Logic Activities in Europe[*]

Yuri Gurevich[†]

1 Introduction

During Fall 1992, thanks to ONR, I had an opportunity to visit a fair number of West European centers of logic research. I tried to learn more about logic investigations and applications in Europe with the hope that my experience may be useful to American researchers. This report is concerned only with logic activities related to computer science, and Europe here means usually Western Europe (one can learn only so much in one semester).

The idea of such a visit may seem ridiculous to some. The modern world is quickly growing into a global village. There is plenty of communication between Europe and the US. Many European researchers visit the US, and many American researchers visit Europe. Neither Americans nor Europeans make secret of their logic research. Quite the opposite is true. They advertise their research. From ESPRIT reports, the Bulletin of European Association for Theoretical Computer Science, the Newsletter of European Association for Computer Science Logics, publications of European Foundation for Logic, Language and Information, publications of particular European universities, etc., one can get a good idea of what is going on in Europe and who is doing what. Some European colleagues asked me jokingly if I was on a reconnaissance mission. Well, sometimes a cow wants to suckle more than the calf wants to suck (a Hebrew proverb).

It is amazing, however, how different computer science is, especially theoretical computer science, in Europe and the US. American theoretical computer science centers heavily around complexity theory. The two prime American theoretical conferences — ACM Symposium on Theory of Computing (STOC) and IEEE Foundation of Computer Science Conference (FOCS) — are largely devoted to complexity theory (in a wider sense of the term). That does not exclude logic. As a matter of fact, important logic results have been published in those conferences. However, STOC and FOCS logic papers belong, as a rule, to branches of logic intimately related to complexity. Finite model theory is a good example of that; see [Fagin 1990, Gurevich 1988] and especially [Immerman 1989]. The difference between theoretical computer science and the rest of computer science (and computer engineering) is relatively well delineated in this country.

In Europe, the line between theoretical computer science and the rest of computer science and engineering is much more blurred, partially because computer science and engineering in general are more theoretical. A much greater proportion of European research goes into programming language theory, semantics, specification languages, proof systems, verification methods, etc. For brevity and the lack of a better term, I will use the term "formal methods" for all of these areas. Europeans put much more faith and efforts into

[*]ACM SIGACT NEWS 25, No. 2, June 1994, 11–24.

[†]Supported by ONR grant N00014-91-J-1861. EECS Dept., University of Michigan, Ann Arbor, MI 48109-2122, gurevich@umich.edu

foundational investigation of software and hardware technology and into developing formal methods for use in software and hardware.

Of course, things are not black and white. For example, the American conference on Principles of Programming Languages is very theoretical (and European influence can be easily discerned there), and there are excellent complexity theorists in Europe. Nevertheless the gap is huge and, to a great extent, it is the gap between the complexity and formal methods communities. The first is centered heavily in the US; there is a fair number of reputable complexity theorists in Germany, but your fingers may suffice to count them in France or Great Britain. The second spreads more evenly but plays a much greater role in Europe. Only a few people work both in complexity and formal methods. An interesting example of the European version of the split between the two communities is the Computer Science Department of Edinburgh University in Scotland. They have two disjoint seminars. One is their Theory Seminar. It is huge (as theory seminars go) and devoted to semantics and formal methods in general; it is as good as any formal-methods seminar in the world. The other one is devoted to complexity theory, also of excellent quality, but tiny. (I spoke in both seminars and was told that such incidents are rare indeed.)

Too often complexity theorists are not interested at all in semantics and (what we call here) formal methods, and too often experts in formal methods are not interested in and do not know modern complexity theory. Furthermore, the formal methods field itself splits into a number of areas speaking very different languages. Yes, there are numerous visits of Americans to Europe and Europeans to the US, but all too often complexity theorists go to speak to complexity theorists and formal methods experts go to speak to formal methods experts in their area even if they all go to the same conference. One meets American researchers dismissing European computer science as largely irrelevant, pure logic. And one militant French computer scientist told me that complexity theory is irrelevant pure mathematics. He is not typical and the phrase was thrown during a heated discussion, but the sentiment of that phrase is not completely foreign to many in Europe.

I had to decide whether to concentrate on a few carefully chosen areas or to try to cover European logic activities in general. The latter task seemed frighteningly hard. Logic permeates all areas of computer science in Europe, and European logic activities are so intensive and diverse that it would take a large committee and longer time to survey them properly. On the other hand, particular fields are surveyed from time to time by experts who know the researchers and the papers and can write surveys without the opportunities of special learning tours, whereas I didn't know of any general study of European logic activities. The opportunity of a learning tour seemed to suggest the more ambitious general study. After a period of hesitation, I decided to do that, and I spent much of the European tour learning things I did not know and talking with people with whom I had had little or no interaction before.

In this report, I am not going to catalog who is doing what in Europe. It makes no sense for one person to attempt to compile a comprehensive survey of European logic activities in few months. As I mentioned earlier, it would take more people and more time to write a comprehensive survey. Moreover, such a survey has been published recently: [van Leewen 1990, volumes A and B]. But, to illustrate how much is going on in Europe, British logic activities are sketched in Section 2. Why British? Well, my division of computer science into European and American is not strictly geographic. I am trying to discern typical and distinguishing European tendencies, and Great Britain is very European in that sense. If you want to know how it feels to bring coals to Newcastle, give a formal methods talk in UK. That is how I felt in March 1992 at the 8th British Colloquium for Theoretical Computer

Science in ... Newcastle upon Tyne, England. It is somewhat surprising that theoretical computer science is so different in the UK and the US; after all, we share roughly the same language. But theoretical computer science is not the only field where British hold their own; British TV is another such field. Another reason for concentrating on UK is that Wilfrid Hodges volunteered to write that sketch for me. In fact, I spent more time in France with its own rich logic culture.

Section 3 is devoted to positive sides of the European way of doing things; there are things for us to learn. Section 4 is devoted to other comments.

The learning tour in Fall 1992 affected me a great deal even though I had visited Europe many times before and was familiar with European computer science and many prominent European computer scientists. The tour and the report have been hard work, but I enjoyed it. The tour gave me a wider perspective. I was especially impressed by European boldness in promoting formal methods. That boldness is contagious. In spite of the criticism that I have for European ways of doing things, the tour gave me more confidence that a logician may be of great help to software engineers and even himself/herself may be a successful software engineer.

Acknowledgments. It is a pleasure to thank ONR for the support and for trusting my judgment and not micromanaging the tour; to thank INRIA Roquencourt for the excellent condition for my work in Paris; to thank my European colleagues for invitations, explanations and for being so generous with their time; to thank Wilfrid Hodges for the survey of British logic activities; to thank my colleagues Krzysztof Apt, Andreas Blass, Egon Börger, Thierry Coquand, Gordon Plotkon, Dean Rosenzweig, Victor Vianu and Moshe Vardi for comments on earlier versions of the report. I shamelessly used some of the comments.

2 Logic activities in Great Britain

By Wilfrid Hodges, Dean, Queen Mary College, London, England, especially for this report.

2.1 Organisation

There is a huge amount of activity in Britain on the frontier between logic and computer science. Most of the research and development is still taking place in the universities and polytechnics, and a few major centres (Cambridge, Edinburgh, Imperial College London, Manchester, Oxford) account for a high proportion of it. But there are an increasing number of good logicians employed in public or private industry (British Telecom, ICL, Hewlett-Packard) or software houses (Adelard, Logica, Praxis). This makes for fragmentation.

Also the field develops very quickly. A typical example is the specification language Z, based on set theory, which was developed at Oxford in the early 1980s. By early 1991 the subject had already accumulated 200 published papers and an e-mail forum with 150 subscribers, and undergraduate courses on it were being taught at 33 British universities and polytechnics. Industrial pressure certainly helped this along.

There have been no serious attempts to bring together the various researchers in logic and computer science; probably it is too late now. A major journal, the Journal of Logic and Computation (Oxford University Press), was founded in 1990. But it is an international effort with editors in Texas, Paris and Kyoto. Another recent journal in the same area, Formal Aspects of Computing, was founded by a specialist group of the British Computer Society in 1989; it is also international and restricts itself to formal methods.

In fact most of the unifying ventures that involve Britain are international. Probably the broadest in its aims is the European Association for Computer Science Logic, founded in 1992 on the initiative of Egon Börger. The Secretary is Karl Meinke from Swansea, and British members of the Scientific Council are Abramsky and Hodges, both in London; its 1993 meeting is at Swansea. Another umbrella organisation is the European Foundation for Logic, Language and Information (FoLLI), led by Johan van Benthem in Amsterdam, which has logic and computer science components but is slanted towards linguistics; it held its 1992 summer school at the University of Essex. About seven years ago the Centre for Theoretical Computer Science at Leeds (led by Tucker and Wainer) instituted annual conferences in Theoretical Computer Science; these are largely a forum for British logicians and mathematicians who have moved into computer science.

There are also a number of important British-European collaborations in more specific areas, with EEC funding. Thus there are ESPRIT projects on Categorical Logic in Computer Science and on Types and Proofs, and a Human Capital and Mobility project on Lambda Calculus, and so on.

Probably the one activity in Britain that did bring together people from the whole field was the Logic for Information Technology Initiative (LogFIT) of the Science and Engineering Research Council. For several years this very valuable initiative funded instructional conferences in aspects of mathematics - mainly logic - which are needed for computer science. It also funded research students and paid for retraining of mathematicians. In 1991 the Initiative started to run into financial problems, and a closing conference was held in early 1993. As the number of Computer Science graduates and PhDs increases, it becomes harder to persuade an engineering committee that money is needed for what are largely conversion activities. Nevertheless several leading British computer scientists (for example Abramsky) are publicising the need for a mathematical logic input into theoretical computer science.

It may be worth mentioning that although a number of philosophical and mathematical logicians in Britain have converted to computer science, the pressure to do this has been noticeably less in Britain than in some European countries (for example Germany).

2.2 Particular areas

One area on the logic/CS frontier where Britain has had a distinctive voice for some two decades is the theory of concurrency. I put this first because of its importance, though I am not sure how close its ties are with logic. Certainly work in this area does sometimes invoke logical methods. Robin Milner in Edinburgh recently won the Turing award for his work on parallelism and concurrency; his formal system CCS and Tony Hoare's comparable system CSP are still paradigms for much of the research being done. Britain has a number of people working in this field, or in the related Dutch paradigm of process algebras. They include Hennessy at Sussex, Stirling in Edinburgh, Kwiatkowska and Ambler at Leicester. A London-Cambridge study group around Abramsky works on game-theoretic approaches to concurrency; their field takes in the semantics of linear logic too.

Concurrency is about the interaction of systems in time. Several logicians at Imperial College London (Gabbay, Hodkinson, Richards) study temporal logic directly, its axiomatisations and expressive power. Maibaum at Imperial College and Barringer in Manchester apply temporal logic to formal specification.

Turning to specification for a moment, Goguen in Oxford continues to work on the fundamentals, for example the theory of modularisation. Britain is strong in set-theoretic specification (Cliff Jones in Manchester with VDM, Hayes and Spivey at Oxford with Z, Schuman and Pitt at Surrey). Current research includes comparisons between these languages, and development of a better semantics for Z. Tucker and Meinke at Swansea work on several aspects of specification, for example higher-order datatypes and the logic of hardware specification.

One can find various other logical languages being developed for special purposes, for example causal logics for the control of robots (e.g. Bell at QMW, London) or logics of belief for use in expert systems (e.g. Jeff Paris in Manchester). But in Britain it is widely felt that computer science needs broad settings in which various logical systems can be represented and compared. Thus a number of people are using the Edinburgh Logical Framework LF (due to Harper, Honsell and Plotkin) as a general setting for defining logics. Aczel at Manchester asks what logical notions a general framework might usefully include, and has offered a number of suggestions including some novel forms of set theory. Gabbay's 'labelled deductive systems' are another move towards generality.

Turning to constructive logics, Hindley at Swansea is a leading authority on lambda calculus. Ray Turner in Essex is one of several people working on constructive logics as a foundation for functional programming. In Cambridge, Pitts works on categorical logic, while Hyland is involved in several category-theoretical aspects of formal methods.

Links between recursion theory and computer science are not very active in Britain at the moment. But one should mention the work on termination by Ursula Martin and her term-rewriting group (recently moved to St Andrews); work by Wainer in Leeds on translating proofs into algorithms; and work of Simmons in Manchester on the classification of primitive recursive functions. Iain Stewart (recently moved to Swansea) is active in complexity theory and related areas of finite model theory.

During the 1980s there was a good deal of activity in Britain, largely among philosophers, in computerised logic teaching. For example my textbook [Hodges 1977] was turned into programs in Manchester, Leicester and Oxford. This activity has largely died out, I suspect because the philosophers realised they couldn't compete with American high technology. One of the main journals of the field, the Computerised Logic Teaching Bulletin, was run by Read at St Andrews, but closed down about two years ago. Some interest continues but in different forms. For example a cognitive science group in Edinburgh is conducting experiments to find out the effectiveness of the semantic approach to teaching elementary logic found in the computer packages 'Tarski's world' and 'Hyperproof' (which were produced at Stanford by Barwise and Etchemendy).

At the same time there are a range of activities in computerised proof systems. An important example is Gordon's HOL, a computerised proof system for higher order logic. Paulson's theorem prover Isabelle, developed in Cambridge, is 'generic' in the sense that it allows the user to specify a proof system; thus recent work in Manchester studied how to implement the theory of constructions and several type theories in Isabelle. Another generic theorem prover is currently being designed by Bornat and Sufrin (QMW and Oxford) for teaching purposes.

Bundy and his group in Edinburgh continue to work on the artificial intelligence project of making computers carry out logical and mathematical reasoning. Kowalski at Imperial College is working on aspects of the formalisation of reasoning.

In logic programming the most striking recent development is the creation of a new logic programming language, Gödel, by Hill in Leeds and Lloyd in Bristol. It has about the same expressive power as Prolog but seems to be much cleaner from the point of view of logic. Since Strand became available commercially in 1989, activity in developing new parallel versions of Prolog seems to have quietened down. Pym in Birmingham and Wallen in Oxford continue to study theoretical aspects of proof search for various calculi of computational interest. Dyckhoff in St Andrews studies intuitionist logic from a computational point of view.

3 Positive sides of European experience

In this section, I list some positive sides of European logic activities. Hopefully, we can benefit from European experience. Describing any of the positive sides, I do not intend to imply that the American situation is the direct opposite of the European one.

3.1 Realization of foundational crisis

"What crisis?" may say the optimist. "We are in the middle of a successful computer revolution and will grow out of whatever seems like a crisis to you."

Well publicized software failures indicate that there is a problem, however. A recent article in Scientific American gives some examples [Littlewood and Strigini 1992]. Software bugs caused the series of large-scale outages of telephone service in the US. A software problem may have prevented the Patriot missile system from tracking the Iraqi Scud missile that killed 28 American soldiers during the Gulf War. A single incorrect character in the specification of a control program for an Atlas rocket, carrying the first US interplanetary spacecraft, Mariner 1, ultimately caused the vehicle to veer off course. Both rocket and spacecraft had to be destroyed shortly after launch. After a long and interesting discussion of inherent limits on testing, the authors of the Scientific American article arrive at the conclusion that "an inherent uncertainty in reliability may mean limiting a computer's role, especially in systems where software is critical for safety." I do not buy that pessimistic conclusion but it is certainly not obvious that we will simply grow out of the reliability problem.

The reliability problem is not overlooked in this country. See in this connection the regular column "Risks to the Public in Computers and Related Systems," edited by Peter Neimann in Communications of ACM. (Association for Computing Machinery is largely an American institution.) The reliability problem raises foundational questions in complexity theory and formal methods. Complexity questions related to testing are addressed by the American theoretical-computer-science community seriously. Two developments come to mind. One is the progress achieved recently by Manuel Blum of Berkeley and his students; see [Blum and Kannan 1989] for example. It turns out that, in many cases, one can exploit the structure of the given problem and, using some mathematics, reduce relevant instances to random instances. It is usually much easier to test random instances. The second development may be relevant to the nasty problem of one wrong bit messing up the whole program. According to recent advances in complexity theory, a formal proof can be rewritten in such a way that it becomes a little longer but every error is spread all over the place [Babai et al. 1991]. A similar approach may possibly be useful in safety-critical software.

However the testing is necessarily one-sided. It may catch errors, but it does not show the absence of errors directly. That is where verification comes in. What can be verified? What are inherent limitations on verification? Whereas testing belongs to a greater extent to the complexity-theoretic half of theoretical computer science, the verification problem belongs to a greater extent to formal methods.

Verification is only one of the foundational problems related to formal methods. One other problem is that of specification. Specifications are usually written in English in this country and often are interpreted differently by different people, e.g., by software developers and their clients, or by software developers and their marketing people. All too often the precise task of the product is not clear. Moreover, it may be not clear how to make that task clear.

Still another though related foundational problem is that of semantics. Even programming languages with their precise syntax may be open to interpretation. Sometimes, people are surprised to learn that the same program (say in C) may give different results being run in different places (using different compilers). As a matter of fact, different compilers usually give somewhat different interpretations to the same language. Sometimes the intended meaning of a phrase in the language is not clear at all.

One example of the current lack of proper foundation is the modern Babel tower of programming languages. Why these hundreds and hundreds of languages? Aren't there languages to be discovered rather than invented?

In general, there is a greater realization in Europe that computer engineering is running ahead of computer science, that computer science does not provide a firm scientific foundation for computer engineering. This contributes to the greater role of formal methods in European computer science. "Formal methods are more relevant than complexity theory," insist some of my European colleagues.

Academia cannot compete with industry in producing computer products, but academics can sit back and think and analyze and try to build a solid foundation for our science.

3.2 Faith in formal methods

In spite of the fact that the US leads the software revolution, gloom and doom are typical in this country when it comes to formal methods. Formal specifications are rare in industry. Verification is often pronounced impossible. "Formal methods," you hear, "had their chance and failed." That attitude toward formal methods is known in Europe as well. The Scientific American article, mentioned above, is written by two Europeans. In the article, the possibility of verified software is dismissed without any arguments; the non-verifiability is treated as an obvious and well-known fact. "Perfect software is a practical impossibility."

It is not true that formal methods had their chance and failed. It is true that particular approaches had their chance and failed. And the criticism of formal methods is justified to a great extent. Indeed, certain approaches were oversold. I believe, for example, that denotational semantics was oversold as a practical tool. I am far from trying to declare denotational semantics useless. It was a great advance in our understanding of computing and it is playing an important role in theoretical computer science, but it is not a practical tool.

Computers are a relatively recent artifact, and computer science is a relatively young science. Naming something a science does not make it a science. (Some people joke that only bad sciences have "science" in their names. Mathematics, physics, chemistry don't.) I believe though that computer science is a science ... in the making. It seems that more ideas flow from computer applications to computer science than the other way round. I believe this will eventually change. Computer science will emerge as a real hard core science and will drive applications.

That kind of optimism is typical in Europe. People working in formal methods often view themselves on the front line of computer science. This positive attitude toward formal methods is not restricted to academia. Some industry and even some government organizations seem to share it to some extent. I was told at Orsay University in Paris that, in connection to some passenger-security problems in the Metro system, the French government required the bidders to provide formally verified solutions. The winner, Jean Raymond Abrial, indeed provided such a verified solution.

Of course I am not saying that there are no successful applications of formal methods in this country. Furthermore, I am not saying that there are more successful applications in Europe than in the US. I am saying is that formal methods play more more central role in European computer science and in European computer science curricula. This brings us to the issue of education.

3.3 Education

British, French, German, etc. high schools give better mathematical training than American high schools do. For example, in France, in order to enter an elite engineering school, one goes through thorough examinations based mostly on mathematics. The undergraduate mathematical training of US computer-science graduates is in general inferior in comparison with European universities.

In addition, we lag behind, very substantially so, in teaching formal methods. All British universities, at least all visible British universities, teach specification and verification methods. For years and years, British CS departments produce graduates that know formal specification methods, trust them and are willing to use them. This is a tremendous advantage.

3.4 The incredible breadth of European foundational research

European activity in foundations of computer science is amazing in its scope and depth. There is a tremendous amount of experimentation with formal methods going on in Europe. It is a playground of ideas. The two volumes [van Leewen 1990] give plenty of evidence.

In this connection, let me mention the field of operational semantics and related specification methods which is close to my interests. Probably, the most popular specification language today is Z, originated by Frenchman Jean Raymond Abrial and developed in Oxford, England. I mentioned above that all British universities teach specification languages; choice number one is Z. Z is not that popular outside the UK, but it is used here and there in Germany, other European countries and also in the US. An annotated bibliography on Z can be found in the [Stepney and Barden 1993]. One of the earliest operational semantics, Vienna Definition Method (VDM), is still alive and used [Jones 1990]. VDM originated in the 1960s in the Vienna Lab of IBM under the name of Vienna Definition Language and was used to specify programming language PL1 formally. Structured Operational Semantics was developed by Gordon Plotkin in Edinburgh, Scotland [Plotkin 1981]. All of these methods not only originated in Europe but were developed there as well. A relatively new approach, Evolving Algebras, originated in Michigan but most of the work has been done in Europe (by Egon Börger of Pisa, Italy, and Dean Rosenzweig of Zagreb, Croatia, and their collaborators and students) [Gurevich 1993].

Functional Programming, logic programming, automata and formal language theory are more popular in Europe.

3.5 ESPRIT

To overcome the fragmentation of the European research by countries, European Community funds ESPRIT, European Strategic Program for Research and Development in Information Technology. ESPRIT projects invariably involve researchers from several countries, and the approach works well in general.

What has all this to do with us? The USA is one country. It looks like we don't the problems of integration. But we do have it in a somewhat different way. Various regions of the country differ greatly in research and funding. Maybe, there is room for supporting inter-regional cooperation and coordination.

3.6 Conference centers

German International Conference and Research Center for Computer Science hosts weekly workshops in its Dagstuhl Castle; every week there is another workshop. Dagstuhl follows the tradition set by the older Center for Mathematical Conferences in Oberwolfach, Germany. Participants are provided with all the necessities. This and the isolated location and attentive stuff create a fruitful working atmosphere.

International Center for Mathematical Conferences in Luminy, France (near Marseille) hosts a continuing series of Logic in Computer Science conferences. The nature is dramatically different, Mediterranean fiords instead of mid European forest, but the scientific atmosphere is similar. International Center of Mechanical Sciences in Udine, Italy hosts computer science schools from time to time.

I do not know whether the Dagstuhl model would work in this country. It certainly makes the organization of meetings much easier.

3.7 Disparate remarks

National barriers are a problem in Europe, but maybe because of that problem Europeans succeeded in creating strong, unifying and active organizations. The organizations most relevant to logic are: European Association for Theoretical Computer Science, European Association for Computer Science Logics, and European Foundation for Logic, Language and Information. In spite of national divisions, there is a stronger sense of community in European logic.

There are many female computer scientists in France and Italy although the situation in most other countries, e.g. Germany or the UK, is closer to that in the US.

4 Other comments

I was asked to be subjective and critical. Well, the following comments are critical to various degrees (although in some cases I intended just to pose questions), and they range from more objective and general (at the beginning) to subjective and technical (at the end).

There are of course problems with criticizing scientific directions or even scientific organization. A perceived weakness may unexpectedly turn out to be a strength, impractical may turn out to be practical, and irrelevant may turn out to be very important and relevant. I remember vaguely reading somewhere that, in his student years, Albert Einstein ignored lectures on modern "fancy" geometries. He was interested in physics rather than pure mathematics. But later he needed those geometries for his relativity theories.

Let me recall also the difference between plain truth and profound truth. Plain truth is a kind of truth you deal with in mathematics. In particular, a statement contradicting a plainly true statement is plainly false. For example, $2 \times 2 = 4$ is plainly true, and $2 \times 2 = 5$ is plainly false. Profound truth is different. A button on my friend's refrigerator says, "Life is more complicated that you think it is, stupid." Profoundly true, isn't it? Once in a while, the friend turns the button around. The other side says, "Life is simpler that you think it is, stupid." Profoundly true again. We deal more with profound rather than plain truth here.

4.1 Scientific organization

In many European countries, for example in Germany, universities are less independent than American universities. Ministries have final words in hiring professors and even in curriculum matters. That makes the system more conservative. It is possible that there are positive sides to such government intervention (guarantees of certain standards, for example); I failed to find colleagues willing to defend it.

The level of government bureaucracy varies from one country to another and usually exceeds the American level. A student of bureaucracy will be interested in France and exhilarated with Italy. "It is the same system in both countries, set by Napoleon by the way," says a colleague with much Italian and French experience, "the difference is only that what takes a month in France, takes a year in Italy."

The emerging European Community structure is not devoid of bureaucracy either. Even ESPRIT, rightfully praised for putting money into research and bridging between countries, is criticized by some beneficiaries for endless meetings and report writing, and a somewhat mysterious decision-making process. "I think NSF runs a leaner operation that is more merit-based," says a colleague with both American and European experience.

There is a substantial fragmentation of effort related to the division of Europe into still very independent countries. For example, the Z specification method, so popular in the UK is virtually unknown in France, even in Paris where Abrial, the originator of Z, lives.

Finally, let me mention the greater importance and influence of scientific schools in Europe. In the US, even the most successful Ph.D. students usually have to leave their nests and look for jobs elsewhere. In Europe, the best Ph.D. graduates often stay in the home institutions. This helps to create strong scientific schools in single locations. The atmosphere of such a school may be conducive for young insiders to grow. On the other hand, you may hear complaints from outsiders trying to develop alternative approaches.

4.2 Gap between academia and industry

There is, in general, a greater gap between academia and industry in Europe as far as logic research is concerned. Sometimes this is expressed with "They are more theoretical," which is true too but does not mean the same thing. Theoretical should not be the opposite of practical. Sound theory may be very practical (modern cryptography is a good example), and of course lousy theory may be impractical.

Computer science attracts mathematically talented logicians, which is wonderful all by itself; there are plenty of foundational and practical problems where such talents can be used. But the gap between academia and industry is so big that too often logicians spend their time doing things whose practicality, even in the long run, seems doubtful. (Another side of that situation is that industry leaves solving various logic questions to people with no appropriate logic training or adequate time. No wonder that the solutions are so often too ad-hoc-ish. But this is a different story.) Too often toy examples are not only the beginning but also the end of a story. Looking recently for a text for my class on Principles of Programming Languages, I saw books where a version of λ calculus is the main programming language. The field of logic programming seems to be another good source of examples of such over-theoretical work.

4.3 Pure functional programming

I move from more general comments to more subjective and technical ones. In this subsection, my goal is to raise a question.

Functional programming is a very active field involving enormous programming effort and much theoretical work. It is also an application field for type theory, denotational semantics, etc. The idea of functional programming isn't new. It inspired McCarthy to create Lisp, one of the first programming languages. Lisp is far from the ideal of pure functional language which is supposed to have no assignments or any side effects whatsoever. Numerous other functional languages acquired some imperative features on the way. A good example is that of ML which acquired assignments and pointers (sorry, references) and thus became impure. Historically these languages are pure functional languages that went a sinful way of imperative programming, but one can see them as imperative languages with extensive functional components. The current pure functional languages like Miranda and Haskell are not sufficiently efficient yet.

Functional programming inspired people on both sides of Atlantic. In this connection, see the passionate Turing Award Lecture by John Backus [Backus 1977]. But the idea of all-purpose pure functional programming is more popular in Europe. A skeptic may wonder if there are objective limitations to the efficiency of pure functional languages. It is true that λ calculus suffices to program all computable functions, but this does not say that much. It is true that the functional programming paradigm is closer to mathematics as we know it, but the development of mathematics has been driven by different applications. We philosophized on that subject before [Gurevich 1988], so let us jump to the conclusion. It is not impossible that the price for avoiding explicit states is too high, that indeed there exist objective limitations on the efficiency of pure functional languages. The usual reaction to my raising these questions was criticism of imperative languages, usually quite valid. Of course the progress of functional programming is most welcome. There are numerous tasks for which functional programming is just right, and functional programs may be easier to verify. Still, it is worth examining possible limitations. Functional programming may never replace imperative programming completely. Maybe, a right mixture of imperative and functional programming is needed, an imperative language with a clean and powerful functional component.

4.4 Pure logic programming

Similarly to the previous question, one may raise a question of objective limitations on the efficiency of pure logic programming. Again, there are numerous tasks for which logic programming is just right and logic programs may be easier to verify than the corresponding imperative programs. In this case too the languages in wide use are "dirty" and a more pragmatic ideal may be that of the right mixture of logic and imperative programming.

The gap between pure logic programming of theoretical papers on one side, and Prolog and other existing logic programming languages on the other side is troubling. Prolog has become a more accepted language through its very efficient implementations and its success as a software engineering tool coincided with developing a programming style drifting away from the logic programming paradigm. "The theoretical logic programming community loses interest in Prolog; Prolog acquires the same status that Lisp has in the functional programming community," say Egon Börger and Dean Rosenzweig, whose paper [Börger and Rosenzweig 1992a], addressing the practical issues of Prolog, will appear in The Science of Computer Programming rather than in a logic programming journal.

There are plenty of papers in logic programming, often related to default and other non-monotonic logics, whose relevance to programming escapes me. And default logic itself, which started with a promise of simplified reasoning, turned out to be less practical than classical logic.

4.5 Proofs as programs

A proof of a specification in constructive (intuitionistic) logic yields (e.g. via classical Kleene's realizability or more modern type-theoretical modified realizability) a program which is a provably correct implementation of the specification. This idea inspired many; in particular it is an inspiration for the proof system Coq, one of the best known and influential proof system, developed in Inria Roquencourt near Paris. This idea conveniently separates two concerns — proofs and programs — which some find an improvement over the predicate transformer technique (involving weakest preconditions) championed by Edsger Dijkstra [Dijkstra 1976]. The following paragraph is a little more technical and can be skipped if you are not interested in technical details.

In certain cases, proofs and programs correspond very closely to each other. Consider the typed combinatory calculus with constants $Kxy = x$ and $Sxyz = (xz)(yz)$. It is easy to see that the type of K is

$$\alpha \rightarrow (\beta \rightarrow \alpha)$$

where α is the type of x and β is the type of y, and the type of S is

$$(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$$

where the types of z, y, x are $\alpha, \alpha \rightarrow \beta$ and $\alpha \rightarrow (\beta \rightarrow \gamma)$ respectively. The two displayed types happen to be the two axioms for the minimal constructive logic of implication. The only derivation rule of that logic, *modus ponens*, corresponds naturally to the composition of functions: if X is of type α and Y is of type $\alpha \rightarrow \beta$, then $Y(X)$ is of type β. Any proof of a formula ϕ in the logic gives a unique program (that is, a constant in the combinatory calculus) of type ϕ, and the other way around. This is a special case of the so-called Curry-Howard correspondence described for example in [Simmons 1993]. This beautiful correspondence between programs and proofs has been extended somewhat but nowhere near real-life programming languages.

People working in the field tell me that there are problems with the implementation of the idea. The program extracted from the proof is often very slow. One reason for the slowness is that the program verifies not only the final result but also various intermediate results. It isn't obvious how to free the program from unnecessary verification. In fact, people do a bit of reverse engineering analyzing faster programs to see which proofs can generate something similar. Sometimes this analysis is revealing. Jean-Louis Krivine was explaining to me a simple proof theoretic meaning of some relatively obscure feature of the C programming language [Krivine 1993].

But so far, this intercourse between logic and programming was more important to logic than programming. The combinatory calculus mentioned above isn't a real programming language; it is another logical calculus. Krivine does work with a real programming language but he has, I understand, a long way to go. I wonder how would a logic tailored to C programming language look like. (We were recently looking at C [Gurevich and Huggins 1993].) I would like to see a critical examination of the idea; may be there are some objective limitations out there. It is possible that the proofs-as-programs approach will play a role in software engineering. It is important to understand what that role should be.

4.6 Mathematics and pedantics

I know, I know, there is no such word "pedantics". But I need a new term. Let me explain. The rigor of classical mathematics may not suffice in the verification area. Good mathematical papers happen to have errors. You may see that the proof of a lemma is not quite correct but it is clear how to fix it. The lemma itself may be incorrect as stated but again it is clear what the intention is, how the lemma is used and how to fix it. Such freedom to deal with "inessential" details may not exist in verification business. Imagine yourself flying on an airplane for which you verified the autopilot. (This example is "stolen" from somewhere but for the life of me I do not remember from where.) The weather is bad and the control is given to the autopilot that you had designed and proved correct. You realize suddenly that you forgot to check a case or two out of a great many cases. Your proof is correct in principle and the autopilot is fine ... modulo some "inessential" details, but somehow this does not bring the desired comfort.

Many draw the conclusion that verification in general should be carried out within some formal system. To an extent, this explains the popularity of temporal logic and other formalisms and the proliferation of reasoning systems.

It seems to me we should distinguish better between formal and precise. Precise mathematical proofs are rarely carried out within formal systems though they are formalizable. (Thanks to the foundational revolution in the beginning of this century, usual mathematical proofs can be rewritten in, e.g., ZFC (the first-order Zermelo-Fraenkel Set Theory with the Axiom of Choice); in many cases, Peano Arithmetic is sufficient.) And in practice proofs carried out within formal systems are not necessarily precise.

Why don't "normal" mathematicians prove their theorems within formal systems? The following argument occurred to me 10 or so years ago in a discussion with proof theorist Jonathan Stavi. When you search for a proof, you don't want to restrict your logic in any way. You want to free your imagination. You search for arguments first and worry about their logic complexity later. This is not to say that proof theory is unnecessary, not at all. It just serves a different purpose. In particular, if and when you find your proof, you may subject it to a revealing proof-theoretic analysis; see [Simpson 1986] in this connection. Excessive formalization makes proofs harder to find and comprehend.

I believe that we should clearly separate concerns and distinguish between mathematical proofs, like those in mathematical literature, and what I propose to call *pedantic proofs*, that is, proofs written in all details and verified by computer. First find a mathematical proof and then — if necessary — convert it into a pedantic one. The term "pedantic" is positive in this context. As we saw above, in that autopilot example, pedantic proofs are necessary and important. And the science of converting mathematical proofs to pedantical ones is not trivial by any means.

The obvious objection to "normal" mathematical proofs in verification is that verification proofs are much different from normal mathematical proofs. That they are plain and boring, like checking and checking and checking numerous cases. That computers do that kind of work better and one may want some formal system to guide the computer. I buy that. Indeed the whole task reduces in many cases to pedantics, and computers are better in many aspects of pedantics than humans, and a formal system may be most useful. It is not true, however, that all verification proofs are like that. There may be an interesting mathematical structure behind the plenitude of boring details, in which case mathematical analysis may be useful. One example, close to my research interests, is the correctness proof for compiling Prolong to Warren Abstact Machine (the standard implementation of Prolog) in [Börger and Rosenzweig 1992b].

References

Babai et al. 1991 L.Babai, L.Fortnow, L.Levin, M.Szegedy, "Checking computations in polylogarithmic time", ACM Symposium on Theory of Computing, 1991, 21–31.

Backus 1977 John Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," in "ACM Turing Award Lectures: The First Twenty Years," ACM Press, 1987, 63–130.

Blum and Kannan 1989 Manuel Blum and Ravi Kannan, "Designing Programs that Check their Work", ACM Symposium on Theory of Computing, 1989, 86–97.

Börger and Rosenzweig 1992a Egon Börger and Dean Rosenzweig, "A Simple Mathematical Model for Full Prolog", TR 33/92, Dipartimento di Informatica, Universita di Pisa, 1992, 23 pages, to appear in The Science of Computer Programming.

Börger and Rosenzweig 1992b Egon Börger and Dean Rosenzweig, "The WAM — Definition and Compiler Correctness", TR 14/92, Dipartimento di Informatica, Universita di Pisa, 1992, 57 pages, to appear in "Logic Programming: Formal Methods and Practical Applications", Eds. C. Beierle and L. Pluemer, North-Holland, 1994.

Dijkstra 1976 Edsger W. Dijkstra, "A Discipline of Programming", Prentice-Hall, 1976.

Fagin 1990 Ron Fagin, "Finite-Model Theory: A Personal Perspective", Theoretical Computer Science 116, 1993, 3–31.

Garey and Johnson 1979 Michael R. Garey and David S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Competeness", Freeman and Company, New York, 1979.

Gurevich 1988 Yuri Gurevich, "Logic and the challenge of computer science", in "Current Trends in Theoretical Computer Science", Ed. E. Börger, Computer Science Press, 1988, 1–57.

Gurevich 1993 Yuri Gurevich, "Evolving Algebras: an Attempt to Discover Semantics", in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, Series in Computer Science, Volume 40, 1993, 266–292.

Gurevich and Huggins 1993 Yuri Gurevich and Jim Huggins, "Semantics of the C Programming Language", Springer Lecture Notes in Computer Science 702, 1993, 274–308.

Hodges 1977 Wilfrid Hodges, “Logic”, Penguin Books, 1977.

Immerman 1989 Neil Immerman, “Descriptive and Computational Complexity,” in “Computational Complexity Theory”, Ed. J. Hartmanis, Proc. Symp. in Applied Math. 38, American Math. Soc., 1989, 75-91.

Jones 1990 C. B. Jones, “Systematic Software Development using VDM”, Second edition, Prentice Hall International, 1990, 333 pages.

Krivine 1993 Jean-Louis Krivine, Private communication.

Lamport 1993 Leslie Lamport, “How to Write a Proof”, Research Report 94, Digital Equipment Corporation, Systems Research Center, 1993.

Littlewood and Strigini 1992 Bev Littlewood and Lorenzo Strigini, “The Risks of Software”, Scientific American, Nov. 1992, 62–75.

Neiman Peter Neimann, Editor, “Risks to the Public in Computers and Related Systems”, regular column in Communications of ACM.

Plotkin 1986 Gordon Plotkin, “A Structural Approach to Operational Semantics”, Tech. Report DAIMI FN-19, Aarhus University, Computer Sci. Dept, Denmark, 1981.

Simmons 1993 Harry Simmons, “Logic and Computation: Taking the Curry-Howard Correspondence Seriously”, Manuscript, Computer Science Dept., Manchester University, UK, 1993.

Simpson 1986 Stephen G. Simpson, “Subsystems of Z_2 and Reverse Mathematics”, appendix to “Proof Theory”, second edition, by G. Takeuti, North-Holland, Amsterdam, 1986, 434–448.

Stepney and Barden 1993 Susan Stepney and Rosalind Barden, “Annotated Z Bibliography”, Bull. of the Euro. Assoc. for Theoretical Computer Sci. 50, June 1993, 280–313.

van Leewen 1990 Jan van Leewen, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, Amsterdam, 1990.