

Constructivism.
A Computing Science perspective *

Bengt Nordström

Department of Computing Science

Chalmers University of Technology and the University of Göteborg

S-41296 Göteborg, Sweden

1. I would like to talk about the symbiotic relationship between Computing Science and Logic and the growing interest in foundational questions which has followed.
2. It is not an exaggeration to say that Logic gave birth to Computing Science. Already in the 1930's, logicians like Gödel, Kleene, Church and Turing were studying the limits of computability. This was at a time when a computer still was a human being performing calculations. In particular Turing's theoretical ideas had important consequences for the architecture of the first electronic computers in the 1940's.
3. Now, in the turn of the new millennium, computers are influencing every day's life in the rich world. Society becomes more and more dependent on computers. We use computers to control complicated processes, like power plants, air-planes, trains and money transfer systems. These systems are so complicated that no human being understands them and it is of vital importance that they function correctly.

*Notes as of December 28, 2001 for the meeting on Foundations and the Ontological Quest in Pontifical Lateran University, Vatican City, January 2002.

During the last 10 years we have seen a number of failures in these kind of systems. I just mention four well-known examples.

The first example is the Ariadne rocket, a common European space project. The rocket exploded a few seconds after takeoff, due to a software error.

Another well known example is the baggage-handling system of the Denver International Airport. Errors in the software that controls the system required postponement of the official opening (Oct. 1993). By June 1994 the \$ 193 million system was still not functioning, but costing \$ 1.1 million per day in interest and other costs. In early 1995 a manual baggage system was installed in order to open the airport.

Some cancer patients in the USA have received fatal radiation overdoses from the Therac-25, a computer-controlled radiation-therapy machine.

The last example is the Sizewell B nuclear power plant in England. Some years ago it was decided to test the subsystem which is used to close down the reactor if a dangerous situation occurs. The results were not comforting: the software failed almost half of them. They were not able to find the errors in the 100 000 lines of code. Instead, they reduced the overall expectation of the plant's performance from one failure every 10,000 years to one every 1,000 years.

4. I don't think that these examples are exceptional cases. Most computer users have frustrating experiences from error-prone computer programs.
5. The risk of trusting a tool from empirical evidence.

When it comes to tools which are easy to understand or tools which are produced according to high engineering standards it is common to trust the tool. The trust in the tools comes from empirical evidences and from the way the tool is designed. In contrast with other tools, computer programs are very easy to produce even if the final product is very complicated. To write a program is not much more difficult than to

write a piece of text using a word processor. And as we know, the final product can be very complicated. But when it comes to other tools, a complicated tool needs a complicated production. The mere effort in producing a complicated tool serves as an evidence of the correct functioning. And therefore, the trust in the tool comes only from the evidences of correct functioning. This phenomena has been taken over in the trust of computer artifacts. When we have used a program for some time without errors, we start to trust it. But then we have forgotten that it is easy to write a program which works sometimes.

This combination of simplicity of producing and complexity of the final product is unique for computer programs. The move from a few empirical tests to a trust in the tool is a step which we are used to take but we must learn not to make it.

6. The simplicity of programming is of course only superficial, it is easy to write a program, but it is difficult to write a program which fulfills its task. The current practice of making sure that programs are correct is to test them. And this is of course not satisfactory. We have to prove that they are correct for all inputs, not just for the test data. That a program should be written together with its proof of correctness is an old idea but it is not reflected in common practice.
7. We have to be able to prove properties of programs. We can distinguish between different levels of proof when it comes to programming. The least satisfactory (but very common) is that the proof of correctness only remains in the head of the programmer, it is never written down. The next level is that the program comes together with a piece of documentation, containing for instance an informal specification of the program and an argument why it is correct. This argument can then be read by other people and faulty steps of reasoning can be discovered. A natural idea is to see if we can use the computer to check a proof of a program. It is possible to give the proof in such detail that all steps

can be mechanically checked. You have to have a completely formal logic in which you can express a proof and where you can express the properties the program should have. That the logic is formal means exactly that a computer can check if a proof is a proof or not. There is a long tradition of formal proofs; it was more than 100 years ago that modern Logic started with Frege and Peano. The next step of development is of course to let the computer construct the proof, this is possible and already used for logically simple but combinatorically complicated problems (like hardware verification).

8. But Computing Science not only benefit and demands a development of Logic, it can also contribute to Logic in an essential way.
9. A formal system like a programming language or a logic can be studied in three different ways. We can reason about it theoretically (mathematically), we can implement (make) it and make practical experiments with it. This is the same as with other human constructions, like houses, cars and tools.

When we study a logic or some other formal system theoretically we look at it as a *mathematical object* and use the language of mathematics to prove properties of it (like normalization, subject reduction etc), this is the metamathematical view. This has also been the traditional and only way to study logic. But computers give us new ways. We can *implement* formal systems. For a programming language, this means to write an interpreter or compiler for it, so that we can execute programs written in it. For a logic this means either to write a proof checker for it, so that we can type formal proofs and let the computer check the correctness of the proofs, or write a theorem prover so that the computer can search for proofs of theorems. Given an implementation, we can *experiment* with the formal system. We write programs or proofs and use the computer to run them. Mathematical logicians have always been mainly interested in the metamathematical aspect of the

formal systems. This means that Logic has mainly been a theoretical study. This is natural since it is not until the last decades that it has been possible to implement and experiment with logical systems. Practical aspects of logics have been neglected. A metamathematical study of a logical language necessarily reduces this to a mathematical object, the language aspect of a logic is forgotten.

10. As the need for useful logics will increase, there will be an increased development of practical aspects of logics by implementing logics and making experiments with them. Computing Science applications will necessitate a growing interest in the semantics of logical systems. And by semantics, I do not mean mathematical semantics, but the real semantics of propositions and proofs.

11. Against formalism.

We cannot have a formalistic attitude when we reason about programs. By formalism I mean that we look at reasoning as only following rules (like playing chess). Properties must have a meaning. When we say for instance that a certain program always closes a certain valve if the temperature is too high then we must be certain that the program really closes the valve. If we for instance prove that the program has the property $A \& B$ then we must be certain that the program has the property A and the property B . When we have proven that a program has the property (expressed in some formal language) that for all inputs there exists an output such that a certain property holds, then this cannot just be a formal derivation of a certain formula with no meaning. When the derivation has been done, then we must also be sure that the program has the required property. An escape to formalism may save the mathematician in his worries about semantics, but it is not an option for programmers and computer scientists who are going to apply logical reasoning to concrete objects. This does not contradict the fact that a completely formalistic view can be fruitful

for mathematical studies of logical system. Then we view the system as rules and look at the mathematical properties of the formal system. These properties can of course be useful for the applicability of the system, but when we use the logic we will return to the state when we look at logic not just as a formal language.

12. Programs are concrete mathematical objects.

Programs are concrete objects like building, chairs and tables. A programmer constructs a program in the same spirit as a carpenter makes a table. The object which is constructed has certain properties and it is an important part of the construction process to make sure that these properties hold. But programs are not only concrete objects, they are also mathematical objects. Most programming languages are extremely complicated but the development of functional languages during the last 20 years is an important step towards diminishing the difference between a mathematical language and a programming language. These languages are of course rather rudimentary viewed as languages to express mathematics in. It is only possible to express mathematical objects in inductively defined sets. It is not possible to express properties about these objects. However, this can be done in programming languages based on different kinds of type theories based on constructive mathematics.

13. Constructive mathematics fits well with programming.

During the last 20 years I have been interested in versions of Martin-Löf's type theory as a basis for programming. Martin-Löf has developed his theory as a foundational language for mathematics. It is based on constructive mathematics. This language seems to fit extremely well with programming. In it, a proof of a proposition A is a *program* which when executed yields a direct proof of the proposition. The fundamental idea is that a proposition is explained by explaining what counts as a direct proof of it. When we define a proposition we give an inductive

definition of its direct proofs. We see that the notion of program is used to explain the notion of proposition in mathematics. The notion of set is explained in exactly the same way as a proposition (we only say element instead of proof and canonical element or value instead of direct proof). Therefore, the two notions of being a set and being a proposition are identified.

14. As an example, the proposition $A \vee B$ is explained by giving an inductive definition of its direct proofs, a direct proof of $A \vee B$ is either on the form $\text{inl } a$ where a is a proof of A or on the form $\text{inr } b$ where b is a proof of B . This is completely analogous to the inductive definition of the set of natural numbers: a value in \mathbf{N} is either on the form 0 or on the form $\text{succ } n$, where n is an element in \mathbf{N} . From the fact that this is an inductive definition we may conclude that if we have an element in \mathbf{N} we can compute it to a canonical form, i.e. either to the form 0 or the form $\text{succ } n$, for some $n \in \mathbf{N}$. Similarly, if we have a proof of a proposition $A \vee B$, we can compute it to a direct proof of it, either to the form $\text{inl } a$ where a is a proof of A or to the form $\text{inr } b$ where b is a proof of B . So, from a proof of $A \vee B$ we can see which one of the propositions A and B are true. And of course, this breaks with classical mathematics, where we can prove a disjunction without having any idea about which one of the disjuncts that are true.

We can look at a proposition A not only as a set, but also as a specification of a program. And a proof p of a proposition A can be seen as a program which satisfies the specification A . To compute the proof p to a direct proof p' is then the same as computing a program to its value. This view of programs and proofs has mainly had theoretical consequences so far. One exception is the idea of proof carrying code, where an untrusted program is transported over the Internet together with its proof of correctness.

15. What are the requirements on a proof-checking system?

A proof editor is a computer tool to interactively build proofs. The editor must be transparent in the sense that the different aspects of the proving process must be supported. Examples of these steps are making an assumption, using an assumption, to apply a rule, to introduce an abbreviation, a lemma, a theorem, a theory etc. But also more practical aspects like being able to regret certain proof steps, to manipulate a data base of theorems, sharing proofs over the internet etc.

A computerized proof system must of course be correct. First it must be clear what logic is implemented. This fundamental requirement is not fulfilled by the most commonly used proof systems of today. Secondly the logic itself must be correct. This means that a propositions in the logic must have a meaning and a proof of a proposition guarantees that the proposition is true. Thirdly the logic must be correctly implemented in the sense that the program which checks the steps of the proof must reject erroneous steps. There is also a possibility of having an independent proof checker which checks the correctness of a complete proof. This task is much simpler than checking the steps in an interactive proof.

16. It is a common misunderstanding that it is computationally expensive to check formal proofs. I think that the misunderstanding comes from the semantics of the judgment $p \in P$, that p is a proof of P means that the value of p is a direct proof of P . This could lead one to believe that to check whether p is a proof of P the computer first computes p and checks if the result is a direct proof of P . This is not necessary.

Let us take a simple example. We want to check a proof p of the proposition $2^{1000} + 1 = 1 + 2^{1000}$. We know that the meaning of the proposition is that the values of $2^{1000} + 1$ and $1 + 2^{1000}$ are equal (i.e. that they are either both 0 or the successor of two equal numbers). So we could expect that a computer checks the proof by first computing the values of $2^{1000} + 1$ and $1 + 2^{1000}$. But this is not what happens!

If the proof is using the commutativity of addition specialized to the two numbers 2^{1000} and 1, where commutativity has been proven using mathematical induction, then there is no need to compute the numbers, it is only necessary to check that the lemmas and rules are correctly applied. The lemma for commutativity of addition is valid for all natural numbers, and it is correctly applied if it is instantiated to natural numbers. Similarly, the law of mathematical induction is correctly applied if the premises of the law are fulfilled, i.e. that the first argument is a natural number, the second a proof of a proposition $C(0)$ and the third a proof of the induction step $(x \in \mathbf{N})(C(x))C(\text{succ}(x))$. In general, we only need to check that we have the correct type of argument.

17. The dominant role of Analysis in Mathematics is going to be taken over by Logic. In the same way as the advances of Physics during the last centuries were a driving force to develop Analysis I think that the growing importance of complicated computer programs will force a development of Logic. This does not only mean that Logic will be studied more than today, but there will be a shift of interest from syntax to semantics and from theory to practice. The new possibilities of making computerised proof systems will make Logic into an experimental science, thus allowing for practical aspects to play a role. This will make foundational issues more important.

References

- [1] Coq homepage. <http://pauillac.inria.fr/coq/>, 1999.
- [2] T. Hallgren. Home Page of the Proof Editor Alfa. www.cs.chalmers.se/~hallgren/Alfa/, 1996-2000.
- [3] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, 1992. Distributed by the University of Chicago Press.

- [4] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
- [5] G. Necula. Proof-carrying code. In *Principles of Programming Languages '97*, January 1997.
- [6] R. Pollack. The lego proof assistant. www.dcs.ed.ac.uk/home/lego/, 1997.