Algorithms and Programs
Erasmus Lecture 2014

Kurt Mehlhorn*

August 14, 2015

Abstract

This article is based on the Erasmus Lecture that I delivered at the 2014 Annual Meeting of the Academia Europaea in Barcelona. I will discuss my early fascination for the field, algorithms, programs, laws of computation, the double role of informatics as a mathematical and engineering discipline, and my effort to teach informatics to non-majors and the general public.

1 Introduction

Information technology (IT) has changed and will continue to change the world in profound ways. Informatics (computer science) is the new science behind this revolution. I assume that the reader is convinced of the usefulness and economic impact of information technology and uses some of its amenities, e.g., computers, smartphones, the Internet, search engines, navigation systems, and electronic banking. However, I also assume that most readers know very little about Informatics as a scientific discipline. The main goal of this article is to contribute to a better understanding of Informatics as a scientific discipline. The article is based on the Erasmus Lecture that I delivered at the 2014 Annual Meeting of the Academia Europaea in Barcelona. I created a companion webpage[1] for this article, which contains additional material.

This article is divided into four parts. First, I will discuss my early fascination for the field: Why did I study informatics? Then, I will discuss algorithms, programs, and laws of computation. Physicists discover laws of nature, informaticians discover laws of computation. Informatics is a mathematical and an engineering discipline. In the third part, I will discuss the interplay between theory and practice. Finally, I will report on my effort to teach informatics to non-majors and the general public.

Information technology has revolutionized the world. IT is much more than Informatics. It also rests on physics, electrical engineering, material science, chemistry, and mathematics. Our daily lives are very different from what they were 20 years ago. We can now communicate whenever and wherever we want. We get off a plane, turn on our phones, and we are reachable again almost instantly. This is amazing. It means that the mobile phone system knows, at any given time, the exact location of every mobile phone in the world. We use search engines to find information. For example, when you type *Kurt Mehlhorn* into Google, it will return after

*Max-Planck-Institut für Informatik, Saarbrücken, Germany

[1] `http://people.mpi-inf.mpg.de/~mehlhorn/ErasmusLecture.html`

less than half a second that it is aware of several hundred thousand documents containing the words *Kurt* and *Mehlhorn* and that it "believes" that my homepage and the Wikipedia entry about me are the most relevant documents for the query. This is amazing. Professions such as type setters have disappeared, and professions such as web designer, have appeared. There are many more examples of how IT has changed the world. In the future, we will see autonomous robots, systems that can make inferences across documents, and machines that can learn. Of course, IT also has its dark sides: IT-security and privacy are two of them.

2 Early Fascination

How did I get interested in Informatics? In a certain sense, I am the oldest German Informatician. Informatics was introduced in Germany as a field of study in 1968, and I started my university studies in 1968. So, I belong to the first group of Informaticians that were trained as such. It has been clear to me since the age of 14 that I would study mathematics. Not that I knew what a mathematician did for a living. The only mathematicians I knew were my high school teachers. So, I started with mathematics and physics. One of my friends informed me that there was also a new subject: Informatics. Since we had spare time, we also took this course. I had no idea what Informatics was about. I had never seen a computer except in pictures.

Figure 1: F.L. Bauer

The lectures in Mathematics and Informatics differed widely from each other. The Math lectures were polished, the level of difficulty was consistent, and it was clear that we were being introduced to the foundations of a huge body of knowledge. Moreover, the lecturer had the material at his fingertips.

F.L. Bauer, one of the founding fathers of German Informatics, taught the Informatics course. It was very different. Some lectures were trivial, some lectures were extremely challenging, some lectures, I am still sure, not even the instructor understood. F.L. Bauer was able to convince his audience that we were entering a new age. He could not say what this world would look like; however, he was absolutely convinced, and he convinced us, that this new age would be full of opportunities and surprises. He convinced us that we had the chance to be the pioneers in this new field.

Figure 2: Al-Khwarizmi, 780 – 850

What was it precisely, that fascinated me? I found the concept of a *formal language* intriguing. Hermann Maurer, the former dean of the Informatics Section of AE, had written an excellent textbook about formal languages, which I studied very carefully. I was intrigued by the fact that one can define languages in a precise way, by the fact that these languages can be processed by machine, and by the fact that one define languages in which one can tell machines what to do. We call such languages *programming languages*.

F.L. Bauer used Algol 68 as the *programming language* in his lectures. As the name reveals, the language was designed in 1968 and the language report [vWMPK69], i.e., the document defining the syntax and the semantics of the languages, was completed in the fall of '68. The syntax defines which sentences belong to the language. The

Algorithm	Sample Execution
write the equation as $x^2 + bx + c = 0$	$x^2 + 8x - 9 = 0$
move the constant term to the other side	$x^2 + 8x = 9$
add $(b/2)^2$ *on both sides*	$x^2 + 8x + 4^2 = 9 + 4^2$
write LHS as $(x + b/2)^2$*, simplify RHS*	$(x + 4)^2 = 25$
if RHS is negative, STOP (no solution)	
remove ^{2} *on LHS, replace RHS by* $\pm\sqrt{RHS}$	$x + 4 = \pm\sqrt{25}$
move constant term from LHS to RHS	$x = -4 \pm \sqrt{25}$

Figure 3: An algorithm for solving quadratic equations. The left column shows the steps of the algorithm. The right column shows a sample execution. LHS abbreviates left-hand side and RHS abbreviates right-hand side.

semantics defines the meaning of well-formed sentences. F.L. Bauer gave us the language report to read in the spring of 1969, at the end of my first semester. The language report is the most complicated book that I have ever tried to read. I tried very hard, but never finished reading it. The experience made clear that entering the new world would be a demanding task.

I was also intrigued by algorithm design and programming. What is an *algorithm*? An algorithm is a step-by-step procedure for solving a certain class of problems. For every instance of the problem, the execution of the algorithm must yield a solution for the instance. There is no thinking required when one executes an algorithm; it is purely mechanical. The name goes back to Al-Khwarizmi, a Persian mathematician, who lived in the 8th and 9th centuries. He wrote a book in which he discusses algorithms for doing calculations with numbers (The Compendious Book on Calculation by Completion and Balancing).

I give two examples to make the concepts *algorithm* and *program* more concrete. You all know programs in the form of cooking recipes. The author/inventor of the recipe is the programmer. The cook is the machine that executes the program. Of course, programmer and machine may be one and the same person.

My second example comes from Al-Khwarizmi's book: an algorithm for solving quadratic equations. See Figure 3. Many readers will have learned this algorithm in high school, probably without being told that it is an algorithm.

Figure 4: The PERM

What is the difference between a program and an algorithm? A program is intended for execution by a computer. It specifies all details and is formulated in a programming language. It is completely clear what every step means, and there is no intelligence needed to execute the program. On the other hand, algorithms are intended for human consumption. My examples above are algorithms, not programs. For example, the formulation of the algorithm in Figure 3 assumes that you understand the concept of left- and right-hand side of an equation and that you know that a term changes signs when it is moved from one side of the equation to the other.

Computers were huge in 1968. Figure 4 shows the PERM, the *programmierbare elektronische Rechenanlage München* (programmable electronic computer Munich), on which I learned to program. It is now at the technical museum in Munich. The technical advancement since

'68 is well-illustrated by the fact that the speed and storage capacity of today's notebooks is about one hundred thousand times that of the PERM. At the same time, the size and cost have also been reduced by orders of magnitude.

I close this part by highlighting three formative insights from my first years of studying informatics.

First, there are algorithms for intellectually demanding tasks. For example, we learned an algorithm for computing shortest paths between any two locations on a map. An instance of this problem would be to find the shortest path from here to my home in Saarbrücken. I had planned routes for vacation trips and found it challenging. Now, I learned that this task can be mechanized and performed by a machine, an eye-opening experience.

The second insight was that computers amplify brainpower. Mankind has always built machines to ease everyday life. All machines existing before computers amplify only muscular power. We use cranes to lift heavy items, and we use cars to go faster than we could with bicycles. Computers amplify brainpower, and it was immediately clear that this would open up a new world.

Finally, designing algorithms and writing programs is an act of creation. It is not just a creative activity, it is more. It creates an object that "lives", interacts with its environment by consuming inputs and producing outputs, and that performs, on its own, computations that the programmer could never do on his/her own.

3 Laws of Computation

Physicists discover laws of nature, informaticians discover laws of computation. A law of nature connects basic physical quantities, e.g., acceleration, mass, and force. A law of computation connects basic computational quantities, e.g., the amount of resources needed to perform a certain computational task. The most important resources are time and space: time measures the number of elementary steps needed for a task and space measures the amount of storage needed for intermediate results. Time and space are different in the sense that time passes and space can be reused.

Der MIPS R10K Prozessorchip.

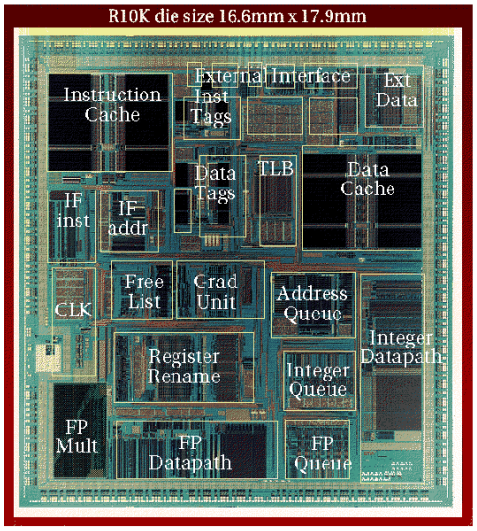


Figure 5: MIPS R10K

I next discuss a particular law of computation in more detail. I have selected it because it is easily explained pictorially and because I had a part in its discovery. Figure 5 shows the MIPS-1000, a vintage 1995 microprocessor. In the bottom left corner, there is a block called FP-Mult. It multiplies numbers; more precisely, it multiplies numbers with 52 binary digits. It occupies about 3 by 4 mm of space and performs a multiplication in about five microseconds.

Is this good?

Should we be impressed by the design of the multiplier? We next perform an abstraction. This is standard when we want to discover fundamental laws. For example, Newton's law of motion ignores friction.

How Hard is it to Multiply Numbers? We study the problem of multiplying numbers with n digits by integrated circuits for large n. Considering only large n allows us to concentrate on the essence of the problem. We measure the complexity of a design by two quantities, namely

$$\begin{aligned} A &= \text{the area of the circuit, and} \\ T &= \text{the execution time of the circuit.} \end{aligned}$$

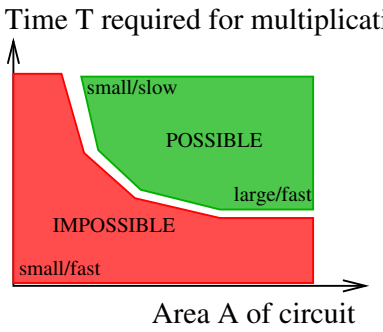


Figure 6: Area-time tradeoff: Every combination of A and T in the red region is impossible, and every combination of A and T in the green region can be realized.

Since multiplication is a fundamental computational problem, there are many different circuit designs for it. There are fast circuits, but all of them are large in area. There are small circuits, but all of them are slow in computation time. *It is natural to ask whether there is a circuit that is small and fast?* The answer is *NO*! There can be no such circuit. Figure 6 illustrates the design space. Each design corresponds to a point in this space. The x-coordinate of the point is the area of the circuit, and the y-coordinate of the point is given by the execution time of the circuit. Designs that are fast and small would sit in the bottom left corner, designs that are large are located on the right, and designs that are slow are located at the top. The red region indicates designs that are impossible, namely, there is a constant c depending only on the technology such that $AT^2 \geq cn^2$ and $T \geq c \log n$ for any circuit that multiplies n-digit numbers. There is another constant C depending only on the technology such that any pair (A, T) with $AT^2 \leq Cn^2$ and $T \geq C \log n$ can be realized. The green region comprises all these designs. Since C is larger than c, there is a white area between the red and green regions, where we do not know, whether designs exist or not.

4 Mathematician in the Morning, Engineer in the Afternoon

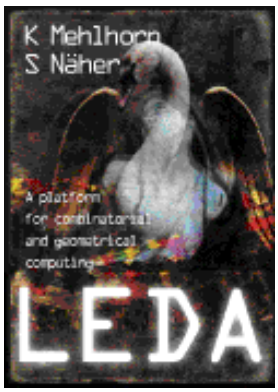


Figure 7: The LEDA book

I like to characterize myself as a mathematician in the morning and an engineer in the afternoon. This reflects the double nature of computer science. It is a mathematical as well as an engineering discipline. As a mathematician, I work on problems of a fundamental nature that are intrinsic to the field, e.g. laws of computation or new algorithms, or on abstracted versions of problems that come up in applications and are too hard for engineers. As an engineer, I turn algorithms in programs, design and build systems, and turn ideas into working and useful systems, a task too mundane for mathematicians. I also identify shortcomings in theories and the need for further theory building.

I next discuss some of my work where this interplay between theory and practice has played an important role, the *Library of Efficient Data Structures and Algorithms (LEDA) project.* I started teaching algorithms and data structures in the mid '70s. I wrote a widely used three-volume book on the subject in the mid 80s. Yet, when I asked

former students whether they had implemented any of the solutions that I taught them, I always got polite answers along the lines of: "Yes, the problems that you discussed do come up. However, my boss has never given me the time to implement any of the complex solutions that you taught us. We always settle for the simplest solution that requires the smallest implementation effort." By '85, I had reached the conclusion that writing books and articles does not suffice. In 1988, Stefan Näher and I started the LEDA project. We wanted to make the knowledge of the field available in the form of easy-to-use, efficient, and correct software modules. Our hope was that in this way, we would make the results of the field accessible to non-experts. Our dream came true. Ten years later, LEDA was in use at thousands of academic and industrial sites. Algorithmic Solutions GmbH, which Stefan and I founded together with Christian Uhrig, marketed the library, and LEDA became a role model for other algorithmic library projects such as CGAL, STXXL, and BALL.

However, the road to success was not smooth. In fact, the project almost ruined my reputation. We had widely announced that the programs in LEDA would be easy-to-use, efficient, and correct. They were easy-to-use and efficient; however, *some of our programs for problems on graphs and most of our programs for geometric computations were incorrect.* What had gone wrong? We had followed the state of the art in program development. However, there was no scientific basis for exact geometric computation at the time. We and others have only created it over the past 20 years. I refer to [KMP$^+$08] and [MN99, Chapter 9] for more information. In the case of our programs for graphs, we had made programming mistakes and turned correct algorithms into incorrect programs. Some of these errors were quite subtle and showed up only rarely so that testing would not reveal them. Also, for some problems, we had only a quite limited number of examples for which we knew the correct output. *What did we do about it*? *By 1995, we had adopted a new design principle, certifying algorithms, and this decision was the turning point. Today, LEDA is a major source of my reputation.*

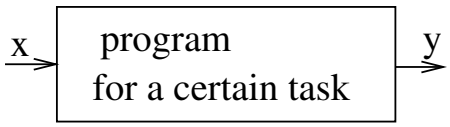


Figure 8: The correctness problem. How can a user be sure that y is the correct result for input x?

Figure 8 illustrates the problem. We have a program that allegedly computes a certain function f. A user feeds an input x to the program, and the program returns y. How can the user be sure that y is indeed the correct output for input x? In general, the user has no way to know. He may know the developer of the program to be reliable person. However, I am generally accepted to be a reliable person, and nevertheless I have put incorrect programs into LEDA. Assume for the moment that you are the boss of a construction company (or any other business) and want to bid for a contract. You ask your team to work out a bid. Some time later, the team comes back with a number. You should charge 5,345,124 euros, and this will guarantee a 10% profit. Of course, you would not accept this number at face value, but you would ask the team to justify its result. This is exactly the principle that we adopted for LEDA.

Design Principle for LEDA: Programs must justify (prove) their answers in a way that can easily be checked by their users.

Figure 9 illustrates the technical realization of this principle. A certifying program for a function f takes an input x and produces the function value y and a witness (certificate, proof) w; w should be convincing evidence that y is indeed equal to the value of f at input x. The witness can be inspected by the user of the program or, more elegantly, by a checker

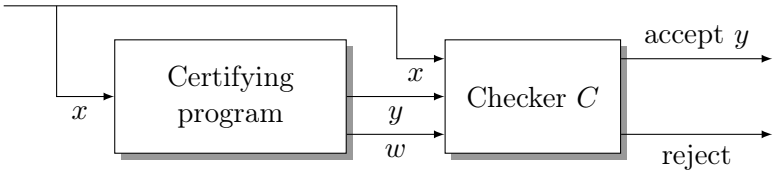


Figure 9: Certifying program and accompanying checker.

program C. The checker program C receives x, y, and w and accepts if w proves the equality $y = f(x)$. Otherwise, it rejects and declares an error.

A certifying program for the planarity test of graphs. A graph consists of a set of nodes and edges connecting them. Think of cities with roads connecting them. A graph is *planar* if it can be drawn in the plane without edge crossings. Figure 10 shows four drawings of graphs. The left-most picture shows a planar drawing; there are four nodes and six edges connecting them. The edges are drawn such that no two of them cross. The second picture shows a nonplanar drawing of a planar graph. In this drawing, the edges that connect opposite corners of the square cross. However, there is an alternative drawing of the same graph, namely the drawing on the left, in which no edges cross. Thus, the graph is planar. The two graphs on the right are the complete graph K_5 on five nodes and the complete bipartite graph $K_{3,3}$ with three nodes on each side. Every pair of nodes is connected in the K_5, and every pair with nodes on different sides is connected in the $K_{3,3}$. The drawings are nonplanar and it is not too hard to show that these graphs cannot be drawn without crossing. In a video accompanying this paper, I convince you that the K_5 is not planar.

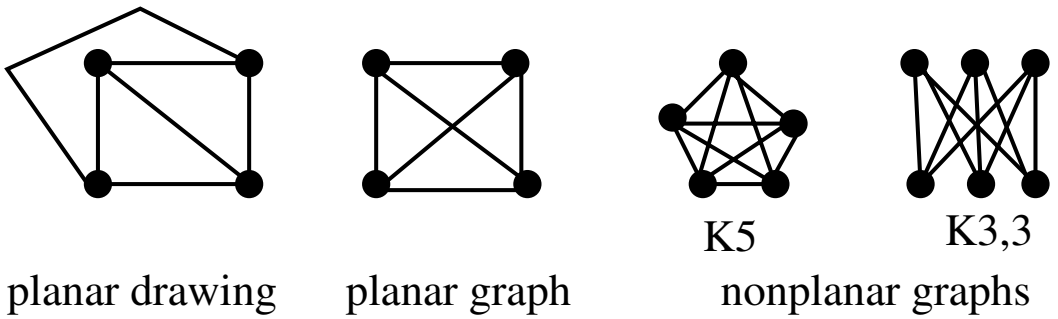


Figure 10: Four graphs.

We added a planarity test to LEDA in 1994. I had asked one of my students to implement an algorithm by Hopcroft and Tarjan. A French mathematician (unfortunately, I do not recall the name) used our program to determine the value of the planarity threshold. It was known at the time that if the ratio of the number of edges and the number of nodes in a random graph is above a certain threshold, the graph is likely to be non planar, and if the ratio is below the threshold, the graph is likely to be planar. It was known that the threshold exists, but its value was not known. Since our program could test very large graphs for planarity, it was natural to use our program to determine the value of the threshold experimentally. This is what was done. Then, somebody proved by mathematical proof that the threshold value

must lie in a certain interval. However, the experimentally determined value did not lie in this interval. Either our program had to be incorrect, or the proof had to be fallacious. The author of the proof was convinced of the former and found a planar graph which our program declared nonplanar. He sent a drawing of the graph to Stefan Näher. Stefan came to my office, told me the story, and concluded with the remark: "This program was written by one of your students". I answered that I would fix the error. I found an error after a day of work and fixed it. The program now worked correctly on all test cases including the example provided by the French mathematician. However, Stefan and I were not satisfied with the situation. Could we be sure that I had removed the last bug? Not really. After long discussions, we concluded that is had not been my student who was to blame for making the mistake, but my specification of his task. One should not ask for a program that gives a Yes/No-answer to a complex question such as "Is a given graph G planar?" One should ask for a program that justifies its answers. What does this mean for the planarity test?

- If the program declares a graph planar, it should also output a planar drawing.
- If the program declares a graph nonplanar, it should also output an obstacle to planarity. Kuratowski, a Polish mathematician, showed in 1910 that every nonplanar graph contains a Kuratowski graph K_5 or $K_{3,3}$.

This is exactly what the planarity test in LEDA does. If the input graph is planar, it returns a planar drawing, and if the input graph is nonplanar, it exhibits a Kuratowski graph. Please have a look at the accompanying video for a demonstration.

History of certifying programs. The concept of a certifying algorithm is by no means new. AL-Khwarizmi (780 – 850) already described casting-out-nines as a method for partially checking multiplications of integers. In the 18th century, Euclid's algorithm for computing the greatest common divisor of two integers was generalized to the extended Euclidean algorithm. This algorithm is certifying. All primal-dual algorithms in combinatorial optimization are certifying. Manuel Blum and co-workers [BK95, Blu93] wrote a sequence of papers in the '90s about programs that check their work.

Stefan Näher and I were the first to adopt the concept as the design principle for a software project. We adopted the principle for LEDA in 1995, and by early 2000, we had made almost all of the algorithms in the library certifying. We refer the reader to the LEDA-book [MN99] and the survey [MMNS11] for details.

Who checks the checker? The output of a certifying program is inspected by a checker program C. It receives x, y, and w and accepts if w proves that y is equal to $f(x)$. Otherwise, it rejects and declares the triple as faulty. *How can one be sure that the checker program is correct*? Have we not replaced the problem of getting the original program correct by the equally hard problem of getting the checker program correct? No, because checkers are simple and short programs, and hence, getting them correct is a lot easier. Note that this answer is not completely satisfactory as it only says that checkers are simpler and therefore the hope of getting them correct is larger. Since about a year, we can give a much more satisfactory answer: *We can prove the correctness of checker programs using formal mathematics.*

What is formal mathematics? It is mathematics carried out in a formal language without any ambiguities. Figure 11 shows an example. Because the syntax and the semantics of the language are well-defined, proofs and arguments can be machine-checked. We use the system

```
definition disjoint-edges :: (α, β) pre-graph ⇒ β ⇒ β ⇒ bool where
  disjoint-edges G e1 e2 = (
      start G e1 ≠ start G e2 ∧ start G e1 ≠ target G e2 ∧
      target G e1 ≠ start G e2 ∧ target G e1 ≠ target G e2)

definition matching :: (α, β) pre-graph ⇒ β set ⇒ bool where
  matching G M = (
      M ⊆ edges G ∧
      (∀e1 ∈ M. ∀e2 ∈ M. e1 ≠ e2 ⟶ disjoint-edges G e1 e2))

definition edge-as-set :: β ⇒ α set where
  edge-as-set e ≡ {tail G e, head G e}

lemma matching_disjointness:
  assumes matching G M
  assumes e1 ∈ M   assumes e2 ∈ M  assumes e1 ≠ e2
  shows  edge-as-set e1 ∩ edge-as-set e2 = {}
  using assms
  by (auto simp add: edge-as-set_def disjoint-edges_def matching_def)
```

Figure 11: Formal mathematics in Isabelle/HOL

Isabelle/HOL developed by L. Paulson and T. Nipkow [NPW02]. We [ABMR14, NRM14] formally verify

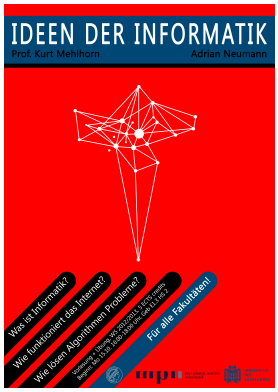
- the *witness property*, i.e., the mathematical theorem underlying certification, e.g., the fact that the existence of a Kuratowski subgraph implies nonplanarity,
- *the termination of the checker program*, i.e., the checker program halts for every input after a finite time and gives either the answer "accept" or "reject", and
- the *correctness of the checker program*, i.e., the checker accepts if and only if w certifies the correctness of y as an output for x in the sense of the first item.

What does this give us? We have now reached *formal instance correctness: If a formally verified checker accepts a triple (x, y, w), we have a formal proof that y is the correct output for input x.* The formal verification of checkers has lifted the trust in our implementations to a new level. It is a way to build large libraries of trusted algorithms.

5 Ideas and Concepts of Informatics

Since 2012, I have been teaching a course called *Ideen und Konzepte der Informatik* for non-majors (Studium Generale), first together with Kosta Panagiotou and now with Adrian Neumann. We pursue three goals.

First, an understanding of fundamental concepts of informatics: What is an algorithm? What is a computer? Are all computers equal? Are there tasks that cannot be solved by a machine?



Second, the familiarization with basic techniques of informatics, such as searching and sorting, data bases, the Internet, cryptography, web search, machine learning, and computer vision, and also their applications, such as search engines, navigation systems, electronic banking, electronic mail, automatic face recognition, and social networks.

Finally, we hope that our students acquire sufficient knowledge in informatics so that they can discuss societal and ethical consequences of information technology on a sound basis. We discuss some of these issues: IT-security, threats to privacy, the disappearance of professions and the appearance of new ones, and the liability of autonomous systems.

We also discuss how Informatics has changed the scientific view of the world. What is intelligence? Will social networks become the experimental playground for the social sciences? What does big data mean for the sciences? The reader can find more information about the course at `http://resources.mpi-inf.mpg.de/departments/d1/teaching/ws14/Ideen-der-Informatik/`. In the fall term 2015–2016, I will offer the course as a massive open online course (MOOC).

References

[ABMR14] E. Alkassar, S. Böhme, K. Mehlhorn, and Ch. Rizkallah. A Framework for the Verification of Certifying Computations. *Journal of Automated Reasoning (JAR)*, 52(3):241–273, 2014. A preliminary version appeared under the title "Verification of Certifying Computations" in CAV 2011, LCNS Vol 6806, pages 67 – 82.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995. preliminary version in STOC'89.

[Blu93] Manuel Blum. Program result checking: A new approach to making programs more reliable. In *ICALP*, pages 1–14, 1993.

[KMP+08] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom Examples of Robustness Problems in Geometric Computations. *Computational Geometry: Theory and Applications (CGTA)*, 40:61–78, 2008. a preliminary version appeared in ESA 2004, LNCS 3221, pages 702 – 713.

[MMNS11] R.M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of Lecture Notes in Computer Science. Springer, 2002.

[NRM14] Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. Verification of certifying computations through autocorres and simpl. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 46–61. Springer International Publishing, 2014.

[vWMPK69] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, and C.H.A. Koster. Report on the algorithmic language 68. *Numerische Mathematik*, 14:79–218, 1969.