

From Hilbert's Program to a Logic Tool Box

J.A. Makowsky

Department of Computer Science
Technion – Israel Institute of Technology
Haifa, Israel
janos@cs.technion.ac.il
www.cs.technion.ac.il/~janos

Abstract. In this paper I discuss what, according to my long experience, every computer scientists should know from logic. We concentrate on issues of modeling, interpretability and levels of abstraction. We discuss what the minimal toolbox of logic tools should look like for a computer scientist who is involved in designing and analyzing reliable systems. We shall conclude that many classical topics dear to logicians are less important than usually presented, and that less known ideas from logic may be more useful for the working computer scientist.

For Witek Marek, first mentor,
then colleague and true friend,
on the occasion of his 65th birthday.

1 Introduction

In this position paper I address the question what aspects of Mathematical Logic and the Foundations of Computer Science should be taught to computer professionals whose responsibilities go beyond the mere production of consumer software, web-page content and support, and other profane activities; to computer professionals who are *responsible* for the creation of *life-critical* software systems which are used in airplanes, medical equipment, atomic power plants, traffic control systems, to name just a few; to computer professionals, in short, who have a global grasp of their technical abilities and limitations, and who can take full responsibility for the safe functioning of their creations. Roger Godement expressed this well in his preface to his "Cours d'Algèbre" [God66]¹.

Even in teaching mathematics we can at least attempt to teach the students the flavor of freedom and critical thought, and to get them used to the idea of being treated as humans empowered with the ability of understanding.

¹ My emphasis and translation. The original is in French.

More specifically, Nikolaus Wirth, who won the Turing Award in 1984 “for developing a sequence of innovative computer languages, EULER, ALGOL-W, MODULA and PASCAL” criticized vehemently the current practices of program development and stated clearly, what has to be required from a computer professional²:

- *Reliable and transparent programs are usually not in the interest of the designer.*
- *A good designer must rely on experience, on precise, logic thinking; and on pedantic exactness. No magic will do.*
- *In our profession, precision and perfection are not a dispensable luxury, but a simple necessity.*

The following text is not a scientific paper. It is really a prose version of a set of slides which I used at several occasions to present³ my ideas on the subject. I therefore use the colloquial first person and allow myself to reveal some autobiographic details to stress the personal experience which is behind my exposition.

Between Theory and Industrial Experience

It may be useful for the reader to be aware of my own research background, my own undergraduate teaching experience, and my industrial experience. It is the combination of the three which shaped my view of the subject.

Research. My own research started in Mathematical Logic, shifted to the foundations of Computer Science, and then returned again to more mathematical topics. More precisely, although a strict separation of my interests is impossible, my research roughly can be divided as follows:

My early work (1970-85) was mainly in mathematical logic, in particular in classical model theory ([Mak74]), abstract model theory and generalized quantifiers ([BF85,Mak85b,MM85,Mak85a]). In the period of 1980-1995 my work shifted to applications of logic to semantics of programming languages, logic programming and databases ([Mak84,Mak92]), after which I shifted again (starting in 1995) to applications of logic to algorithmics and combinatorics, as documented in [Mak04,Mak06].

Industrial consulting. My views on computing have been shaped also, and significantly, by three exposures to computing in the real world of commercial enterprises: In 1971-1972 I worked as an employee of a subcontractor of the European Space Agency (ESA), for whom I designed and implemented algorithms to compute the orbits of geo-stationary satellites. Here it was essential to prove

² My emphasis, quoted from Wikipedia

³ I have presented various version of these slides at the following conferences: LPAR’07 in Yerevan, Armenia, in October 2007, ISAIM’08 in Fort Lauderdale, in January 2008 (Special Session in Honor of W. Marek’s 65th birthday), CSR’08 in Moscow, in June 2008 and CiE’08 in Athens, also in June 2008.

the programs correct, although program verification was not yet recognized as a technology to be developed. I had to invent my own methods which I later recognized in the literature as the intermediate assertion method.

In 1986 I started my activity as founding scientific advisor to **mental images**⁴, the recognized international leader in providing rendering and 3D modeling technology. I remained their scientific advisor formally till 2007. During these twenty years I got a first hand experience of the development process of highly reliable, highly portable and well documented software products. Some of this experience came to fruition in the commissioned feasibility study (1987-1989) for **Zurich Financial Services**, then **Zurich Insurances**, about questions concerning the insurability of software and the liability of software engineers. I realized that the issue of creating *reliable* and *certifiable* software was of utmost importance and it shaped my own understanding of what is required from a professional in the field of algorithmic design and software engineering. From 1987-1990 I published a regular column in the Swiss paper **Finanz & Wirtschaft** dealing with computing related topics, in which I reflected upon my experiences.

Teaching. During my time in the Computer Science Department of the Technion – Israel Institute of Technology in Haifa my undergraduate teaching dealt and deals with Logic for Computer Science, both on introductory and advanced levels; Database Systems and Database Theory; Foundations of Logic Programming and of Automated Theorem Proving.

Caveat. My views expressed in the sequel are definitely biased. Reader be aware!

The Students I Have in Mind

I want to examine what we should teach from logic to our *non-specialized undergraduate students who are aiming for Bologna-style graduate program*. I mean, what does *every* graduate of Computer Science have to learn in/from logic? The current syllabus is often justified more by *the traditional narrative* than by *the practitioner's needs*. The practitioner's needs are determined by what she needs to understand her own activity in dealing with his computing environment. As a computer/computing engineer she should be aware of the inherent difference between *consumer products* and *life-critical hardware and software*. The occasional failure of consumer goods is beneficial to the functioning of the Fordistic consumer society in that it maintains the consumption cycles needed for its functioning. The failure of life-critical products is disastrous for all the parties involved. Life-critical products have to be properly *specified, verified, tested and certified* before they can be released. The practitioner therefore needs a basic understanding of what it means to properly specify, test, verify and, possibly, certify a product. Practically speaking,

⁴ **mental images** was founded by Rolf Herken in 1986 <http://www.mentalimages.com>. In 2007 ownership passed to NVIDIA,

- she should understand the meaning and implications of modeling her environment as precise mathematical objects and relations;
- she should understand and be able to distinguish intended properties of this modeling and side-effects;
- she should be able to discern different level of abstraction;
- she should master the (non-formalized) language of sets and second order logic which enables her to speak about the modeled objects;
- she should understand what it means to prove properties of modeled objects and relations;
- but she should also understand the *inherent limitations* of what can be achieved, and the limitations of her own activity.

2 Sets and the Logical Foundations of Mathematics

Whether we like it or not depends on our philosophical position, if we have one at all, but it is a fact supported by a large social consensus, that the language of sets is the *most used and most accepted way of modeling mathematical objects*. A very convincing discussion, why sets are used that way, is given in [BG08]. We are used to modeling automata of all sorts including Turing machines as tuples of sets, functions and relations. We do the same when we discuss behavior of hardware and software, when we prove properties of modeled artefacts, and when we show that certain combinations of properties of such artefacts cannot be achieved.

The emergence of the language of sets goes back to the work of G. Cantor, R. Dedekind, G. Frege and G. Peano, who felt the need to put mathematics on new rigorous foundations upon which the growing edifice of real and complex analysis could be built. Cantor initiated the use of sets for modeling natural, real and complex numbers and their functions, and Frege wanted to derive the rules of set formation from logic. Frege's program intended to derive the foundations of mathematics from logical principles. It did this using sets as the universal data structure for modeling mathematical objects from logic. The history of logic in the years between 1850 and 1950 is the history of successes and failures of Frege's program. This history forms the traditional narrative along which we are accustomed to teach logic. I will argue that this narrative is misleading as far as the working mathematician or computer scientist or engineer is concerned.

Let us look at this traditional narrative. We start by paraphrasing the history of Logicism from Frege to Gödel, and further to the re-evaluation of Frege's program.

Act I: Cantors Paradise

- First G. Cantor (1874 - 1884) created the Paradise of Sets.
- Then G. Frege (1879) created the modern Logical Formalisms, including the *correct binding rules for quantification*, and
- set out to lay the Foundations of Mathematics with his *Die Grundgesetze der Arithmetik*, Volume1 (1893), see [Bur05].

- The book was not well received. Only G. Peano, author of *The principles of arithmetic, presented by a new method* (1889), [Ken73] wrote a positive review of it.

Act II: Paradise lost

- On 16 June 1902, Bertrand Russell pointed out, with great modesty, that the Russell paradox gives a *contradiction* in Frege’s system of axioms.
- ... and with Russel’s paradox started the *crisis* of the *Foundations of Mathematics*. Putting it this way, the role of the paradoxes is overstated.
- G. Cantor already had sensed the problems, when he noticed trouble with the ”set of all sets” and his notion of cardinality:
Let V be the set of all sets. Then its power set $P(V)$ is a subset of V . But Cantor proved that the cardinality of the power set $P(A)$ of a set A is always strictly bigger than the cardinality of A . On the other hand the cardinality of a subset of A is at most the cardinality of the set A , a contradiction.

Act III: Hilbert’s Program

D. Hilbert around 1920 designed a program to provide a secure foundation for all mathematics. In particular this should include⁵:

- *Formalization* of all mathematics: all mathematical statements should be written in a precise formal language, and manipulated according to well defined rules.
- *Completeness*: a proof that all true mathematical statements can be proved in the formalism.
- *Consistency*: a proof that no contradiction can be obtained in the formalism of mathematics. This consistency proof should preferably use only ”finitistic” reasoning about finite mathematical objects.
- *Conservation*: a proof that any result about ”real objects” obtained using reasoning about ”ideal objects” (such as uncountable sets) can be proved without using ideal objects.
- *Decidability*: there should be an algorithm for deciding the truth or falsity of any mathematical statement.

Hilbert’s Logic Lectures

In 1928, D. Hilbert and W. Ackermann published *Grundzüge der theoretischen Logik*, [HA28,HA49,HA50]. The points of interest to us are:

- The Logic in question is *Second Order Logic*.
- What we call *First Order Logic*, is called there the *restricted calculus*.
- They *prove* soundness of the calculus, and ask the question of *completeness*.

⁵ This subsection is a quote from Wikipedia. Its author is unknown.

The book is soon translated into English, French and Russian and remains the most widely used reference for more than thirty years. K. Gödel, as a graduate student, reads the book in 1928. The original book contains several technical mistakes which are fixed in subsequent editions. The first English edition [HA50] gives credit to A. Church and W. Quine for pointing out some mistakes in the second German edition.

The book states as the main problem of Logic its axiomatization and proofs of the

- Independence of the axioms
- Consistency of the axioms
- Completeness of the axioms
- The decidability of the consequence relation known as the *Decision Problem*.

Act IV: Rise and Fall of Hilbert's Program

Initial successes:

- Leopold Löwenheim (1915), Thoralf Skolem (1920), Mojżesz Pressburger (1929), Alfred Tarski (1930), Frank Plumpton Ramsey (1930), László Kalmár (1939) and many others prove partial *decidability* results for fragments of Logic, and for Arithmetic, Algebra, Geometry.
- In 1929 Kurt Gödel proves the *completeness* of the Hilbert-Ackermann axiomatization of the *restricted* (first order) calculus.

Final blows:

- 1931 K. Gödel proves that every recursive theory which contains arithmetic is *incomplete*.
- 1931 K. Gödel proves that every recursive consistent theory which contains arithmetic cannot prove its own consistency.
- 1936 Alonzo Church and Alan Turing show that already for the *restricted* calculus with free relation variables the set of tautologies is not computable (but is semi-computable). Hence, they gave a negative solution to the Decision Problem.
- 1951 Boris Trakhtenbrot showed that Logic restricted to finite interpretations is not axiomatizable.

The most comprehensive account of solvable and unsolvable cases of the Decision Problem can be found in [BGG97].

Act V: Clarifications and Repairs

Out of the ashes rise the four classical sub-disciplines of mathematical logic:

Set Theory arises from work by E. Zermelo, D. Mirimanoff, J. von Neumann, A. Fränkel, K. Gödel and P. Bernays. Alternative approaches were developed by, among others, W. Quine, W. Ackermann, and J.L. Kelley and A.P. Morse, and more recently, by P. Aczel.

Set theory, in contrast to using sets in mathematical practice, is mostly concerned with settling questions around the axiom of choice and cardinal arithmetic, or in formulating alternatives, such as the axiom of determinacy, and in clarifying their impact on questions in topology and analysis. Today, set theory is a highly specialized branch of technical mathematics with little impact on computer science.

Proof Theory arises from work by W. Ackermann, G. Gentzen, J. Herbrand, D. Hilbert and P. Bernays.

It has developed into a full-fledged theory of proofs, comprising the analysis of (transfinite) consistency proofs in terms of ordinals and fast-growing functions, program extraction from proofs, and resource analysis related to provability. It also plays an important role in all aspects of automated reasoning, an important branch of Artificial Intelligence.

Recursion Theory arises from work by E. Post, J. Herbrand, K. Gödel, A. Church, A. Turing, H. Curry. Recursion theory developed at one side into degree theory, classifying non-computable functions according to their different levels of complexity, at the other side it developed into Computability Theory and has become one of the pillars of Computer Science education in its own right.

Model theory arises from work by T. Skolem, A. Tarski, A. Robinson, R. Fraïssé and A. Mal'cev.

Two main directions evolve, classification theory, and a more algebraic and geometric theory, linking model theory with algebraic geometry and number theory. Although finite model theory has its early origin in the study of the “*spectrum problem*” initiated by H. Scholz in [Sch52], it was through Automata Theory, Database Theory and Complexity Theory that it evolved into its own discipline with a legitimate place in advanced Computer Science education.

... and for long this remained the classical divide of **Mathematical Logic**.

J. Shoenfield’s monograph [Sho67] is possibly the only monograph covering all aspects of Mathematical Logic up to the boundaries of research of his time. Since then the four classical disciplines have pursued their own paths, and among the younger generations of researchers it cannot be taken for granted that they have studied the four disciplines in depth.

Act VI: 100 years later - Fixing Frege

If only G. Frege had not been so intimidated by B. Russel’s letter. C. Wright, P. Geach and H. Hodes suggested, and G. Boolos proved (1987) that a modified Frege program actually is feasible, [Boo98a,Boo98c,Boo98b]. They noted that the famous contradiction stemmed from the axiom which states, roughly, that

the extension of any concept is a set. However, this axiom is only used to derive an abstraction principle, called Hume's principle, which states, again roughly, that two extensions have the same cardinality if and only if there is a bijection.

So we have for the modified Frege system:

Frege: The Peano Postulates can be deduced in dyadic second order logic from Hume's principle and suitable definitions of the natural numbers (Frege's Arithmetic).

Boolos: Frege's arithmetic is interpretable in second order Peano Arithmetic.

Some (the Neo-Logicians) argue that this justifies a revival of Logicism. But it also creates new problems. A thorough discussion of the pros and cons of Neo-Logicism can be found in J. Burgess [Bur05]. A thorough discussion of *abstraction principles* similar to Hume's Principle can be found in K. Fine [Fin02].

So much for the "big crisis".

However, the set theory needed for the foundations of Computer Science can be derived from logical principles without running into new foundational problems. In the sequel we discuss how exactly this can be done with our goals in mind, i.e., how such a foundation will help our responsible computer expert in understanding what he tries to do.

3 The Foundations of Mathematics and Computer Science

What Frege and Russel and Whitehead had in mind, viz. to build the foundations of mathematics from scratch, was done in a more intelligible (but still not too user friendly) way by E. Landau in his *Foundations of Analysis* first published in German in 1930, and in English in 1951, with many reprints, the latest in 1999 with a German-English vocabulary by the American Mathematical Society, [Lan99]. In this book he explicitly constructs the real and complex numbers from the standard model of Peano arithmetic. Landau's style is very dry and concise, and the text was written for mature mathematicians. A more pedagogical version of the same constructions can be found in S. Feferman's [Fef64], which I also would love to see reprinted. One can view such a foundation of Analysis as a *pragmatic* version of Frege's program. Roughly, we proceed as follows:

- One starts with a cumulative hierarchy of sets, based on the empty set alone (or with urelements) and natural set construction principles which also allow the construction infinite sets.

- Then one defines inductively the natural numbers with a successor function, and the sets of finite words over a (not necessarily) finite alphabet (a set), with an append operation for each element of the alphabet.
- One then proceeds with defining the number systems \mathbb{N} , \mathbb{Z} , \mathbb{Q} and their arithmetic operations *inductively* and using *quotient structures*.
- Then one constructs the reals \mathbb{R} , using Dedekind cuts.
- In similar ways one constructs other structures, say, groups, fields, topological spaces, Banach spaces, Lie algebras, which are specified *axiomatically*.
- *Existence* of axiomatically defined objects has to be established by an *explicit sequence of set construction steps within the cumulative hierarchy*.

Set Theory in Computer Science

Clearly, one can apply the same methods to model objects of the computing world, such as automata, formal languages, programs, data structures, etc. One *should* adapt Landau's way for modeling the basic data structures of Computer Science. I have attempted to do this in the course *Sets and Logic for Computer science*, which we teach in the third semester in our Computer Science undergraduate program. In Section 4, I will outline the main point of this program.

Besides using set theory for modeling purposes, the computer scientist uses only few ingredients of set theory:

1. The Cantor-Bernstein Theorem to prove equicardinality,
2. The fact that a countable union of countable sets is countable,
3. The fact that the cardinality of the power set of a set A is always bigger than that of A ,
4. The relationship between the termination of processes and well-orderings.
5. The Recursion Theorem.
6. Some Fixed-Point Theorems.

The first three of these go ultimately back to Cantor's original work and are very basic. The Recursion Theorem and Fixed Point Theorems should probably be taught in a more advanced course, using, say, Y. Moschovakis' book [Mos94].

Recursion Theory vs Computability Theory

Recursion Theory got its name for a good reason: The computable functions over the natural numbers were defined recursively, and early Recursion Theory consisted in studying the strength of various proposed recursion schemes. Recursion schemes can be replaced by register machines. Computability Theory studies usually the computable relations and functions over sets of words. The three approaches are inter-translatable, but they are not the same. It is a pity that teaching all these complementary notions of computability is not always part of the Computer Science curriculum. Be that as it may, Computability Theory has emancipated itself from Logic and in Computer Science the two are often taught independently. A good exception is the monograph [Pap94].

Proof Theory

Proof theory evolved around the question of what type of consistency proofs are at all possible. As a spin-off the field of deduction-based automated reasoning and automated theorem proving came into being. Proof theory is also used in the foundations of programming languages, where it generated a rich literature of deep insights into the nature of programming. Some basic principles of automated reasoning do belong in a beginner's course on Artificial Intelligence, and some basic facts about functional programming do belong in a basic course on Programming Languages. However, only little of this rich material is suitable for the student I have in mind. A comprehensive survey of Proof Theory is [Bus98]. It remains to be seen to what extent Girard's ambitious program of using Linear Logic (a resource conscious version of natural deduction) is valuable for the sophisticated practitioner. The current status of this approach is described in [HO00,Gir01,Abr03]. In any case, this work is far too sophisticated to be presented in the course framework I have in mind.

Model Theory

The main tools of classical model theory almost all derive from the compactness theorem and variations of the model existence theorem. Using these tools one proves preservation theorems, the omitting types theorem and develops a general understanding of the possible structure of models of first order theories. I have described how to use these tools in the Computer Science context, in database theory, the foundations of Logic Programming, and the specification of data types, [Mak84,Mak92]. It turned out, however, that the Compactness Theorem is mostly suitable when dealing with infinite structures, and the most prominent application of the Compactness Theorem in Computing is Herbrand's Theorem with its ramifications in Automated Reasoning and Logic Programming. The most important tools from Model Theory in algorithmic applications are the Ehrenfeucht-Fraïssé games and the Feferman-Vaught Theorem and its variations. The former is omnipresent in Finite Model Theory, cf. [EF95,Lib04] and the survey of the uses of the latter can be found in [Mak04]. For other failures of classical theorems of First Order Logic when restricted to the finite case, such as the compactness theorem and various preservation theorems, see Y. Gurevich [Gur88] and B. Rossman [Ros08].

One should add here that the combination of the Compactness Theorem with the Ehrenfeucht-Fraïssé games leads to Lindström's characterization of First Order Logic. The attempts to develop an *Abstract Model Theory* are documented in the monumental [BF85]. This line of reasoning had a considerable impact on Finite Model Theory and Descriptive Complexity in providing techniques for defining logics which capture complexity classes. For the advanced student, applications of Finite Model Theory to Computer Science are surveyed in [GKL⁺07].

The classical textbooks in Logic

The available undergraduate texts of Logic for Computer Science follow too often the narrative of the *Rise and Fall of Hilbert's Program*. They emphasize the classical Hilbertian topics.

- Logic is needed to resolve the paradoxes of set theory.
- First Order Logic is THE LOGIC due to its completeness theorem.
- The main theorems of logic are the *Completeness Theorem* and the *Compactness Theorem*
- The tautologies of First Order Logic are not recursive.
- Arithmetic Truth is not recursive enumerable.
- One cannot prove CONSISTENCY within rich enough systems.

This is NOT what a Practitioner
of Computing Sciences NEEDS !

Other texts are often written with a very special agenda reflecting the author's research interest or his particular tastes. Finally, there are texts which are really written with the undergraduate Computer Science student in mind. But then they are often either written specifically with programmers in mind, or do not deal with the data modeling issues⁶. Logic is first of all a language in which we express ourselves before we prove statements. We first have to formulate a *specification*, a *database query*, an *intermediate assertion*, a *loop invariant*, before we prove them to hold, to be valid, or for two of them to be equivalent or not. Logic deals with **definability** issues as much as with **provability** issues, something which in the Hilbertian tradition is easily forgotten. An introductory text in Logic for Computer Science should choose its topics in way that the student meets the topics taught again in later courses. When we teach linear algebra in the first year of a mathematics curriculum, most of the topics reappear in vector analysis, differential equations, physics, statistics etc. We have to build our syllabus of logic keeping this in mind.

4 Foundations for the Practitioner

It is customary in the discourse of education to distinguish between knowledge of *theoretical orientation* and *practical* knowledge, which consists of *tools* and *skills*. In our case this means:

⁶ A recommendable exception is the recent book by R. Bornat[Bor05] is a lovely introduction to Logic for programmers.

Theoretical orientation:

- *awareness* that our domain of discourse is an *idealized world of artefacts* which models fairly accurately the artefacts which allow us to run and interact with computing machinery.
- *awareness* of the different levels of abstractions.
- *awareness* that in this world of artefacts there are *a priori limitations*. Not everything is realizable, computable, etc.

Practical knowledge:

- *tools* which allow us to *model new artefacts*, whenever they arise;
- *tools* which allow us to *prove properties* of the modeled artefacts.

The student needs a carefully adapted blend of the *practical Frege* program, with the knowledge of its *limitations*. He needs both *proficiency* and *performance* in his practical knowledge.

5 Lessons from 150 years of History

I have spent so much space reviewing what I consider noteworthy in the evolution of the Logicists program because I do want to draw some lessons from it which are *not foundational* but *practical*. I would like the graduates of Computer Science who intend to deal with the production and maintenance of life-critical reliable and certifiable hardware and software products to be capable of understanding not only the functioning of the product itself, but also its interaction with its surroundings at large. This requires a sophisticated use of many skills which can be learned within the framework logic. Let us examine some of these one by one.

Lesson 1. Modeling the world

The first lesson to learn concerns understanding the modeling process and its pitfalls.

Our scientific language: In our scientific and engineering discourse we use our natural language *enhanced* by standardized and precise use of boolean operations, quantification and the use of naive language of sets.

Our universal data structure: The naive language of sets allows us to build a cumulative world of sets which serves as our universal data structure.

Modeling the world: All the abstract and concrete artefacts of our discourse can be modeled within this world of sets. In particular, we should be accustomed to model *all* artefacts of our computing world by objects in the world of sets constructed according to the rules of set construction.

Modeling involves side effects: However, modeled artefacts have properties not intended.

Digression: Side effects: We shall use the term “side effect” here for any *accidental property* of the modeled artefact, be it *intentional* or *accidental*. This deviates from the usage of the term in the programming community, where the intentional aspect is emphasized. The Wikipedia explains:

In computer science, a function or expression is said to produce a side effect if it modifies some state in addition to returning a value. For example, a function might modify a global or a static variable, modify one of its arguments, write data to a display or file, or read some data from other side-affecting functions. Side effects often make a program’s behavior more difficult to predict. A “Safe” operation is one that is guaranteed to be free of side-effects, i.e., it may be relied upon to leave the state of the system unchanged. Queries are the canonical safe transactions. Safe operations are also idempotent, although the reverse is not necessarily true.

In medicine, however, the *accidental*, if not *harmful* aspect of side effects is prevalent. We again quote the Wikipedia:

In medicine, an adverse effect is a harmful and undesired effect resulting from a medication or other intervention such as chemotherapy or surgery. An adverse effect may be termed a “side-effect” (when judged to be secondary to a main or therapeutic effect) and may result from an unsuitable or incorrect dosage or procedure (which could be due to medical error)

In mathematical modeling the use is *neutral*. A side effect may be a curiosity of no further importance, or it may become a *feature*, like in modeling ordinals as so called von Neumann ordinals⁷, i.e., as special transitive sets.

Digression: the ordered pair: Ordered pairs could be introduced via an abstraction principle:

$(x, y) = (x', y')$ if and only if $x = x'$ and $y = y'$.

But usually the ordered pair is modeled directly:

N. Wiener: $(x, y)_W := \{\{\{x\}, \emptyset\}, \{\{y\}\}\}$,

K. Kuratowski: $(x, y)_K = \{\{x\}, \{x, y\}\}$,

Simplified : $(x, y)_S = \{x, \{x, y\}\}$.

Now one has to verify that $(x, y) = (x', y')$ if and only if $x = x'$ and $y = y'$. All the proposed versions do satisfy this, but the proofs differ. Kuratowski’s version is the accepted definition today. But all definitions have **side effects**, e.g., x is an element of $\{x, \{x, y\}\}$ but not of $\{\{x\}, \{x, y\}\}$. Proving properties of objects which depend on the use of ordered pairs should not use these side effects, but only the defining property. How do the three modeled ordered pairs

⁷ J. von Neumann studied ordinals in his Ph.D. Thesis (1926). However, J. von Neumann, in his thesis, credits D. Mirimanoff for some of his ideas. J. von Neumann’s approach was widely adopted, cf. [Lev79, Jec78, Mos94], or for the less sophisticated reader, [Hal60] and [Dev93].

differ from each other? Syntactically, Kuratowski's pair and its simplified version are shorter than Wiener's version, and are simpler in the sense that they appear earlier in the cumulative hierarchy. However, the simplified version has a serious drawback: to prove that it satisfies its specifications the axiom of Foundation is needed, which is not the case for Kuratowski's version. Kuratowski's pair has become the universally accepted definition of the ordered pair, because it finds the right balance between its simplicity in form and simplicity in the proof of its required properties.

The distinction between specified properties and side effects should be taught early on!

Fixing levels of abstraction: In the process of modeling we introduce structures, and fix which sets are not further to be analyzed. We are accustomed to define a graph as a pair $\langle V, E \rangle$, where V is any (finite) set and $E \subset V^2$ is a binary symmetric relation. A finite automaton is defined as a tuple $\langle S, \Sigma, R, I, T \rangle$, where S is any (finite) set (of states), Σ is any finite set (the alphabet), $R \subset S \times \Sigma \times S$ is a ternary transition relation, and $I, T \subset S$ are the initial and terminal states respectively. Our students should be familiar with this kind of modeling, in particular, they should understand that the properties of the elements of S and Σ are side-effects and should be ignored.

Like in the foundations of Analysis, as practiced by R. Dedekind, E. Landau and N. Bourbaki, we need the *precise language mix* of *normalized natural language* augmented by the *language of sets* to *model* the *idealized artefacts* of computer science. However, to model the artefacts we also need basic tools. These are:

- inductive definitions and proofs by induction;
- enumerations, proving countability and uncountability;
- (countable) well-orderings (for termination)

Using these tools we can model and analyze the most basic artefacts modeled, such as

- strings with the concatenation operation;
- natural numbers with the arithmetic operations;
- stacks, arrays;
- graphs, relational structures;
- circuits, Turing machines, register machines;
- specification and programming languages.

Digression: *Is this not "too denotational" ? . . .* a friend asked. Yes, this approach *does* map everything into *sets*. But "truth" does not necessarily *presuppose* a world of sets. Truth in the sense of Frege's world is defined by the laws (introduction and elimination rules) of logic and of the Fregean constructs. It does leave your *foundational options* open.

Lesson 2. Modeling Computability

We have already said that computability is usually taught in a separate course, be it together with formal languages or with an introduction to basic complexity theory. Nevertheless, our student should understand that computability is modeled over different domains, computing operations, and resource restrictions.

Natural numbers and recursion: The old fashioned name *Recursion Theory* for *Computability Theory* refers to the classification of computable functions using various forms of recursion over the natural numbers. This is the original definition of the set of *recursive functions*.

Turing machines and words: Turing machines model the recognition and manipulation of strings and were used before the advent of sophisticated computing devices. They led to the notion of von Neumann computers and are close to assembly languages.

Random access register machines: Register machines were introduced to model computations as performed by humans and are close to early programming languages. The content of a register may be a natural number or a string, or, for that matter, any element of a computing domain, as in relational databases. The addresses of the registers are natural numbers and have to be computed as well, possibly using special address registers. Random access machines are register machines with indirect addressing capabilities and are used to discuss program complexity.

Other models: Logic programs, Lambda calculus, Cellular Automata, Quantum computing, and many more.

Passing from one computational model to another involves modeling also the translation between the domains and the translations between programs (interpreters and compilers). Here I want to stress the different basic structures involved, and their bi-interpretability. In terms of knowledge of orientation and practical knowledge we have:

Orientation:

Not everything is computable.

Not everything is feasibly computable.

Tools:

Using the non-solvability of the Halting Problem to prove non-computability.

Using different types of reducibilities (and simulations) to pass between the models and various resource restrictions.

Using various diagonalization techniques to separate complexity classes.

I have observed that even my colleagues are sometimes imprecise: The Church Turing Hypothesis is often carelessly invoked. There is also a trend to say *computable* when actually one means *feasibly computable*, where *feasibly computable* may mean computable in *deterministic* polynomial time, sometimes computable in polynomial time with *randomized algorithms*.

It is also important to *distinguish* between complexity classes defined as *equivalence classes of problems* under certain reductions, and sometimes defined as

classes of decision (counting, approximation) problems solvable in a specific computational model.

Using polynomial time Turing reductions, the class $[SAT]_T$ of problems reducible to SAT is of the first type, NP is of the second type, and $NP = [SAT]_T$ is a theorem. In the case of counting problems $[\#SAT]_T$ is of the first type, $\#P$ is of the second type, and $\#P = [\#SAT]_T$ is not true. Using reductions in First Order Logic we still have $[SAT]_{FOL} = NP$, but not every problem X , which is NP -complete with respect to polynomial time Turing reductions satisfies $[X]_{FOL} = [SAT]_{FOL}$.

It is important to insist that slogans are replaced by precise definitions.

Lesson 3. Modeling Syntax and Semantics

In the creation of reliable software or hardware we want to be able to prove that our creation does satisfy its specification. To make such a task meaningful we have to model syntax and semantics of the language in which we express the properties of our creation in the world of sets. Only then the notion of a rigorous proof that a modeled artefact has certain properties specified in the modeled language becomes meaningful.

We look at Propositional, First Order, Second Order Logic, or any other logic of *assertions*. Again we model them in our framework of sets.

Syntax: The syntax is modeled is an inductively defined set of words, the well formed expressions. The expressions of our language contain uninterpreted symbols (propositional variables, individual variables, relation variables, etc), the non-logical symbols.

Semantics: *Structures* are interpretations of the basic non-logical symbols. *Assignments* are interpretations of the variables. The *meaning function* associates with structures, assignments, and formulas a truth value.

What is the meaning of an assertion ?

The traditional approach in logic is to consider sentences, i.e., formulas without free variables.

Without free variables: The meaning of an assertion is a *truth value*.

But this is misleading!

With free variables: The meaning of an assertion is the totality (not necessarily a set) of interpretations of all its free variables, including relation and function variables. In the case of first order variables only, where the interpretation of relation and function variables is fixed, the meaning of an assertion in a fixed structure is a *relation*. As in Classical Geometry one speaks of the *geometrical lieu* of all points satisfying an equation, we can speak of the *logical lieu* defined by a formula. In modern data base parlance this is called a *query*.

We define usually *logical validity* via truth values. It would be preferable to define *validity* and *logical consequence* directly for formulas *with free variables*. Our student is more likely to meet in the sequel of courses formulas with free variables that just formulas without.

Do we need the Completeness Theorem?

For the **practical knowledge** we need:

- The semantic notion of *logical consequence*.
- Enough basic logical equivalences to prove the *Prenex Normal Form Theorem (PNF)*.
- Introduction and elimination rules for quantifiers (via constants).
- A *game theoretic* interpretation of formulas in PNF.

For the **knowledge of orientation** we might state (but not prove) the Completeness Theorem for our redundant set of manipulation rules.

Here are the arguments for and against proving the Completeness Theorem in the first course of Logic.

The classical arguments pro:

- Completeness and its corollary, *Compactness*, is at the heart of logic.
- A logic for which one cannot prove a completeness theorem is problematic.
- Logic is identified with first order logic (in Hilbert's terms, the restricted calculus). A point of view which is strengthened by Lindström's characterization of first order logic as the only logic (over relational structures) for which every satisfiable sentence has a countable model and for which a completeness theorem can be proved.

My arguments against:

- None of these are part of the practical knowledge we aim at.
- The proof of the Completeness Theorem is a waste of time at the expense of teaching more the important skills of understanding the manipulation and meaning of formulas.
- First Order Logic is not privileged in our context. We deal very often with finite structures, where the Completeness Theorem is not true.
- *Second Order Logic* in any case is the natural logic we work in, and not taking that seriously confuses the student.

We should instead concentrate on
understanding quantification

- Although Lindström’s Theorem has been used to argue that First Order Logic is the *most natural logic*, one can easily argue otherwise: Completeness, compactness and axiomatizability are rare phenomena, true for the restricted calculus only, and we have to live without them when using formal languages suitable for the computer scientist. What may be a valid argument for the philosophy of logic need not be an appropriate argument for our discussion.

Replacing the completeness theorem: As **tools** of understanding the interplay of syntax and semantics of logical formalisms we have to

- Read, write and *understand the meaning* of First Order **and** Second Order formulas.
- Understand the relationship between *projection of relations* and first order quantification.
- Understand that *Relational Calculus* and First Order Logic are really the same (i.e., bi-interpretable).
- Introduce immediately after the proof of the Prenex Normal Form Theorem the *Ehrenfeucht-Fraïssé Game*, and proceed to show the easy direction of the *Ehrenfeucht-Fraïssé Theorem*, i.e., if a formula (say in Prenex Normal Form) of quantifier rank k is true in one but not in another structure, then we can derive from the formula a winning strategy for player I (the spoiler) for the game with k moves.
- Play with the *game interpretation* of quantifiers to analyze the *amount of quantification* needed to express, say ”there exists at least n elements x such that $\phi(x)$ ”.
- One can even point out that (the easy direction of) *Ehrenfeucht-Fraïssé Theorem* holds also for Second Order Logic.

Lesson 4. Limitations of formalisms: Definability

Before we find time to prove the Completeness Theorem, I would like the students to understand the difference between First Order (FO) and Second Order (SO) Logic. More generally, they should realize that modeling languages (modeled in the language of sets) have limited expressibility, and not everything we would like to say can be said within a given formalism. Consider the statement

”There are an equal number of x with $P(x)$ and with $Q(x)$ ”

where P, Q are unary predicate symbols. This is expressible in Second Order Logic (SO) but not in First Order Logic (FO), and we can even show the proof since we have the Ehrenfeucht-Fraïssé Games available. We can even go further, and show that connectedness on finite graphs is SO-definable, but not FO-definable. In the natural numbers \mathbb{N} , *multiplication* is SO-definable, but not FO-definable, using addition only. However, multiplication *is* FO-definable using addition and squaring. The negative result we cannot prove in an undergraduate course, as we need the decidability of FO Presburger Arithmetic. But we can explain it.

Lesson 5. Interpretability and Reducibility

Again before we use our time to prove the Completeness Theorem I would like the students to understand what it means that a structure is interpretable in another structure. As an example we may consider FO-interpretability. Let us look at the case of the natural numbers \mathfrak{N} and the integers \mathfrak{Z} with their arithmetic operations.

The integers \mathfrak{Z} with their arithmetic are *FO-interpretable* inside the natural numbers \mathfrak{N} with their arithmetic.

To get the interpretation we define a new structure from \mathfrak{N} , called a *transduction* $T(\mathfrak{N})$, and which will be isomorphic to \mathfrak{Z} , as follows.

- The new universe consists of equivalence classes of pairs of natural numbers such that $(x, y) \sim (x', y')$ iff $x + y' = x' + y$.
- The new equality is this equivalence.
- The new addition is the old addition on representatives.
- Same for multiplication.

T is a semantic map. Its syntactic counterpart is the *interpretation*

$$S : \text{Formulas} \rightarrow \text{Formulas},$$

defined as follows: For any SO-formula ϕ we let $S(\phi)$ be the result of substituting the *new* definitions of addition and multiplication and equality for the corresponding symbols. In the exact definition one has to be careful with the renaming of free variables. S and T are intimately related:

$$T(\mathfrak{N}) \models \phi \text{ iff } \mathfrak{N} \models S(\phi)$$

which is the *Fundamental Property of Transductions and Interpretations*. In our special case above we have $\mathfrak{Z} = T(\mathfrak{R})$.

In the same way we can see that

- The Cartesian product is interpretable in the disjoint union.
- Many graph transformations are given as transductions.
- All implementations of one data structure in another are of this form.
- Transductions and interpretations are everywhere

Interpretability is a kind of
logical coordinate transformation
and is at the very core of understanding
where things looking different are actually the same.

6 The Fundamental Properties of SO and FO

In a first logic course we should teach our students to think in second order concepts, because these are the concepts which are naturally used in dealing with the structures which model our artefacts. As in the book by Hilbert and Ackermann, first order logic should have its place as a *restricted* calculus.

Speaking Second Order Logic

In teaching our students to *think and speak* Second Order Logic, we should

- concentrate on developing their expressive skill in formulating properties of artefacts in the appropriate language of second order logic;
- stress that isomorphic structures satisfy the same SO sentences;
- show that checking whether a finite structure satisfies an SO-sentence can be done in polynomial space, and, in particular, checking existential SO-sentences is in non-deterministic polynomial time (NP);
- teach the usefulness of the Fundamental Property of Transductions and Interpretations;
- prove and use the Prenex Normal Form Theorem and its visualization as a two person game.

We should really *practice* thinking in SO as the natural language of specifying properties of modeled artefacts. A good example is the notion of a planar graph, where its very definition (embeddable in the Euclidean plane where edges become non-crossing Jordan curves) is not amenable to an SO-definition in the language

of graphs, but nevertheless, using Kuratowski's characterization via forbidden topological minors, planarity is SO-definable. In my experience, students often already have difficulties in expressing the notion of connectedness in graphs as an SO-definable property.

It does not make sense to prove meta-theorems for a formalized language before one masters its use in expressing concepts of its application domain.

The Fundamental Properties of FO

Second order logic has many useful fragments, out of which first order logic (FO) and monadic second order logic (MSO) are the easiest to introduce. They have one fundamental property which also holds when we restrict our attention to finite structures.

Ehrenfeucht-Fraïssé Theorem : We can introduce the Spoiler/Duplicator game played on two structures at an early stage in our logic course with great didactical benefit. Developing actual playing skills greatly enhances the students feeling for the restricted expressive power of FO and MSOL with respect to SO.

Two structures *can be distinguished* by a sentence of quantifier depth k iff the Spoiler can force a win in the Spoiler/Duplicator-game of length k .

or, equivalently

Two structures *cannot be distinguished* by a sentences of quantifier depth k iff the Duplicator can force a win in the Spoiler/Duplicator-game of length k .

We say that two structures are k -isomorphic if the Duplicator can force a win in the Spoiler/Duplicator-game of length k . With this definition we have:

k -isomorphism is preserved under the formation of disjoint unions of structures.

The game can be adapted to MSO and SO without conceptual problems. However the analysis of the game becomes much more involved. Modified versions of the statements above also hold for Monadic Second Order Logic, but *not* for SO, and the analysis of winning strategies for the SO-version are currently beyond our capabilities.

Combining EF-Games and Interpretations. Combining games and interpretations gives a very powerful tool to compute the meaning function of a FO formula in a complex structure by reducing this computation to simpler structures.

If G is obtained from graphs H_1, H_2 by applying disjoint unions, Cartesian products, and first order definable transductions T_1, T_2 , say

$$G = T_1(H_1 \times T_2(H_2))$$

then the truth of the formulas of quantifier rank k in G is uniquely and effectively determined by the truth of the formulas of quantifier rank k which hold in H_1 and H_2 .

This is the *Feferman-Vaught Theorem*. It allows us to compute the meaning function for FO-formulas (or MSO-formulas) of composite structures by reducing its computation to the meaning functions of the formulas on the components. I have surveyed how to use the Feferman-Vaught Theorem in Computer Science in [Mak04,GM03]. The point to mention here is that large artefacts are usually defined using some generalized modular construction, similar to graph grammars, and this modular construction is actually given in advance. Very often this fact can be used to design dynamic programming algorithms to verify properties of the artefact which run much faster than the worst case analysis of the property verification would predict. A case in point is Courcelle's Theorem, which states that on graphs of tree-width at most k , properties definable in monadic SO can be verified in linear time (with constants depending on k), whereas in general properties definable even in monadic SO can be arbitrarily hard within the polynomial hierarchy. There is a rich literature on generalizations of Courcelle's Theorem, most of it cited in [Mak04,GM03].

7 My Logic Tool Box

So we finally come to the description of the **Logic Tool Box** I would like to give to our students. Tools to do what, you will ask. Tools to think *rigorously* in order to approach the disciplines of programming and information processing, tools to model accurately new artefacts as they occur, tools to grasp the scope of abstraction and modularity. Our students are not logicians. Logic per se is not their main interest. Logic is for Computer Science what Hygienics is to Medicine. They should learn **rigorous informal reasoning before** they learn to model this kind of reasoning as *formalized proof sequences*. Needless to say that each tool comes with a **required skill** of how to use it.

My Logic Tool Box contains:

Modeling tools:

- Basic set construction principles;
- Inductive definitions;
- Proofs by induction;
- Basic cardinality arguments.

Logic tools:

- Propositional Logic and its axiomatization.
- Second Order Logic as the main formalism to express properties of the modeled artefacts.
- The semantic notion of logical consequence and validity.
- Validity over finite structures.
- Quantifier manipulation rules.
- Skolem functions.
- The Fundamental Property of Transductions and Interpretations.
- First Order Logic as an amenable fragment of Second Order Logic.
- The *Ehrenfeucht-Fraïssé Theorem* and its refinements.
- The Feferman-Vaught Theorem and its variations.

We said before that the Completeness Theorem for First Order Logic holds only if we define validity over all First Order Structures. For Second Order Logic one would have to explain the difference between Henkin’s notion of validity and standard second order quantification. Just stating the Completeness Theorem for First Order Logic misleads the student, and explaining its true subtleties may be beyond the undergraduate level.

Where these tools work

I have chosen the Logic Tool Box with a view to the courses our student has to take during his undergraduate studies. Ideally, the course I have in mind, **Sets and Logic for Computer Science**, should be taught in the second or third semester. The student should have already studied **Discrete Mathematics** and **Algorithms and Data Structures**, so the teacher can rely on the intuition of the students, and the examples developed in these courses. The course should play the same role as the course Number Systems used to play when it was still customary to teach it, cf. [Fef64],

The modeling skills taught in our course should help him in the following (usually compulsory) courses:

- Automata and Formal Languages
- Introduction to Computability
- Database Systems
- Graph Algorithms
- Principles of Programming Languages
- Computer Architecture
- Introduction to Artificial Intelligence

The more specialized topics of Logic should be taught in advanced courses: A course *Advanced Topics in Logic* could have three parts, covering the Completeness and Incompleteness Theorem, Ordinals and Termination, and Temporal and Modal Logic. Other topics belong where they are really used, in the courses on *Verification*, *Automated Theorem Proving*, *Principles of Logic Programming*, *Database Theory*, *Functional Programming* and so forth.

Digression: Logic and the Turing Award. The prestigious *Turing Award* has been given to logic-related work rather rarely.

- 1976:** to M. Rabin and D. Scott, for their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their classic paper has been a continuous source of inspiration for subsequent work in this field;
- 1981:** to E.F. Codd, for his fundamental and continuing contributions to the theory and practice of database management systems, esp. relational databases;
- 1991:** to R. Milner, for three distinct and complete achievements:
- 1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
 - 2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
 - 3) CCS, a general theory of concurrency.
- In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.
- 1996:** to A. Pnueli, for seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification.
- 2007:** to E. M. Clarke, E. A. Emerson and J. Sifakis, for [their roles in] developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

We could add the recipients who created and advanced the analysis of algorithms and complexity theory, D. Knuth (1974), S.A. Cook (1982), R. Karp (1985), J. Hopcroft and R. Tarjan (1986), Y. Hartmanis and R.E. Stearns (1993), M. Blum (1995). We can assume that this list reflects well how the world of computing appreciates work in logic and computability. My logic toolbox most definitely is motivated by similar considerations, which led the Turing Award Jury to the choice of these Turing Award recipients.

8 Building a syllabus

It would go beyond the scope of this paper to discuss in detail a syllabus based on the thoughts I have expressed so far. But a few remarks are in order.

Like in the 19th century tradition of the great French mathematicians, who saw their life-time task, rather than in writing many smaller papers, in writing a "Cours d'Analyse" or a "Cours d'Algèbre", a monograph which presented a comprehensive view of what they thought was worth teaching in their field, I would like to write such a book on Logic for the Computer Scientist. Two of the last great books in this tradition were written by Laurent Schwartz ([Sch67]) and was used as a course for the elitist École Polytechnique, and by R. Godement ([God66]) which was used for the equally elitist École Normal Supérieure. My

French colleagues now tell me, that even at these prestigious schools one could no longer teach a course on this ambitious level.

The student I have in mind is a student who from the beginning of his studies is determined to finish a Master's program as prescribed by the Bologna guidelines. Under this assumption the material I described should be distributed into four semester courses distributed over three semesters.

Semester 2: Sets and Logic: Concentrating on modeling artefacts within the language of sets and describing the properties of the modeled artefacts in second order logic.

Semester 3: Computability: Covering the classical material of computability and automata theory and the effect of resource bounds, but building on the rigor introduced in Sets and Logic.

Semester 3: Semantics: Discussing both the semantics of fragments of second order logic and of simple programming languages, again building on the rigor introduced in Sets and Logic.

Semester 4: Computability and Definability: Showing the basic undecidability results and the interplay between computability and definability.

9 What Was Omitted?

I have not included in my discussion what is called in the standard classification non-classical logics. These logics can also be modeled using set-theoretic tools, and indeed they are. When they find applications to Computer Science, as temporal logic [MP95] or the logic of knowledge, [FHMV95], they also find their way into more advanced courses. I have also omitted a discussion of the role of logic research in the development of more and more sophisticated programming languages and programming concepts, mostly due to lack of space. The results of these efforts are usually taught in courses dealing with programming languages. To understand the logical background of these efforts requires sophisticated tools, and is beyond the scope of what I have in mind. Neither have I discussed alternative approaches to modeling using, instead of the language of sets, the language of category theory or the language of higher types. Both approaches have their merit, but have failed to become accepted as main stream.

I have not included in my discussion two classical concerns of the debate around the Foundations of Mathematics and Computer Science: Epistemology and degrees of constructivism. A delightful and insightful presentation and discussion of these matters from a contemporary point of view can be found in [Sha00]. Although I tend to be a Platonist, viewing mathematical concepts as *real*, I am aware of the difficulties inherent in this position, cf. [Mad90, Mad98]. I am also aware of the social and cultural mechanisms at work which strongly influence how science evolves, cf. [Wil81]. However, I strongly object to the arguments which take the social and cultural mechanisms at work as a justification for the erroneous claim that scientific truth is purely social and cultural.

From a more pragmatic point of view I tend to be a Formalist, viewing the observable part of the mathematical and logical enterprise as happening on (virtual) paper written with (virtual) pencils. Concerning the degrees of constructivism I subscribe to, I only want to remark that often *non-constructive* is confused with *lack of detail*. The axiom of choice is an example of *lack of detail*. I assume that the choice function exist and I want to proceed from there, filling in the details (implementation) later, or leaving them to others. Software engineering always proceeds like this and is not considered non-constructive even by the most extreme constructivist. Using a cardinality argument to prove the existence of, say, transcendental numbers, expander graphs or other combinatorial objects, is considered non-constructive, but can be explained in the same way.

Our students, however, should rather follow the advise of the Rabbinic Sages, who admonish us not to study Kabbala (Jewish Mysticism) before the mature age of forty years and before serious exposure to the more down-to-earth matters of Talmud and Torah. Our students should view the Philosophy of Mathematics and of Computer Science as something to be left for later. Children do not question linguistic principles before they learn their first language. Scientists should not question Science before they master the craft.

Acknowledgments

I would like to thank N. Francez, D. Giorgetta, S. Halevy and D. Hay for stimulating discussions and suggestions about how to teach Logic to Computer Science students.

I would like to thank the editors of this volume, M. Kaminski and M. Truszczynski, for their patience, and their two anonymous referees for their many constructive suggestions.

I would like to thank the Trade Union of University Professors (Irgun HaSegel) of Israel for giving me time to prepare the first version of this paper. At the moment of completion we were in the fourth week of our teaching strike which lasted 10 weeks (October 2007-January 2008), and which was a resounding success.

References

- [Abr03] Samson Abramsky. Sequentiality vs. concurrency in games and logic. *Mathematical. Structures in Comp. Sci.*, 13(4):531–565, 2003.
- [BF85] J. Barwise and S. Feferman, editors. *Model-Theoretic Logics*. Perspectives in Mathematical Logic. Springer Verlag, 1985.
- [BG08] A. Blass and Y. Gurevich. Why sets? In A. Avron, N. Dershowitz, and A. Rabinowich, editors, *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, page In press. Springer, 2008.
- [BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.
- [Boo98a] G. Boolos. The consistency of Frege’s “foundations of arithmetic”. In *Logic, Logic, Logic*, pages 182–201. Harvard University Press, 1998.
- [Boo98b] G. Boolos. *Logic, Logic, Logic*. Harvard University Press, 1998.

- [Boo98c] G. Boolos. On the proof of Frege’s theorem. In *Logic, Logic, Logic*, pages 275–90. Harvard University Press, 1998.
- [Bor05] R. Bornat. *Proof and Disproof in Formal Logic*. Number 2 in Oxford Texts in Logic. Oxford University Press, 2005.
- [Bur05] J.P. Burgess. *Fixing Frege*. Princeton University Press, 2005.
- [Bus98] S. Buss, editor. *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers, 1998.
- [Dev93] K. Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory*. Springer, 1993.
- [EF95] H.D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer, 1995.
- [Fef64] S. Feferman. *The number systems: foundations of algebra and analysis*. Addison-Wesley, 1964.
- [FHMV95] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
- [Fin02] K. Fine. *The Limits of Abstraction*. Oxford University Press, 2002.
- [Gir01] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11.3:301–506, 2001.
- [GKL⁺07] E. Grädel, P. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Vardi, Y Venema, and S. Weinstein. *Finite Model Theory and its Applications*. Springer, 2007.
- [GM03] A. Glikson and J.A. Makowsky. NCE graph grammars and clique-width. In H.L. Bodlaender, editor, *Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2003), Elspeet (The Netherlands)*, volume 2880 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2003.
- [God66] R. Godement. *Cour d’Algèbre*. Hermann, Paris, 1966.
- [Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Trends in Theoretical Computer Science*, Principles of Computer Science Series, chapter 1. Computer Science Press, 1988.
- [HA28] D. Hilbert and W. Ackermann. *Grundzuge der theoretischen Logik*. Springer, 1928.
- [HA49] D. Hilbert and W. Ackermann. *Grundzuge der theoretischen Logik, 3rd edition*. Springer, 1949.
- [HA50] D. Hilbert and W. Ackermann. *Principles of Mathematical Logic*. Chelsea Publishing Company, 1950.
- [Hal60] P. Halmos. *Naive Set Theory*. Van Nostrand, 1960.
- [HO00] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for pcf: I, ii, and iii. *Information and Computation*, 163.2:285–408, 2000.
- [Jec78] T. Jech. *Set Theory*. Academic Press, 1978.
- [Ken73] H.C. Kennedy. What Russel learned from Peano. *Notre Dame Journal of Formal Logic*, 14.3:367–372, 1973.
- [Lan99] E. Landau. *Die Grundlagen der Analysis*. American Mathematical Society, 1999.
- [Lev79] A. Levy. *Basic Set Theory*. Springer, 1979.
- [Lib04] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
- [Mad90] P. Maddy. *Realisms in Mathematics*. Oxford University Press, 1990.
- [Mad98] P. Maddy. *Naturalisms in Mathematics*. Oxford University Press, 1998.
- [Mak74] J.A. Makowsky. On some conjectures connected with complete sentences. *Fundamenta Mathematicae*, 81:193–202, 1974.

- [Mak84] J.A. Makowsky. Model theoretic issues in theoretical computer science, part I: Relational databases and abstract data types. In G. Lolli et al., editor, *Logic Colloquium '82*, Studies in Logic, pages 303–343. North Holland, 1984.
- [Mak85a] J.A. Makowsky. Abstract embedding relations. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Perspectives in Mathematical Logic, chapter 20. Springer Verlag, 1985.
- [Mak85b] J.A. Makowsky. Compactness, embeddings and definability. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Perspectives in Mathematical Logic, chapter 18. Springer Verlag, 1985.
- [Mak92] J.A. Makowsky. Model theory and computer science: An appetizer. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter I.6. Oxford University Press, 1992.
- [Mak04] J.A. Makowsky. Algorithmic uses of the Feferman-Vaught theorem. *Annals of Pure and Applied Logic*, 126.1-3:159–213, 2004.
- [Mak06] J.A. Makowsky. From a zoo to a zoology: Descriptive complexity for graph polynomials. In A. Beckmann, U. Berger, B. Löwe, and J.V. Tucker, editors, *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe, CiE 2006, Swansea, UK, July 2006*, volume 3988 of *Lecture Notes in Computer Science*, pages 330–341. Springer, 2006.
- [MM85] J.A. Makowsky and D. Mundici. Abstract equivalence relations. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Perspectives in Mathematical Logic, chapter 19. Springer Verlag, 1985.
- [Mos94] Y. Moschovakis. *Notes on Set Theory*. Springer, 1994.
- [MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems*. Springer, 1995.
- [Pap94] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
- [Ros08] B. Rossman. Homomorphisms preservation theorems. *J.ACM*, 55.3:1–54, 2008.
- [Sch52] H. Scholz. Problem # 1: Ein ungelöstes Problem in der symbolischen Logik. *Journal of Symbolic Logic*, 17:160, 1952.
- [Sch67] L. Schwartz. *Cour d'Analyse*. Hermann, Paris, 1967.
- [Sha00] S. Shapiro. *Thinking about Mathematics*. Oxford University Press, 2000.
- [Sho67] J. Shoenfield. *Mathematical Logic*. Addison-Wesley Series in Logic. Addison-Wesley, 1967.
- [Wil81] R.L. Wilder. *Mathematics as a Cultural System*. Pergamon Press, 1981.