymbol{C}, 0) \to \boldsymbol{A}$, $(\neg(\boldsymbol{B} \vee \boldsymbol{C}), \boldsymbol{B}, 0) \to \boldsymbol{A}$, and $(\neg(\boldsymbol{B} \vee \boldsymbol{C}), \boldsymbol{C}, 0) \to \boldsymbol{A}$.

(5) follows from (1) and from the tautologies $(\boldsymbol{B} \to \boldsymbol{C}, \boldsymbol{B}, \neg\boldsymbol{C}, 0) \to \boldsymbol{A}$, $(\neg(\boldsymbol{B} \to \boldsymbol{C}), \neg\boldsymbol{B}, 0) \to \boldsymbol{A}$, and $(\neg(\boldsymbol{B} \to \boldsymbol{C}), \boldsymbol{C}, 0) \to \boldsymbol{A}$. ■

5.3 Quasi-tautological reduction

Identity truth set $\mathbf{S}$ is a propositional truth set such that such that

(6) $(\boldsymbol{a} = \boldsymbol{a}) \in \mathbf{S}$,

(7) if $(\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S}$ then $(\boldsymbol{b} = \boldsymbol{a}) \in \mathbf{S}$,

(8) if $(\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S}$ and $(\boldsymbol{b} = \boldsymbol{c}) \in \mathbf{S}$ then $(\boldsymbol{a} = \boldsymbol{c}) \in \mathbf{S}$,

(9) if $(\boldsymbol{as} = \boldsymbol{bs}) \subset \mathbf{S}$ then $(\boldsymbol{f}(\boldsymbol{as}) = \boldsymbol{f}(\boldsymbol{bs})) \in \mathbf{S}$,

(10) if $(\boldsymbol{as} = \boldsymbol{bs}) \subset \mathbf{S}$ and $\boldsymbol{p}(\boldsymbol{as}) \in \mathbf{S}$ then $\boldsymbol{p}(\boldsymbol{bs}) \in \mathbf{S}$.

Definition 6 (Quasi-tautological consequence) A formula $\boldsymbol{A}$ is a *quasi-tautological consequence* of a set of formulas $\mathbf{T}$ if for every identity truth set $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$.

The relation between identity and propositional truth sets is captured by the following straightforward lemma:

Lemma 21 $\mathbf{S}$ is an identity truth set iff $\mathbf{S}$ is a propositional truth set containing $\mathbf{Eq}$.

We now get a theorem corresponding to the definition 2:

Theorem 22 (Quasi-tautological reduction) $\mathbf{T} \models_i \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{Eq} \models_p \boldsymbol{A}$.

Proof: $\mathbf{T} \models_i \boldsymbol{A}$ iff for all identity truth sets $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$ iff by the lemma 21 for all propositional truth sets $\mathbf{S}$ containing $\mathbf{T} \cup \mathbf{Eq}$ also $\boldsymbol{A} \in \mathbf{S}$ iff $\mathbf{T}, \mathbf{Eq} \models_p \boldsymbol{A}$. ■

5.4 Henkin Reduction

Witnessing truth set $\mathbf{S}$ is an identity truth set such that

(11) if $(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$ then $\boldsymbol{A}[\mathit{Sc}(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}])] \in \mathbf{S}$,

(12) if $\boldsymbol{A}[\boldsymbol{a}] \in \mathbf{S}$ then $(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$,

(13) if $(\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$ then for all terms $\boldsymbol{a}$ we have $\boldsymbol{A}[\boldsymbol{a}] \in \mathbf{S}$,

(14) if $\boldsymbol{A}[Sc(\exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}])] \in \mathbf{S}$ then $(\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$.

Witnessing truth sets have a natural interpretation of quantifiers:

Lemma 23 For a witnessing truth set $\mathbf{S}$ we have

(15) $(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$ iff $\boldsymbol{A}[\boldsymbol{a}] \in \mathbf{S}$ for some term $\boldsymbol{a}$,

(16) $(\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$ iff $\boldsymbol{A}[\boldsymbol{a}] \in \mathbf{S}$ for all terms $\boldsymbol{a}$.

Proof: (15)$\rightarrow$ by (11), (15)$\leftarrow$ by (12), and (16)$\rightarrow$ by (13). (16)$\leftarrow$: if $\boldsymbol{A}[\boldsymbol{a}] \in \mathbf{S}$ for all terms $\boldsymbol{a}$ then also $\boldsymbol{A}[Sc(\exists \boldsymbol{x} \neg \boldsymbol{A}[\boldsymbol{x}])] \in \mathbf{S}$. Hence $(\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]) \in \mathbf{S}$ by (14). ■

The relation between witnessing and identity truth sets is captured by the following straightforward lemma:

Lemma 24 $\mathbf{S}$ is a witnessing truth set iff $\mathbf{S}$ is an identity truth set containing $\mathbf{Q}$.

The relations of Henkin and logical consequence will be defined with the help of structures in the standard way. A *structure* $\mathcal{M}$ is a quadruple

$$\mathcal{M} = \langle D, I_f, I_s, I_p \rangle \tag{5}$$

where D, called the *domain* of $\mathcal{M}$, is a non-empty set; I_f, called the *interpretation of function symbols*, is a mapping from $\mathbf{N}^2$ to functions over D such that for the function symbol $\boldsymbol{f} = n, m$ we have $I_f(n, m) : D^n \mapsto D$ $(D^0 \mapsto D = D)$; I_s, called the *interpretation of special constants*, is a mapping form $\mathbf{N}$ to D; and I_p, called the *interpretation of predicate symbols*, is a mapping from $\mathbf{N}^2$ to relations over D such that for the predicate symbol $\boldsymbol{p} = n, m$ we have $I_p(n, m) \subset D^n$ $(D^0 = \{\emptyset\})$.

A function $\alpha : \mathbf{N} \mapsto D$ is called an *assignment in* $\mathcal{M}$. The intention of assignments is to assign values from the domain D to the free variables of semi-terms and semi-formulas.

For the structure (5) we define a mapping I from semi-terms and assignments to D, in writing $\boldsymbol{a}^I[\alpha]$, giving *denotations* in D to semi-terms. The denotations are obtained by an inductive definition on semi-terms $\boldsymbol{a}$ as follows:

- if $\boldsymbol{a} = Bvar(n)$ then $\boldsymbol{a}^I[\alpha] = \alpha(n)$,
- if $\boldsymbol{a} = Sc(n)$ then $\boldsymbol{a}^I[\alpha] = I_s(n)$.
- if $\boldsymbol{a} = Applf(m, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_n, 0)$ then
$$\boldsymbol{a}^I[\alpha] = I_f(n, m)((\boldsymbol{a}_1)^I[\alpha], \cdots, (\boldsymbol{a}_n)^I[\alpha]).$$

The reader should note that if two assignments α_1 and α_2 agree on all variables occuring in the semi-term $\boldsymbol{a}$ then $\boldsymbol{a}^I[\alpha_1] = \boldsymbol{a}^I[\alpha_2]$. For a term $\boldsymbol{a}$ we have $\boldsymbol{a}^I[\alpha_1] = \boldsymbol{a}^I[\alpha_2]$ for any assignments α_1 and α_2.

We denote by $\alpha(n/d)$ an assignment α_1 which agrees with α everywhere except that $\alpha_1(n) = d$. For the structure (5) we define the relation *the semi-formula* $\boldsymbol{A}$ *is true in* $\mathcal{M}$ *for an assignment* α, in writing $\mathcal{M} \models \boldsymbol{A}[\alpha]$, by induction on semi-formulas $\boldsymbol{A}$ as follows:

- if $\boldsymbol{A} = Applp(m, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_n, 0)$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff $\langle (\boldsymbol{a}_1)^I[\alpha], \cdots, (\boldsymbol{a}_n)^I[\alpha] \rangle \in I_p(n, m)$ $(\langle\rangle = \emptyset)$,
- if $\boldsymbol{A} = Eq(\boldsymbol{a}, \boldsymbol{b})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff $\boldsymbol{a}^I[\alpha] = \boldsymbol{b}^I[\alpha]$,
- if $\boldsymbol{A} = \neg \boldsymbol{B}$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff not $\mathcal{M} \models \boldsymbol{B}[\alpha]$,
- if $\boldsymbol{A} = (\boldsymbol{B} \wedge \boldsymbol{C})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff $\mathcal{M} \models \boldsymbol{B}[\alpha]$ and $\mathcal{M} \models \boldsymbol{A}[\alpha]$,
- if $\boldsymbol{A} = (\boldsymbol{B} \vee \boldsymbol{C})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff $\mathcal{M} \models \boldsymbol{B}[\alpha]$ or $\mathcal{M} \models \boldsymbol{C}[\alpha]$,
- if $\boldsymbol{A} = (\boldsymbol{B} \rightarrow \boldsymbol{C})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff not $\mathcal{M} \models \boldsymbol{B}[\alpha]$ or $\mathcal{M} \models \boldsymbol{C}[\alpha]$,
- if $\boldsymbol{A} = \exists(n, \boldsymbol{B})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff there is $d \in D$ such that $\mathcal{M} \models \boldsymbol{B}[\alpha(n/d)]$,
- if $\boldsymbol{A} = \forall(n, \boldsymbol{B})$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$ iff for all $d \in D$: $\mathcal{M} \models \boldsymbol{B}[\alpha(n/d)]$,
- if $\boldsymbol{A} = \top$ then $\mathcal{M} \models \boldsymbol{A}[\alpha]$,
- if $\boldsymbol{A} = \perp$ then not $\mathcal{M} \models \boldsymbol{A}[\alpha]$.

The reader should note that if two assignments α_1 and α_2 agree on all free variables in the semi-formula $\boldsymbol{A}$ then $\mathcal{M} \models \boldsymbol{A}[\alpha_1]$ iff $\mathcal{M} \models \boldsymbol{A}[\alpha_2]$. For a formula $\boldsymbol{A}$ we have $\mathcal{M} \models \boldsymbol{A}[\alpha_1]$ iff $\mathcal{M} \models \boldsymbol{A}[\alpha_2]$ for any assignments α_1 and α_2. For this reason we abbreviate $\mathcal{M} \models \boldsymbol{A}[\alpha]$ to $\mathcal{M} \models \boldsymbol{A}$ (read: $\boldsymbol{A}$ is true in $\mathcal{M}$) for formulas $\boldsymbol{A}$.

Lemma 25 For all variables $\boldsymbol{x} = Bvar(n)$, semi-terms $\boldsymbol{b}[\boldsymbol{x}]$, semi-formulas $\boldsymbol{A}[\boldsymbol{x}]$, terms $\boldsymbol{a}$, and assignments α we have

$$\begin{array}{rcl} (\boldsymbol{b}[\boldsymbol{a}])^I[\alpha] & = & (\boldsymbol{b}[\boldsymbol{x}])^I[\alpha(n/\boldsymbol{a}^I[\alpha])] \\ \mathcal{M} \models (\boldsymbol{A}[\boldsymbol{a}])[\alpha] & \text{iff} & \mathcal{M} \models (\boldsymbol{A}[\boldsymbol{x}])[\alpha(n/\boldsymbol{a}^I[\alpha])]. \end{array}$$

Proof: By induction on semi-terms $\boldsymbol{b}$ and semi-formulas $\boldsymbol{A}$ respectively. ■

For a structure $\mathcal{M}$ the set $Th(\mathcal{M})$, called the *theory of* $\mathcal{M}$, is

$$Th(\mathcal{M}) = \{\boldsymbol{A} \in Form \mid \mathcal{M} \models \boldsymbol{A}\}.$$

A structure $\mathcal{M}$ is a *model* of a set of formulas $\mathbf{T}$ if $\mathbf{T} \subset Th(\mathcal{M})$. *Henkin structure* is a model of $\mathbf{H}$.

Definition 7 (Logical consequence) A formula $\boldsymbol{A}$ is a *logical consequence* of a set of formulas $\mathbf{T}$ if for every model $\mathcal{M}$ of $\mathbf{T}$ we have $\mathcal{M} \models \boldsymbol{A}$.

Definition 8 (Henkin consequence) A formula $\boldsymbol{A}$ is a *Henkin consequence* of a set of formulas $\mathbf{T}$ if for every Henkin structure $\mathcal{M}$ which is a model of $\mathbf{T}$ we have $\mathcal{M} \models \boldsymbol{A}$.

We clearly have

Lemma 26 $\mathbf{T} \models_h \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{H} \models \boldsymbol{A}$.

Lemma 27 For a Henkin structure $\mathcal{M}$ the set $Th(\mathcal{M})$ is a witnessing truth set.

Proof: The proof that $Th(\mathcal{M})$ is an identity truth set is straightforward. The properties (11), (13), and (14) are shown similarly as the property (12) where we assume $\boldsymbol{A}[\boldsymbol{a}] \in Th(\mathcal{M})$, i.e. $\mathcal{M} \models (\boldsymbol{A}[\boldsymbol{a}])[\alpha]$ for any α. Hence by the lemma 25 also $\mathcal{M} \models (\boldsymbol{A}[\boldsymbol{x}])[\alpha(n/\boldsymbol{a}^I[\alpha])]$, i.e. $\mathcal{M} \models \exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$. ■

Lemma 28 For every witnessing truth set $\mathbf{S}$ there is a Henkin structure with the domain a subset of terms such that $Th(\mathcal{M}) = \mathbf{S}$.

Proof: Since $(\boldsymbol{a} = \boldsymbol{a}) \in \mathbf{S}$ the set $\{\boldsymbol{b} \in Term \mid (\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S}\}$ is non-empty. Hence we can define a function $r : Term \mapsto Term$ by

$$r(\boldsymbol{a}) = min\{\boldsymbol{b} \in Term \mid (\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S}\}.$$

Note that $(\boldsymbol{a} = r(\boldsymbol{a})) \in \mathbf{S}$. Since $\mathbf{S}$ is an identity truth set the following can be easily seen true

$$\begin{array}{rl} (\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S} & \text{iff} \\ \{\boldsymbol{c} \in Term \mid (\boldsymbol{a} = \boldsymbol{c}) \in \mathbf{S}\} = & \\ \{\boldsymbol{c} \in Term \mid (\boldsymbol{b} = \boldsymbol{c}) \in \mathbf{S}\} & \text{iff} \\ r(\boldsymbol{a}) = r(\boldsymbol{b}). & \end{array}$$

We will now construct a structure $\mathcal{M}$ from (5). The domain D is defined as

$$D = \{r(\boldsymbol{a}) \mid \boldsymbol{a} \in Term\}.$$

We wish to define the interpretations I_f and I_s in such a way that $\boldsymbol{a}^I[\alpha] = r(\boldsymbol{a})$ for every term $\boldsymbol{a}$ and assignment α. We define $I_f(0, m) = r\ Applf(m, 0)$ and for $n > 0$

$$I_f(n, m)(r(\boldsymbol{a}_1), \cdots, r(\boldsymbol{a}_n)) = r\ Applf(m, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_n, 0).$$

That the definition is a legal one can be seen from the property (9) of identity truth sets. With the interpretation of special constants as

$$I_s(n) = r\ Sc(n)$$

we can now prove by induction on terms $\boldsymbol{a}$ that the desired property $\boldsymbol{a}^I[\alpha] = r(\boldsymbol{a})$ holds.

We next interprete the predicate symbols by setting

$$I_p(n, m) = \{\langle r(\boldsymbol{a}_1), \cdots, r(\boldsymbol{a}_n)\rangle \mid Applf(m, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_n, 0) \in \mathbf{S}\}.$$

The desired property $\mathcal{M} \models \boldsymbol{A}$ iff $\boldsymbol{A} \in \mathbf{S}$ follows by induction on formulas $\boldsymbol{A}$. If $\boldsymbol{A}$ is an identity formula $\boldsymbol{a} = \boldsymbol{b}$ we have $\mathcal{M} \models \boldsymbol{a} = \boldsymbol{b}$ iff $\boldsymbol{a}^I[\alpha] = \boldsymbol{b}^I[\alpha]$ for all α iff $r(\boldsymbol{a}) = r(\boldsymbol{b})$ iff $(\boldsymbol{a} = \boldsymbol{b}) \in \mathbf{S}$.

If $\boldsymbol{A}$ is $\boldsymbol{p}(\boldsymbol{as})$ we can easily see that if $\boldsymbol{p}(\boldsymbol{as}) \in \mathbf{S}$ then also $\mathcal{M} \models \boldsymbol{p}(\boldsymbol{as})$. Vice versa, if $\mathcal{M} \models \boldsymbol{p}(\boldsymbol{as})$ then for some $\boldsymbol{bs}$ such that $\boldsymbol{bs} = \boldsymbol{as}$ we have $\boldsymbol{p}(\boldsymbol{bs}) \in \mathbf{S}$ and by the property (10) of identity truth sets also $\boldsymbol{p}(\boldsymbol{as}) \in \mathbf{S}$.

The cases when $\boldsymbol{A}$ is a propositional formula are straightforward.

If $\boldsymbol{A}$ is $\forall \boldsymbol{x} \boldsymbol{B}[\boldsymbol{x}]$ with $\boldsymbol{x} = Bvar(n)$ then $\mathcal{M} \models \forall \boldsymbol{x} \boldsymbol{B}[\boldsymbol{x}]$ iff $\mathcal{M} \models (\forall \boldsymbol{x} \boldsymbol{B}[\boldsymbol{x}])[\alpha]$ for all α iff $\mathcal{M} \models (\boldsymbol{B}[\boldsymbol{x}])[\alpha(n/d)]$ for all $d \in D$ and α iff $\mathcal{M} \models (\boldsymbol{B}[\boldsymbol{x}])[\alpha(n/r(\boldsymbol{a}))]$ for all $\boldsymbol{a}$ and α iff $\mathcal{M} \models (\boldsymbol{B}[\boldsymbol{x}])[\alpha(n/\boldsymbol{a}^I[\alpha])]$ for all $\boldsymbol{a}$ and α iff by the lemma 25: $\mathcal{M} \models \boldsymbol{B}[\boldsymbol{a}]$ for all $\boldsymbol{a}$ iff by IH: $\boldsymbol{B}[\boldsymbol{a}] \in \mathbf{S}$ for all $\boldsymbol{a}$ iff by (16) of the lemma 23: $(\forall \boldsymbol{x} \boldsymbol{B}[\boldsymbol{x}]) \in \mathbf{S}$. The case when $\boldsymbol{A}$ is an existential quantifier formula is similar.

The structure $\mathcal{M}$ is a model of $\mathbf{H}$ as by the lemma 24: $\mathbf{H} \subset \mathbf{S} = Th(\mathcal{M})$. ■

Theorem 29 (Henkin consequence) A formula $\boldsymbol{A}$ is a Henkin consequence of a set of formulas $\mathbf{T}$ iff for all witnessing truth sets $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$.

Proof: ($\rightarrow$): Assume $\mathbf{T} \models_h \boldsymbol{A}$ and that $\mathbf{S}$ is a witnessing truth set containing $\mathbf{T}$. Construct by the lemma 28 a Henkin structure $\mathcal{M}$ such that $Th(\mathcal{M}) = \mathbf{S}$. Hence $\mathcal{M}$ is a model of $\mathbf{T}$ and so $\boldsymbol{A} \in Th(\mathcal{M}) = \mathbf{S}$.

($\leftarrow$): Assume that for all witnessing truth sets $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$ and that $\mathcal{M}$ is a Henkin structure which is a model of $\mathbf{T}$. By the lemma 27 the set $Th(\mathcal{M})$ is a witnessing truth set containing $\mathbf{T}$. Hence $\boldsymbol{A} \in Th(\mathcal{M})$, i.e. $\mathcal{M} \models \boldsymbol{A}$. ∎

We are now ready to prove a theorem corresponding to the definition 3:

Theorem 30 (Henkin reduction) $\mathbf{T} \models_h \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{Q} \models_i \boldsymbol{A}$.

Proof: $\mathbf{T} \models_h \boldsymbol{A}$ iff by the theorem 29 for all witnessing truth sets $\mathbf{S}$ containing $\mathbf{T}$ also $\boldsymbol{A} \in \mathbf{S}$ iff by the lemma 24 for all identity truth sets $\mathbf{S}$ containing $\mathbf{T} \cup \mathbf{Q}$ also $\boldsymbol{A} \in \mathbf{S}$ iff $\mathbf{T}, \mathbf{Q} \models_i \boldsymbol{A}$. ∎

5.5 Pure formula reduction

Let us denote by $Form_i$ the set of formulas $\boldsymbol{A}$ with the level of special constants i, i.e. such that $Scfl(\boldsymbol{A}) = i$. Hence the set of pure formulas is $Form_0$. Structures $\mathcal{M}$ and $\mathcal{M}_1$ are *elementarily equivalent on* $\mathbf{T}$ if $Th(\mathcal{M}) \cap \mathbf{T} = Th(\mathcal{M}_1) \cap \mathbf{T}$.

Lemma 31 (Witnessing expansion) To every structure $\mathcal{M}$ there is a Henkin structure $\mathcal{M}_1$ elementarily equivalent with $\mathcal{M}$ on pure formulas with.

Proof: Let us assume that $\mathcal{M}$ has the form (5). We define a sequence of interpretations of special constants I_i and a sequence of structures $\mathcal{N}_i$ such that

$$\mathcal{N}_i = \langle D, I_f, I_i, I_p \rangle.$$

We set $I_0 = I_s$ and

$$I_{i+1}(x) = \begin{cases} d & \text{if } R(i,x,d) \\ I_i(x) & \text{otherwise.} \end{cases}$$

$R(i,x,d)$ is an abbreviation for a set-theoretical formula saying that for some i, $\boldsymbol{x}$, $\boldsymbol{A}$, and n we have $x \in Form_i$, and $x = \exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]$, $\boldsymbol{x} = Bvar(n)$, and d is the least element of D in some fixed well-order such that $\mathcal{M}_i \models (\boldsymbol{A}[\boldsymbol{x}])[\alpha(n/d)]$. Here the assignment α is such that $\alpha(k) = 0$ for all k. Note that the well-order of D requires the axiom of choice.

By induction on i we can see that

(i) $\mathcal{N}_{i+1}$ and $\mathcal{N}_i$ are elementarily equivalent on $\bigcup_{j \le i} Form_j$,

(ii) $\mathcal{N}_{i+1} \models \exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \rightarrow \boldsymbol{A}[Sc(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}])]$ where $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \in Form_i$.

The condition (i) assures that once an interpretation of a formula $\boldsymbol{A} \in Form_i$ has been settled in the structure $\mathcal{N}_i$ it does not change in the structures $\mathcal{N}_{i+k}$. The condition (ii) says that the special constants for formulas from $Form_i$ obtain their desired denotations in the structure $\mathcal{N}_{i+1}$. By (i) the Henkin witnessing axioms for formulas $\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}] \in \bigcup_{j<i} Form_j$ are true in $\mathcal{N}_i$.

The structure $\mathcal{M}_1$ is defined as a 'union' of $\mathcal{N}_i$ by $\mathcal{M}_1 = \langle D, I_f, I, I_p \rangle$ where the interpretation of special constants I is defined as

$$I(x) = \begin{cases} I_{i+1}(x) & \text{if } x \in Form_i \text{ and } x \text{ is an} \\ & \text{existential q. formula} \\ I_0(x) & \text{otherwise.} \end{cases}$$

It is not too difficult to see that $\mathcal{M}_1$ is indeed a union as we have

$\mathcal{M}_1 \models \boldsymbol{A}$ iff $A \in Form_i$ and $\mathcal{N}_i \models \boldsymbol{A}$ for some i.

$\mathcal{M}_1$ is a Henkin structure by (ii). As $\mathcal{M} = \mathcal{M}_0$ and all structures $\mathcal{N}_i$ are elementarily equivalent on pure formulas thus also $\mathcal{M}_1$ and $\mathcal{M}$ are elementarily equivalent on pure formulas. ∎

We are now ready to prove a theorem corresponding to the definition 4:

Theorem 32 (Pure formula reduction) For a pure formula $\boldsymbol{A}$ and a set of pure formulas $\mathbf{T}$ we have $\mathbf{T} \models \boldsymbol{A}$ iff $\mathbf{T} \models_h \boldsymbol{A}$.

Proof: ($\rightarrow$): Obvious. ($\leftarrow$): Assume $\mathbf{T} \models_h \boldsymbol{A}$ and that $\mathcal{M}$ is a model of $\mathbf{T}$. We construct a Henkin structure $\mathcal{M}_1$ elementarily equivalent on pure formulas with $\mathcal{M}$ by the lemma 31. $\mathcal{M}_1$ is a model of $\mathbf{T}$ and consequently $\mathcal{M}_1 \models \boldsymbol{A}$. Thus also also $\mathcal{M} \models \boldsymbol{A}$. ∎

The following is a fundamental theorem of logic which states in different words the same thing as our reductions, i.e. that in studying the semantics of predicate calculus we do not have to go beyond sets of natural numbers:

Theorem 33 (Löwenheim-Skolem) To every structure there is a Henkin structure with the domain a subset of $\mathbf{N}$ such that both structures are elementarily equivalent on pure formulas.

Proof: Given a structure $\mathcal{M}$ we first construct by the lemma 31 a Henkin structure $\mathcal{M}_1$ elementarily equivalent with $\mathcal{M}$ on pure formulas. By the lemma 27 $Th(\mathcal{M}_1)$ is a witnessing truth set and so by the lemma 28 there is a Henkin structure $\mathcal{M}_2$ with the domain a subset of $Term \subset \mathbf{N}$ such that $Th(\mathcal{M}_2) = Th(\mathcal{M}_1)$. Hence $\mathcal{M}_2$ is the desired structure. ■

6 Predicate Calculus

In this chapter we are concerned with *syntactic*, as opposed to *semantic*, properties of predicate logic. 'Syntactic' means based on proofs, 'semantic' means based on models. Predicate calculus is the name for Predicate logic developed by syntactic means.

6.1 Provability

The concept of proof is used in mathematics to turn the notion of logical consequence which is in general non-decidable into the decidable property of recognizing that a proof p proves a formula $\boldsymbol{A}$ from axioms $\mathbf{T}$.

Let us assume that a given set of pure formulas $\mathbf{T}$ is a decidable set, i.e. that there is a recursive predicate Ax such that $Ax(x) \leftrightarrow x \in \mathbf{T}$. The complexity of the predicate Ax is usually not to big so we can safely assume that it will be a $\mathcal{LESP}$ predicate. We can find out whether a given formula $\boldsymbol{A}$ is an *axiom*, i.e. whether $\boldsymbol{A} \in \mathbf{T}$, by computing $Ax(\boldsymbol{A})$.

For a decidable predicate $\mathbf{T}$, the predicate $Valid_T(\boldsymbol{A}) \leftrightarrow \mathbf{T} \models \boldsymbol{A}$ *and* $\boldsymbol{A}$ *is pure* is a recursively enumerable predicate. This is because from the reduction theorems of the previous chapter we know that the formula $\boldsymbol{A}$ logically follows from $\mathbf{T}$ iff $\mathbf{T} \models_h \boldsymbol{A}$ iff there is a list of formulas $\boldsymbol{As} \subset \mathbf{T} \cup \mathbf{Eq} \cup \mathbf{Q}$ such that $\boldsymbol{As} \to \boldsymbol{A}$ is a tautology. The list $\boldsymbol{As}$, if it exists, is called an *associate* of $\boldsymbol{A}$. As the sets $\mathbf{Eq}$ and $\mathbf{Q}$ are decidable by p.r. $\mathcal{LESP}$ predicates we can see that the predicate $Assoc_T(x, y)$ holding if

> 'x is a list of formulas such that $x \subset \mathbf{T} \cup \mathbf{Eq} \cup \mathbf{Q}$ and $x \to y$ is tautology'

is a recursive $\mathcal{LESP}$ predicate (primitive recursive in $\mathbf{T}$). We have

$$Valid_T(y) \leftrightarrow \exists x\, Assoc_T(x, y).$$

An associate $\boldsymbol{As}$ of $\boldsymbol{A}$ can be viewed as a proof of $\boldsymbol{A}$.

Associates are a basis for the notion of provability which can be called *Hilbert systems*. Given a set of formulas $\mathbf{T}$ (not necessarily pure) as *axioms* we say that the formula $\boldsymbol{A}$ is a *theorem* of the Hilbert system $\mathbf{T}$ if there is a list $\boldsymbol{As}$ (which is a proof) with the last formula $\boldsymbol{A}$ such for that every formula $\boldsymbol{B} \in \boldsymbol{As}$ one of the following holds:

(i) $\boldsymbol{B}$ is a *non-logical axiom* where $\boldsymbol{B} \in \mathbf{T}$,

(ii) $\boldsymbol{B}$ is a *logical axiom* where $\boldsymbol{B}$ is either a tautology or $\boldsymbol{B} \in \mathbf{Eq} \cup \mathbf{Q}$,

(iii) $\boldsymbol{B}$ is the *conclusion* of rule *Modus ponens* where there is a formula $\boldsymbol{C}$ which together with a formula $\boldsymbol{C} \to \boldsymbol{B}$ precede $\boldsymbol{B}$ in the list $\boldsymbol{As}$. The formulas $\boldsymbol{C}$ and $\boldsymbol{C} \to \boldsymbol{B}$ are the *premises* of modus ponens.

It should be clear that the predicate $Proof_T(\boldsymbol{As}, \boldsymbol{A})$ holding iff $\boldsymbol{As}$ is a proof of $\boldsymbol{A}$ is primitive recursive in $\mathbf{T}$.

Theorem 34 (Soundness and completeness of Hilbert systems) A formula $\boldsymbol{A}$ is a theorem of $\mathbf{T}$ iff $\mathbf{T} \models_h \boldsymbol{A}$.

Proof: $(\to)$: By induction on the length of a proof we show that every theorem has an associate. The empty list 0 is an associate of a tautology. As $\boldsymbol{A} \to \boldsymbol{A}$ is a tautology, the list $\boldsymbol{A}, 0$ is an associate of $\boldsymbol{A} \in \mathbf{T} \cup \mathbf{Eq} \cup \mathbf{Q}$. If by IH the lists $\boldsymbol{As}_1$ and $\boldsymbol{As}_2$ are associates of $\boldsymbol{A}$ and $\boldsymbol{A} \to \boldsymbol{B}$ respectively, then $\boldsymbol{As}_1 \oplus \boldsymbol{As}_2$ is an associate of $\boldsymbol{B}$. By the semantic reduction we have $\mathbf{T} \models_h \boldsymbol{A}$. This proves that Hilbert systems are *sound* in the sense that theorems od $\mathbf{T}$ are Henkin consequences of non-logical axioms $\mathbf{T}$.

$(\leftarrow)$: If $\mathbf{T} \models_h \boldsymbol{A}$ then $\boldsymbol{A}$ has an associate $\boldsymbol{As}$. The formula $\boldsymbol{As} \to \boldsymbol{A}$, being a tautology, is a theorem and we get $\boldsymbol{A}$ from it by a series of applications of modus ponens. This proves the *completeness* of Hilbert systems, i.e. that every Henkin consequence of $\mathbf{T}$ is a theorem of $\mathbf{T}$. ∎

Our definition of provability in Hilbert systems coincides with the notion of Henkin consequence. By the Pure formula reduction Henkin consequence coincides with logical consequence on pure formulas. Thus we get:

Corollary 35 If $\mathbf{T}$ and $\boldsymbol{A}$ are pure then $\boldsymbol{A}$ is a theorem of a Hilbert system $\mathbf{T}$ iff $\mathbf{T} \models \boldsymbol{A}$.

The reader familiar with the usual development of formal systems based on modus ponens will note that by formalizing the concept of provability as Henkin consequence we have significantly simplified the axiomatization. We do not need the usual rules of inference of *universal and existential introduction* where, for instance, one proves the formula $\boldsymbol{A} \rightarrow \forall \boldsymbol{x}\, \boldsymbol{B}$ from the premise $\boldsymbol{A} \rightarrow \boldsymbol{B}$ provided that the conclusion satisfies the *eigen-variable condition* that the variable $\boldsymbol{x}$ does not freely occur in it. This simplification explains why we have introduced the concept of Henkin consequence.

6.2 Tableaux for predicate logic

It is possible to develop the finitary logic within Hilbert systems. Indeed, the text of Shoenfield [13] does it. We have decided against the Hilbert-style provability for two reasons.

The first reason is that while modus ponens corresponds to the usual practice of proving theorems in mathematics it makes the proofs of theorems in the form of implications quite difficult. In classroom situations tableau-based systems are preferable to Hilbert-style systems as students seem to find tableau proofs easier.

The second reason is that the analysis of proofs with modus ponens is quite difficult as a proof proceeds by a series of *detours*, where in the process of proving $\boldsymbol{B}$ we first prove $\boldsymbol{A}$ and $\boldsymbol{A} \rightarrow \boldsymbol{B}$. The discarded auxiliary theorem $\boldsymbol{A}$ does not show in the final theorem. Detours make it hard to *characterize* the theorems proved by a formal system because the theorems depend on all discarded formulas. The difficulty of analysis of proofs in Hilbert systems makes it difficult to write programs operating on the proofs.

We now extend the propositional tableaux introduced in the chapter 4 to predicate logic. The extended tableaux, called simply *tableaux*, are designed to cover the notion of Henkin consequence.

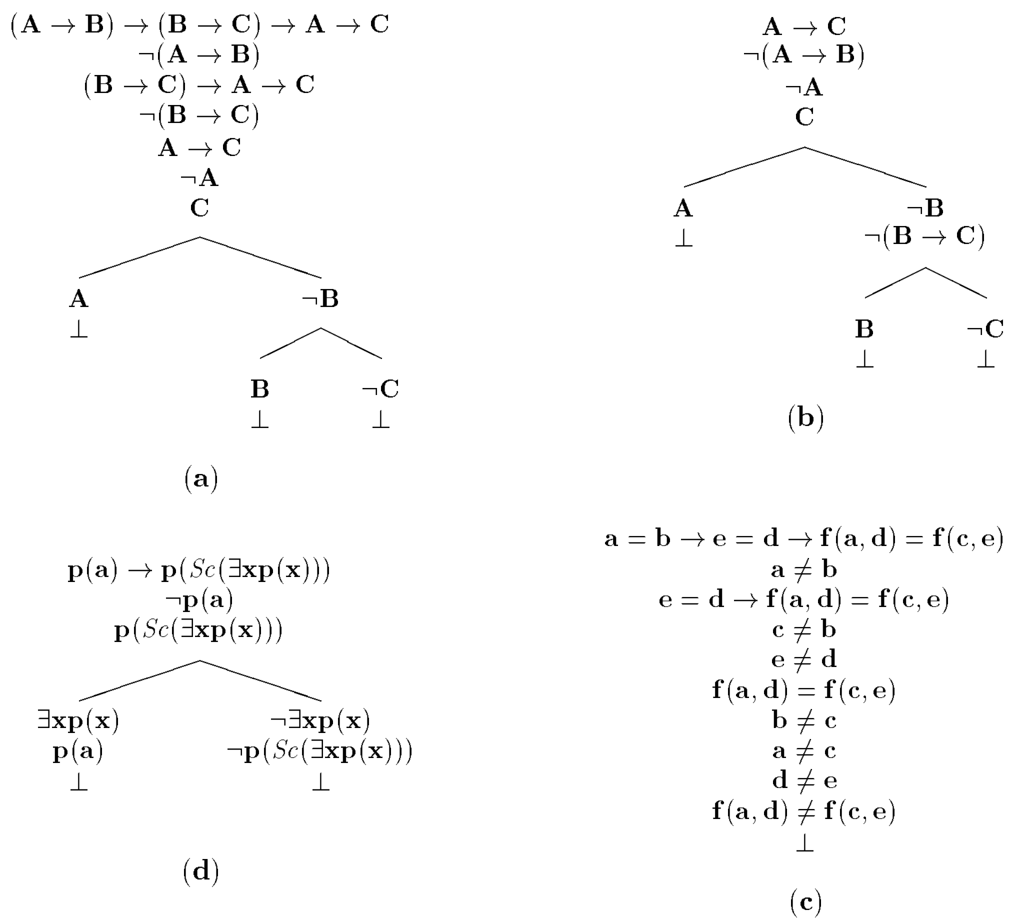


Figure 1: Examples of tableaux

Definition 9 (Tableaux) For a list of formulas $\boldsymbol{As}$ and a set of formulas $\mathbf{T}$ we say that the formula $\boldsymbol{X}$ *is a tableau for* $\boldsymbol{As}$ *in* $\mathbf{T}$ if $\boldsymbol{X}$ is formed like a propositional tableau but the tableau expansion rules are, in addition to the propositional rules, also the *identity*, *quantifier*, *cut*, and *axiom* rules.

The *identity rules* correspond to the identity axioms **Eq**:

$$\frac{}{\boldsymbol{a} \neq \boldsymbol{a}} \quad \frac{\boldsymbol{a} \neq \boldsymbol{b}}{\boldsymbol{b} \neq \boldsymbol{a}} \quad \frac{\boldsymbol{a} \neq \boldsymbol{b} \quad \boldsymbol{b} \neq \boldsymbol{c}}{\boldsymbol{a} \neq \boldsymbol{c}}$$

$$\frac{\boldsymbol{as} \neq \boldsymbol{bs}}{\boldsymbol{f}(\boldsymbol{as}) \neq \boldsymbol{f}(\boldsymbol{bs})} \quad \frac{\boldsymbol{as} \neq \boldsymbol{bs} \quad \neg \boldsymbol{p}(\boldsymbol{as})}{\neg \boldsymbol{p}(\boldsymbol{bs})}.$$

The quantifier rules correspond to the quantifier axioms **Q**:

$$\frac{\neg \exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]}{\neg \boldsymbol{A}[Sc(\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}])]} \quad \frac{\forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]}{\boldsymbol{A}[Sc(\exists x \neg \boldsymbol{A}[\boldsymbol{x}])]}$$

$$\frac{\exists \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]}{\boldsymbol{A}[\boldsymbol{a}]} \quad \frac{\neg \forall \boldsymbol{x} \boldsymbol{A}[\boldsymbol{x}]}{\neg \boldsymbol{A}[\boldsymbol{a}]}.$$

The *cut rules* are

$$\frac{}{\boldsymbol{A} \quad \neg \boldsymbol{A}}$$

and the *axiom rules* are for every $\boldsymbol{A} \in \mathbf{T}$:

$$\frac{}{\neg \boldsymbol{A}}.$$

A tableau $\boldsymbol{X}$ for $\boldsymbol{As}$ in $\mathbf{T}$ is *closed*, in writing $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{As}$, if each of its branches is closed.

Example 3 All four tableaux in the figure 1 are closed. They are discussed in more detail in the example 4 given below.

We note that among the identity expansion rules reflexive rules are without a premise, transitive rules have two premises, and function and predicate substitution rules have $n \geq 0$ premisses depending on the arity of the function and predicate symbols involved.

Cut and axiom rules have no premises while symmetry and quantifier rules are with a single premise. With the exception of cut rules which are binary, all new rules are unary.

Every unary rule $\frac{\boldsymbol{As}}{\boldsymbol{A}}$ has the property

$$\mathbf{T} \models_h \boldsymbol{A} \rightarrow \boldsymbol{As} \vee \bot.$$

Every binary rule $\frac{\boldsymbol{As}}{\boldsymbol{A}\ \boldsymbol{B}}$ has the property

$$\mathbf{T} \models_h \boldsymbol{A} \wedge \boldsymbol{B} \rightarrow \boldsymbol{As} \vee \bot.$$

As a consequence if $\boldsymbol{X}$ is a tableau for $\boldsymbol{As}$ in $\mathbf{T}$ then

$$\mathbf{T} \models_h \boldsymbol{As} \vee \bot \text{ iff } \mathbf{T} \models_h \boldsymbol{As} \vee \boldsymbol{X},$$

i.e. tableaux preserve the Henkin consequence. If $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{As}$ then $\boldsymbol{As} \vee \boldsymbol{X}$ is a tautology and we get $\mathbf{T} \models_h \boldsymbol{As} \vee \bot$. This suggests that the tableaux are sound. This fact will be proved in detail in the theorem 37.

We take over the abbreviations introduced for propositional tableaux. We write $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{A}$ when $\boldsymbol{As} = \boldsymbol{A}, 0$. We say that $\boldsymbol{As}$ is provable in $\mathbf{T}$, in writing $\mathbf{T} \vdash^{+} \boldsymbol{As}$ if $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{As}$ for some $\boldsymbol{X}$.

We also introduce some new abbreviations. We will abbreviate $\boldsymbol{X} : \emptyset \vdash^{+} \boldsymbol{As}$ to $\boldsymbol{X} : \ \vdash^{+} \boldsymbol{As}$. This and the above abbreviations are naturally used in a combination with the following abbreviations.

If $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{As}$ and the tableau $\boldsymbol{X}$ is not expanded by any quantifier rules we will write $\boldsymbol{X} : \mathbf{T} \vdash^{+}_{i} \boldsymbol{As}$ and say that $\boldsymbol{X}$ is an *identity* tableau. We say that the tableau $\boldsymbol{X}$ is a *propositional* tableau, in writing $\boldsymbol{X} : \mathbf{T} \vdash^{+}_{p} \boldsymbol{As}$, if it is an identity tableau without using any identity expansion rules. In all forms we will omit the qualifier $+$ if $\boldsymbol{X}$ is a *cut-free* tableau, i.e. a tableau obtained without using any cut rules.

The reader will note that the predicate $\boldsymbol{X} : \ \vdash_p \boldsymbol{As}$ of $\boldsymbol{X}$ *being a closed propositional tableau for* $\boldsymbol{As}$ as introduced in the chapter 4 Tautologies coincides with the eponymous predicate of $\boldsymbol{X}$ *being a closed propositional and cut-free tableau for* $\emptyset$ as introduced above.

It should be clear that the predicate $p(x, y)$ holding iff $x : \mathbf{T} \vdash^{q_1}_{q_2} y$ in all qualified forms is a primitive recursive in $\mathbf{T}$ predicate.

Example 4 For the tableau (a) of the figure 1 we have

$$\boldsymbol{X}_1 : \ \vdash_p (\boldsymbol{A} \rightarrow \boldsymbol{B}) \rightarrow (\boldsymbol{B} \rightarrow \boldsymbol{C}) \rightarrow \boldsymbol{A} \rightarrow \boldsymbol{C}$$

where $\boldsymbol{X}_1$ starts with second formula. For the tableau (b) we have

$$\boldsymbol{X}_2 : \neg(\boldsymbol{B} \rightarrow \boldsymbol{C}) \vdash_p (\boldsymbol{A} \rightarrow \boldsymbol{C}), \neg(\boldsymbol{A} \rightarrow \boldsymbol{B}), 0$$

where $\boldsymbol{X}_2$ starts with the third formula. For the tableau (c) we have

$$\begin{aligned}\boldsymbol{X}_3 : \boldsymbol{c} = \boldsymbol{b} \vdash_i \boldsymbol{a} = \boldsymbol{b} \rightarrow \\ \boldsymbol{e} = \boldsymbol{d} \rightarrow \boldsymbol{f}(\boldsymbol{a}, \boldsymbol{d}) = \boldsymbol{f}(\boldsymbol{c}, \boldsymbol{e})\end{aligned}$$

where $\boldsymbol{X}_3$ starts with the second formula. For the tableau (d) we have

$$\boldsymbol{X}_4 : \vdash^{+} \boldsymbol{p}(\boldsymbol{a}) \rightarrow \boldsymbol{p}(Sc(\exists \boldsymbol{x}\, \boldsymbol{p}(\boldsymbol{x})))$$

where $\boldsymbol{X}_4$ starts with the second formula.

6.3 Completeness of tableaux

We now state the *syntactic* counterparts of the semantic reductions. Syntactic means proved by finitary means as opposed to semantic proofs which are in general non-constructive. The syntactic reductions (i), (ii), and (iii) in the following theorem correspond in that order to the tautological, quasi-tautological, and Henkin reductions.

Theorem 36 (Syntactic Reductions)

(i) $\mathbf{T} \vdash_p^+ \boldsymbol{A}$ iff $\vdash_p^+ \boldsymbol{As} \to \boldsymbol{A}$ for some $\boldsymbol{As} \subset \mathbf{T}$.

(ii) $\mathbf{T} \vdash_i^+ \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{Eq} \vdash_p^+ \boldsymbol{A}$.

(iii) $\mathbf{T} \vdash^+ \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{Q} \vdash_i^+ \boldsymbol{A}$.

Sketch of the proof: (i): This is the so-called *Deduction theorem for propositional logic.* It is proved with the help of two auxiliary p.r. *LESP* functions such that

$$\frac{\boldsymbol{X} : \boldsymbol{As} \vdash_p^+ \boldsymbol{A}}{\mathit{Ded1}(\boldsymbol{As}, \boldsymbol{A}, \boldsymbol{X}) : \vdash_p^+ \boldsymbol{As} \to \boldsymbol{A}}$$

$$\frac{\boldsymbol{X} : \vdash_p^+ \boldsymbol{As} \to \boldsymbol{A}}{\mathit{Ded2}(\boldsymbol{As}, \boldsymbol{A}, \boldsymbol{X}) : \boldsymbol{As} \vdash_p^+ \boldsymbol{A}}.$$

Thus $\mathbf{T} \vdash_p^+ \boldsymbol{A}$ iff $\boldsymbol{As} \vdash_p^+ \boldsymbol{A}$ for some $\boldsymbol{As} \subset \mathbf{T}$ iff $\vdash_p^+ \boldsymbol{As} \to \boldsymbol{A}$ for some $\boldsymbol{As} \subset \mathbf{T}$.

(ii): We use the correspondence between the axioms and rules of identity. We have a p.r. *LESP* function *Irlelim* which if given an identity tableau replaces the identity rules by the corresponding identity axioms and yields a propositional proof. This is shown in the figure **??** (a) for a symmetric rule. Vice versa, a p.r. *LESP* function *Iaxelim* takes a propositional tableau and yields an identity tableau by eliminating the axioms of identity. This is done by introducing the corresponding identity rule and one or more cuts. The figure **??** (b) shows this for a symmetric axiom. The two functions work as follows:

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash_i^+ \boldsymbol{A}}{\mathit{Irlelim}(\boldsymbol{A}, \boldsymbol{X}) : \mathbf{T}, \mathbf{Eq} \vdash_p^+ \boldsymbol{A}}$$

$$\frac{\boldsymbol{X} : \mathbf{T}, \mathbf{Eq} \vdash_p^+ \boldsymbol{A}}{\mathit{Iaxelim}(\boldsymbol{X}) : \mathbf{T} \vdash_i^+ \boldsymbol{A}}.$$

Consequently, $\mathbf{T} \vdash_i^+ \boldsymbol{A}$ iff $\mathbf{T}, \mathbf{Eq} \vdash_p^+ \boldsymbol{A}$.

(iii): This is similar to (ii) and uses the correspondence of the quantifier axioms and rules. We have p.r. *LESP* functions *Qrelim* and *Qaxelim* such that

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash^+ \boldsymbol{A}}{\mathit{Qrelim}(\boldsymbol{A}, \boldsymbol{X}) : \mathbf{T}, \mathbf{Q} \vdash_i^+ \boldsymbol{A}}$$

$$\frac{\boldsymbol{X} : \mathbf{T}, \mathbf{Q} \vdash_i^+ \boldsymbol{A}}{\mathit{Qaxelim}(\boldsymbol{X}) : \mathbf{T} \vdash^+ \boldsymbol{A}}.$$

The function *Qrelim* replaces quantifier rules by the corresponding quantifier axioms. The function *Qaxelim* replaces quantifier axioms by the corresponding quantifier rules and a cut. The figure **??** (c) and (d) shows this for an existential substitution rule and axiom, respectively. ∎

Theorem 37 (Soundness and Completeness of Tableaux)

(i) $\mathit{Taut}(\boldsymbol{A})$ iff $\vdash_p^+ \boldsymbol{A}$

(ii) $\mathbf{T} \models_p \boldsymbol{A}$ iff $\mathbf{T} \vdash_p^+ \boldsymbol{A}$

(iii) $\mathbf{T} \models_i \boldsymbol{A}$ iff $\mathbf{T} \vdash_i^+ \boldsymbol{A}$

(iv) $\mathbf{T}, \mathbf{H} \models \boldsymbol{A}$ iff $\mathbf{T} \vdash^+ \boldsymbol{A}$

(v) $\mathbf{T} \models \boldsymbol{A}$ iff $\mathbf{T} \vdash^+ \boldsymbol{A}$ for pure $\mathbf{T}$ and $\boldsymbol{A}$.

Proof: (i): This is a fairly standard finitary proof and can be done by adapting the methods used, say, in [13].

(ii): $\mathbf{T} \models_p \boldsymbol{A}$ iff by tautological reduction $\mathit{Taut}(\boldsymbol{As} \to \boldsymbol{A})$ for some $\boldsymbol{As} \subset \mathbf{T}$ iff by (i) $\vdash_p^+ \boldsymbol{As} \to \boldsymbol{A}$ for some $\boldsymbol{As} \subset \mathbf{T}$ iff by (i) from the theorem 36 $\mathbf{T} \vdash_p^+ \boldsymbol{A}$.

(iii): $\mathbf{T} \models_i \boldsymbol{A}$ iff by quasi-tautological reduction $\mathbf{T}, \mathbf{Eq} \models_p \boldsymbol{A}$ iff by (ii) $\mathbf{T}, \mathbf{Eq} \vdash_p^+ \boldsymbol{A}$ iff by (ii) from the theorem 36 $\mathbf{T} \vdash_i^+ \boldsymbol{A}$.

(iv): $\mathbf{T}, \mathbf{H} \models \boldsymbol{A}$ iff by Henkin reduction $\mathbf{T}, \mathbf{Q} \models_i \boldsymbol{A}$ iff by (iii) $\mathbf{T}, \mathbf{Q} \vdash_i^+ \boldsymbol{A}$ iff by (iii) from the theorem 36 $\mathbf{T} \vdash^+ \boldsymbol{A}$.

(v): This is straightforward. ∎

As a complete example of a function operating on proofs we give the following finitary theorem.

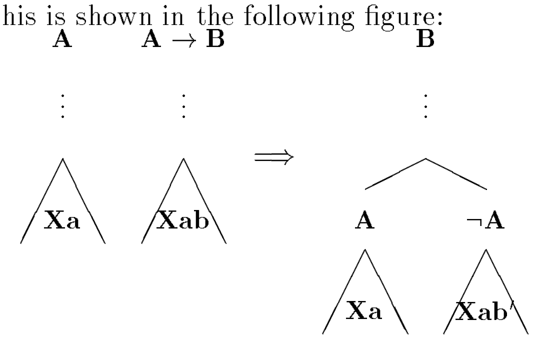
Theorem 38 Modus ponens is a derived rule of inference. Specifically, there is a p.r. function *Mp* such that

$$\frac{\boldsymbol{Xa} : \mathbf{T} \vdash^+ \boldsymbol{A} \quad \boldsymbol{Xab} : \mathbf{T} \vdash^+ \boldsymbol{A} \to \boldsymbol{B}}{\mathit{Mp}(\boldsymbol{Xa}, \boldsymbol{A}, \boldsymbol{Xab}, \boldsymbol{B}) : \mathbf{T} \vdash^+ \boldsymbol{B}}.$$

Proof: We first remove from the tableau $\boldsymbol{Xab}$ the conclusions $\neg\boldsymbol{A}$ and $\boldsymbol{B}$ of disjunctive rules with $\boldsymbol{A} \to \boldsymbol{B}$ as the premise. We get a tableau $\boldsymbol{Xab'}$ for $\neg\boldsymbol{A}, \boldsymbol{B}, 0$ in $\mathbf{T}$. By a cut rule we get the desired tableau

$$(\boldsymbol{A} \vee \boldsymbol{Xa}) \wedge (\neg\boldsymbol{A} \vee \boldsymbol{Xab'}) : \mathbf{T} \vdash^{+} \boldsymbol{B}.$$

This is shown in the following figure:



Thus we can define the function *Mp* by an explicit $\mathcal{LESP}$ definition

$Mp(\boldsymbol{Xa}, \boldsymbol{A}, \boldsymbol{Xab}, \boldsymbol{B}) =$
$\quad(\boldsymbol{A} \vee \boldsymbol{Xa}) \wedge (\neg\boldsymbol{A} \vee \boldsymbol{Xab'}) \leftarrow$
$\quad Dinvx(\neg\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{Xab}) = \boldsymbol{Xab'}$

where the auxiliary function *Dinvx* which does the removal has the property

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{B} \vee \boldsymbol{C}, \boldsymbol{As}}{Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{X}) : \mathbf{T} \vdash^{+} \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{As}}$$

and a p.r. $\mathcal{LESP}$ definition

$Dinvx(\boldsymbol{B}, \boldsymbol{C}, \bot) = \bot$
$Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{A} \vee \boldsymbol{Xt}) = \boldsymbol{Xt'} \leftarrow$
$\quad(\boldsymbol{A} = \boldsymbol{B} \vee \boldsymbol{A} = \boldsymbol{C}) \wedge$
$\quad Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Xt}) = \boldsymbol{Xt'}$
$Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{A} \vee \boldsymbol{Xt}) = \boldsymbol{A} \vee \boldsymbol{Xt'} \leftarrow$
$\quad\neg(\boldsymbol{A} = \boldsymbol{B} \vee \boldsymbol{A} = \boldsymbol{C}) \wedge$
$\quad Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Xt}) = \boldsymbol{Xt'}$
$Dinvx(\boldsymbol{B}, \boldsymbol{C}, (\boldsymbol{D} \vee \boldsymbol{Xd}) \wedge (\boldsymbol{E} \vee \boldsymbol{Xe})) =$
$\quad(\boldsymbol{D} \vee \boldsymbol{Xd'}) \wedge (\boldsymbol{E} \vee \boldsymbol{Xe'}) \leftarrow$
$\quad Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Xd}) = \boldsymbol{Xd'} \wedge$
$\quad Dinvx(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Xe}) = \boldsymbol{Xe'}$. ■

6.4 Cut elimination

Careful reader has noticed that because of cut rules we have detours in our tableaux. He can see from the proof of the theorem 38 that modus ponens is transformed into a cut. We need cut rules as they are instrumental in the proof of the theorem 37 where the syntactic reductions eliminate axioms by introducing cuts. However, one of the reasons why we have introduced tableaux was to eliminate the detours. This cannot be achieved in general as the figure 1 (d) suggests. The cut rule cannot be eliminated since the tableau requires application of a quantifier rule corresponding to a Henkin witnessing axiom. The quantified formula must be introduced into the tableau by a detour via a cut.

The situation is better with pure formulas. The following theorem was proved originally by Gentzen [4] for his sequent calculus. We have adapted the proof for our tableaux where it is somewhat simpler than in the sequent calculus.

Theorem 39 (Cut elimination) If $\mathbf{T}$ and $\boldsymbol{A}$ are pure then there is a p.r. $\mathcal{LESP}$ function *Cutelim* such that

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{A}}{Cutelim(\boldsymbol{X}, \boldsymbol{A}) : \mathbf{T} \vdash \boldsymbol{A}}.$$

Proof: We need some auxiliary lemmas which we do not prove here. This proof resembles in its structure the sequent calculus proof given in the chapter on Cut elimination by H. Schwichtenberg in [1]. If the reader wants to know more about some of the lemmas he is referred to the article.

The application $Cutelim(\boldsymbol{X}, \boldsymbol{A})$ proceeds in three stages. The tableau $\boldsymbol{X}$ is first converted into a *regular* tableau $\boldsymbol{X}_1$. Then $\boldsymbol{X}_1$ is transformed into an *atomically closed* tableau $\boldsymbol{X}_2$. The stage three is a recursive process which repeatedly decreases the *cut rank* of the tableau $\boldsymbol{X}_2$ until a tableau $\boldsymbol{X}_3$ of the cut rank 0 - i.e. without cuts - is obtained. This tableau is yielded as the result of the call to *Cutelim*.

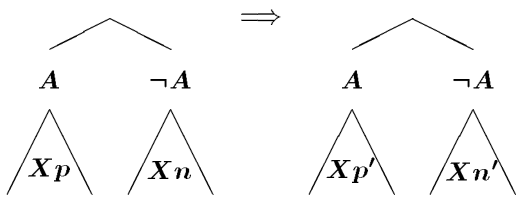
A tableau is *regular* if every special constant $\boldsymbol{r}$ which appears in it appears in, and below, the conclusion of the Henkin witnessing or counterexample rule corresponding to $\boldsymbol{r}$. Every tableau for a pure formula $\boldsymbol{A}$ in a pure theory $\mathbf{T}$ can be brought into a regular form by a rather complex p.r. function (we need this as a lemma). Incidentally, the tableau (d) in the figure 1 cannot be regularized because the special constant which occurs in it occurs already in the formula to be proved. It is impossible to 'hide' it into a conclusion of a Henkin rule.

Atomically closed tableau is such that if any of its branches closes because of a pair of formulas $\boldsymbol{B}$ and $\neg\boldsymbol{B}$ then the formula $\boldsymbol{B}$ is atomic ($\boldsymbol{p}(\boldsymbol{as})$ or $\boldsymbol{a} = \boldsymbol{b}$). Every tableau can be brought into an atomically closed form by a fairly straightforward p.r. function which inserts new inferences into every branch closed by a pair of formulas $\boldsymbol{B}$ and $\neg\boldsymbol{B}$ where $\boldsymbol{B}$ is not atomic (we need this as a lemma).

The *rank* $\rho(\mathbf{A})$ of a formula $\boldsymbol{A}$ is the number of logical operators occuring in it. The *rank* of a tableau $\boldsymbol{X}$, denoted by $\rho(\mathbf{X})$, is the maximal rank of left cut formulas (i.e. the first conclusions of cut rules) in $\boldsymbol{X}$ increased by one. Thus a cut-free tableau has a rank 0. The stage three of the cut elimination process repeatedly calls on a p.r. *LESP* function *Redcut* which reduces the rank of a tableau by one. The function *Redcut* has the following property:

> If a regular atomically closed tableau $\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{As}$ is such that $\rho(\mathbf{X}) \leq n+1$ then the tableau $\mathit{Redcut}(n+1, \boldsymbol{X}, \boldsymbol{As})$ is a regular and atomically closed tableau for $\boldsymbol{As}$ in $\mathbf{T}$ of a rank at most n.

The function *Redcut* recurs on (the size of) the tableau $\boldsymbol{X}$. It can be extracted from the following proof done by induction on $\boldsymbol{X}$. We therefore assume the antecedent of the above property. We may also assume that the tableau $\boldsymbol{X}$ is of the form $(\boldsymbol{A} \vee \boldsymbol{Xp}) \wedge (\neg\boldsymbol{A} \vee \boldsymbol{Xn})$ with $\rho(\mathbf{A}) = n$ (i.e. $\rho(\mathbf{X}) = n+1$) because otherwise the conclusion follows by a straightforward application of the inductive hypothesis. Hence $\boldsymbol{Xp} : \mathbf{T} \vdash^{+} \boldsymbol{A}, \boldsymbol{As}$ and $\boldsymbol{Xn} : \mathbf{T} \vdash^{+} \neg\boldsymbol{A}, \boldsymbol{As}$ with both tableaux regular, atomically closed, and of cut ranks at most $n+1$. This is shown on the left of the following figure:



By the induction hypothesis we obtain the tableaux on the right $\boldsymbol{Xp'} : \mathbf{T} \vdash^{+} \boldsymbol{A}, \boldsymbol{As}$ and $\boldsymbol{Xn'} : \mathbf{T} \vdash^{+} \neg\boldsymbol{A}, \boldsymbol{As}$ which are both regular, atomically closed, and of cut ranks at most n. We now proceed by the form of the cut formula $\boldsymbol{A}$.

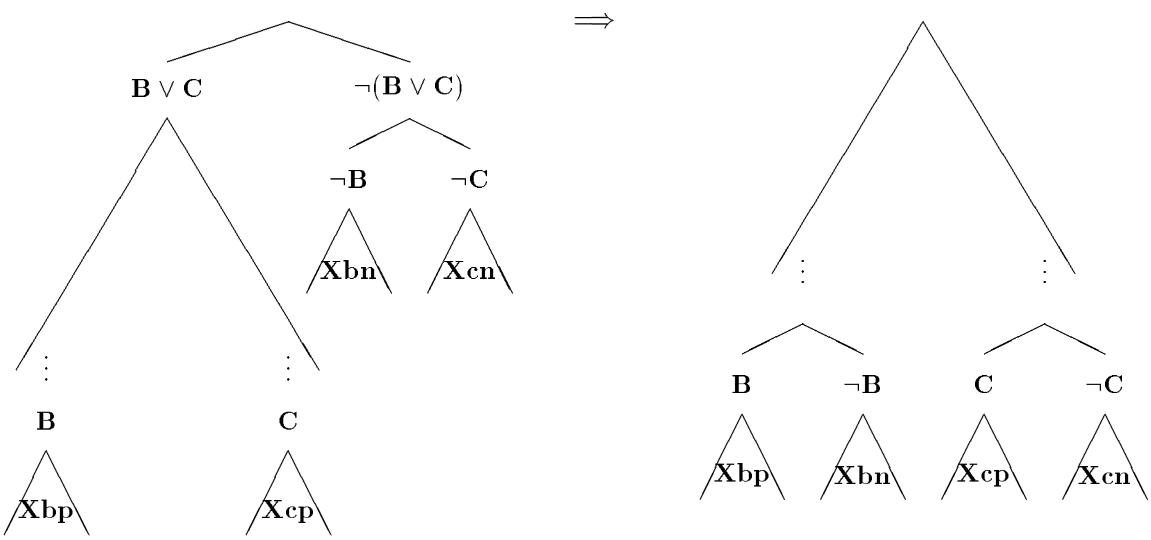


Figure 2: Reduction of a disjunction in the cut formula

Suppose that $\boldsymbol{A} = \boldsymbol{B} \vee \boldsymbol{C}$. We may assume that

$$\boldsymbol{Xn'} = (\neg\boldsymbol{B} \vee \boldsymbol{Xbn}) \wedge (\neg\boldsymbol{C} \vee \boldsymbol{Xcn})$$

where

$$\boldsymbol{Xbn} : \mathbf{T} \vdash^{+} \neg\boldsymbol{B}, \boldsymbol{As}$$

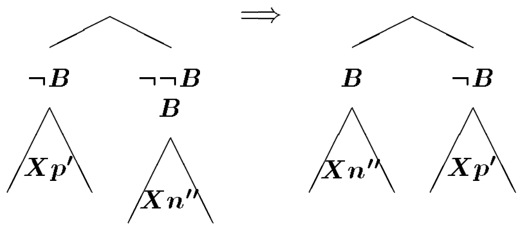
and

$$\boldsymbol{Xcn} : \mathbf{T} \vdash^{+} \neg\boldsymbol{C}, \boldsymbol{As}.$$

This is shown in the tree on the left of the figure 2. The assumed situation can be brought about by an *inversion* lemma which postulates a p.r. function modifying the tableau $\boldsymbol{Xn'}$ in such a way that all conclusions of conjunctive rules applied to the premise $\neg(\boldsymbol{B} \vee \boldsymbol{C})$ are removed from $\boldsymbol{Xn'}$ and placed as a single conclusion right at the begin of the tableau. We have indicated in the tree $\boldsymbol{Xp'}$ two of possibly many (or none) conclusions of disjunction rules applied to the formula $\boldsymbol{B} \vee \boldsymbol{C}$.

We construct a new tableau from $\boldsymbol{Xp'}$ as shown on the right of the figure by removing the topmost cut and inserting into $\boldsymbol{Xp'}$ new cuts of rank n instead of **all** conclusions of disjunction rules for the premise $\boldsymbol{B} \vee \boldsymbol{C}$. The reader will note that the formula $\boldsymbol{B} \vee \boldsymbol{C}$ can be removed from the resulting tableau as no branch in the tableau $\boldsymbol{Xp'}$ for $\boldsymbol{A}, \boldsymbol{As}$ can close on the pair $\boldsymbol{A}$ and $\neg\boldsymbol{A}$ because the tableau is atomically closed.

Suppose that $\boldsymbol{A} = \neg\boldsymbol{B}$. We may assume that $\boldsymbol{Xn'} = \boldsymbol{B} \vee \boldsymbol{Xn''}$ where $\boldsymbol{Xn''} : \mathbf{T} \vdash^{+} \boldsymbol{B}, \boldsymbol{As}$. This can be done by an auxiliary p. r. inversion function. The situation is shown in the tree on the left of the following figure:



By performing the indicated transformation we get a tableau on the right which is for $\boldsymbol{As}$ in $\mathbf{T}$. The rank of the cut formula has decreased from $n+1$ to n so the tableau has a cut rank at most n.

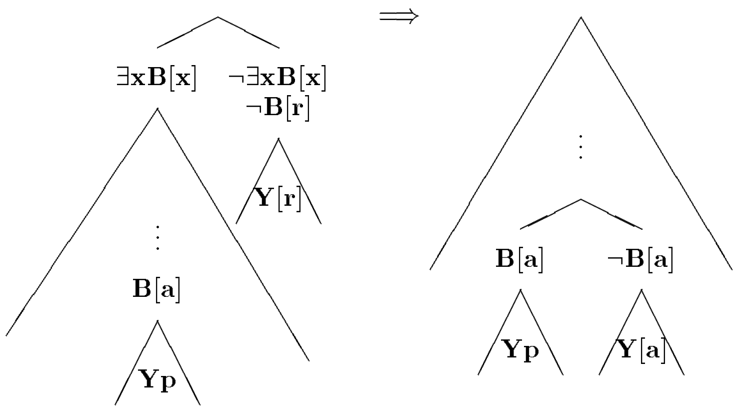


Figure 3: Reduction of an existential quantifier cut formula

Suppose that $\boldsymbol{A} = \exists \boldsymbol{x}\boldsymbol{B}[\boldsymbol{x}]$. We may assume that

$$\boldsymbol{Xn'} = \neg\boldsymbol{B}[\boldsymbol{r}] \vee Y[\boldsymbol{r}]$$

where $\boldsymbol{r} = Sc(\exists \boldsymbol{x}\boldsymbol{B}[\boldsymbol{x}])$ and

$$\boldsymbol{Y}[\boldsymbol{r}] : \mathbf{T} \vdash^{+} \neg\boldsymbol{B}[\boldsymbol{r}], \boldsymbol{As}.$$

The assumption means that the tableau is already 'inverted'. The situation is shown on the left of the figure 3. We have indicated the possible presence of the special constant $\boldsymbol{r}$ in the tableau $\boldsymbol{Y}$. We will denote by $\boldsymbol{Y}[\boldsymbol{a}]$ the tableau obtained from $\boldsymbol{Y}$ by the replacement throughout the tableau of the special constant $\boldsymbol{r}$ by the term $\boldsymbol{a}$. We need a *substitution* lemma which asserts that such a replacement does not change the character of the tableau, i.e. that we have

$$\boldsymbol{Y}[\boldsymbol{a}] : \mathbf{T} \vdash^{+} \neg\boldsymbol{B}[\boldsymbol{a}], \boldsymbol{As}.$$

The reader will note that this is where the assumption of regularity is critical. The special constant $\boldsymbol{r}$ may not occur in the list $\boldsymbol{As}$.

By performing the indicated transformation to the tableau $\boldsymbol{Xp'}$ we get a tree on the right of the figure 3 which represents a tableau for $\boldsymbol{As}$ in $\mathbf{T}$ with a cut rank at most n. The transformation consists of locating **all** conclusions of the form $\boldsymbol{B}[\boldsymbol{a}]$ of the quantifier substitution rule with the premise $\exists \boldsymbol{x}\boldsymbol{B}[\boldsymbol{x}]$ in $\boldsymbol{Xp'}$ (the figure shows just one conclusion) and replacing it by a cut with the rank n as indicated. The character of the tableau does not change as the tableau is atomically closed and so we can safely remove the formula $\exists \boldsymbol{x} \boldsymbol{B}[\boldsymbol{x}]$ from the tableau.

Suppose that $\boldsymbol{A}$ is an atomic formula (we must have $n = 0$). The situation is shown by the tree on the left in the figure 4 which indicates two out of possibly many branches (there may be even none) which close on the pair of formulas $\boldsymbol{A}$ and $\neg \boldsymbol{A}$. The transformation applied to the tableau $\boldsymbol{X}\boldsymbol{p}'$ shown on the right of the figure removes the cut and closes each of the branches with the tableau $\boldsymbol{X}\boldsymbol{n}$ instead of the formula $\boldsymbol{A}$ which can be now removed as it cannot be used as a premise of any rule. The resulting tableau is cut-free.

Transformations for the remaining forms of the cut formula $\boldsymbol{A}$ are variations of the discussed cases and are left to the reader. ■

Propositional and identity tableaux can be brought into a cut-free form even for non-pure formulas because the assumption of regularity is needed only for Henkin rules applied to cut formulas.

6.5 Theories

A *theory* is given by a $\mathcal{LESP}$ decidable set of pure axioms $\mathbf{T}$. A *language* of a theory $\mathbf{T}$ consists of all function and predicate symbols occurring in the axioms $\mathbf{T}$ together with the special constants for formulas constructed from these symbols. A formula $\boldsymbol{A}$ *is in the language of* $\mathbf{T}$ if all its function and predicate symbols are from the language of $\mathbf{T}$. A formula $\boldsymbol{A}$ is a *theorem* of the theory $\mathbf{T}$ if it is in the language of $\mathbf{T}$ and $\mathbf{T} \vdash \boldsymbol{A}$.

Theorem 40 (Analytic tableaux) There is a primitive recursive $\mathcal{LESP}$ function *Analytic* such that for pure $\mathbf{T}$ and $\boldsymbol{A}$ we have

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash^{+} \boldsymbol{A}}{\mathit{Analytic}(\boldsymbol{X}, \boldsymbol{A}) : \mathbf{T} \vdash \boldsymbol{A}}$$

where the tableau $\mathit{Analytic}(\boldsymbol{X}, \boldsymbol{A})$ is in the language of $\mathbf{T}$ and $\boldsymbol{A}$.

Sketch of the proof: The application $\mathit{Analytic}(\boldsymbol{X}, \boldsymbol{A})$ eliminates first the cuts in $\boldsymbol{X}$ and obtains a tableau $\boldsymbol{X}_1$ for $\boldsymbol{A}$ in $\mathbf{T}$. Terms not in the language of $\mathbf{T}$ and $\boldsymbol{A}$ are introduced into $\boldsymbol{X}_1$ by reflexive and by function and quantifier substitution rules. As such terms do not occur in $\boldsymbol{A}$ or $\boldsymbol{T}$ we can replace them under certain conditions by terms in the language of $\mathbf{T}$ and $\boldsymbol{A}$ without destroying the character of the tableau $\boldsymbol{X}_1$. Function substitution rules may cause problems. We have to be careful if a 'bad' term, say $\boldsymbol{f}(\boldsymbol{a})$, is in the conclusion $\boldsymbol{f}(\boldsymbol{a}) \neq \boldsymbol{f}(\boldsymbol{b})$ of a function substitution rule. In such a case we must replace $\boldsymbol{f}(\boldsymbol{a})$ by $\boldsymbol{f}(\boldsymbol{b})$. This will cause problems if we have in the same branch the conclusion $\boldsymbol{f}(\boldsymbol{a}) \neq \boldsymbol{f}(\boldsymbol{c})$ of another function substitution rule because the replacement will invalidate the rule. See the situation on the left-hand-side of the figure 5 (a). If we make the formula $\boldsymbol{f}(\boldsymbol{a}) \neq \boldsymbol{f}(\boldsymbol{c})$ the conclusion of a transitive rule then the replacement will leave the rule intact. Necessary modifications to the tableau consisting of new applications of identity rules are shown in the right-hand-side of the figure 5 (a). ■.

Tableaux obtained by the theorem 40 are called *analytic* as they proceed by analyzing the formula to be proved into its subformulas without introducing any new formulas. Analytic tableaux can be strenthened to posses a *subterm* property where no new terms are introduced. As a consequence the predicate $\vdash_i \boldsymbol{A}$ of being a quasi-tautology is primitive recursive. This is because there is only a finite number of identity axioms $\mathbf{Eq}$ whose terms occur in $\boldsymbol{A}$. $\boldsymbol{A}$ is a quasi-tautology iff $\boldsymbol{Es} \to \boldsymbol{A}$ is a tautology where the list $\boldsymbol{Es}$ consists of such axioms.

A consequence of cut-free tableaux is the lemma 41 which eliminates quantifier substitution rules from certain tableaux. We need some definitions first. We call a formula an *existential* (*universal*) formula if it contains at most existential (universal) quantifiers after it has been brought into a prenex normal from. For instance, the formula $\forall \boldsymbol{x} \boldsymbol{A} \to \exists \boldsymbol{y} \boldsymbol{B}$ with $\boldsymbol{A}$ and $\boldsymbol{B}$ quantifier-free is an existential formula as the universal quantifier occurs in a negative context.

A *quantifier instance* of a formula $\boldsymbol{A}$ is a

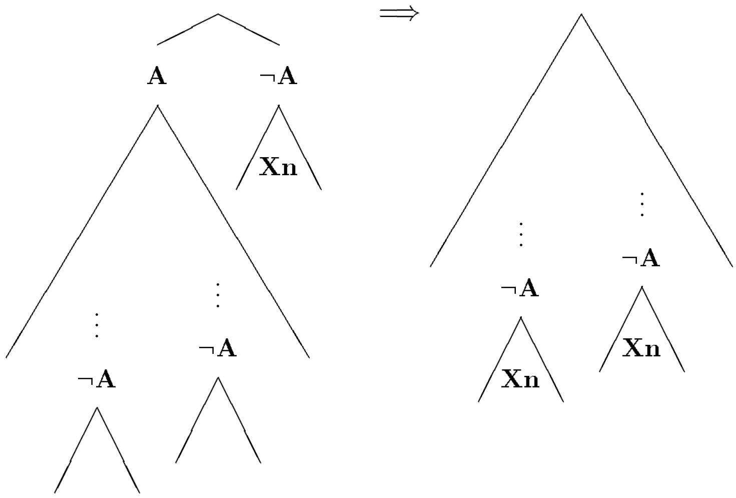
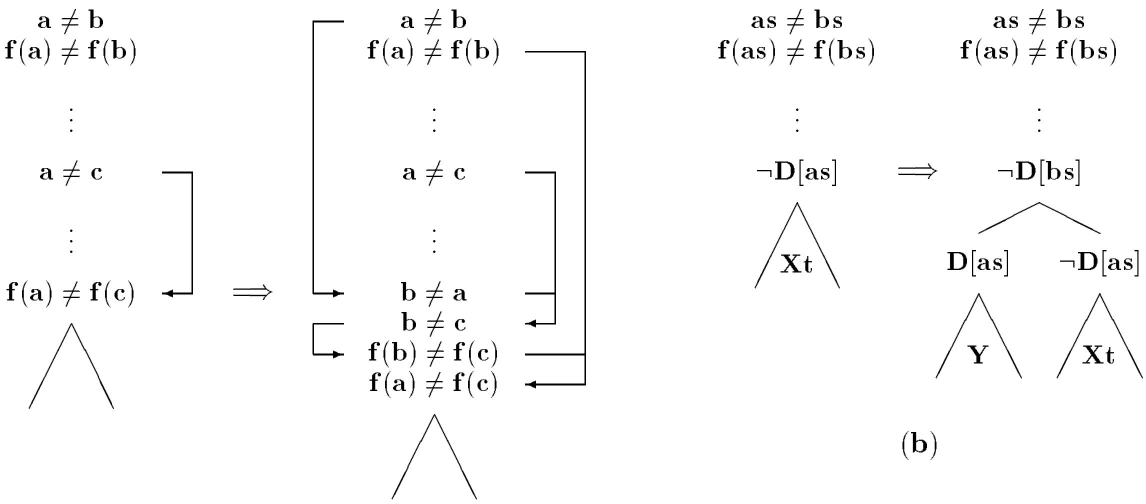


Figure 4: Reduction of an atomic cut formula



(a)

Figure 5: Elimination of collisions with function substitution rules

quantifier-free formula such that every existential quantifier $\exists x B[x]$ occurring in the formula is replaced by a disjunction $Bs \lor \bot$ where the list Bs consists of formulas which are quantifier instances of $B[b]$ for some term b. Every universal quantifier is replaced by a conjunction $Bs \land \top$ where the list Bs is similar. A quantifier instance of the above example formula has the form $As \land \top \to Bs \lor \bot$. Note that we have $Bs \lor \bot \to \exists x B[x]$ in the first case and $\forall x B[x] \to Bs \land \top$ in the second case. As a consequence we have $\vdash A_1 \to A$ for an existential formula A and $\vdash A \to A_1$ for an universal formula A where in both cases A_1 is a quantifier instance of A.

Lemma 41 (Herbrand) For an existential formula A we have $\vdash A$ iff $\vdash_i A_1$ for some quantifier instance A_1 of A.

This is finitary as we have p.r. *$\mathcal{LESP}$* functions $Qinstance_1$ and $Qinstance_2$ such that **(a):**

$$\frac{X : \vdash A}{X_1 : \vdash_i A_1}$$

where $Qinstance_1(X, A) = X_1, A_1$ and **(b):**

$$\frac{X : \vdash_i A_1}{Qinstance_2(X, A_1, A) : \vdash A}.$$

Sketch of the proof: (a): Since the formula A is existential the only quantifier rules which can be used in X are the quantifier substitution rules. The function $Qinstance_1$ is a recursive one which does the following transformation. Suppose that at a particular stage it is working on the tableau $\exists x B[x] \lor X_1$ for As where all quantifier rules have been already removed from X_1. All conclusions $B[a]$ of quantifier substitution rules with the premise $\exists x B[x]$ in X_1 are collected into the list Bs and they are removed from it. The resulting tableau X_2 is an identity tableu for $Bs \oplus As$. The tableau X_2 is modified to be for $(Bs \lor \bot), As$ by possibly inserting some applications of disjunctive rules whereby we get X_3. We note that the formula $Bs \lor \bot$ is a quantifier instance of the formula $\exists x B[x]$ so the tableau $(Bs \lor \bot) \lor X_3$ is for As_1 where the last list is obtained from As by essentially replacing all occurrences of $\exists x B[x]$ by $Bs \lor \bot$. The transformation for the universal substitution rule is similar.

(b): The function $Qinstance_2$ is constructed to make use of the property $\vdash A_1 \to A$ of existential formulas. ■

We now turn to the elimination of certain non-logical axioms from tableaux. A theory $\mathbf{T}$ is an *extension* of the theory $\mathbf{S}$ if every theorem of $\mathbf{S}$ is a theorem of $\mathbf{T}$. The theory $\mathbf{T}$ is a *conservative extension* of $\mathbf{S}$ if $\mathbf{T}$ is an extension of $\mathbf{S}$ such that every theorem of $\mathbf{T}$ in the language of $\mathbf{S}$ is also a theorem of $\mathbf{S}$. The concept of conservative extension captures the idea that while an extension of a theory may introduce new theorems in the extended language it does not add any new theorems in the language of $\mathbf{S}$. As a consequence if the original theory $\mathbf{S}$ is *consistent*, i.e. $\bot$ is not a theorem of $\mathbf{S}$, every conservative extension $\mathbf{T}$ must remain consistent. Conservative extensions we are interested in are *finitary* where we have a *$\mathcal{LESP}$* function f such that from $X : \mathbf{T} \vdash A$ where A is in the language of $\mathbf{S}$ we get $f(X, A) : \mathbf{S} \vdash A$. If $\mathbf{T}$ was obtained from $\mathbf{S}$ by the addition of new non-logical axioms, i.e. if $\mathbf{T} = \mathbf{S} \cup \mathbf{S}_1$ then $f(X, A)$ eliminates from X the use of all non-logical axioms $\mathbf{S}_1$.

The central theorem on finitary conservative extensions is the theorem 42 about extensions by *Skolem axioms* which are the formulas $\forall xs D[xs]$ where

$$D[xs] = (\exists y\, A[xs, y] \to A[xs, f(xs)]).$$

The reader will note that Skolem axioms are a generalization of Henkin witnessing axioms although historically Skolem functions were used before witnesses. Indeed, the formula $\forall xs D[xs]$ is true in Henkin structures where $\forall xs\, (f(xs) = Sc(\exists y A[xs, y]))$ is true.

Theorem 42 (Addition of Skolem Axioms) If the semi-formula $A[xs, y]$ is in the language of $\mathbf{T}$ and the language does not contain the function symbol f then the theory $\mathbf{T} \cup \{\forall xs D[xs]\}$ is a finitary conservative extension of $\mathbf{T}$, i.e. there is a p.r. *$\mathcal{LESP}$* function *Skelim* such that

$$\frac{X : \mathbf{T}, \forall xs D[xs] \vdash B}{Skelim(X, \forall xs D[xs]) : \mathbf{T} \vdash B}$$

for all formulas B in the language of $\mathbf{T}$.

Sketch of the proof: The application $\mathit{Skelim}(\boldsymbol{X}, \forall \boldsymbol{xs} \boldsymbol{D}[\boldsymbol{xs}])$ first modifies the tableau $\boldsymbol{X}$ to a tableau $\boldsymbol{X}_1 : \mathbf{T}, \mathbf{S} \vdash \boldsymbol{B}$ where the set $\mathbf{S}$ consists of instances of Skolem axioms of the form $\boldsymbol{D}[\boldsymbol{as}]$, i.e.

$$\exists \boldsymbol{y}\, \boldsymbol{A}[\boldsymbol{as}, \boldsymbol{y}] \to \boldsymbol{A}[\boldsymbol{as}, \boldsymbol{f}(\boldsymbol{as})].$$

This is done by simply omitting from $\boldsymbol{X}$ the Skolem axioms $\forall \boldsymbol{xs} \boldsymbol{D}[\boldsymbol{xs}]$ and all conclusions of substitution axioms applied to it as long as they contain universal quantifiers binding some of the variables from $\boldsymbol{xs}$. We then proceed to eliminate the applications of axiom rules for $\mathbf{S}$ from $\boldsymbol{X}_1$. For instance, the formula $\neg \boldsymbol{D}[\boldsymbol{as}]$ is replaced by the conclusion of the axiom rule for a Henkin witnessing axiom

$$\neg(\exists \boldsymbol{y}\, \boldsymbol{A}[\boldsymbol{as}, \boldsymbol{y}] \to \boldsymbol{A}[\boldsymbol{as}, \mathit{Sc}(\exists \boldsymbol{y} \boldsymbol{A}[\boldsymbol{as}, \boldsymbol{y}])]).$$

The replacement is done by replacing the term $\boldsymbol{f}(\boldsymbol{as})$ by the special constant $\mathit{Sc}(\exists \boldsymbol{y} \boldsymbol{A}[\boldsymbol{as}, \boldsymbol{y}])$ throughout the tableau $\boldsymbol{X}_1$. However, the process is quite subtle as such a replacement can destroy conclusions of function substitution rules of the form $\boldsymbol{f}(\boldsymbol{as}) \neq \boldsymbol{f}(\boldsymbol{bs})$ occurring in the tableau $\boldsymbol{X}_1$. In this case we employ the technique from the proof of the theorem 40 and replace $\boldsymbol{f}(\boldsymbol{as})$ by $\boldsymbol{f}(\boldsymbol{bs})$. As this replacement will destroy the above conclusion $\neg \boldsymbol{D}[\boldsymbol{as}]$ we have to eliminate it by other means. The situation is shown on the left-hand-side of the figure 5 (b). For the elimination of $\neg \boldsymbol{D}[\boldsymbol{as}]$ we use the identity theorem $\vdash \boldsymbol{as} = \boldsymbol{bs} \to \boldsymbol{D}[\boldsymbol{bs}] \to \boldsymbol{D}[\boldsymbol{as}]$, more specifically the tableau

$$\boldsymbol{Y} : \vdash \neg \boldsymbol{D}[\boldsymbol{bs}], \boldsymbol{D}[\boldsymbol{as}], (\boldsymbol{as} \neq \boldsymbol{bs})$$

and introduce a cut in the right-hand-side of the figure 5 (b). Now the formula $\neg \boldsymbol{D}[\boldsymbol{as}]$ is not introduced into the tableau by an axiom rule but by a cut rule and the replacement of $\boldsymbol{f}(\boldsymbol{as})$ by $\boldsymbol{f}(\boldsymbol{bs})$ will not destroy the character of the tableau. We continue in this style and eliminate all instances of Skolem axioms which cannot be turned into Henkin axioms. Then we take care of function substitution rules in the style of the theorem 40. At the end we obtain a tableau $\boldsymbol{X}_2 : \mathbf{T}, \mathbf{H} \vdash^+ \boldsymbol{B}$. It remains to eliminate the Henkin witnessing axioms and cuts by yielding the tableau

$$\mathit{Cutelim}(\mathit{Qaxelim}(\boldsymbol{X}_2), \boldsymbol{B}) : \mathbf{T} \vdash \boldsymbol{B}$$

by applying the theorems 36 and 39. ■

We note that in a theory where $\forall \boldsymbol{xs} \boldsymbol{D}[\boldsymbol{xs}]$ is a theorem we have

$$\forall \boldsymbol{xs}(\exists \boldsymbol{y}\, \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{y}] \leftrightarrow \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{xs})].$$

In a theory where the Skolem axiom

$$\forall \boldsymbol{xs}(\exists \boldsymbol{y}\, \neg \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{y}] \to \neg \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{xs})])$$

is a theorem we have

$$\forall \boldsymbol{xs}(\forall \boldsymbol{y}\, \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{y}] \leftrightarrow \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{xs})]).$$

This suggests that at a cost of introducing new function symbols we can eliminate both existential and universal quantifiers in theories with the appropriate Skolem axioms as theorems. This idea is used in three fundamental theorems about finitary conservative extensions known as the *Addition of functions symbols* (the theorem 43), *Skolem*, and *Herbrand* theorems (the theorems 44 and 45).

Theorem 43 (Addition of function symbols) If for a theory $\mathbf{T}$ and a formula $\boldsymbol{A}$ in the language of $\mathbf{T}$ which does not contain $\boldsymbol{f}$ we have $\mathbf{T} \vdash \forall \boldsymbol{xs} \exists \boldsymbol{y}\, \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{y}]$ then there is a p.r. $\mathcal{LESP}$ function *Funelim* such that

$$\frac{\boldsymbol{X} : \mathbf{T}, \forall \boldsymbol{xs} \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{xs})] \vdash \boldsymbol{B}}{\mathit{Funelim}(\boldsymbol{X}, \boldsymbol{B}) : \mathbf{T} \vdash \boldsymbol{B}}$$

for every formula $\boldsymbol{B}$ in the language of $\mathbf{T}$.

Proof: Let us denote the extended theory by $\mathbf{T}_1$ and the theory obtained from $\mathbf{T}$ by the addition of the Skolem axiom $\forall \boldsymbol{xs} \boldsymbol{D}[\boldsymbol{xs}]$ by $\mathbf{T}_2$. The application $\mathit{Funelim}(\boldsymbol{X}, \boldsymbol{B})$ first forms a tableau $\boldsymbol{X}_1$ for $\boldsymbol{B}$ in $\mathbf{T}_2$. This is possible as $\forall \boldsymbol{xs} \boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{x})]$ is a theorem of $\mathbf{T}_2$. The function *Funelim* then yields the tableau

$$\mathit{Skelim}(\boldsymbol{X}_1, \boldsymbol{B}, \forall \boldsymbol{xs} \boldsymbol{D}[\boldsymbol{xs}]) : \mathbf{T} \vdash \boldsymbol{B}. \ ■$$

By simultaneous recursion we can define p.r. $\mathcal{LESP}$ functions $\boldsymbol{B}^h$ and $\boldsymbol{B}^s$ which take a formula $\boldsymbol{B}$ and yield formulas called respectively the *Herbrand* and *Skolem normal form of* $\boldsymbol{B}$. The formula $\boldsymbol{B}^h$ is an existential formula and the formula $\boldsymbol{B}^s$ is a universal formula. The normal forms are formed by locating in $\boldsymbol{B}$ certain quantified semi-formulas of the form $Qy\boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{y}]$ and replacing them by semi-formulas $\boldsymbol{A}[\boldsymbol{xs}, \boldsymbol{f}(\boldsymbol{xs})]$ for a new function symbol $\boldsymbol{f}$. The quantifier semi-formulas whose quantifiers are eliminated by this process are those sub-semi-formulas of $\boldsymbol{B}$ whose quantifiers show in a prenex form of $\boldsymbol{B}$ as universal quantifiers in the case of $\boldsymbol{B}^h$ and as existential quantifiers in the case of $\boldsymbol{B}^s$. This is the reason why the resultant normal forms are existential and universal formulas respectively. The variables in $\boldsymbol{xs}$ are all the variables bound by existential quantifiers in the case $\boldsymbol{B}^h$ and universal quantifiers in the case $\boldsymbol{B}^s$ in the quantified semi-formulas surrounding the semi-formula $\boldsymbol{A}$ in $\boldsymbol{B}$.

If, for example,

$$\boldsymbol{C} = \forall \boldsymbol{x} \exists \boldsymbol{y} \forall \boldsymbol{z} \exists \boldsymbol{w} \boldsymbol{E}[\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w}]$$

then

$$\begin{aligned} \boldsymbol{C}^s &= \forall \boldsymbol{x} \forall \boldsymbol{z} \boldsymbol{E}[\boldsymbol{x}, \boldsymbol{f}(\boldsymbol{x}), \boldsymbol{z}, \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{z})] \\ \boldsymbol{C}^h &= \exists \boldsymbol{y} \exists \boldsymbol{w} \boldsymbol{E}[\boldsymbol{c}, \boldsymbol{y}, \boldsymbol{f}(\boldsymbol{y}), \boldsymbol{w}]. \end{aligned}$$

To every quantifier eliminated by the process of forming the Skolem and Herbrand normal forms corresponds a Skolem axiom of the form mentioned above. Hence $\boldsymbol{As} \vdash \boldsymbol{B} \leftrightarrow \boldsymbol{B}^h$ and $\boldsymbol{Bs} \vdash \boldsymbol{B} \leftrightarrow \boldsymbol{B}^s$ in the theories $\boldsymbol{As}$ and $\boldsymbol{Bs}$ consisting of such axioms. It is not difficult to see that $\vdash \boldsymbol{B} \to \boldsymbol{B}^h$ and $\vdash \boldsymbol{B}^s \to \boldsymbol{S}$.

The theorems 44 and 45 are fundamental in that they solve the *characterization problem* for theories. The characterization problem calls for finding of necessary and sufficient conditions (apart from the existence of proofs) which state when a formula $\boldsymbol{A}$ is a theorem of a theory $\mathbf{T}$. We call a theory $\mathbf{T}$ *open* if all formulas in $\mathbf{T}$ are universal formulas. The Herbrand's theorem 45 characterizes the theorems of open theories and the Skolem's theorem 44 turns arbitrary theory into an open theory.

Theorem 44 (Skolem) Every theory has an open conservative extension.

Proof: For a given theory $\mathbf{T}$ let us denote by $\mathbf{T}^s$ the open theory consisting of the Skolem normal forms of formulas in $\mathbf{T}$. We first show that $\mathbf{T}^s$ is an extension of $\mathbf{T}$. For a $\boldsymbol{A} \in \mathbf{T}$ we have $\vdash \boldsymbol{A}^s \to \boldsymbol{A}$. Since $\boldsymbol{A}^s \in \mathbf{T}^s$ we have $T^s \vdash^+ \boldsymbol{A}$. Hence every theorem of $\mathbf{T}$ is provable in $\mathbf{T}^s$. In order to show $\mathbf{T}^s$ conservative over $\mathbf{T}$ we define a function F such that for a tableau $\boldsymbol{X} : \mathbf{T}^s \vdash \boldsymbol{B}$ where $\boldsymbol{B}$ is in the language of $\mathbf{T}$ we will have $F(\boldsymbol{X}, \boldsymbol{B}) : \mathbf{T} \vdash \boldsymbol{B}$. The application $F(\boldsymbol{X}, \boldsymbol{B})$ extracts first from $\boldsymbol{X}$ a list $\boldsymbol{As}^s$ consisting of all axioms of $\mathbf{T}^s$ used in $\boldsymbol{X}$. Let $\boldsymbol{As}$ be the list like $\boldsymbol{As}^s$ but consisting of the corresponding axioms of $\mathbf{T}$. The function F then constructs a tableau $\boldsymbol{X}_1 : \mathbf{T}_1 \vdash \boldsymbol{B}$ where $\mathbf{T}_1$ is the theory $\mathbf{T}$ extended by the finite set of Skolem axioms obtained from the construction of Skolem normal forms for the formulas in $\boldsymbol{As}$. The construction of $\boldsymbol{X}_1$ is possible as for every formula $\boldsymbol{A}$ in $\boldsymbol{As}$ we have $\mathbf{T}_1 \vdash \boldsymbol{A} \leftrightarrow \boldsymbol{A}^s$. The function F yields a tableau for $\boldsymbol{B}$ in $\mathbf{T}$ obtained from $\boldsymbol{X}_1$ by repeated applications of the theorem 42 where Skolem axioms are eliminated from the tableau $\boldsymbol{X}_1$ by calls to the function *Skelim*. The function F is a $\mathcal{LESP}$ function provided we can go from the list $\boldsymbol{As}^s$ to the list $\boldsymbol{As}$ by a $\mathcal{LESP}$ function. If this is the case then the open extension of $\mathbf{T}$ is finitary conservative. ■

Theorem 45 (Herbrand) For an open theory $\mathbf{T}$ and a formula $\boldsymbol{A}$ in the language of $\mathbf{T}$ which does not contain the Skolem functions introduced in the formation of $\boldsymbol{A}^h$ we have $\mathbf{T} \vdash \boldsymbol{A}$ iff $\vdash_i \boldsymbol{Ts}_1 \to (\boldsymbol{A}^h)_1$ for a list $\boldsymbol{Ts}_1$ of quantifier instances of formulas of $\mathbf{T}$ and a quantifier instance $(\boldsymbol{A}^h)_1$ of $\boldsymbol{A}^h$.

This is finitary as we have p.r. $\mathcal{LESP}$ functions *Herbrand*$_1$ and *Herbrand*$_2$ such that **(a):**

$$\frac{\boldsymbol{X} : \mathbf{T} \vdash \boldsymbol{A}}{\boldsymbol{X}_1 : \vdash_i \boldsymbol{Ts}_1 \to (\boldsymbol{A}^h)_1}$$

where $\mathit{Herbrand}_1(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{X}_1, \boldsymbol{Ts}_1 \to (\boldsymbol{A}^h)_1$ and **(b):**

$$\frac{\boldsymbol{X} : \vdash_i \boldsymbol{Ts}_1 \to (\boldsymbol{A}^h)_1}{\mathit{Herbrand}_2(\boldsymbol{X}, \boldsymbol{Ts}_1 \to (\boldsymbol{A}^h)_1, \boldsymbol{Ts}, \boldsymbol{A}) : \mathbf{T} \vdash \boldsymbol{A}}.$$

Proof: **(a):** The application *Herbrand*$_1(\boldsymbol{X}, \boldsymbol{A})$ constructs first a tableau $\boldsymbol{X}_2 : \vdash \boldsymbol{Ts} \to \boldsymbol{A}$ where $\boldsymbol{Ts}$ is a list of axioms of $\mathbf{T}$ used in $\boldsymbol{X}$. Then it constructs a tableau $\boldsymbol{X}_3$ for the existential formula $\boldsymbol{Ts} \to \boldsymbol{A}^h$ by using the property $\vdash \boldsymbol{A} \to \boldsymbol{A}^h$. We have $(\boldsymbol{Ts} \to \boldsymbol{A})^h = \boldsymbol{Ts} \to \boldsymbol{A}^h$ and we can apply the lemma 41 to yield *Qinstance*$_1(\boldsymbol{X}_3, \boldsymbol{Ts} \to \boldsymbol{A}^h)$.

(b): The application $Herbrand_2(\boldsymbol{X}, \boldsymbol{Ts}_1 \rightarrow (\boldsymbol{A}^h)_1, \boldsymbol{Ts}, \boldsymbol{A})$ first uses the lemma 41 to obtain the tableau

$$\boldsymbol{X}_1 = Qinstance_2(\boldsymbol{X}, \boldsymbol{Ts}_1 \rightarrow (\boldsymbol{A}^h)_1, \boldsymbol{Ts} \rightarrow \boldsymbol{A}^h)$$

for $\boldsymbol{Ts} \rightarrow \boldsymbol{A}^h$. This is turned into a tableau $\boldsymbol{X}_2 : \mathbf{T} \vdash \boldsymbol{A}^h$ and then to a tableau $\boldsymbol{X}_3 : \mathbf{T}_1 \vdash \boldsymbol{A}$ where $\mathbf{T}_1$ is obtained from $\mathbf{T}$ by the addition of the Skolem axioms needed in the formation of $\boldsymbol{A}^h$. The tableau $\boldsymbol{X}_3$ can be constructed as we have $\mathbf{T}_1 \vdash \boldsymbol{A} \leftrightarrow \boldsymbol{A}^h$. Now we repeatedly apply the theorem 42 to eliminate by the function *Skelim* the Skolem axioms from the tableau $\boldsymbol{X}_3$ and to yield a tableau for $\boldsymbol{A}$ in $\mathbf{T}$. ■

The theorem 42 on Skolem axioms seems to be our contribution to the finitary presentation of the characterization problem. The theorems 43, 44, and 45 are relatively simple consequences of the theorem 42. The reader should compare this with the extremely complicated proofs in the Characterization chapter in Shoenfield [13]. The reader should note especially the Shoenfield's proof of the Herbrand's theorem which is done only for a special case and uses an extremely technical lemma on the equality of special constants.

7 Conclusions

We have stated our goals in the introduction. We hope that we have justified them by the development in the two main sections.

We also hope that we have demonstrated how the techniques and methodology of declarative computer programming can be applied to non-trivial programs with tremendously rich data structures. When teaching computer programming there is only so much one can do with abstract lists and trees. After a while the exercises become boring. The manipulation of formulas and proofs by programs presents a tremendous richness of interesting challenges. Our experience in the classroom is that the students actually enjoy the developing and reasoning about such programs.

References

[1] J. Barwise (editor). *Handbook of Mathematical Logic.* North-Holland, 1977.

[2] P. Borovansky, P.J. Voda. *Types as Values Polymorphism.* In Proceedings of SOFSEM conference, 1993.

[3] M. Fitting. *First-order Logic and Automated Theorem Proving.* Springer Verlag, New York, 1990

[4] G. Gentzen. *Untersuchungen über das logische Schliessen.* Math. Zeitschr., **39**, 1934.

[5] J. Girard. Proofs and types, Cambridge University press 1989.

[6] A. Grzegorczyk. *Some classes of recursive functions.* Rozprawy Mate. No. IV, Warsaw, 1953.

[7] J. van Leeuwen (editor). *Handbook of Theoretical Computer Science. Volume B. Formal Models and Semantics.* The MIT Press, 1990.

[8] Z. Manna, R. Waldinger. The logical basis for computer programming (vol 1), Addison-Wesley 1985.

[9] J. McCarthy. *A basis for a Mathematical Theory of Computation.* In Computer Programming and Formal Systems, North-Holland, 1963.

[10] R. Péter. *Recursive Functions.* Academic Press, 1967.

[11] H.E. Rose. *Subrecursion, Functions and Hierarchies.* Clarendon Press, Oxford, 1984.

[12] H. Schwichtenberg. *Elimination of higher type levels in definitions of primitive recursive functionals by means of transfinite recursion.* In Logic Colloquium'73, North-Holland, 1975.

[13] J.R. Shoenfield. *Mathematical logic.* Addison-Wesley, 1967.

[14] R. Smullyan. *First Order Logic*, Springer Verlag, 1968.

[15] W.W. Tait. *Nested recursion.* Math. Annalen, **143**, 1961.

[16] P.J. Voda. *Types of Trilogy.* In Proceedings of the Fifth International Conference on Logic Programming, MIT Press, Cambridge MA, 1988.

[17] P.J. Voda. *Subrecursion as Basis for a Feasible Programming Language.* In Proceedings of Csl'94 (Logic in Computer Science) conference, Lecture Notes in Comp. Sci., Springer Verlag, 1995.

[18] A. Yasuhara. Recursive function theory and logic, Academic Press 1971.