Cultures of programming

Understanding the history of programming through controversies and technical artifacts

TOMAS PETRICEK, University of Kent, UK

Programming language research does not exist in isolation. Many programming languages are designed to address a particular business problem or as a reflection of more wide-ranging shifts in the computing community. Histories of individual programming languages often take their own contexts for granted, which makes it difficult to connect the dots and understand the history of programming in a holistic way.

This paper documents the broader socio-technological context that shapes programming languages. To structure our discussion, we introduce the idea of a *culture of programming* which embodies a particular perspective on programming. We identify four major cultures: hacker culture, engineering culture, managerial culture and mathematical culture. To understand how the cultures interact and influence the design of programming languages, we look at a number of historical strands in four lectures that comprise this paper. We follow the mathematization of programming, the development of types, discussion on fundamental limits of complex software systems and the methods that help make modern software acceptably reliable.

This paper makes two key observations. First, many interesting developments that shape programming languages happen when multiple cultures meet and interact. Those include both the development of new technical artifacts and controversies that change how we think about programming. Second, the cultures of programming retain a strong identity throughout the history. In other words, the existence of multiple cultures of programming is not a sign of an immaturity of the field, but instead, a sign that programming developed a fruitful multi-cultural identity.

The methodology used in this paper is inspired by historically grounded philosophy of science. We look at the four aforementioned historical strands and find a conceptual framework capable of effectively explaining the key historical developments. Using the prism of cultures of programming to look at the history of computing sheds a new light at the controversies and the development of technical artifacts that we discuss in this paper. It provides an explanation for a number of, otherwise hard to grasp, historical developments and it explains why certain debates keep recurring over the history. For a programming language researcher, the paper might also point at new, yet unexplored, intersections between cultures.

ACM Reference Format:
Tomas Petricek. 2018. Cultures of programming: Understanding the history of programming through controversies and technical artifacts. 1, 1 (September 2018), 75 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Author's address: Tomas Petricek, University of Kent, Canterbury, UK, tomas@tomasp.net.

XXXX-XXXX/2018/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computer programming originated at the intersection of many distinct disciplines including logic, electrical engineering, psychology, business and art. Although much of work on computer programming is now hosted by academic computer science departments and by the software industry, the different cultures arising from different ways of thinking about programming are still present and provide, sometimes incompatible, perspectives on programming.

In the early days of computing, this could have been just a sign of the immaturity of the field. When the computing appeared, the early contributors inevitably came from diverse backgrounds. However, 70 years after the first electronic digital stored-program computer was built, the idea that computing is still an immature field and that the different cultures will eventually merge seems even more unrealistic.

Looking specifically at programming languages, there is a number of programming languages that combine ideas from multiple cultures, but equally, new programming languages keep appearing that are strongly rooted in one specific culture of programming and are heavily criticised by proponents of other cultures. On one hand, programming languages such as Scala and F# reconcile concepts developed by the mathematical culture (their functional heritage) with practical concerns of the engineering culture (as dictated by their runtimes and ecosystems). On the other hand, programming languages such as Agda and TypeScript continue advancing the state of the art solely within one particular culture; Agda develops the mathematical culture by making proofs a part of the program, while designers of TypeScript are committed to the engineering culture and make decisions based on how large JavaScript systems are built, even if that means admitting unsoundness.

In this paper, we identify four different cultures of programming. The mathematical culture sees programming as a mathematical activity and advocates the use of formal methods. The hacker culture sees programming as tinkering with computers and relies on the practical skills of an individual. The managerial culture advocates solving programming problems via organizational structures and processes. Finally, the engineering culture develops tools and methods that help individual programmers and teams in a technical way.

Most of this paper is dedicated to examining how the different cultures interact using a number of case studies that are organised into lectures. Each lecture starts with a dialogue that illustrates some of the disagreements and surprising discoveries that resulted from the interaction of cultures, followed by extensive notes that document the historical strand covered less precisely in the dialogue.

The interactions we look at are twofold. First, we consider a number of cases where a particular controversy triggered a disagreement and debate between cultures. The debates we cover revealed important issues, disturbed the established practices within individual cultures and, consequently, led to the development of new ideas. Second, we consider a number of cases where multiple cultures contributed to the development of a technical artifact such as a programming language feature. This often advances the state of the art of programming in ways that would unlikely happen within a single consistent culture.

The four lectures that form the core of this paper document a number of strands in the history of programming languages. We consider programming languages in a broad sense to accommodate debates that were not originally about programming languages, but influence programming language design. We aim to make the historical account in this paper as accurate as possible, but our key contribution is that we present the historical developments, many of which were documented elsewhere, within a new conceptual framework. Although we certainly "study the past with the reference to the present" [Butterfield 1965], we try to avoid the simplifying view of seeing the past

as a linear progression toward the present that some computer scientists make when talking about the history of our discipline. To this end, we extensively rely on primary references in the lecture notes and often refer to reviews written by historians.

The conceptual framework used in this paper allows us to shed a light on the history of programming language features such as types or exceptions, programming concepts such as tests and debates about formal verification and the limits of correctness of large software systems. For a computer scientists, perhaps equally interesting aspect of this paper might be the interactions between cultures that have not happened yet. If our premise that interesting ideas appear when cultures interact is, indeed, true then the missing interactions point to areas where new ideas for programming languages could come from.

Analytical index: An outline of the paper

Lecture: Mathematization of programming. Programming starts shifting from being a black art to being an engineering discipline. The very idea of a programming language appears when managerial, mathematical and hacker cultures meet. Mathematical culture argues that programming is a mathematical discipline. These developments are largely ignored by the data processing industry with the exception of the idea of structured programming, which gains a managerial interpretation. Engineers question the belief that proofs give a guarantee of correctness; hackers question the assumption that we can provide formal specifications and philosophers debate whether mathematical proofs can talk about the physical world.

Lecture: Multicultural notion of types. After some confusion about the terminology, the engineering culture introduces the notion of a type in the context of programming languages. Many ideas about types are imported from logic by the mathematical culture, where a related notion of type existed in work on the foundations of mathematics. The engineering culture uses types for structuring data, while the managerial culture adopts types as a mechanism for information hiding. Many of the ideas about types meet and the ML language appears. The mathematical culture starts thinking of types as a lightweight formal method and develops type systems that can prevent various kinds of errors. Meanwhile, the engineering culture starts using types as a tool for documenting, writing and organising code. In this process, it diverges from the mathematical culture and breaks a number of core assumptions that the mathematical culture makes about types.

Lecture: Software as a socio-technological system. Digital computers escape the lab and start being used by businesses. The lack of trained personnel to program them marks the first stage of a computing crisis. Managerial culture starts looking for ways of training programmers, while the hacker culture starts building "automatic programming systems", that will later evolve into programming languages. The programming cultures meet at the 1968 NATO conference and the term "software engineering" is born, however, each culture has its own interpretation of what "turning the black art of programming into a proper engineering discipline" means. Early work on building an anti-ballistic missile systems sparks a debate on fundamental limitations of software systems. The engineering culture points out that software systems are not algorithms and considerations about building large and complex software systems convince some authors that software systems always need to be considered as socio-technological entities, together with humans that use and build them.

Lecture: How software gets reliable. A strong advocate of the mathematical culture points out that, despite the very limited use of formal methods in the industry, computer systems are surprisingly reliable. In an attempt to investigate how this happened, we look at good engineering practices such as debugging, testing and error recovery. Debugging and testing appear as "program checkout"

in the hacker culture during the early days of computing. While debugging keeps its hacker status, testing is adopted and adapted first by the managerial culture and, later, but the engineering culture. Error recovery also appears within the hacker culture. The early ideas get integrated into programming languages in the form of exceptions. In systems where errors are unavoidable, the engineering culture starts exploring systems that not only recover from errors, but use errors as a key component in system architecture.

LECTURE: MATHEMATIZATION OF PROGRAMMING

In the late 1960s, academics, businesses and military realised that computers have the power to radically transform the way they work. Equally, computer glitches became increasingly visible and anecdotes about canceled or massively delayed programming projects that significantly exceeded the planned budget have become a commonplace in the computing industry.

In this lecture, we follow one particular strand of history that, in the words of its proponents, aims to set computing on solid mathematical foundations. The following debate is inspired by the history between the *1968 NATO Conference on Software Engineering* and the 1988 paper *Program Verification: The Very Idea* by James Fetzer.

Teacher: You all have experience with programming electronic computers for different purposes. Let me start with a very open-ended question. Do you think there is a method for programming computers that will minimize or even eliminate glitches?

Alpha: Computer programming is a completely new discipline, so I'm not surprised that we need more time to figure out how to do it right. If you are curious enough, computers are a great fun and once you play with them a bit, you just learn how to do it. I don't think there is any special method to it, just a lot of experience.

Gamma: If you look at the advertisements from companies hiring programmers, you will notice that they are looking people who enjoy mathematics, but also musical composition and arrangement, like chess and possess a lively imagination. That makes me think that the ideal method might have more common with how art and music are done.

Omega: Right, but it is *black art*! I was lucky enough to work with some great programmers, but they rely on private arcane methods that are reinvented with every single project. It seems that good programmers are *born*, not trained. How are we supposed to scale our projects when we rely on primadonna programmers and cannot train new ones?

Epsilon: This is because we are still treating programming as playing with toys, not as an engineering discipline. We can build bridges reliably and on budget, so why not computer systems? The black art of programming has to make way for the science of software engineering!

Omega: That sounds very nice, but how do you propose to do this? Engineering has a lot of methods such as building prototypes, rigorous testing, reliability engineering and hierarchical management structures. I'm not even sure where to start.

Tau: Sorry to interrupt, but we can do much better than building bridges. As we learned with the development of Algol, computer programs are just syntactic structures in a formal language defined by the rules of logic. All properties of program execution can be known from its text by purely deductive methods.

Omega: Even better! Once IBM finally releases an Algol compiler for our machine, we can surely check it out. But I'm still not sure how are these deductive methods supposed to help my team without making programming even more expensive.

Tau: If you structure your programs well, you will be able to reason about them with a mathematical certainty. This gets even easier when you follow the ideas of *structured programming* where the syntactic structure of the program follows the logical structure of the evaluation. So, your understanding of higher-level structures will compose from the understanding of lower-level structures in your program.

Omega: I see how this idea could be used to manage large programming teams in a top-down fashion. You can proceed from the general design goal to specific implementation detail and let one architect manage the decomposition. If the languages and tools can support this pattern, we can hire small number of experienced programmers to work on the high-level parts and a large number of unskilled programmers to solve the low-level problems.

Epsilon: You seem to be treating programming as a completely management problem. This is a wrong way of thinking about it and I, for one, absolutely refuse to regard myself as either a managerial architect or technical puzzle-solver.

Teacher: I find it interesting that the concept of structured programming can be interpreted in such two very different ways – as a programming practice and a management method. However, I want to learn more about what *Tau* suggested earlier. How can we reason about computer programs formally to guarantee that they work correctly?

Tau: Sure, I'm happy to give you an example. Let's take a look at the following simple program in an Algol-like formal language:

$$k = 1; \text{while } n > 0 \text{ do } k = k * n;\ n = n - 1 \text{ end}$$

As you probably already guessed, this calculates the factorial of the number n. To reason about it formally, we can associate we can associate a pre-condition and post-condition to every statement in the program. We'll write $\{P\}s\{Q\}$ to denote that, if P holds before the execution of the statement s, then Q will hold after the execution of s. Let's go to the whiteboard and let me show you how to do this for our factorial...

Tau: (*runs to the whiteboard and spends 30 minutes writing a formal proof*)

Epsilon: You convinced me that your code snippet does, indeed, calculate a factorial, but why does the proof have to be so long and tedious? Mathematical proofs that I learned at the university always had some insight. This is just tediously showing something that you know is true just by looking at the program.

Gamma: This is why formal verification is bound to fail. Formal proofs about computer programs are not the same thing as formal proofs in mathematics. A mathematical proof is shared with colleagues, discussed at whiteboard sessions and checked by every student who learns it. This is what establishes its truth! Nobody will excitedly rush into your office with a printout of a 20 page tedious proof of some trivial fact about a simple program.

Tau: This is a claim from a political pamphlet from the middle ages, rather than a sensible argument against formal verification! The validity of a proof does not depend on how many people excitedly run into a colleague's office to discuss it. If you have a formal proof and check it once, it is a proof!

Gamma: It might be a proof, but the social processes of ordinary mathematics give you more guarantees. For example, mathematical proofs are more stable – when there are small errors, they are often easy to correct. This is because the community that accepts the proof has the right intuition about the proof.

Epsilon: I can understand your doubts, but there is a similar social activity about the correctness of standard algorithms that we teach our students at the university.

Gamma: I'm not worried about the proofs of correctness of simple algorithms, but about proofs that involve large systems. Those are the most tedious once that you want to get done, but nobody will want to read them. Just like mathematical proofs, they aim to give you guarantees, but unlike mathematical proofs, they offer no new or deeper insight.

Teacher: I think we can better understand the arguments about proofs if we recognise that there are different cultures of proving. What a mathematician considers a proof is clearly different from what programmers consider proof. There might even be multiple different cultures around program proofs...

Epsilon: I think the main point of proof of program correctness should be to convince the programmer and their team that the program does what it is intended to do. If you rigorously discuss every subroutine in this way, you can be reasonably sure that your code is correct and, more importantly, that everyone in your team understands it and agrees on the specification.

Lambda: Excuse me for jumping in, but do you claim this method will fully eliminate all errors?

Epsilon: No. I don't think that is a realistic expectation. You will still get some errors that you need to fix during debugging, but the error rate will be significantly smaller.

Lambda: Why settle for small error rate when you can reliably eliminate all of them! Thanks to the modern proof assistants, you can use the computer itself to check that your proof is valid. The proof assistant serves as an implacable skeptic that insists on all assumptions being stated and all claims justified. If you write a formal proof and check it using a proof assistant, you can be absolutely certain that it is right.

Epsilon: But who verifies the verifier?

Lambda: Oh, the proof checkers can be constructed such that they produce a full proof derivation that lists the applied rules. Checking that the derivation does, indeed, follow the rules is very simple algorithm that can be easily checked by hand or, if you prefer, even by another proof checker.

Gamma: I'm willing to accept that your proof checker works correctly, but I think that this ruins the proof for another reason. Once you leave proof checking to the machine, you no longer need to understand why the proof works. This is a disaster, because ordinary mathematical proof plays a crucial role in explaining the theorem. Now you just end up with theorems that might hold, but you have no idea how and why they are formulated in the way they are. You are much more likely to get the theorem wrong.

Tau: The theorems that you need to prove about programs capture very high-level properties of the system. They will be intuitive and easy to understand, even without knowing the details of the proof.

Teacher: To guide the discussion, could you perhaps offer an example?

Tau: If I may, I would propose the following: Let's say that you are writing a sorting function `sort`. Assume that $b = \texttt{sort}(a)$ for some array a of n elements. You should prove two properties. First, for some permutation p of indices from 0 to $n-1$ and for all indices i, it holds that $b[p(i)] = a[i]$. Second, for any indices i, j, it holds that $i \leq j \implies b[i] \leq b[j]$. The properties guarantee that, first, the function rearranges existing elements and, second, produces a sorted array.

Alpha: *(opens a laptop and types something vigorously for a few minutes)*

Alpha: I have a perfectly fine sorting algorithm, but the property does not always hold! It works fine with integers, but not if you want to search floating-point numbers and one of your values is *NaN*. For example, $\texttt{sort}([3.14, 2.71, \mathit{NaN}]) = [\mathit{NaN}, 2.71, 3.14]$ but $\mathit{NaN} \not\leq 2.71$!

Tau: I could have expected an issue with floating-point numbers. They are just nasty. But you can easily fix that by saying that the $\leq$ in the theorem is not defined when either side is *NaN*.

Teacher: To generalise from this example, the problem seems to be that our mathematical model, i.e. real numbers, do not match what the actual computer uses, i.e. floating-point numbers. Is this an inherent limitation of all formal verification of computer programs?

Lambda: But that is because the model and the property proposed by *Tau* was a naïve. If we want to prove anything about real programs, we need a precise mathematical model, including proper floating-point numbers. That's the only way to avoid bugs caused by *NaN* and other subtle aspects of floating-point arithmetic!

Philosopher: If I may join the debate, I think we are finally getting close to the fundamental problem with any attempt to prove programs correct. The proofs about program may lack the social processes of proofs in mathematics, but that is something that the community could change. What is more fundamental is that, no matter how you refine your model, you are still working with a formal model of a physical computer!

Lambda: Of course I'm working with a model, but I insist on using a fully accurate one!

Philosopher: I see no problems as long as we are talking just about algorithms, which are merely formal objects. However, a program is not just a syntactic entity. It is a causal model of an algorithm implemented in a form suitable for execution by a machine.

Now, where do axioms about this machine that executes programs come from? They can be either empirical observation of an actual computer or they can be definitional. In the former case, you gain empirical knowledge that is still fallible and so you need to test your system anyway. In the latter case, you gain formal knowledge, but about an abstract system, not a real physical computer.

Omega: Are you suggesting that all formal proofs are useless?

Philosopher: I merely want to make sure people have realistic expectations! If you prove that a program satisfies a specification, you are proving compatibility of two abstract descriptions of the real world. That's still a useful check, but not an infallible guarantee.

Epsilon: Speaking of specifications, I feel like we are ignoring the elephant in the room. The kinds of systems that I would want to formally verify implement complex logic and have correspondingly long and complex specification that evolves during the development...

Teacher: This might be a good time for a brief break. We use the problem of large and complex real-world systems as the opening topic for our next session.

NOTES: MATHEMATIZATION OF PROGRAMMING

The use of mathematics for constructing and analysing computer programs is perhaps the most significant research paradigm in the modern academic programming community. Programming languages are described as formal languages using the methods of logic, we prove that algorithms employed by programming language implementations are correct and we use concepts from abstract algebra to structure our programming language abstractions.

In this section, we trace the origins of the mathematical approach to programming and how it fits with the birth of computer science as a standalone academic discipline. The integration of the mathematical culture into the computing community had wide-ranging consequences. It led to the development of the Algol programming language. It traded ideas with the managerial culture of programming, leading to a controversy around the idea of structured programming. The application of formal mathematical methods to problems in computer programming brought with itself a new set of concerns that were not faced by mathematicians before. We explore the debates that this caused within the mathematical paradigm of computer science.

The fact that the mathematical culture of programming became a significant part of computing had a huge impact on how programming looks today. The details of how this happened are interesting for two reasons. First, the controversies within the mathematical paradigm in the 1980s are precursors of similar debates about programming that we are facing today. Second, the grounding of programming in mathematics is something that we often take for granted today. Understanding that this was not always the case lets us see the benefits and drawbacks of this approach and, perhaps, envision alternatives.

1 THE BIRTH OF COMPUTER SCIENCE

The first computers were built to perform military, scientific and later business computations. People building those machines came from a variety of backgrounds including physics, electrical engineering and mathematics. In the academic environment, computers were seen as tools for other sciences, rather than an object worth studying on its own. Many believed that programming errors will, over time, become as infrequent as hardware errors Priestley [2011, p.254]. To the physicists and the military, building and programming a computer was seen as an auxiliary task that can be handed to a sub-contractor who will build the system according to a specification within a fixed budget [Slayton 2013, p.57].

While solving programming problems, the early computer programmers developed highly sophisticated methods and sophisticated tricks. As John Backus later said, "Programming in the 1950s was a black art, a private arcane matter."[Ensmenger 2012, p.40] The knowledge that programmers developed was a kind of personal knowledge [Polanyi 1958] that was not written down and shared within the community, leading to a notion that programming was black art or alchemy. An analogy with alchemy might be more than just a pejorative remark. As pointed out by Wootton [2015], the secretive nature of alchemy was one of the factors that contributed to the dominance of chemistry.

Programming lacked "the sound body of knowledge that could support it as an intellectually respectable discipline." [Dijkstra 1972] This lack meant that programmers relied on personal knowledge, but it also threatened programming to remain a low-status profession.

The mathematical approach to computer science that was born in universities at this time was a remarkable achievement. As noted by Ensmenger [2012, p.130] the notion of an *algorithm* became the fundamental concept of modern computer science and allowed it to develop into normal science [Kuhn and Hacking 1962]. Algorithms also provided a practical agenda with many interesting open unsolved puzzles that the community can address. A prime example of normal science work is "The Art of Computer Programming" series [Knuth 1968]. Perhaps more importantly, the

mathematical approach also helped computer science to establish itself within the modern university environment which values theory higher than practice.[Hacking 1983, p.150] This provided space for the development for programming languages based on mathematical foundations and for much of the developments on program verification discussed in this section.

1.1 When technology became language

As noted by Nofre et al. in a paper that gave the title to this section, the idea of thinking about programming as a linguistic activity and of thinking about programming languages as *languages* became so ubiquitous in the programming discipline that we rarely even notice it. The linguistic metaphor came through cybernetics and the treatment of computers as electronic brains. The problem of programming became conceptualized as a translation problem—how do we translate the problem description from a language that the human understand to a language that the electronic brain can understand. As noted by Nofre et al. [2014] "during the second half of the 1950s, the language metaphor lost its anthropomorphic connotation and acquired a more abstract meaning, closely related to the formal languages of logic and linguistics."

The most important aspect of the development, however, was that programming languages became standalone objects, independent of hardware that implemented them, and began to be studied on their own. The birth of programming language as an independent object happens at the intersection of three programming cultures. There was the hacker culture formed by the "black art" programmers who actually implemented the first programming languages. There was the mathematical culture that provided most of the resources of logic for developing programming languages as formal mathematical objects. However, and perhaps somewhat surprisingly, there was also the managerial culture focused on solving problems with computers that provided motivation for the birth of programming languages.

As noted by Nofre et al. [2014, p.42] "the early users of the term *programming language* were principally computer-user groups and computer-installation managers attempting to bring about the cross-machine compatibility of programs; common or universal notations would facilitate the exchange of programs among and within organizations, and would also provide a suitable vehicle for teaching programming in universities." Thus, the concept of a programming language was born at the intersection of three very different programming cultures with very different motivations. The first programming languages that emerged from this development were ALGOL, which emphasized the mathematical culture and COBOL, which emphasized the managerial culture.

The three cultures of programming met, gave rise to the very idea of programming language as well as two largely influential concrete programming languages and slowly began to part ways. COBOL was widely adopted by the data processing industry and remains widely used in legacy systems today, but has never been adopted and studied by universities. ALGOL was recognised as an "object of stunning beauty" and a remarkable achievement of computer science, but was never widely adopted in the United States. As documented by Nofre [2018], IBM partly viewed ALGOL as a threat and may have been reluctant to fully support it. The emergence of the ALGOL programming language as a mathematical entity facilitated a range of research that aims to analyse programs and programming languages using the devices of logic. For the academic community, the Algol research programme provided a scientific research paradigm to which they could subscribe and contribute [Priestley 2011, p.229].

The 1962 paper *Toward a Mathematical Science of Computation* by John McCarthy is a manifesto of the Algol research programme. McCarthy notes that "in a mathematical science, it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science" and that programs in particular programming languages are one kind of entities that can be treated by such mathematical computer science. McCarthy outlined a number of program properties that

proponents of the mathematical paradigm are still concerned with half a century later such as: Are two procedures equivalent? Does a translation algorithm correctly translate procedures between two programming languages?

A practical contribution toward reasoning about programs has been offered by Hoare [1969] who introduced the formalistm that is now known as Hoare triples – a notation $P\{Q\}R$ asserting that if an assertion P holds before the execution of program Q, then the assertion R will hold afterward. It is worth noting that Hoare also used a method that has become ubiquitous in mathematical treatment of programming languages to this day. He described how to apply his method to a small subset of the ALGOL language.

The idea of treating programming languages and programs as mathematical objects that can be formally analysed is something that we take for granted. However, this was not an obvious idea when the work was first done. The very idea of using resources of logic for analysing programs only became possible once programming languages became standalone entities, which required the meeting of mathematical, managerial and hacker programming culture.

The academic focus on questions that can be answered by formal mathematical methods has been enormously fruitful, but also found its critics. Naur [1993] argued that it overemphasize "minor issues" and ignores crucial informal aspects of programming that are usually glossed over, such as reflecting how people actually think, elegance of certain solutions or our ability to utilise our intuitions. Naur further notes that "it is curious to observe how the authors in this field, who in the formal aspects of their work require painstaking demonstration and proof, in the informal aspects are satisfied with subjective claims that have not the slightest support, neither in argument nor in verifiable evidence." It is certainly possible to imagine that a different mix of programming cultures would emphasize such informal issues over the formal ones that became the fundamental questions of theoretical programming language research.

1.2 Structured programming

The birth of the mathematical computer science was motivated by practical programming problems, but it remained academic and did not address the problems that the industry was facing in a direct way. However, the different programming cultures exchanged knowledge. Two mechanisms, documented by sociologists, that enable such exchange are trading zones [Gorman 2010], which provide a platform through which cultures without a shared language can exchange ideas and boundary objects [Star and Griesemer 1989], which are concrete entities that the cultures exchange, even though they might interpret them differently. One example of this transfer of ideas is the concept of *structured programming*, which behaves as a boundary object. Structured programming was developed as part of the computer science culture, but was soon given a new meaning by the managerial culture.

Structured programming was popularised following the publication of Dijkstra [1968] letter "Go to statement considered harmful". Dijkstra argued that avoiding the goto statement makes reasoning about programs easier, because the static structure of the text better corresponds to the dynamic structure of program execution. In modern terms, structured programming allows composable reasoning about code.

At the same time, the computing industry was facing a crisis caused by the shortage of programming personnel, burgeoning complexity of software and the professional and political tensions associated with the computer revolution. In response, the community started talking about the need to make a transition from the black art of programming to the science of software engineering, which culminated with the 1968 NATO Conference on Software Engineering [Ensmenger 2012, p.195].

The software engineering movement was less motivated by technological concerns and more motivated by the problem of control over complexity, budgets and workforce [Ensmenger 2012, p.198]. The management culture took the technical idea of structured programming and transformed it into a more managerial notion of top-down approach to program design [Slayton 2013, p.153]. This was appealing as the top-down approach resembled the stratified organisational structure of large corporations [Ensmenger 2012, p.199]. Thus the technical idea of structured programming was reinterpreted to support an authoritarian and rigid structuring of software engineering teams that was appealing to the management of 1970s [Ensmenger 2012, p.209].

The idea of top-down approach to the management of software projects has been largely influential and led to the development of methods such as the Waterfall model [Benington 1983]. The case of structured programming also shows that, while the different programming cultures exchange ideas, they do not fully blend. The originator of the term *structured programming*, Edsger Dijkstra never accepted the managerial interpretation and disdained the "American management philosophy aimed at making companies independent of the competence of their employees". Dijkstra claimed that the banner of software engineering is used to replace intellectual discipline by management discipline to the extent that it has now accepted as its charter "How to program if you cannot." [Slayton 2013, p.166]

It is worth noting that the case of structured programming sets a precedent that would repeat itself in later software development methodologies. For example, the modern notion of microservice architecture has both technological and managerial aspect. At the technological side, software is built in terms of small independent components that make the architecture more modular and aid understanding of individual components and structure of interactions between them. At the managerial side, it allows development by small autonomous teams that can follow different development methodologies and use technologies of their own choice.

2 LIMITATIONS OF THE MATHEMATICAL PARADIGM

The promise of the mathematical paradigm was that it would eliminate errors and debugging. As noted by McCarthy [[n. d.]b], "one should prove that [a program] meets its specifications, and this proof should be checked by a computer program." Instead of relying on private arcane methods of those who mastered the black art of programming, programs would be constructed systematically, using deductive formal methods that would eliminate any doubt that one might have in the correctness of the program.

Half a century later, most software systems built by the industry still ship with bugs and rely on empirical testing. Proponents of the mathematical computer science paradigm today claim, just like their ideological ancestors did half a century earlier, that it is only a matter of economical reasons and education. A number of arguments claiming that program verification, as advocated by the proponents of the mathematical culture, is infeasible used arguments that are not merely political, but approach the question from a deeper philosophical perspective.

The idea of correctness proof imports notions from mathematics into programming. If we see programming as a branch of mathematics, as the mathematical culture does, then this does not lead to any issues. However, if we see programming as a discipline of its own, various problems arise. We consider a number of arguments that are share a common theme. They point out that program correctness and its proof are not the same notions as mathematical theorems and their proofs as practiced by mathematicians.

2.1 Proofs in mathematical computer science

The nature of mathematical proof is not as simple as it appears at a first glance. In formal logic, "a proof is a finite sequence of propositions, each of which is an axiom, an assumption or follows from

the preceding sentences in the sequence by a rule of inference."[Wikipedia contributors 2018] The idealised view of mathematical proof is often based on this notion from formal logic. In practice, proofs that mathematicians work with are quite different. They are written in a mix of natural and formal language and they are written to convince another, adequately qualified, colleague that a theorem holds, i.e. that a formal logical proof could be constructed.

Mathematical proofs are central objects of mathematical practice. They are taught at universities, shared with colleagues and discussed in whiteboard sessions where mathematicians marvel at the elegance of a proof and clever tricks that make it work. They almost never bother fully expanding the proof to a proof as understood in formal logic.

In contrast, proofs of program correctness are bound to be secondary. The main reason for the existence of a computer program is to run it. Proofs do not exist as main objects of knowledge, but merely as certifications that programs match their specifications. This also explains some of the differences in how the two communities treat mechanised proofs. On one hand, many mathematicians remain cautious about the use of computer in proofs [Gold and Simons 2008]. On the other hand, the idea that correctness proofs should be checked by a computer has been proposed as early as 1962 by McCarthy [[n. d.]b] and has since become a main-stream method in computer science.

In 1977, De Millo, Lipton and Perlis pointed out the difference between proofs in mathematics and computer science in a paper "Social processes and proofs of theorems and programs" [De Millo et al. 1979], first published in the ACM Symposium on Principles of Programming Languages (POPL) and later in the Communications of the ACM. The paper triggered a controversy and was equally labeled as "Marvelous, marvelous, marvelous!" and as "a political pamphlet from the Middle Ages" [MacKenzie 2001, p.197].

According to De Millo et al., mathematical proofs encompass a rich structure of social processes that determine whether mathematicians feel confident about a proof. Contrary to what its name suggests, a proof is just one step in the direction of confidence, but that is how actual mathematicians construct actual mathematics. In other words, social processes are what gives mathematicians confidence in their proofs. The same social processes do not exist in computer science for proofs of correctness and so they do not warrant the same confidence.

De Millo et al. focused merely on the confidence that we can have in the correctness of programs, but the lack of social processes typical to mathematics could be even more significant. As illustrated by Lakatos et al. [1976], the attempts to produce a proof and refutations of earlier imperfect proofs is what builds mathematical knowledge. A failed attempt to construct a proof lets us find issues with our definitions and theorems and encourages us to revise those. If we treat proofs merely as certifications, as computer scientists might, we lose an invaluable force that shapes the our knowledge.

It is difficult to trace the direct effect that the De Millo et al. paper had on computer science. As noted by MacKenzie [2004], the controversies triggered by the paper likely contributed to the decrease of funding for formal verification, especially from the US military. However, many further developments around program correctness address the issues raised by De Millo et al., even if they are not a direct response to the debate triggered by their paper. Two approaches that we discuss next reflect two sides of the mathematical culture.

2.2 Mechanistic and intuitionalist culture of proving

The lack of social processes in proofs of programs can be addressed in two ways, depending on who do we want to convince about program correctness. If we want to convince a human that a program is correct, we need social processes, though not necessarily exactly the same as the ones that mathematics has. If we want to convince a computer that a program is correct, we need

to accept that program correctness proofs are of a different epistemological nature. For example, mechanised proofs are perhaps better seen as technological artifacts [Turner and Angius 2017]. These two approaches can be seen as two sides of the mathematical culture of programming and we will refer to them as the *mechanist culture* and the *intuitionalist culture*.

The split between the mechanist culture and the intuitionalist culture can be traced (at least) to the birth of artificial intelligence and the first use of computers for automated theorem proving in 1950s. The Logic Theory Machine designed by Newell and Simon (which would eventually prove 38 of the first 52 theorems in Principia Mathematica) captured heuristic rules that humans use and was designed to understand how human beings reason. The work aimed at human understanding and the prover later produced new proofs, regarded as elegant by human readers. In contrast, methods advocated by logicians such as Wang [1960] used exhaustive search methods that were inscrutinizable, but were able to prove all of the 52 theorems much faster than the method of Newell and Simon [MacKenzie 2001, p.73]. As in the context of program correctness later, the proponents of the intuitionalist culture such as Newell and Simon focused on human understanding while Wang and other advocates of the mechanist culture focused on using computers in a way that is not accessible to humans.

The cleanroom software engineering methodology, developed by Dyer and Mills [[n. d.]] combines the intuitionalist mathematical culture, with its focus on convincing humans via social processes, with a managerial culture that shifts focus from individuals to organizations. The methodology proposes "a technical and organizational approach to software development with certifiable reliability". It aims to prevent defects, as opposed to debugging them later, by using software specifications and rigorous proof. However, the goal of a proof was to convince fellow team members that the program was correct. This was done through a social process akin to modern code reviews. The social processes that were lacking in proofs of program correctness are restored through the business organization, which recognized that additional time spent while reviewing (perhaps tedious) conformance to the specification will save time later. The process of proving in the cleanroom methodology is fundamentally human. "It is obvious" is an acceptable proof if everyone agrees. It does not provide an absolute certainty, but when practiced well, it does notably decrease the defect rate [MacKenzie 2004].

The proponents of the mechanist culture were not satisfied with proofs checked by humans and argued, as McCarthy did in 1962 that proofs of program correctness should, themselves, be checked by a computer program. Unlike the early work in AI on automatic theorem proving, which was able to obtain some, albeit limited, results automatically, verification of non-trivial programs required (and still requires) human input. The computer thus "does not act as an oracle that certifies bewildering arguments for inscrutable reasons but as an implacable skeptic that insists on all assumptions being stated and all claims justified." [MacKenzie 2004, p.272]

The first tool that was born from the mechanistic mathematical culture was the LCF theorem prover, introduced by Milner [1972]. The work on LCF was motivated by program correctness and, more specifically, compiler correctness proof [Milner 1979]. To simplify construction of LCF proofs, Milner's group developed a meta-language (ML) which could be used to develop proof procedures. Those were, in turn, checked to be correct by the ML type checker. This powerful combination gave rise to a family of interactive theorem provers that are still popular today (HOL, Coq, Nuprl [Constable 1986; Coquand and Huet 1988; Gordon 1988]) that directly follow in the mechanist mathematical tradition. However, the ideas developed as part of the ML metalanguage soon took a new form as a stand-alone programming language.

Mechanized proofs developed using LCF and similar tools are no longer the complex human constructs that mathematicians know as proofs that require social processes for their verification. Instead, they are a computer representation of a sequences of propositions where a prover program

can derive each proposition from earlier sentences, axioms or assumptions. This notion is close to the notion of proof in logic, with the caveat that we are now talking about computer representations.

2.3 Theorems and formal specifications of ALGOL

The critique of De Millo et al. focused on the notion of proof. A proof of program correctness is not the same kind of proof as a proof in mathematics. However, this is not the only discrepancy. The theorems proved by proponents of the mathematical programming culture are also quite different notions than the theorems proved by mathematicians. In programming, the structure of the theorems is mainly shaped by the programs they talk about and the programs, in turn, are shaped by the aim of building software that can be run to achieve some task. In contrast, mathematical theorems are shaped by the proofs, counter-examples and the preferences of the mathematical community which strives for simplicity and elegance. As a result, the specifications that computer scientists work with (and hence also the theorems about them) are either extremely complex or significantly simplified and, quite often, both of these.

The two mathematical sub-cultures can, again, help us understand how specifications are written. On one hand, the aims of the mechanist culture are to provide specifications that can be, at least in principle, checked automatically against an actual implementation. This requires that it is written in a fully formal language and that it is as complete as possible. On the other hand, the aims of the intuitionist culture are to explain key aspects of the program behaviour to a human. For this reason, the specification can use natural language where appropriate and does not necessarily have to cover everything. It can focus on a subset of the program that is expected to be tricky and cannot be skipped as "obvious".

An illuminating example is provided by the various attempts of formally defining the ALGOL 60 programming language that have been documented in detail by Jones and Astarte [2016]. The *Revised report on the algorithmic language ALGOL 60* [Backus et al. 1963] defines the syntax of ALGOL mathematically using the BNF notation, but the semantics of the language is defined in carefully-crafted English language. The report was produced under the influence of many cultures, but the main focus was to explain ALGOL 60 to humans including programmers working in the industry, as well as mathematicians.

McCarthy [[n. d.]a] followed his aim of defining a mathematical science of computation and defined a formal mathematical semantics of a subset of ALGOL named Microalgol. McCarthy "advocates an extension of [his] technique as a general way of describing programming languages," but keeps the language to a trivial subset (including goto, but omitting many other aspects of real ALGOL). Similarly, in his *An axiomatic basis for computer programming*, Hoare [1969] chooses to study properties of a very simple language (with assignment and iteration, but without goto) Hoare notes that there "does not appear to be any great difficulty in dealing with [most other aspects of ALGOL]" with the exception of jumps. He optimistically claims that "the practice of supplying proofs for nontrivial programs will (...) not be easy, [but] the practical advantages of program proving will eventually outweigh the difficulties."

Providing semantics for the full ALGOL 60 language was significantly more work. The operational semantics given by Lauer [1968] from the Vienna IBM laboratory is 67 pages and provided a motivation for follow-up work aiming to give simpler semantics, including simpler operational semantics Allen et al. [1972] and Vienna denotational semantics [Henhapl and Jones 1978] which is "simple enough to fit a book chapter." [Jones and Astarte 2016] The different sub-cultures can, perhaps, be seen in the notations used. The Oxford denotational semantics [Mosses 1974] uses mathematical symbols and is thus written to explain ALGOL 60 to a mathematician whereas the Vienna denotational semantics [Henhapl and Jones 1978] is written in the VDM language and is more accessible for mechanical processing, although the tool support only provided basic checking.

Following one strand, it took a decade before a formal specification of a programming language was used to develop a mechanically verified implementation [Moore 1989] but a very small low level language and even longer before the same was done for a realistic main-stream programming language [Leroy 2009]. Following another strand, the use of simplified mathematical models (in the style of Microalgol) has become a main-stream academic method for talking about programming language features [Igarashi et al. 2001; Milner 1999]. In both cases, we fact the problem that theorems about programs are determined by the programs, which tend to be structurally complex, rather than by a community of mathematicians which strives for simplicity. The case of formal language specifications is illuminating, because a compiler is structurally very simple (a semantics preserving transformation), yet the task of formally specifying it is huge. For more structurally complicated software systems, the situation is more difficult as we will see in Chapter X.

2.4 Mathematical models vs. physical systems

The next limitation of the idea of program verification as advocated by the mathematical culture was pointed out by a philosopher James Fetzer. The idea came from an outsider rather than a member of any specific programming culture, but Fetzer [1988] does not question the validity of proofs or the structure of theorems, but the very idea of proofs about programs. Although both programs and theorems are syntactic structures, the meaning of a program is that it controls a physical machine. Programs are not purely mathematical entities, but causal models that "implement a [certain] function in a form suitable for execution by a machine". Mixing the mathematical reasoning and physical effects is a *category mistake*.

Depending on how we choose to treat the nature of the machine that executes a program, a program verification can mean two things. The first option is that the machine is an abstract machine and a program verification is mathematical activity. In this case, we have mathematical proofs, but they are about mathematical models, not actual programs that run. The second option is that, the machine is an empirical description of a physical system. In this case, we are only able to obtain (more or less reliable) empirical knowledge about program behaviour, but never a *proof*.

Fetzer points out that there is a gap *somewhere* between the mathematical knowledge that includes proofs about programs and the physical reality. A concrete example has been explored by Avra Cohn who worked on the team aiming to formally verify the Viper microprocessor. Documenting this experience, Cohn [1989] notes that "a device can be described in a formal way, and the description verified; but (...) there is no way to assure the accuracy of the description. Indeed, any description is bound to be inaccurate in some respects, since it cannot be hoped to mirror an entire physical situation (...); a model of a device is necessarily an abstraction." In other words, even if we decided to formally verify a system at all its levels, ranging from the high-level programming language to the low-level chip it runs on, we will inevitably work only with abstract models that cannot give us certainty about the system. This does not make formal verification less useful in practice, but it means that it should not have a unique status among other methods for building correct systems.

We saw earlier that the intuitionist and mechanist cultures developed methods that (perhaps indirectly) addressed the issue of social processes raised by DeMillo, Lipton and Perlis. Similarly, the way formal methods are typically used by the two sub-cultures (perhaps indirectly) address the issue of category mistakes as raised by Fetzer.

The mechanist sub-culture understood the importance of minimizing the gap between the program itself and the proof about it. Gradually, this led to the development of systems where the proof and the program are written in the same programming language. A good example is the CompCert compiler, where the program is written using a theorem prover alongside its correctness proof. This approach avoids the category mistake by changing the nature of the proof from an abstract mathematical entity to a computer program, or a *technical artifact* [Turner and Angius

2017]. The correctness of the mechanical proof is decided by a computer and it is subject to the same limitations as any other program. As long as we do not treat it as a mathematical proof, we avoid making a category mistake.

The second approach is to be more conscious about working with an abstract model. The way the intuitionist sub-culture builds mathematical models is that they are often very simple (following McCarthy's Microalgol language). Proofs about such models do not provide guarantees about the reality (and the culture is aware of that), but the attempt to develop a proof often leads to useful insights about the system. Although we cannot guarantee a correctness when using a simple model, a failure to prove correctness often points at an issue that also exists in the actual system and can be empirically observed. This way, a poof becomes an invaluable debugging tool and we avoid making a category mistake by recognizing that we work with models.

As with the critique by DeMillo et al., it is difficult to trace any concrete influence by Fetzer's critique. However, we can again make the *post hoc* claim that the mechanist and intuitionist sub-cultures evolved in ways that alleviate the critique. This suggests that Fetzer, as well as DeMillo et al. before, were pointing out a problem that the community was perhaps sub-consciously aware of.

3 THE DECISIVE MEETING OF CULTURES

Computer programming of 1950s and early 1960s was often perceived a black art that was done by programmers, who were born to be programmers, using arcane tricks that had to be reinvented with every single project. We label this focus on individuals, clever tricks and craftsmanship as the hacker culture. In 1960s and 1970s, two cultures aimed to make programming less ad-hoc. The first was the managerial culture, which was focused on controlling the development process, and the computer personnel involved, and achieving predictable and reproducible results. The second was the mathematical culture, which saw programming as a mathematical discipline that should be done in a systematic way using the devices of logic.

The mix of the three cultures gave birth to a programming language as a stand-alone object that can be described and studied independently of any of its implementations. The proponents of the mathematical culture argued that programming languages are mathematical objects and should thus be studied using the resources of logic. This did not have immediate influence on how programming was done in practice, but ideas such as structured programming were communicated across cultures and developed a managerial interpretation.

The idea of studying programs and programming languages mathematically turned out to be harder than initially thought. Computer programs differ from mathematical theorems in many ways. They are created to be executed and tend to be complex, while proofs about programs are not interesting enough to develop the same social processes as mathematical proofs. These issues have been addressed in ways that can be broadly categorised as two sub-cultures. The mechanist sub-culture believes that proofs should be mechanical, i.e. written for and checked by a machine. The intuitionist sub-culture believes that proofs should mainly guide human understanding and can be about an illuminating subset of the full system. Nevertheless, both of these approaches have to abandon the idea that we can gain mathematical certainty about the correctness of a program.

An interesting fact about the mathematical culture has been suggested by Ensmenger [2012, p.117] who says that "[it's] rise was anything but inevitable." He also argues that "the advocates of theoretical computer science pursued a strategy that served them well within the university, but that increasingly alienated them from their colleagues in industry." This suggests that the early developments contributed to the long-lasting division between the mathematical and managerial culture of programming. It is also entirely possible to imagine that, given a slightly different arrangement of actors in the early days, the notion of a *programming language* could have been born not as a mathematical object, but as something quite different. Perhaps, programming languages

could have been business entities that are ignored by the academia (there is no university department for the railroad, radio, airplane or television technologies [Ensmenger 2012, p.115]) or an entity studied by designers.

LECTURE: MULTICULTURAL NOTION OF TYPES

In the previous lecture, we saw how the different programming cultures interact while aiming to produce more correct and more reliable software. We saw that the very idea of programming language was born at the intersection of multiple cultures and that ideas such as structured programming were born in one culture but acquired meaning in another culture. We also saw cases where ideas born in one culture provoked criticism from other cultures and, eventually, found ways of avoiding it.

In this lecture, we return back to the end of 1950s, but rather than following the mathematization of programming and its controversies in general, we follow a strand through the history that is centered around the concept of types as it appeared in programming.

Teacher: Let's start with the FORTRAN system. Looking at the 1956 reference manual for the FORTRAN automatic coding system for the IBM 704[1], does it tell us anything about types?

Alpha: What do you mean by types? There is 32 types of statements, two types of constants and two types of variables, three basic types of decimal-to-binary and binary-to-decimal conversions... But I suspect that you are actually asking about the *modes* of functions and their arguments.

Arguments of a function can be either fixed or floating point; similarly the function itself may be in either of these modes. You indicate that a function is in the fixed point mode by starting its name with the letter "X" and it also has to end with "F" to indicate that it is a function.

Epsilon: I'm glad this is not how we program today. Having just a floating point and a fixed point as the only two primitive types is funny, but how can you write any code without some form of custom data types, that I really do not understand.

Omega: If you looked outside of your scientific programming bubble, you would discover that this is something we in the data processing industry are already working on. The Flow-Matic language has a concept of records. You can define that your record consists of multiple fields such as product number, name and price, which virtually eliminates your coding load![2]

Epsilon: I have read non-ACM articles on business data processing, but they suffer from one basic fault: They fail to report fundamental research in the data processing field. I'm not interested in how a particular Univac system implements records, but in the fundamental ideas behind the problem.[3]

Tau: The fundamental research that we need to develop is a mathematical theory of computation. This way, we can focus on fundamental ideas behind computing rather than on bits in computer memory. As for records, we can model basic numerical data spaces as sets and then construct derived data spaces using set operations such as product to model records.[4]

Teacher: I'm sorry to interrupt this interesting debate, but could we please clarify our terminology? So far, we used the term mode, data type, type, record and even data spaces. Are these the same thing, or are they a different thing?

Epsilon: Oh, I think they are all just natural English words to use in this context, but I believe we are talking about the same idea here, so let's settle on the term 'type'.[5]

Tau: The concept of a type seems fitting. It is familiar to logicians who developed types to avoid paradoxes in logical reasoning and there is a high degree of commonality between the concept

[1] [Backus et al. 1956]
[2] [Univac 1957]
[3] [Postley 1960]
[4] [McCarthy 1959]
[5] [Martini 2016]

in logic and the concept in programming. In both, types can prevent meaningless constructions. In programming, they also determine the method of representing and manipulating data on a computer. Furthermore, the types we are interested in are already familiar to mathematicians; namely, sets, Cartesian products, discriminated unions, functions and sequences.[6]

Alpha: I can see how sets are a nice way of thinking about numbers and records, but what about pointers? That's a pretty important type if you ask me!

Tau: You really should stop thinking about programming as fiddling with bits. What do you need pointers for? To represent data structures such as lists or trees. Those can be mathematically modeled as recursive structures, which gives you higher level of abstraction, more suitable for reasoning about programming.

Epsilon: I don't think language designers should think of types as sets.[7] While this way of thinking may work well in mathematics, I feel that it misses an important aspect of types. Rather than worrying about what types are, we should look at how they are used. In programming, type systems check for two things that I will call *authentication* and *secrecy*.

The authentication problem is to make sure that program operates only on values that are valid representations of its expected inputs. This is what types as sets can model well. The secrecy problem is to make sure that programmer uses only using provided procedures, rather than relying on the representation. The secrecy problem is not something that "types as sets" can model.

Omega: I came across the idea that you are calling secrecy before, but under the name *information hiding*[8]. This is a great concept for managing a large team of programmers, because it lets you decompose a large system into smaller parts that can be managed and developed independently. This is a much underappreciated aspect of types.

Epsilon: As a matter of fact, we've been working on a programming language that focuses exactly on this idea. We call this aspect "abstract data types" and implement it through a language mechanism called 'clusters'. The data representation is hidden to a cluster and you can only manipulate it through the operations that the cluster exposes.[9]

Lambda: I also think that types are important for hiding representation, but we should be doing this based on solid logical foundations. After all, we decided to use the term 'type' because of the work done on types in logic. I have been working on an extension of simply typed lambda calculus that lets us *prove* that programs do not rely on their data representations.[10]

Teacher: I do worry a little that we might be doing ourselves a disservice by unifying all those different ideas under the name of 'type'. Is it really possible that one programming language could combine all those distinct aspects in one language feature?

Lambda: I'm glad you asked! We've been working on a programming language called Metalanguage (ML) for the LCF proof assistant that combines pretty much all of these ideas. It is inspired by the lambda calculus and uses a strict type system for both secrecy and authentication.

A theorem is an abstract data type; pre-defined values represent the axioms and operations on the theorem model the inference rules. Secrecy then gives us a guarantee that your proofs are constructed correctly. It turns out that the language is also great for general purpose programming

[6] [Hoare 1972]
[7] [Morris 1973]
[8] [Parnas 1972]
[9] [Liskov and Zilles 1974]
[10] [Reynolds 1974]

and then authentication gives you a compile-time guarantee that your functions are only applied to compatible values. We also have set-theoretical data types such as records, functions, discriminated unions and recursive types...[11]

Gamma: Hang on *Lambda*, your focus on compile-time guarantee is changing the focus of our discussion. Types are useful for describing the structure of your data even without the compile-time checking. I think that by mixing the two topics, we are making the discussion confusing.

Lambda: I don't agree. The most amazing aspect of types is that they make some of the ideas from work on formal verification, which we discussed in the previous lesson, available to a wide audience. Type systems are by far the most popular and best established lightweight formal methods.[12]

Teacher: Let me just point out that we now also started using the term *type system*. Now might be a good time to clarify what we mean by that. Can anyone propose a definition?

Lambda: Let me try: A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.[13]

Alpha: This is a neat definition, but just like all other definitions that academics like you propose, it excludes a large number of type systems actually used in practice. Some type systems serve more as a documentation tool, some are merely a mechanism to support better tooling and some use types for other things than classifying values.

Teacher: That is a bold claim and I'm sure *Lambda* is eager to comment. I would like to return it in a few minutes. Perhaps our zeal for unification was too great and we produced a monster that cannot be even defined!

Now, I would like to better understand how types work as a formal method. How do types relate to ideas that we talked about in the last lecture such as specifications and proofs about programs?

Lambda: Type systems guarantee that your program behaves in a certain way. The types play the role of a specification that defines what values can a function accept as arguments and what values it is allowed to produce. In the previous lecture, *Tau* introduced the notation $\{P\}s\{Q\}$. If s is a function call, then the pre-condition P and post-condition Q restrict the values that the function can consume and produce. Except that it is now an inherent part of the programming language, rather than an external mathematical method for reasoning about programs.

Epsilon: The relationship seems quite clear to me, but saying that a function produces, say, a list of numbers is not very detailed specification, is it? It might prevent some basic errors, but it is not really specifying what the program does.

Lambda: You can do much more if you use a more powerful type system. Recent work on refinement types and dependent types allows you to write more detailed specifications as types. The returned list satisfies `IsSorted` predicate where $\texttt{IsSorted}\,(x :: xs) = x \leq^* xs \longrightarrow \texttt{IsSorted}\,xs \longrightarrow \texttt{IsSorted}\,(x :: xs)$, i.e. if x is smaller than all elements in xs and xs is sorted, then the list $x :: xs$ is also sorted.

Tau: Yes, yes, we had this discussion before. I expect that *Alpha* will now again remind us about *NaN* values and *Lambda* will propose a more precise type.

Philosopher: The situation is different now. The property is now a part of your program, rather than a separate mathematical entity. This means that we work in the realm of empirical knowledge, but at least we are not making a category mistake like we did in the previous lecture. That said, we

[11] [Gordon 1988]
[12] [Pierce 2002]
[13] [Pierce 2002]

are still just building two technological models of the real world and the type system ensures that they are compatible...

Omega: Are you saying that, if we are using type systems as lightweight formal method, then we basically have to write the same thing twice? That sounds quite inefficient and expensive.

Philosopher: This aspect of types might actually be quite valuable. In the previous lecture, *Gamma* pointed out that program proofs do not have the social processes of mathematics that ensures their correctness. Well, with type systems, people often review, check and talk about the types, so it gives us an entity that encourages social processes and this, in turn, might make them more useful than any formal property they have.

Lambda: In my experience, your point about social processes is a very valid one. People certainly talk about types. However, I also want to comment on what *Manager* said. In many of the dependently typed languages, you do not write everything twice, because the environment can often help you write code based on the type, which is your specification.[14]

Alpha: It sounds like we might avoid some of the duplication but then, isn't it easier to just write your program rather than trying to come up with a formal property that would fully specify what it does? I can see how that's manageable for a sorting function, but how do you formally specify, say, an accounting system or an anti-ballistic missile system?

Teacher: We will return to large systems in the next lecture, but we said enough about types as a lightweight formal method for now. I wanted to get back to a point that *Alpha* made earlier. Are there programming languages with type system that does not match *Lambda*'s definition?

Tau: In logic, type systems were never about "classifying phrases according to the kinds of values they produce". They were introduced to eliminate logical paradoxes. Later, types turned out to be useful in work on the foundations of mathematics. The equivalence between programs and proofs means that you can use proof assistants like Coq to construct proofs in the same way as you construct programs. In those systems, types are logical propositions, not kinds of values.

Alpha: Even in the programming world, types are not sets. I mentioned pointers already, but I found one more example. Programming languages that use effect system to track how your program accesses memory, you can get types such as `int` & {`read` ρ}. This says that the expression returns `int`, but might also read information from a memory location ρ. This type is clearly more than just a set of numbers.

Lambda: You are right. In this case, it is better to think of types as relational constraints on the compiled code.[15] The property denoted by the type in your example is that, the state of the world after running it will be the same as the state before (because it does not write) and that the resulting value will not change if you modify any memory location *outside* of the region ρ.

Teacher: It seems that we can find different ways of interpreting what types denote, but other than that, the definition given by *Lambda* was quite accurate.

Alpha: I have one more counter-example. In TypeScript, the type system does not prove the absence of certain behaviors. It was intentionally designed to be simple and usable, but that leaves certain tricky holes in the type system. However, this is not a problem for TypeScript. There is a

[14] [McBride 2008; Petricek 2017]
[15] [Benton et al. 2006]

runtime checking, so the system is still safe and types work great as a tool and documentation for programmers in practice.[16]

Lambda: I turn aside with a shudder of horror from this lamentable plague of unsound type systems! The very point of type systems is to guarantee soundness. If they cannot fix their system, they should call it a hint system, because it certainly is not a type system![17]

Alpha: Well, it is a type system in all other respects. It classifies phrases according to the kinds of values they compute, it provides an abstraction mechanism, it serves as a documentation and enables tooling such as auto-complete...

Gamma: I also would not be that strict. In languages like ML, the static type system makes guarantees that allow eliminating runtime checks, but that is just one particular design. Languages like Clu in 1970s had static type systems, but relied on runtime checks for some of their guarantees.[18]

Lambda: Excuse me, but referring to languages from the 1970s is not particularly convincing. Surely, we learned how to do things better!

Gamma: You have a very narrow view of what types are. Plenty of dynamically typed languages use types and check their use at runtime.[19]

Lambda: I do not disregard those, but we were talking about type systems and I believe the whole point of type systems is to guarantee whole-program safety that allows you to eliminate runtime checks and makes your program correct and efficient.[20]

Gamma: There are type systems for dynamically typed languages too. Optional typing allows you to statically type check some part of your dynamic program without affecting its runtime semantics. Gradual typing allows you to gradually introduce types with static checking where possible and dynamic checking where needed.[21]

Alpha: This is funny. If you look at languages like TypeScript, you can almost say that there are two type systems in place. First, there is a type system of the TypeScript languages. This gives a lot of help to the programmer, but makes only weak guarantees. Second, there is a hidden type system in the runtime system that runs the generated JavaScript. This makes stronger guarantees and it is used to run the code more efficiently, but the programmer never even sees it. So, the many uses of types are split between two type systems!

Teacher: It seems to me that if we look at all the different ways in which people think and talk about types, there is a strong relationship between all of them, but we keep failing to find a definition that would cover them all. Is it possible to talk about types without having a clear conception of what they are?

Tau: The fact that people do not have a clear definition makes them make confused claims[22], so I think that the lack of a clear definition is a real obstacle to the progress in programming language research.

Lambda: I agree. If I could change the past, I would make sure that we use a completely different term for types arising from the logical tradition and types used by engineers for data abstraction.

[16] [Cavanaugh 2018]
[17] [Lakatos et al. 1976; Petricek 2017]
[18] [Liskov and Zilles 1974]
[19] [Gabriel 2012]
[20] [Petricek 2015]
[21] [Meijer and Drayton 2005]
[22] [Gabriel 2012; Kell 2014]

Epsilon: I wonder what we would lose if you did that. The fact that those overlapping concepts use the same name likely makes it easier to exchange ideas between different communities. Would a language like ML, which brings all those ideas together, be ever born?

Tau: That might be a valid point, but how do you want to talk about something that you cannot even clearly define?

Gamma: That might be a problem for a mathematician, but humans do that using natural language very day. The meaning is use. The meaning of a 'type' is its use in the practice and theory of programming languages.[23]

Lambda: This is a very clever-sounding answer that does not actually say anything. How do you expect that science can progress with such lazy thinking that says 'anything goes'?[24]

Alpha: You might not be able to directly compare and synthesize different theoretical contributions concerning types, but that is not what programming is about! In the end, we care about what kinds of things we can achieve using types and those effects are something you can assess regardless of your definitions. Many different kinds of types can be used to build developer tooling, so just look at the end results!

Teacher: You made a very interesting point. You are arguing for a very pragmatic, applied approach to research and you formulated it as a moral principle, but it seems to be very fitting as a descriptive statement about the history of types. It does seem that programmers and computer scientists just exploit what they find useful for the design of more elegant, economical and usable artifacts.[25]

NOTES: MULTICULTURAL NOTION OF TYPES

Programmers today, especially those belonging to the logical programming culture, are keen to trace the notion of types in programming all the way back to the simple theory of types by Church [1940] and even the theory of types by Russell [1908]. In mathematical logic, types were used to avoid paradoxes of the kind "class of all classes that do not contain themselves as elements". Church [1940] integrates Russel's theory of types with his work on λ-calculus, but this is still work on foundations of mathematics. However, the computational interpretation of the system that most people talking about types in programming use today was developed much later.

In the meantime, types appear, somewhat by accident, in the engineering programming culture at the end of 1950s. As noted by Martini [2016], the early writing on programming including the aforementioned FORTRAN reference manual (Bacus, 1956) uses the term 'type' in its everyday English sense, often interchangeably with words such as 'kind' or 'class'. In a more formal description of how to write functions accepting fixed-point (integers) and floating-point numbers, the manual talks about 'modes'.

The first modern usage of the term 'type' appears in the Algol 58 report (Perlis, Samelson, 1958). Martini [2016] documents that types in the Algol 58 report appeared during a meeting between the ACM group and European group working on Algol in Zurich at the end of May 1958. He also points out a remarkable fact that "the technical term appears to be just a semantical shift from the generic one; in particular, there is no clue that in this process the technical term 'type' from mathematical logic had any role". Over the next two decades, the term 'type' will acquire a multitude of new meanings and the mathematical culture will meet the engineering culture, arguably culminating with the development of the ML language.

[23] [Petricek 2015]
[24] [Feyerabend 1975]
[25] [Martini 2016]

1 MULTITUDE OF TYPES

Types of Algol 58 and Algol 60 were very basic compared to what we know as types today. The two primitive types were integer and floating-point and the only derived type was an array. This was soon seen as a limitation and the first line of work on types focused on developing better ways of structuring data. However, types also soon acquired new meaning as a tool for abstraction (or information hiding). In this section, we follow these two developments leading to the great unification of cultures from which the ML language was born.

1.1 Structuring data with types

Both FORTRAN and Algol 60 were born out of the engineering programming culture. They were designed for scientific applications and focused on representing numerical data. Arrays enabled modeling of vectors and matrices, but textual data were poorly supported and there was no support for richer composed data structures [Priestley 2011, p.244]. In contrast, language designers belonging to the business data processing culture were very much aware of the need for structured data. Flow-Matic developed by Grace Hopper and announced by the Remington Rand Corporation in 1957 [Univac 1957] had a notion of records that was later incorporated into COBOL [Bosak et al. 1962].

Although the idea of records was already in the air, the path toward integrating them to an Algol-like language was more complicated. As discussed in the previous chapter, Algol was the result of mathematization of computer programming and it had strong roots in mathematical and engineering cultures as represented by the Association for Computing Machinery (ACM). Members of the ACM were aware of some work done in the data processing industry, but felt that the articles written by the proponents of the managerial culture "failed to report fundamental research in the data processing field" [Ensmenger 2012, p.174]. The ACM members were not interested in "how someone else solved his payroll problem". "The solution is almost sure to be too specifically oriented to the company for which it was solved and [lacks] general applicability." [Postley 1960]

Rather than just copying records from COBOL, Algol had to wait until a proper theory of data types was developed as part of the mathematical culture of programming. The first step was made by McCarthy [1959] as part of his work on mathematical theory of computation, who acknowledged that the "lack of such a formalism is one of the main weaknesses of Algol" and that business data processing languages such as Flow-Matic and COBOL "have made a start in this direction, even though this start is hampered by concessions to what the authors presume are the prejudices of business men."

McCarthy proposed "a way of defining new data spaces in terms of given base spaces". His data spaces where modeled as sets and could be combined using the usual mathematical operations such as Cartesian product and union of (non-intersecting) sets. McCarthy also noted the importance of recursive definitions. As an example, the data space S defined as $S = A \oplus S \times S$ is the model of S-expressions over the alphabet A. This work was later refined and extended by Hoare [1972], who started using the term type. Hoare presents his ideas using a hypothetical un-implemented programming language with concrete syntax for type definitions and includes records, discriminated unions, arrays and sequences, powersets and recursive type definitions, together with their set-theoretical models and notes on implementing them.

It is worth noting that the term type was still not universally accepted in 1970 and that the idea of adding new user-defined types to a language was not always seen as an inherent feature of a language, but as a way of extending a language in order to produce a new language, suitable for a particular domain. Both of these are exemplified by the paper by the work of Stephen Schuman and Philippe Jorrand who published a paper [Schuman and Jorrand 1970] on definition mechanisms in

extensible programming languages. The paper proposes a language where new *modes* of values can be defined using *mode constructors*. Mode constructors allow introducing new records, unions and sequences but also pointers (as opposed to more set theoretical recursive definitions). In line with the aim of being extensible, the proposed language also introduces an extensible syntax via a macro system.

Mode declarations, including references, records (known as structures) and unions were incorporated into Algol 68, which has been criticized by many, including its design committee members, for abandoning the simplicity and elegance of Algol 60. One of the critics, Niklaus Wirth went on to develop the programming language Pascal [Wirth 1971], which switched the terminology back and started referring to custom data structures as *types*. Pascal had records, (unchecked) variants, arrays, range types and pointers.

The various proposals for programming languages that allow custom data types was, in part, motivated by the aim of producing language that would be also suitable for business data processing. However, the work was mainly done at the intersection of engineering and mathematical cultures. That said, the mathematics involved was mostly set theory which served as a model for records, unions and recursion. References to types as known from mathematical logic appeared only later, which might also explain the confusing terminology of types, modes and data spaces. There also seems to be an interesting borderline between more mathematical and more engineering designs. The mathematical culture favored recursive type definitions, which have a set theoretical model, while the engineering culture favored pointers or references, which are closer to how the machine operates but breaks the otherwise elegant mathematical theory.

1.2 Hiding representation with types

The debate in the previous section focused on understanding what values a data type represents and encodes. However, types will also soon become useful as a mechanism for implementing information hiding. The idea of information hiding was introduced by Parnas [1972] in a paper "On the Criteria To Be Used in Decomposing Systems into Modules". Parnas proposes a way of structuring programs into independent modules that expose a number of public operations, but hide the representation of data they use. This makes the system easier to change, understand and enables independent development of individual components.

The view of types as a tool for information hiding has been put forward by Morris [1973] in a paper aptly titled "Types are not sets". Morris recognises the dominant mathematical culture that focuses on what types are and summarizes the work: "there has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas." Morris proposes to focus on what types are used for: "rather than worry about what types are I shall focus on the role of type checking. Type checking seems to serve two distinct purposes: authentication and secrecy."

As discussed in the lecture, the problem of authentication is to guarantee that only values of the right type can be submitted for processing. The problem of secrecy is to ensure that "only the [given] procedures can be applied to objects of [a certain] type." Morris proposes a programming language with a support for modules that can contain both private and public data and operations. Interestingly, the checking is implemented through a dynamic type system that seals and unseals data into opaque representations at runtime.

The notion of type fully acquired this new meaning when Liskov and Zilles [1974] introduced the concept of "abstract data types". The paper introduces the Clu language and talks about *clusters*, which implement the idea of secrecy through abstract data types. As before, the abstraction is, in

part, enforced by runtime checking, but the paper references personal communication with John C. Reynolds noting that his recent work "indicates that complete compile-time type checking may be possible."

The mention of John C. Reynolds was likely referring to his work that appeared one year later in a paper "Towards a Theory of Type Structure" [Reynolds 1974]. The paper, finally, brings together the development of types in the engineering culture with types known in mathematical logic. Reynolds "introduce[s] an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show[s] that the semantics of this language satisfies a representation theorem." The representation theorem says that the value obtained by evaluating a program does not "depend upon the particular representations used to implement its primitive types."

The paper by Reynolds follows the method of many theoretical programming language papers of the present time. It presents a simple extension of the λ-calculus and proves a property about it. However, an interesting point to note is that the fundamental representation theorem talks about information hiding, rather than about values produced by well-typed programs as is the case in much of the present-day literature on programming.

1.3 The great unification and ML

So far, we traced the history from modes as used by early computer hackers to annotate what a block of memory represents to data types modeled by sets and the history of abstract data types which extended the concept of type with applications for information hiding. The last strand is the integration of types as known in mathematical logical and, in particular, in simply typed λ-calculus.

Researchers studying mathematical foundations of programming languages were well aware of lambda calculus in the 1960s. As documented by Astrate [2017], Christopher Strachey had been introduced to λ-calculus by Roger Penrose around 1958 and employed Peter Landin, who had also been interested in λ-calculus in his consulting business from 1960–1954; During this time, Landin produced a series of papers on semantics that defined semantics of ALGOL 60 using λ-calculus [Landin 1964, 1965a,b, 1966a]

In 1966, Landin wrote "The next 700 programming languages" [Landin 1966b] which makes an interesting logical shift. Rather than using the λ-calculus to give the semantics of an imperative language, the paper proposes the design of a family of languages (ISWIM) *based* on the λ-calculus. ISWIM remained unimplemented and it is not clear whether Landin intended the language to be dynamically or statically typed, but it provided the foundations for a new family of functional programming languages. In particular, ISWIM influenced the design of the ML language, which first appeared around 1973 as a meta-language for the LCF theorem prover. ML was used to implement proof tactics (and higher-order functions for combining tactics) that can be interactively invoked while constructing proofs [Gordon 2000].

The strict ML type system was essential for its application inside LCF. The ML meta-langauge was used to construct theorems by applying inference rules to existing theorems and axioms. A theorem was represented by an abstract data type. The strict ML type system statically guaranteed that the secrecy of abstract data types cannot be violated by the programmers which, in turn, meant that it was only possible to construct theorems with proofs. Milner [1978] provided a description of the ML type system, the type inference algorithm and proved soundness using the, now standard, formulation that "well-typed programs cannot go wrong".

The ML type system brings together aspects from many different programming cultures. It uses types to track how are values stored in memory, which traces back to the hackers implementing modes in FORTRAN. It suppers data types such as pairs and sum types (unions) which were born at the intersection of engineering and mathematical cultures. It employs abstract data types, which were born from the engineering tradition under the influence of information hiding, which also

had notable applications in the managerial culture. Finally, the system was structured using the ideas about types that date back to Church [1940].

Using the philosophy of scientific revolutions proposed by Kuhn and Hacking [1962], it would be tempting to see the era before the birth of ML as pre-scientific and the great unification of ideas on types that happened with ML as the birth of a normal science. This would not be entirely unjustified: there was no consensus on a particular theory of types in the pre-scientific era and the birth of the ML paradigm defined a set of methods and the kinds of questions that it can answer. However, this would give only a very narrow view of types, as understood by a particular strand of the mathematical programming paradigm. In a wider programming culture, types soon acquired even more meanings and developed in directions that are incommensurable with the ML way of thinking. We first follow the ML line of work on types before looking at the diversification of the notion of type that happened following the development of ML.

2 TYPES AS A LIGHTWEIGHT FORMAL METHOD

Although the ML approach to type systems was not the last word to be said on types, it did provide the basis of a paradigm, at least for the mathematical programming culture. The idea that "well-typed programs do not go wrong" was sufficiently open-ended. What other kinds of runtime errors could we eliminate using a more advanced type system? The ML approach also provided a clear methodology. We need to give a formal semantics of the language, define the type system and prove that certain things do not happen for well-typed expressions.

The popular textbook "Types and Programming Languages" (Pierce, 2002), which follows the aforementioned tradition, provides the context by noting that "modern software engineering recognizes a broad range of formal methods for helping ensure that a system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior." Such formal methods include, on one hand, sophisticated and difficult to use methods and, on the other hand, lightweight methods that are easy to use, but have modest power. Pierce then presents types in the general context of the mathematization of computer science discussed in the previous lecture by saying that "by far the most popular and best established lightweight formal methods are type systems, the central focus of this book."

2.1 Evolving the ML tradition

Documenting the history of the work on types in the ML paradigm is a topic for a separate paper. A good basic overview is provided by the collection of papers edited by Pierce [2005]. The first group of papers makes types for various functional languages more precise and capable, for example by tracking how a program uses memory. The second group extends the scope of types to both large-scale systems and low-level programming languages such as the assembly language. There is, however, an interesting shift in the way types are used as they get more expressive that is worth discussing in more detail.

Seen as a lightweight formal method, the ML type system is very basic. ML guarantees that abstractions are not violated and that expressions evaluate to a value of a right type, but that can be hardly seen as a system specification. In order to allow expressing more complex specifications, various ML extensions have been proposed and developed. Effect and coeffect [Petricek et al. 2014; Talpin and Jouvelot 1994] systems allow tracking what a program does to and requires from the environment in which it executes; refinement types allow narrowing down the set of values a type can represent and session types allow specifying how messages are exchange over network in a communication protocol. Another direction has been taken by the Haskell language and, especially, it's implementation in the GHC compiler. The language does not have special-purpose extensions,

but a family of general purpose type-level mechanism that can be used to express very detailed specifications using types.

The challenge with all those extensions is always the same. As the types get more complex, it becomes harder to keep the usability afforded by the type inference and simplicity of type checking as available in the original ML. The type system may require more annotations, explicit type specifications; it may become intractable; at the very least, it becomes harder to use.

However, the principle that a type system should be decidable without explicit help from the programmer has been a rarely questioned part of the paradigm until early 2000s when practically-oriented dependently typed programming languages started appearing. In dependently typed languages such as Agda and Idris, *terms* can appear as part of *types*, making it possible to easily express arbitrary logic in types. A type `Vect` n might represent a vector of length n and a function appending two vectors has a type `Vect` $n \rightarrow$ `Vect` $m \rightarrow$ `Vect` $(n + m)$. The consequence is that type checking becomes, in general, non-decidable and typically requires more detailed type annotations. A programmer typically writes a full type of a function and might also need to provide additional hints for the type checker inside the implementation to guide it.

As we will see later, there are a number of directions in which work on types evolved in 2000s. Dependently typed programming languages represent the direction taken led by the mathematical culture. They take their inspiration from dependent type theories, such as Martin-Löf's intuitionistic theory of types, which were designed as a basis for constructive mathematics and so they do, as a matter of fact, follow the tradition started by Bertrand Russel.

2.2 Specifications and social processes

Types can be seen as a lightweight specification of what the program does, but they are an inherent part of a program and they are written by the same person who writes the implementation. The type checker also guarantees that the implementation is aligned with the specification, which is often not the case with formal specification that relies on checking by humans. It also means that the specification is written alongside the implementation. Other efforts at producing a formally verified software typically require that the specification is written upfront (and often also by a different team).

The development of dependent types makes it possible to produce a more detailed specification, but, perhaps more interestingly, it also changes how we work with the specification. In ML, the type inference can infer the types, or specification, for you. In Agda, you have to write the specification yourself. However, the editing tools for Agda can often assist you when writing the implementation and, to an extent, infer the implementation for you. In both cases, there is an attempt at eliminating some of the overhead arising from the fact that we have to write both the specification and the implementation.

As a formal verification method, types can also provide a response to the two philosophical critiques of formal verification that we discussed in the previous chapter, that is, the lack of social processes and the fact that formal verification is a category mistake. Social processes involving types are not centered around proof checking, i.e. verifying that an implementation matches a specification. The type checker does that. However, types are still actively talked about in two ways. First, developers discuss types of common functions and what they mean. For example, the type of the monadic *bind* operation is $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$ and functional programmers will happily spend a lot of time explaining what this type means. Second, programmers often ask their peers "why is this code not type checking?" Both of these provide important social processes, which ensure that formal verification is not just a black-box that labels programs as 'correct', but a process with a human element, involving human checking and intuition.

The meaning of a program is that it acts on the real world through a machine. Claiming that a program is formally verified is a category mistake. We can formally prove that the source code (as a formal mathematical entity) satisfies a specification (another formal entity), but this does not yet talk about how a machine running the program will affect the world. Types face the very same limitations, but the way the community uses them typically avoids the category mistake. When we say "a program is well typed", we do not mean that it is guaranteed to do exactly what it should; we just mean that certain errors are eliminated. This is significant, because danger is a self-negating prophecy. Believing that a system may still be incorrect leads to actions that reduce this danger; believing that no mistake is possible can have the opposite effect. [MacKenzie 2004, p.301]

3 DIVERSIFICATION OF TYPES

The ML language unified previously disconnected strands of thinking about types and defined a dominant paradigm in academic research on types that is still active today. It also influenced industrial programming languages. The design of .NET generics available in C# was based on the work done in F#, which is a direct descendant of ML. The design of Java generics follows the same tradition although, interestingly, since Java generics are erased, they do not make it possible to fully eliminate runtime checks, which type systems in the ML tradition generally do.

However, a number of developments in the 2000s and 2010s take the idea of types in various directions. As discussed, dependently typed languages were born out of the logical side of the mathematical culture. However, the mathematical culture also revisited the idea that types are sets. More interestingly, the engineering culture also developed different takes on types that emphasize uses of types for documentation and tooling.

3.1 Understanding complex types as relations

The early mathematical work on types studied what types denote using notions from set theory. The problem of denoting became less prominent in the ML tradition. A proof that a well-typed program "does not go wrong" often relies on syntactic guarantees about how type checking works. This has been a sufficient answer for many, but it does not say what a type is. In the mathematical culture, the answer should be that a given type denotes a certain mathematical object. However, what object should it be if sets are not sufficient?

One interesting answer appeared in a work that extends ML types with information about physical units. This makes it possible to annotate numerical types with their physical units, so a function to calculate the speed based on a distance and a time will have a type $\texttt{real m} \rightarrow \texttt{real s} \rightarrow \texttt{real (m/s)}$. Units of measure can be concrete units, such as `m`, but also generic (i.e. unit variables), so the system allows types such as $\forall\alpha.\texttt{real}\ \alpha \rightarrow \texttt{real}\ \alpha^2$. This function takes a number with any unit and returns a number with a unit squared. What does the type denote? Based on the type, we know more than that it is a function that takes a number and returns a number.

All such functions f have the property that for any constant $k > 0$, it holds that $f(k*x) = k^2*f(x)$. Kennedy [1996] terms this property *invariance under scaling* and uses it as a basis for theory that interprets information about units as relations. This idea was later also useful for interpreting the meaning of types with effect annotations. As discussed in the lecture, a computation of type $\texttt{int}\ \&\ \{\texttt{read}\ \rho\}$ returns a number, but it may also read from a memory region ρ. Using a logical relation, we can capture how the computation depends on the world and how it affects the world, for example by saying that values in certain memory regions remain unchanged or may (or may not) affect the value of the computation.

This line of work keeps the properties of type systems following the ML paradigm, but it is interesting because it shows that fundamental questions about the nature of types can be reopened long after the discussion was seemingly closed. In case of physical units and effects, the

new applications motivated by issues from the engineering culture, forced the members of the mathematical culture to develop a new theory of what types are.

3.2 Unsound types for tooling and documentation

We said only little about types in programming languages such as Java, C# or C++. Those follow the engineering tradition and were not largely influenced by other cultures with the exception of Java and .NET generics. However, the engineering culture started using types in novel ways that influence and challenge how other cultures work with types. Type information in object-oriented languages is often used for providing automatic code completion when member access operator is typed. When the programmer types "`person.`" the editor can offer completion based on the type of `person`. Types are also used as a documentation tool, both informally and formally through tools such as Javadoc [Kramer 1999], which uses types as the primary source of information, together with explicit documentation annotation.

An example of a language born from the engineering culture that challenges other cultures is TypeScript. The primary motivation behind TypeScript is to enable better tooling for large-scale JavaScript development. The language is a "typed superset of JavaScript". Many aspects of the design of the TypeScript type system are thus determined by the design of existing popular JavaScript libraries [Rosenwasser 2018]. TypeScript makes it possible to annotate existing JavaScript libraries with types (through a separate definition file). This allows better developer experience, but the annotations are not (and can not) be checked, since the implementation is in plain JavaScript. Moreover, some of the aspects of the TypeScript type system are unsound. For the designers of TypeScript, "100% soundness is not a design goal. Soundness, usability, and complexity form a trade-off triangle. There's no such thing as a sound, simple, useful type system." [Cavanaugh 2018] In places where designing a sound type system would introduce additional complexity, the language intentionally chooses simpler, but unsound design.

The TypeScript design follows a set of consistent design principles which are distinct from the principles that are favoured by the ML paradigm. The motivation on tooling and producing simple useful documentation for programmers means that it makes sense to sacrifice soundness for simplicity. However, since TypeScript compiles to JavaScript, the lack of soundness does not make the system unsafe. It remains to be seen whether the use of types in TypeScript can be reconciled with perspectives of other cultures. So far, most proposals coming from other cultures propose to "fix unsoundness" by making the type system more complex, which goes against the TypeScript design principles [Richards et al. 2015; Vekris et al. 2016], but some also propose to accept this design approach and replace the usual *soundness* requirement with a weaker *soundiness* requirement [Livshits et al. 2015].

3.3 Relatively sound types for working with data

The tooling available through types and especially auto-completion was also one of the motivations for the design of F# type providers [Syme et al. 2013]. Type providers make it possible to write lightweight compiler extensions that generate, typically types based on information from an external data source such as a database, web service or a sample data file. The mechanism is similar to macros or code generation, with one notable exception. The types are generated lazily and it becomes possible to capture more information about the structure of the data at the type level. For example, the World Bank type provider generates types for accessing statistical information about countries. The programmer can access information about CO2 emissions in the UK by typing "worldbank" followed by dot ("."), navigating through the auto-complete list and choosing the "United Kingdom" member and then typing another dot and choosing the "CO2 emissions (kt)"

member (F# provides a mechanism for escaping member names with arbitrary characters in them, including spaces).

Type providers challenge thinking about types in two ways. First, provided types can represent the world in a more fine-grained way than hand-written types. Previously, "United Kingdom" was a value of type "Country", but with type providers, "United Kingdom" can be a type with only a single value. This development is not unprecedented. Indeed, custom data types (records, unions) were introduced as part of the research on *extensible languages* that allow users to define a language (i.e. types) that more closely maps to their domain. Perhaps type providers are the next step in a move from arrays of floating-point numbers and integers to records and unions. However, formally defining what a type such as "United Kingdom" denotes poses a notable problem for the mathematical culture of programming.

The second challenge is that the correctness of a program using type providers may depend on an external data source or, indirectly, the state of the real world. If the UK split into England, Scotland and Wales, the member and type "United Kingdom" would disappear. For some type providers, this property has been formally expressed as *relative soundness* where "well-typed programs do not go wrong", as long as the environment obeys certain constraints. This can be interpreted as an engineering interpretation of the mathematical idea of soundness. The idea that soundness relies on the external environment is not reasonable for the logical culture, but it is quite natural in the engineering culture, where most real-world programs rely on external resources for their correct functioning anyway. Type providers just acknowledge this property and make it tractable to type-based analyses.

4 DEFINITION OF TYPES – SHOULD THERE BE ONE

Should there be one unambiguous definition of what a type is? The proponents of the mathematical culture sometimes see the lack of definition as a burden to research. If we had one definition, multiple people could contribute to work on types, knowing that we all work with the same vision. However, even in mathematics, concepts do not have unambiguous timeless definitions. As documented by Lakatos, they evolve through the process of *proofs and refutations*. In his book, Lakatos looks at the history of the Euler characteristic of polyhedra, which relates vertices V, edges E and faces F in a formula $V - E + F = 2$. Over the history, people proposed counter-examples such as polyhedra with tunnels or intersecting faces. Those led to various refinements of exactly what sort of polyhedra the formula applies to and what it says about other polyhedra.

The situation with types is similar. The term type was adopted, likely independently, by the engineering culture and the mathematical culture. Kell [2014] illustrates the disconnectedness of the engineering culture from the mathematical culture in 1970s by the lack of cross citations between them. Gradually, the term 'type' acquired even more meanings, but the cultures also became aware of work on types done by other cultures.

The concept of a 'type' plays the role of a *boundary object* in programming languages. In sociology boundary objects "are both plastic enough to adapt to local needs and constraints of the several parties employing them, yet robust enough to maintain a common identity across sites." Similarly, the very fact that 'type' does not have an unambiguous single definition means that it can be adapted by different programming cultures, yet it allows the exchange of ideas across cultures. For example, the early work on abstract data types (engineering culture) used runtime checking, but it soon adopted idea of fully static checks enabled by the mathematical culture. In the opposite direction, the idea of using types for editor tooling such as auto-complete in Java influenced similar tooling in languages following the mathematical tradition.

Seeing types as boundary objects explains some of their value, but it limits the scope of what we can say about types to a narrow area of a specific culture. Petricek [2015] suggests a number

of approaches for talking about types in a unified way that does not rely on having a definition and can account for mutually inconsistent notions of types by taking inspiration from the "new experimentalism" movement in philosophy of science [Chalmers 2013; Hacking 1983] In the new experimentalist philosophy, knowledge about scientific entities consists of the empirical experiments that one can perform using an entity. Considering electrons, Hacking [1983] argues that "if you can spray them [on a niobium ball to change its charge] then they are real." Similarly, if we can use types to produce more efficient compiled code, eliminate bugs or build developer tooling, then we obtained knowledge about types that is independent of a particular definition.

In his reflection on types, Martini [2016] reached a similar conclusion and generalizes it to the entire computer science discipline. "The crucial point, here and in most computer science applications of mathematical logic concepts and techniques, is that computer science never used ideological glasses (...), but exploited what it found useful for the design of more elegant, economical, usable artifacts. This eclecticism (or even anarchism, in the sense of epistemological theory) is one of the distinctive traits of the discipline, and one of the reasons of its success." The fact that the notion of 'type' is a multi-cultural mix of ideas that have never been fully integrated might well be the reason behind its success.

LECTURE: SOFTWARE AS A SOCIO-TECHNOLOGICAL SYSTEM

In the previous two lectures, we followed two historical strands. First, we looked at the mathematization of computer programming which gave us the very idea of programming language. Second, we followed how different cultures contributed ideas to the concept of 'type'. Both of these strands were strongly influenced by the mathematical programming culture that is dominant in universities.

In this lecture, we will shift our focus from the mathematical culture to the managerial culture. The managerial culture should be seen in a broad sense, as covering the positions of businesses, managers but also the military. In other words, we take the perspective of those who want to build complex computing systems and care more about the result than the process. We start at the end of 1950s when computers started to be used by businesses.

Teacher: We discussed how formal methods can help us build software and, despite some limitations, they seem to offer great help. We also talked about types, which is a pragmatic multi-cultural concept that should also make it easier for us to build software. Have types and formal methods addressed concerns of managers and businesspeople?

Omega: Going back to the end of 1950s, the problem is not so much producing correct software, but finding programmers to do it!

Tau: Given the salary that programmers get, I expect that you can quite easily find mathematicians and teach them how to program.[26]

Omega: I wish it was this easy! Sadly, it seems that programmers are born, not made! You can never tell upfront if a programmer will be good. We designed a test based on logical puzzles, but even that is not very reliable.

Epsilon: I'm not surprised that you have a hard time finding enough programmers, when the tools we are using are so rudimentary. Writing the whole program in machine code is a certain way of never getting any work done!

Alpha: Once you learn it, it is not actually that hard. Or do you have some better idea for communicating your instructions to the machine?

Epsilon: These days, there are automatic coding systems for most of the computers out there. The automatic coding system almost eliminates the need for programmers. Domain experts can use them to communicate directly with the computer.

Omega: This makes some programmers a bit more efficient, but it is not nearly enough to stop the looming labour crisis in programming that is threatening the health and future of the entire commercial computer industry![27]

Tau: The lack of people is, no doubt, a real problem, but I was hoping we could go back to how mathematical methods help make software more correct, at least the software you have resources to build.

Epsilon: I find that the problem with software in practice is the sheer complexity of what we are building. Systems these days consist of hundreds of thousands of instructions and, according to some studies, programmers working on these systems can produce about 100 instructions per man-month. You do the math.[28]

[26] [Ensmenger 2012, p.18]
[27] [Ensmenger 2012, p.10]
[28] [Licklider 1969]

Tau: I admit that mathematics has no tradition of dealing with expressions on this scale, but a more important issue is that many programmers of the present day have been educated in ignorance and fear of mathematics.[29]

Teacher: We talked about the labour crisis, now we are talking about the complexity crisis. It seems that more people can be trained and new mathematical methods can be developed. Are these just practical limitations of our time, or are there fundamental reasons that limit what software can be built?

Gamma: All the work on formal verification that I have seen is focusing on proving properties of an implementation of some algorithm, but there is no continuity between and algorithm for computing the greatest common divisor and a ticketing system that issues real tickets.[30]

Tau: I don't see why not. The problem is that businesses have wrong expectations, but if we had proper specification that does not keep changing as well as enough time and resources to do things properly, then we can certainly build provably correct software.

Philosopher: You are making two assumptions that are not realistic. There is no proper specification and the environment in which programs need to run is guaranteed to continue changing.

Tau: I admit that writing a proper specification is hard, but I do not see any principal reason why it would be impossible.

Philosopher: We already talked about the principal reason before. The specification is not a merely formal entity. It describes something that exists in the real world and, as such, it has the same limitations as any empirical knowledge.

Epsilon: This is a useful insight. If we take what *Philosopher* says seriously, then we should treat specification as a scientific theory and subject it to empirical testing. When we build a system based on a theory, we should collect feedback on how well the modeling is going.

For example, if an industrial robot develops an internal three-dimensional representation of a wheel assembly passing by on a conveyor belt, and then guides its arm towards that object and tries to pick it up, it can use video systems or force sensors to see how well the model corresponded to what was actually the case.[31]

Teacher: The second problem that *Philosopher* mentioned was that the environment keeps changing. To better understand whether this is a fundamental problem, can someone propose an example?

Gamma: One good example is the development of mathematical security models for operating systems. The popular early model proposed by Bell and LaPadula was supposed to guarantee that an operating system kernel which provably satisfies the model cannot be compromised. However, the work was done in the context of time-sharing systems and failed to account for networking which soon became the main source of problems.

Alpha: I was going to propose a more recent example which has a similar structure. The infamous Y2K problem. Programmers of an early computer systems were using two digits to represent year, because it never occurred to them that the software would still be running after the year 1999. In other words, the environment changed in a way that the developers were not expecting.

[29] [Hoare 1985]
[30] [De Millo et al. 1979]
[31] [Smith 1985]

Gamma: I think that all the partial problems we are talking about boil down to a simple fact. Unlike algorithms which are unchanging formal mathematical entities, real software systems are socio-technological entities. You can never see the software in isolation, without considering the human aspects and so, a software system really is a fundamentally different entity from an algorithm.

Teacher: Do you have an example system in mind that we could discuss more concretely?

Gamma: Any large-scale software system in healthcare, finance or accounting fits this description, but if you want a well-documented concrete example, then we can talk about the development of anti-ballistic missile systems, starting with the Nike Zeus system in 1950s and all the way to the Patriot missile used in the first Gulf War in 1990s.

Tau: Well, you first need to specify what exactly you want to build. There are two parts. You need to specify how to track enemy missiles and how to intercept them. The first part is a matter of building a radar to track missiles, detect decoys and predict their flight path. Then you need to figure out at which altitude to intercept the enemy missiles.

If you want to protect smaller areas such as your own nuclear missile launch facilities, this can be relatively low in the atmosphere, but if you want to protect an entire city or the whole country, this would need to be higher. A military and scientific expert team can answer those questions, write a detailed specification and then we can build the system. Doing that properly will be expensive, but that is a price worth paying for a mission critical defense system.

Epsilon: I would not want my life to depend on such system. Almost all systems I worked on had some form of glitches that had to be fixed and this is an experience that most people in the industry have. However, it is not something that is widely reported in the literature.[32]

Omega: I understand that building large systems is difficult, but I have been using a number of systems in our company that work very reliably.

Epsilon: We can build a reliable system, but they only get reliable through gradual improvement and adaptation. The first version will always have issues, but if you plan for that, you will be able to resolve them and improve the system when you start using it.

Philosopher: Since we have no spare planets on which to fight trial nuclear wars, operational testing of a global anti-ballistic missile system is impossible.[33]

Gamma: I fear that building such anti-ballistic missile system would actually make our country *less* safe. It would give us a false feeling of safety and only encourage politicians to pursue more aggressive politics. Rather than avoiding conflict, they might end up triggering an attack that we wouldn't be able to defend against.

Philosopher: This also illustrates my second point, which is that the environment keeps changing. In the case of anti-ballistic missile systems, the environment is actively hostile. The enemy will learn about your plans and adapt their attack so that it can circumvent your defenses. This makes it impossible to build a successful anti-ballistic missile system.

Tau: Now we are talking about politics. You are including way too much as part of your "environment". I thought we are discussing how to build software systems, not how to decide what software systems are useful or ethical to build.

[32] [Slayton 2013, p.105]
[33] [Slayton 2013, p.184]

Teacher: The case of anit-ballistic missile system is an interesting one, but I agree that it is a very large-scale complex socio-technological system. Are there cases where something as simple as a phone app had similar problems?

Gamma: Well, there is the case of Google Photos app that labeled black people as gorillas using their machine algorithm for tagging. I do not doubt that their algorithm was, mathematically speaking, doing what it was supposed to be doing, but the results are just atrocious. When building software systems, you always need to include humans and, as in this case, account for their biases or for biases in your data.

Teacher: We learned a lot about the difficulties with building real-world software systems. We are facing a paradoxical situation. There is a long list of problems and fundamental limitations, but the industry is still very successful and builds systems that we rely on daily. Let's conclude the lecture by trying to understand what systems are we actually able to build and how.

Philosopher: The first important thing to say is that there is a difference between software systems and algorithms. Software systems face a number of fundamental limitations that do not algorithms. The limitations are the unprecedented complexity of such systems, the fact that a specification is an, inherently incomplete, model of the real world and the fact that the socio-technological environment in which programs are development keeps changing.

Gamma: The concern of complexity is certainly putting some hard constraints on what is possible to build. You could imagine an healthcare system that integrates all medical records, institutions and procedures and also monitors the health of individuals using wearable devices and makes recommendations. Such system would also have to be protected from misuse and attacks. We have to admit that we simply do not know how to design and build such ultra large-scale system.[34]

Epsilon: I agree that this is too big, but there are very big systems that we can build. The important thing is that such systems need to be developed progressively, with the aid of extensive testing and then operated continually in a somewhat lenient and forgiving environ- ment.[35]

Gamma: Speaking of forgiving environment, it is also important to recognize that human users are often very good at living with errors. The fact that there is a small bug in the system does not immediately make the system completely useless. People can frequently find a workaround for the bug and continue using the system happily.[36] They adapt their practice and might even forget that the bug is there.

Tau: If we want to talk about software systems in a more scientific way, we should be able to find a general law. I don't think we are there yet, but I imagine there could be a law relating the rate of change in the environment with the change rate of the system.

This would explain why building an anti-ballistic missile system has been an elusive goal. So far, the rate of the change in the environment has been greater than the rate at which we were able to design and build the system.[37]

[34] [Gabriel 2006]
[35] [Slayton 2013, p.119]
[36] [MacKenzie 2004, p.29]
[37] [Slayton 2013, p.216] and [Slayton 2013, p.125]

NOTES: SOFTWARE AS A SOCIO-TECHNOLOGICAL SYSTEM

As we saw in the previous two lectures, the variety of programming cultures that contribute to the modern notion of computing interacts in many ways. Even the very idea of a programming language was solving practical business need of portability using methods from mathematical logic. However, there are greater gap between some of the cultures than between others. The most notable gap has been between the mathematical and managerial cultures, as represented by academically oriented computer scientists and business programmers. In 1960s, those two groups were, to some extent, represented by the Association for Computing Machinery (ACM) and Data Processing Management Association (DPMA).

The historical strands that we followed in the previous two lectures were primarily centered around the mathematical culture. In this lecture, we shift our focus to the debates driven by the practical problem of developing large industrial and military systems.

The most important development that we follow in this lecture has been described by Slayton (2013) in the context of missile defense systems. In 1950s, the software part of a missile defense system SAGE was seen as unproblematic. The scientists and engineers involved in the planning saw programming as an easier and more flexible alternative to physical electronics. During 1960s and 1970s, computer glitches became increasingly visible. The managerial programming culture attempted to address the issues by developing development methodologies that would make the development of large systems manageable while the engineering programming culture slowly developed a vocabulary for discussing the problems and understanding what is technically possible. Based on those arguments, David Parnas resigned from a computing panel for the Strategic Defense Initiative (also known as "Star Wars") in 1985, claiming that such complex software was prone to unreliability for fundamental mathematical reasons that would not disappear with improved technology [Parnas 1985].

1 PERPETUAL SOFTWARE CRISIS

In his history of computer programmers, Ensmenger [2012] makes an interesting observation. Despite the fact that software is one of the most successful and profitable industries of all time, "corporate managers, academic computer scientists and government officials [have been] releasing ominous warnings about the desperate state of the software industry with almost ritualistic regularity" for the last several decades. [Ensmenger 2012, p.10] Despite the developments discussed in the previous two lectures, and the optimism associated with them, building large software systems has always been difficult enough that commentators felt that the "bright and rosy future" of the industry is being threatened by one crisis or another [Ensmenger 2012, p.18].

1.1 Finding programmers for commercial computing

The first crisis of the software industry started at the same time when the managerial programming culture itself appeared. Until early 1950s, programming was fully a domain of the hacker and mathematical programming cultures, working jointly on scientific and military applications of early computers. As documented by Ensmenger [2012, p.16], "the focus of electronic computing shifted [during 1950s] from scientific and military agendas (...) to electronic data processing and information management", which resulted in a labour crisis, i.e. "a sharp increase in the demand for business programmers."

Two major complementary solutions addressing the lack of professional programmers appeared. The first one was driven purely by the managerial programming culture and it focused on finding and training more programmers. The second solution was pursued mainly by the members of the

hacker programming culture and it aimed to make programming easier through the development of "automatic programming systems."

The first direction has been in detail documented by Ensmenger [2012]. Companies developed vocational courses and developed psychological profiles in order to identify what a suitable programmer looks like. The profile of an ideal programmer included a wide mix of interests and skills. As an example, a series of IBM ads that appeared in The New York Times listed the unsurprising interest in "algebra, geometry and other logical operations", but also more curious interests such as "musical composition and arrangement", "chess, bridge or anagrams", or even just "a lively imagination" [Ensmenger 2012, p.52]. It thus seems that the early conception of an ideal programmer was, in fact, based on the mix of programming cultures that we are following in this paper.

The second direction was the development of "automatic programming systems" that would automate some part of the tedious task of implementing (coding) of an algorithm for an actual computer. Systems that aimed to make coding easier appeared as early as 1950 when John Mauchly developed Short Code, an interpreted system that allowed representing and mathematical computations as expressions, rather than as machine instructions [Priestley 2011, p.188]. Short Code can be seen as a programming language, although it appeared before the idea of a programming language.

The idea of capturing some of the common coding patterns using a shorthand form gradually evolved into two most popular programming languages of 1950s, namely FORTRAN and COBOL. These two languages came at the right time to provide potential solution to the labour crisis. FORTRAN was a product of the mathematical culture and, as such, was mostly used in scientific and military applications. COBOL was a product of the managerial culture and was widely adopted by the data processing industry.

1.2 Turning black art into engineering

The culture of early programming was very much the hacker culture. As discussed in the first lecture, programming in 1950s was a black art that relied on personal knowledge. The results often depended on personal tricks and were not reproducible or reusable. This state of the art was problematic for the managerial programming culture which wanted to be able to produce software of a predictable quality at a predictable cost. There was a broad agreement that this is the next software crisis and it became subject of the NATO Software Engineering Conference in 1968 [Naur et al. 1969].

The conference brought the term "software engineering" into prominence and, according to Ensmenger [2012], historians and practitioners agree that the meeting "marked a major cultural shift" when programming started to make the "transition from being a craft for a long-haired programming priesthood to becoming a real engineering discipline." The conference identified a problem with the hacker culture of programming, but there was no agreement about possible solutions. It was followed by a second conference held in 1969, which "bore little resemblance to its predecessor. [A] lack of communication between different sections of the participants became, in the editors' opinions at least, a dominant feature." [Buxton et al. 1970]

Looking at the history through the perspective of distinct programming cultures, it seems that other cultures agreed on the problems with the hacker culture, but each culture was more interested in pursuing its own agenda than agreeing on a common approach. This is not surprising. The different cultures exchange ideas around specific controversies, such as the (im)possibility of formal verification and technical artifact such as types. The NATO conference series and the abstract idea of "software engineering" did not provide a sufficiently strong link.

Two cultures that developed their own answers to the problem of software engineering were the mathematical and managerial cultures. The mathematical approach was discussed in the first lecture. It focused on developing programs that are provably correct with respect to their specification. For

the managerial culture, "the essence of the software-engineering movement was control: control over complexity, control over budgets and scheduling, and, perhaps most significantly, control over a recalcitrant workforce." [Ensmenger 2012, p.198] Another factor that contributed to the managerial vision of software engineering was the the Defense Department funding which favoured work on "quantifying, predicting and managing complex software." [Slayton 2013, p.152] In a way, the mathematical and managerial cultures came out as winners of the software crisis of 1960s.

The managerial answer to the software crisis was to develop a methodology that could control the development process, not by appropriately controlling and structuring the code being written, but by appropriately controlling and structuring the team writing it. The inspirations for such structure came from the structures of industrial factories, surgical teams or the ideals of a democratic system. [Ensmenger 2012, p.207] However, the dominant approach was a top-level development methodology inspired by the, initially mathematical, idea of structured programming. The specification of the system had to be written, decomposed into parts that can then be developed according by different teams. The most prominent structured approach to software engineering became known as the Waterfall model [Benington 1983].

1.3 Paradigm change in software engineering

A number of researchers and practitioners raised objections against the strictly structured software engineering methodology that was favoured by the managerial culture in the 1970s and 1980s. A clear critical description of the current software engineering methods appeared in 1985 when David Parnas resigned from a computing panel on the Strategic Defense Initiative ("Star Wars"). Parnas [1985] is a collection of brief papers arguing that the proposed Star Wars system cannot be built for technical reasons. One point made by Parnas describes the difficulty of writing an upfront specification: "Unfortunately, it is hard to make the decisions that must be made to write such a document. We often do not know how to make those decisions until we can play with the system."

A number of authors in the late 1980s recognised the heavyweight nature of the dominant top-down development methodologies and proposed alternatives. One direction, presented by Blum [1989], is to use "rapid prototyping as a technique to define (or refine) the initial specification." Blum strikes a conciliatory note. He still accepts the importance of specifications, but proposes a better method for obtaining them for certain projects. A more radical vision has been presented by Christiane Floyd, in her "Outline of a Paradigm Change in Software Engineering" (Floyd, 1987). Floyd describes the dominant paradigm that emerged following the 1968 NATO conference as *product-oriented* and proposes a *process-oriented* alternative.

According to Floyd, "the product-oriented perspective regards software as a product". It "considers the usage context of the product to be fixed and well understood, thus allowing software requirements to be determined in advance." In contrast, the process-oriented perspective advocated by Floyd "views software in connection with human learning, work and communication, taking place in an evolving world with changing needs". The final product "is perceived as emerging from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback, carried out by different groups of people involved in system development in various roles." The process-oriented approach admits that understanding is built gradually and requirements change.

In the product-oriented perspective, process aspects such as requirements definition, quality assurance, user acceptance and software modifiability are considered "as additional aspects outside the realm of systematic treatment." Floyd holds that "this situation is inherently unsatisfactory and can only be remedied if we adopt (...) the richer process-oriented perspective as our primary point of view."

The heavyweight top-down software development methodologies dominated until late 1990s when computers spread even more widely and enterprises adopted PC computing. According to

some authors, businesses innovated faster and change of requirements made software systems obsolete before they were completed. This led to yet another crisis, this time termed "application development crisis" [Varhol 2018]. This eventually led a number of industry professionals to formulate the "Agile Manifesto" [Beck et al. 2001], which was a beginning of an industry transformation as envisioned 10 years earlier by Christiane Floyd.

The paradigm change outlined in this section can be seen as yet another cultural shift in the software industry. We started with the dominance of the "black art" software development as practiced by the hacker culture in the 1950s, followed by rising criticism and the NATO conference at the end of 1960s. We followed the rise to prominence of the managerial programming culture in the 1970s and 1980s, and its efforts to make software production a predictable process by following top-down structured development methodologies. Finally, we saw the disillusionment with the managerial methods in the 1990s and the backlash that culminated with the agile software development movement. The agile movement can, perhaps, be best seen as restoring a balance between the hacker culture, engineering culture and managerial culture. An idealised agile development process includes quick prototyping, good engineering practices such as testing, as well as continuous engagement with the customer. Despite this synthesis, agile methodologies are also subject to criticism and new trends keep emerging, borrowing ideas from crafts [McBreen 2002] and even biological systems [Aitchison 2011].

2 FUNDAMENTAL LIMITATIONS

So far, we have been following a debate on how to best build industry-scale software systems. However, a number of comments has also been made on the fundamental limitations of the systems that we are capable of building. In theoretical computer science, certain limits are well known. The undecidability of the halting problem is a mathematical proof that a certain *algorithm* cannot be defined, but are there similar limits for building *systems*?

Before looking at answers to this question, it is worth noting that the very structure of the answer has been subject to a debate. As part of the debate about the feasibility of the anti-ballistic missile system, Herbert Bright expressed "a profound distaste for saying we believe it won't work when the truth is merely that we suspect it won't work". The system would not violate any known physical or mathematical law and so "it's unfeasible to develop a sound basis for a proof that the proposed system cannot work". Joseph Weizenbaum did not feel obliged to provide a mathematical proof: "My suspicion is strong to the point of being belief. I don't think that my statement as a professional that I hold this belief obligates me to a mathematical proof." **?**, p.125

The fundamental limitations discussed in this section are thus not based on mathematical or physical laws. However, many authors discussing fundamental limitations of software development supported their point by a sound theoretical argument that, sometimes, proposes a possible software engineering law. We follow three such arguments.

2.1 Software systems are not algorithms

The perspective discussed in this section focuses on software systems as built in the industry. Those are often seen as distinct entities from algorithms, but perhaps also from algorithm-centric systems as typically considered by the mathematical culture. The difference has been clearly stated by De Millo et al. [1979] who argue that "there is no continuity between the world of FIND or GCD and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets".

The fundamental difference is that "the specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. The specifications for algorithms are highly stable, stable over decades or

even centuries; the specifications for real systems vary daily or hourly. (...) These are not differences in degree. They are differences in kind."

This position focuses more on the practical differences between implementing a software system and an algorithm than on the fact that an algorithm is a formal mathematical entity. This also suggests that there is no clear boundary. Thus the fundamental limitations of software systems might not necessarily apply to certain software systems that happen to have relatively concise and stable specifications. Examples of those might include compilers and virtual machines, which is also a space where the mathematical programming paradigm achieved relative success [Leroy 2009].

The large production systems that we consider in this section are not "composed of nothing more than algorithms and small programs". De Millo et al. remark that the colorful jargon of practicing programmers seems to be saying something about the nature of the structures they work with. Namely, a large part of the implementation of such large systems is made of "patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink." How do all these constructions get there and why isn't that just a lack of programming skill?

2.2 Inherent complexity of software systems

The first fundamental aspect that limits what kind of software systems can be built is their inherent complexity. This point has been clearly made by Fred Brooks in his well-known paper "No Silver Bullet: Essence and Accidents in Software Engineering" ([Brooks 1987]). Brooks points out that software construction involves *essential tasks* and *accidental tasks*. The former deal with the "complex conceptual structures that compose the abstract software entity", while the latter are concerned with "the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints."

According to Brooks, the hard part of producing a software system is the specification, design, and testing of the abstract software entity. The accidental tasks have already (in 1987) been largely simplified by the development of high-level programming languages and "unless [the accidental tasks comprise] more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement."

One of the irreducible essences of modern software systems according to Brooks is its complexity: "software entities are more complex for their size than perhaps any other human construct." The fundamental reason for this complexity has been clearly explained by Parnas, following his resignation from the Star Wars panel. Parnas [1985] contrasts software systems with analog systems and with digital computers. Analog systems can be modeled using continuous functions and the engineer merely has to make sure that components operate within their normal operating range. Digital systems are discrete and discontinuous. Before the advent of digital computers, the number of states was small enough to permit exhaustive testing. Digital computers have very large number of states, but they are constructed using many copies of the same small sub-systems that can be exhaustively tested. The case with software systems is different. They are not only discrete and discontinuous, consisting of very large number of states, but there is also no repetitive structure. Parnas concludes that this "is a fundamental difference that will not disappear with improved technology".

Brooks [1987] also recognised the fact that software systems are comprised of a large number of distinct, discrete elements and adds that, ass software systems get larger, the situation only gets worse: "scaling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements

interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly."

2.3 Continually changing environment

The second aspect of software systems that gives rise to fundamental limitations is that software systems are embedded in and built in a continually changing environment. To quote Brooks [1987] one more time, "the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product."

The need to reflect the dynamic nature of the environment motivated some of the changes in the software development methodology discussed earlier. The realisation that software development "[takes] place in an evolving world with changing needs" was one of the motivations for the process-oriented paradigm proposed by Floyd [1987]. The process-oriented method makes it possible to reflect the changes in the environment during the whole life-cycle of a system. This includes both the initial development, but also adaptation and modification during later use. To a lesser extent, methods based on prototyping recognise that requirements will evolve as the system is better understood, but this ignores the fact that the environment will continue changing even after the system is specified.

The fact that software needs to adapt to a changing environment also poses a fundamental limitation on what systems can be built. This point was actively discussed as part of the debate on anti-ballistic missile systems. The environment in which such systems operate evolves in a number of ways: new systems are deployed, collaborations with allies add new systems to interface with and, most importantly, counter-measures by enemies are developed and deployed. This means that defensive systems can never be blackboxed and will always remain in state of constant development and adaptation [Slayton 2013, p.216] However, it also means that building certain systems is logically impossible. The principle formulated by Weizenbaum [Slayton 2013, p.125] is that "the environment that [computer systems] are to control or with which they are intended to cooperate must have a change rate smaller than that of the system itself." Weizenbaum used this principle to argue that the Star Wars system cannot be built, because its environment is actively hostile. "This is not a question of not knowing enough technology now. There is a difficulty in principle here."

2.4 Infinite richness of the embedding world

Software systems are implemented and typically also specified in a formal language, but they interact with the messy real world. This gap is the third source of fundamental limitations of software systems. The mathematical culture of programming often writes as if the formal proofs of correctness that it produces were concerned directly with the physical reality. As already discussed, the problematic nature of this assumption has been pointed out, in the context of software verification, by a philosopher Fetzer [1988].

The gap between the complex physical reality of the world that software interacts with and the formal languages of specifications and program implementations has also been discussed from a pragmatic engineering perspective. In "The Place of Strictly Defined Notation in Human Insight", Naur [1993] points out that specification and implementation are two models of some reality. They are constructed using different *building elements*, but they are both models. Brian Smith [1985] argues along similar lines and also points out that "every model deals with its subject matter at some particular level of abstraction, paying attention to certain details, throwing away others." Models have to be abstractions, "otherwise they would drown in the infinite richness of the embedding world." However, the meaning of computer programs is that they perform actions on the real world

and those are not abstract. Models and thinking "may be abstract, some aspects of computation may be abstract, but action is not."

The fact that computer systems operate based on abstract models, but perform concrete actions is a fundamental limitation that cannot be avoided. Naur [1993] argues that major issues of such models are the "modeling insight and the model building insight", both of which are "concerned with the modellee and thus inherently involve items that are not strictly defined" (i.e. informal). In contrast, "minor issues are concerned with models and model building elements, i.e. items composed wholly of strictly defined items." Most notably, this includes the formal relationships between two models such as program correctness.

Naur criticised Dijkstra for claiming that "only formal questions can be of scientific interest." The questions concerning the modeling of the real world are of fundamental importance and are typically glossed over by many computer scientists. To use Naur's own words, "it is curious to observe how the authors in this field, who in the formal aspects of their work require painstaking demonstration and proof, in the informal aspects are satisfied with subjective claims that have not the slightest support, neither in argument nor in verifiable evidence. Surely common sense will indicate that such a manner is scientifically unacceptable."

2.5 Searching for software engineering laws

The mathematical programming culture has powerful mathematical tools for discussing theoretical possibilities and limitations of computer programs. However, those tools allow us to talk only about a highly idealised notion of computer programs, namely algorithms. Algorithms are an essential concept of computer science, but they are less essential when it comes to building software systems. This is understood even by theoreticians such as Hoare, who has reportedly been quoted saying that "in many applications, algorithm plays almost no role, and certainly presents almost no problem." [De Millo et al. 1979]

The discussions reviewed in this section can be seen as an attempt of the non-mathematical programming cultures to understand possibilities and limitations and, possibly, formulate fundamental laws of what software systems can be built. The inherent complexity of software systems is a result of an observation made from the engineering perspective, while the continually evolving nature of the environment is best recognised through the managerial perspective. Finally, the limited power of formal models is a result of philosophical reflection by practitioners who notice the mismatch between the claims that mathematical programming culture makes and the reality of building software systems.

Naur [1993] points out that some researchers disregard questions that cannot be answered with mathematical methods and quotes Dijkstra who contrasts the "scientific questions of correctness" with "the unformalized question whether a tool (...) is in such-and-such unformalized and ill-understood environment a pleasant tool to use." Dijkstra talks of pleasantness which is "a somewhat euphemistic word to use about such calamities as airplanes colliding in mid air or atomic reactors exploding" as a result of misunderstood informal requirement. It is thus important to understand that the fundamental limitations proposed by Naur, Smith, Weizenbaum and Brooks are not less scientific than those given by the theory of algorithms. They are fundamental limitations of a nature that cannot be analysed by other means.

3 SOCIO-TECHNOLOGICAL SYSTEMS

When the Information Processing Techniques Office (IPTO) was founded by ARPA in 1962, the first director J. C. R. Licklider actively pursued a vision of man-computer symbiosis [Licklider 1960] that, among other things, provided funding for work on graphical displays, interactive computing and the internet [Waldrop 2001]. One of the reasons that we do not commonly see the entire computing

field as dealing with socio-technological systems might well be that the early military computing efforts were focused on building fully automatic systems and so the majority of remaining funding from ARPA and the Department of Defense favoured research that benefited this goal. Yet, seeing humans as an inherent part of the computer system provides a helpful perspective for thinking about such systems in a large number of situations.

The positions discussed in this section suggest that there is yet another distinct culture of programming in addition to the mathematical, managerial, engineering and hacker cultures that we discussed so far. The *humanist* culture sees computers primarily through the interaction with humans, be it individuals or societies. This provides a helpful change in perspective, because it allows us to talk about problems that would not be visible otherwise. It also reveals that there are new difficult problems with software systems, but also new possible solutions.

3.1 Helpful change in perspective

Software systems are subject to mathematical, technical, but also social conditions. The fact that certain algorithms have exponential complexity is a mathematical problem, while implementing a compiler that produces efficient code is a technical problem. However, some of the most troublesome problems are the result of social interactions. To quote Brooks [1987], "much of the complexity [of software systems] is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which [the system] interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God."

This is not merely a matter of being able to build systems to a deadline, using an allocated budget. MacKenzie [2004] collected a data set of 1,1000 deaths up to the end of 1992 where a computer system was involved. The result of MacKenzies's analysis is that "over 90 percent of these deaths were caused by faulty human-computer interaction (often the result of poorly designed interfaces or of organizational failings as much as of mistakes by individuals). Physical fault such as electromagnetic interference were implicated in a further 4 percent of deaths, while (...), software bugs, caused no more than 3 percent, or thirty, deaths: two from a radiation-therapy machine whose software control system contained design faults, and twenty-eight from faulty software in the Patriot anti-missile system that caused a failed interception in the 1991 Gulf War." [MacKenzie 2004, p.300] It is thus clear that considering human a part of the system is also fundamental for building systems that do not cause harm to humans.

Yet another example of this appears in machine learning and artificial intelligence. There are many systems that operate according to their specification and could, conceivably, be proven correct, yet, they lead to undesirable behaviour that is frequently labeled as a bug. Examples of these include Google Translate translating Russia to 'Mordor' and Google Photos tagging black people as 'gorillas' [Kasperkevich 2015; Parkinson 2016]. The algorithms behind those bugs were likely correct, as documented by the difficulty of fixing them [Hern 2018], yet, their operation in the socio-technological context was incorrect. Such cases triggered a debate in the field of artificial intelligence, leading to an open letter [Future of Life Institute 2015] that encourages seeing such software systems in a wider context: "Progress in AI research makes it timely to focus research not only on making AI more capable, but also on maximizing the societal benefit of AI. [This] constitutes a significant expansion of the field of AI itself, which up to now has focused largely on techniques that are neutral with respect to purpose." Although the focus of the open letter was on a narrow space of artificial intelligence, we can see that the a similar call for emphasizing the humanist culture of programming could be made more broadly.

3.2 Keeping human in the loop

The ideas discussed in this section repeat the discussion from the first lecture about intuitionalist and mechanist mathematical sub-cultures, but in the context of building reliable software systems. The traditional managerial and engineering approach is that reliable software can be built by having a carefully written specification and correctly implementing it. The humanist approach recognises that the reliability of a system can also be achieved by keeping a human in the loop. [MacKenzie 2004] summarizes the two approaches by contrasting the positions of two well-known historical actors. "Hoare was advocating greater rigor in language and compiler design and the adoption of the proof-based verification techniques (...). Licklider was warning his contemporaries not to place nuclear missiles under the computer's sole control." [MacKenzie 2004, p.300]

A frightening case where keeping human in the loop prevented a disaster happened on October 5, 1960 when "the warning system at NORAD indicated that the United States was under massive attack by Soviet missiles with a certainty of 99.9 percent. It turned out that the Ballistic Missile Early Warning System radar in Thule, Greenland, had spotted the rising moon." [Borning 1991] The system did not trigger a counter-attack automatically and human operators correctly recognised the warning as a false alarm. In this case, there was an *unknown* bug in the software system and a human decision to overrule the alarm was, in part, based on the fact that the system has been in operation only for 4 days and that Khrushchev was attending the General Assembly of the United Nations in New York. [MacKenzie 2004, p.25] If the system was designed with human in mind, it would, perhaps, provide the human operator with more information based on which to make a decision.

Software with *known* bugs can operate perfectly reliably when we consider it together with its users. This is the case even for mission-critical systems such as the Apollo program: "Missions flew with program errors that were thoroughly documented, and the effects were understood before the mission. No *unknown* programming errors were discovered during a mission." [Slayton 2013] The system was not provably correct, but it was reliable when used by trained operators. Parnas made exactly this point when arguing that anti-ballistic missile systems did not need to be perfect, but had to be reliable: "I drive a car every day that I trust and I know that it is not perfect. I just want it to be trustworthy. What I want to know is what its limits are, when it can succeed, and when it can fail. I want to have confidence that it will not fail catastrophically." The common theme among these two examples is that when we consider software systems as socio-technological entities, it becomes more useful to think about their reliability in the actual context in which they are used, rather than to focus on abstract notion of correctness.

3.3 Large scale and wicked problems

Viewing software as a part of a wider socio-technological system is also fundamental when considering large-scale software systems. This issue has been discussed in a study that explores the problems with building ultra large-scale software systems (ULS) that comprise billions of lines of code, millions of computers and sensors, developed gradually in a decentralized fashion over multiple decades. Because of the scale, such system will be continually under development and even rare boundary conditions occur often enough that something will always be failing [Northrop et al. 2006].

The report on ULS summarizes that "to understand the challenges posed by ULS systems, we will need to look at them differently, not just as systems or systems of systems, but as socio-technical ecosystems: *socio-technical* because they are composed of people and technology interacting in complex ways, and *ecosystem* because characterizing them in this way will give us the richest

understanding of the technical and management problems that will be encountered when building, operating, and developing them."

Ultra large-scale software systems suffer from all the fundamental problems we discussed earlier. They are built in a constantly changing environment, they are complex and it is impossible to give a full specification of their behaviour. To quote Northrop et al. [2006], "the requirements to be satisfied by the systems will not be adequately known until the systems are in use." However, ultra large-scale systems have one more fundamental problem that arises from the fact that they are complex systems embedded in a rich social context. They will encounter "so-called wicked problems in which requirements are neither knowable in advance (because there is no agreement about what the problem is) nor stable (because each solution changes the problem, and no solution is considered to have 'solved' the problem)."

4 CULTURES AND SYSTEMS

In this lecture, we shifted our focus, both in terms of the kinds of systems that we are considering and in terms of programming cultures that are involved in building them. The systems that we were considering were large and complex, often with a complicated social implications. We used the well-documented case of a US anti-ballistic missile systems as one example, but many large computer systems built in banking, healthcare or accounting share most of the properties.

In terms of cultures, our focus was more on the managerial and engineering culture of programming. Those were the dominant cultures contributing to the discussion about such systems. We also introduced the humanist culture, which considers software systems as part of a larger socio-technological complex. This culture provides useful insights about cases where human interaction with a system was important, but it might also be growing in importance in the future when more ultra large-scale systems will be built.

4.1 Cultures and continual crisis

One of the fascinating observations that we discussed was the mismatch between the optimism of the mathematical culture, about the prospects of producing correct formally verified software, and the continual crisis narrative of the computing industry. As noted by Ensmenger [2012, p.240]: "The continued existence of a four-decades-long crisis in one of the largest and fastest-growing sectors of the U.S. economy suggests an interesting dichotomy: on the one hand, software is the technological success story of the past half century; on the other hand, its reputation and identity continue to be marked by perceptions of crisis and failure."

The nature of the crisis evolves and it becomes concerned with different programming cultures over time. The labour crisis of early 1950s was the result of commercialization of computers and the need to hire more programmers. At the time, those belonged to the hacker culture of programming and had "black art" skills that were difficult to understand for outsiders. The software crisis of 1960s and 1970s was dominated by the managerial culture of programming. As noted by Ensmenger [2012], the aim to turn programming from "black art" to "software engineering" was mainly motivated by the desire for control over budgets, scheduling, but also the hard-to-understand hacker workforce. However, software engineering also gradually became a technical discipline. As methodologies based on writing a detailed specification upfront reached their limits during the application crisis of 1990s, the engineering programming culture defined a new approach to system building through the agile methodology that favors gradual development, direct interaction with the customer and immediate feedback.

4.2 Systems and their limits

The second topic we discussed in this lecture are the fundamental limitations of software systems. Although the distinction is not always exact, each of the programming cultures we discussed has a somewhat different conception of what a typical software system looks like and, hence, understands its limits differently.

The mathematical culture of programming recognises mathematical laws as only limitations. It is impossible to build a system that will solve the halting problem, but anything else is possible. This extends from abstract algorithm to algorithm-centric systems where a full formal specification can often be provided and the environment does not evolve and is not hostile. For example, a compiler can be specified as a semantics-preserving transformation between two languages that can also be formally specified. Such specification is simpler and more readable than the actual implementation of the compiler, which makes the mathematical approach particularly appealing.

The managerial and engineering cultures are concerned with complex systems, lacking clear specifications, built in an evolving environment. Such complex systems can only be mastered if they are "developed progressively, with the aid of extensive testing, and then operated more or less continually in a somewhat lenient and forgiving environment." [Slayton 2013, p.119] The progressive development makes it possible to gradually understand the problem and specify how the system should address it. The system typically needs to interface and integrate with other systems and so the continual operation is necessary for adaptation to changes in the environment. The "bells and whistles" of such system are a visible part that cannot be abstracted away and so a formal mathematical description of the behaviour of such system is equally messy and complex as the implementation. For this reason, methods developed by the mathematical culture are mainly useful for small sub-components of such systems.

Finally, many computer systems are intertwined with the society in which they operate. This is the case for any complex software system, but also simpler application in artificial intelligence and machine learning that interact directly with humans or use personal data. [Future of Life Institute 2015] This will also be the case for the growing category of ultra large-scale computer systems [Northrop et al. 2006]. For such systems, we need to consider their social effect, however, this can rarely be understood before the system is in operation. We can see an indication that a new humanist culture of programming is appearing to address those issues, but it is too early to talk about the methodologies and technologies associated with this culture.

4.3 Ways of trusting software systems

Yet another aspect of software systems that keeps evolving with different kinds of systems and different cultures building them is the issue of trust. How can we trust a software system to operate well? MacKenzie [2004] discusses the issue of trust in the context of mechanized proof by referring to historical and sociological research on risk and trust in the societies of "high modernity." He summarizes that "in traditional communities face-to-face interactions typically obviated the need for formal structures of objectivity." In a society of a high modernity which, among other things, relies on increasingly complex computer systems, "face-to-face solutions are no longer viable. Modernity's 'trust in numbers' is a substitute for absent trust in people."

In the hacker culture of programming, the trust is mainly placed in individuals. In the 1950s, those were the masters of "black art" of computer programming, but a similar level of trust still exists in highly technical, especially open-source, programming projects such as the Linux kernel or many programming languages where communities even recognised such role through the Benevolent Dictator for Life (BDFL) title. The mathematical programming culture trusts systems based on a mathematical proof about them. This is often the case even if the mathematical proof is not

about the entire system, but about a formal model of a small part of it. The trust in proofs may be established through a social process or through a mechanical checking, although we noted earlier that both of these may pose a philosophical problem. Finally, the only way to build a trust in a socio-technological software system is through its use and operation as there are no proxy individuals or mathematical properties. This is perhaps best captured by a question stated by Slayton about the possibility of building a reliable anti-ballistic missile system: "[W]hat sense did it make to speak of a 'reliable' system that had never been relied upon?" [Slayton 2013, p.191]

LECTURE: HOW SOFTWARE GETS RELIABLE

In 1996, one of the most prominent members of the mathematical culture of programming, Tony Hoare published a paper "How Did Software Get So Reliable Without Proof?" (Hoare, 1996) in which he admits that "formal methods and proof play a small direct role in large scale programming" and surveys the current engineering practice to understand how it achieved the, surprisingly satisfactory, level of reliability.

In this lecture, we follow a number of historical strands that contributed to the possible answer to Hoare's question. We look at the development of debugging, testing and error handling and how those pragmatic concepts evolved from programming language features into development methodologies.

Teacher: In the last lecture, we looked at an analysis of deaths in which a computer was involved up to 1992. Out of the 1,000 deaths, only 30 were due to a software bug and out of these, 28 were caused by the Patriot missile system. That leaves 2 deaths caused by a program bug. How is it possible that the number is this low despite the continuous software crises and a very limited use of formal methods in practice?

Tau: Looking at the industry practices, I think the answer is a mix including rigorous management, quality assurance and testing, debugging and continuous software updating, defensive programming and various forms of over-engineering.[38]

Gamma: I don't want to sound rude, but do you think it still makes sense to dedicate your career to formal methods, even if it turns out that they only matter very little?

Tau: Formal methods play a direct role in practice too, especially in mission-critical systems. More importantly though, they provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement.[39]

Teacher: Let's have a look at *Tau*'s list in a historical order. What were the first industry practices that helped us produce more reliable systems?

Alpha: Debugging programs got a lot easier thanks to the birth of time-sharing and interactive computing at the end of 1960s and in 1970s, because you could interact with the computer while it was running and debug it in almost a modern sense of the word, but some form of debugging goes back to early digital computers. Many had a way of running programs step-by-step and had lights or other visual indicators that you could use to find what the current state of registers is.

Gamma: Ironically, debugging of your system only got worse with digital computers. A light to show you a value in your registers is nice, but nowhere near what we had before!

Epsilon: What do you mean? Debugging calculations made by hand on a piece of paper?

Gamma: It's easy to forget, isn't it? Before digital computers started to dominate in 1950s, there was quite a lot of analog machines. Sure, they were much less powerful and flexible, but they didn't just calculate an answer; they invited you to make a tangible model of the world. Your program become a living thing that you could inspect as a whole![40]

Tau: I admit that some of the analog computers are works of beauty. You can certainly debug an analog machine such as MONIAC easier than a model of economy implemented in FORTRAN. That

[38] [Hoare 1996]
[39] [Hoare 1996]
[40] [Waldrop 2001]

said, I wanted to learn more about early debugging techniques and I'm a bit lost. I can hardly find any paper documenting best practices or developing a theory of debugging...

Alpha: You can check out a couple of tools like FLIT from 1960. They are not really documented in academic papers, but there are memos published by the MIT Lincoln Laboratory that document them. I don't think that an academic paper would be very useful though. You just have to learn how to use them through practical experience...

Omega: I see, this is the black art of programming again!

Tau: Does debugging really matter that much, though?

Epsilon: I would even dare to say that debugging is the next big challenge for programming! After hardware got reliable in 1950s and high-level programming languages FORTRAN and COBOL made it possible to write most useful programs, the fact that the number of bugs in our programs is not getting any smaller is now one of the main limiting factors that limit how we can use computers. I expect that the rest of 1960s and 1970s will be the epoch of debugging![41]

Teacher: In retrospect, what were the most interesting developments of debugging methods following the 1965 prediction that the "epoch of debugging" is starting?

Alpha: Oh, the debugging tools that I was talking about were quite powerful already. You could step through instructions one-by-one, set breakpoints and even create conditional breakpoints. You could also patch your program on the fly to see if your fix resolves the issue. This was available for both assembly language debugging and for high-level languages.

Epsilon: Should I be impressed or disappointed? Honestly, this sounds pretty much like the modern debugging tools that I use today!

Alpha: The only two limitations I can think of are that getting the corrected program after applying a series of patches can be a bit tricky and that, for high-level languages, the debuggers relied on interpretation, rather than incremental compilation. We have quite reliable incremental compilation today, but I don't think we found a satisfactory solution to the first problem.

Teacher: Another industry practice that *Tau* talked about is testing. There is quite a lot happening around testing in modern industry practices, but let's start with the history first. Has testing become popular before or after debugging?

Omega: Proper testing only came when programming started becoming "software engineering" and replaced the arcane methods of development that *Alpha* keeps talking about with a more rigorous methodology following the 1968 NATO conference. Testing became a part of the software development process. It was used to make sure the program solves the specified problem.[42]

Tau: Excuse me, but testing shows the presence, not the absence of bugs. You cannot use testing to show that your program operates correctly.

Omega: I agree that the early way of thinking about testing was quite naïve. At the end of 1970s, the use of testing in software engineering definitely shifted from showing that a program works correctly towards detecting bugs. That said, a good test generation method can help you get a reasonable confidence that you covered the most important test cases.[43]

[41] [Halpern 1965]
[42] [Gelperin and Hetzel 1988]
[43] [Howden 1982]

Gamma: You are talking about a very structured and organized kind of testing, but surely, we should also consider the kind of testing that happens during software development, when I just want to run a component on some ad-hoc input to see how it would work...

Alpha: This way of testing is at least as old as debugging. The early view of programming was that you "wrote" a program and then you "checked it out". For many, the concepts of program checkout, debugging and testing were not clearly differentiated.[44]

Epsilon: This is a strange conflation of ideas. The way I see it, testing is what we do to find bugs and debugging is what we do to remove them.

Omega: I agree with that, but I think you are missing one important aspect of testing and debugging, which is where it fits in the software development life-cycle and who performs it. Testing should be done as an independent step after programming and it should be done by a separate team. If programmers know the tests, they will develop incorrect code that passes.[45]

Epsilon: This is a ridiculous idea. Programmers are professionals who want to do a good job. We understand that our goal is to produce effective software, not software that merely passes tests. Such secretive way of testing not only makes development expensive, but it also creates an unhealthy relationship between testers and programmers.[46]

Gamma: I'm very glad that this was just a managerial fantasy of 1980s heavy-weight software engineering methods and that nobody thinks this we should build software in this way today.

Tau: The problem is that you are still using testing as a poor substitute for program verification. You have a program specification and you are trying to show that the program implements it. You could perhaps develop a probabilistic theory based on how much of your code gets covered by tests and what proportion of possible inputs is tested[47], but that is still only a very weak guarantee.

Epsilon: In modern use, testing is doing much more than just writing checks to see if you implemented certain part of a specification correctly. If you follow the test-driven development, you are using tests to develop your specification, to get a quick feedback during development and also to structure your development workflow.

Tau: I understand what you say about structuring development workflow and getting a quick feedback, but using tests to create your specification is the wrong way round. Surely, you should write tests based on your specification so that you are checking for sensible properties.

Epsilon: The problem is that, in practice, you almost never have a detailed specification of the system. The idea of test-driven methodology is that you first write a test and only then implement the code. When writing the tests, you have to think about what the right behaviour should be. The tests should be as simple as possible, so that your colleagues or even business domain experts can review them and make sure that your specification is correct.[48]

Philosopher: This is an interesting remark. If tests are something that you discuss with your colleagues and look at in order to understand how a software should work, then they provide a focal point for valuable social processes. Exactly the kind of social processes that proofs of program correctness might be lacking.

[44] [Gelperin and Hetzel 1988]
[45] [Gelperin and Hetzel 1988]
[46] [Gelperin and Hetzel 1988]
[47] [Musa 1975]
[48] [Solis and Wang 2011]

Epsilon: Another useful property is that tests do not get out of date. You run them automatically after every change and make sure that all tests pass before you continue. If your tests fail, you either fix your code, or revise your specification and update the test.

Omega: When you put it in this way, the idea that engineers not only see, but even produce the tests makes sense. What you are describing is an appealing light-weight software development methodology. If you have enough time and resources, then I think you should still produce a detailed specification up-front. However, if your timeframe and resources are limited, then your method where specification is developed gradually and written in form of tests sounds quite effective.

Gamma: This debate has taken an unexpected turn! When I think of testing, I just think of writing a simple script to explore an idea, or running a command interactively. You just turned testing from a simple programming practice into an entire software development methodology. I bet it won't be long before someone gives your development methodology a three-letter acronym and starts offering expensive certifications confirming that you are doing it in the 'one true way'!

Teacher: Before we get too distracted speculating about business ideas, I would like to get back to other factors that contributed to software reliability. We talked about debugging and testing already; the next item on *Tau*'s list was over-engineering. What does this idea look like in the context of software engineering?

Omega: In traditional engineering disciplines, you can define over-engineering in terms of safety factors. For example, you calculate the worst case load for each beam of a bridge and then make it ten times stronger.[49] However, I just do not see how this idea could be applied in software engineering. You can certainly insert some checks and defenses, but the fact that we do not even have a concept like safety factor shows that software engineering is simply still not a proper engineering discipline.

Teacher: Does anyone have any ideas on what might such over-engineering look like?

Epsilon: Well, I imagine that one useful thing to do would be to detect that some operation did not finish as expected. Then you could attempt to run it again. You'd have to make sure to undo any state changes it caused already, but that should be doable.

Tau: How is that supposed to work? Given the same initial state, when you run the same program, you will just get the same result. If your program has a bug, it is because you put it there. Running the program again will not make it go away.

Gamma: You are ignoring the fact that programs interact with the outside world. If you are reading data from a magnetic tape or sending data over a network, then running the program again might, in fact, solve the problem.

Epsilon: What you can do depends on what kind of exception you are dealing with. I suggest that we should distinguish between *range failures* and *domain failures*. A range failure occurs when an operation cannot satisfy its output requirements, for example when a function cannot read a valid record from a file. A domain failure occurs if an input for an operation is not acceptable.[50]

Tau: That is quite obvious. You are just distinguishing between input assertions and output assertions. Of course, if you can prove that, given input satisfying domain assertions, your code produces result that satisfies range assertions, then you know there are no errors. But I expect you don't want correctness proofs, but some sort of recovery mechanism.

[49] [Hoare 1996]
[50] [Goodenough 1975]

Epsilon: That's correct. For range failures, the invoker should be able to decide whether to try again, abort the operation or terminate and get incomplete or incorrect partial result. For domain failures, the exception should indicate what condition was violated, so that the invoker can correct the inputs.[51]

Alpha: I wonder how you want to implement all this! The closest language construct I can think of is the idea of ON conditions in PL/I. ON conditions are very limited though. They let you handle 23 infrequent conditions such as end of a file. If the condition occurs, the control is transferred to the code block specified by the ON construct, but you have to pass any additional information via a global variable.

Epsilon: The way exception handling is implemented in PL/I is a good first step, but it is very cumbersome. Rather than using non-local jumps, exception handlers should be associated with operations or blocks of operations. Signaling an exception is then more akin to the "break" statement for multilevel loop exiting.

The ability to resume computation after an exception is triggered makes them also useful for monitoring or to indicate circumstances under which a result has been obtained. It makes exceptions a more general language feature for conveniently interleaving actions belonging to different levels of abstraction.[52]

Tau: This looks like a useful programming language feature, but I do not see how this is over-engineering. If your program does not handle end of file correctly, then it is just wrong. Whether you handle the end of file condition via an ON construct or some other mechanism does not make any difference.

Omega: I have to admit, *Tau* makes a good point. The hard part in programming is not reading files, but managing the complexity of software design. Our over-engineering efforts should focus on redundancy of software design, not simple replication of programs.

Gamma: Intriguing! You are not talking about just restarting the same program. You are suggesting that we design and implement multiple variants of the same logic...

Epsilon: This can certainly be done. For each operation that you want to make fault tolerant, you would specify one or more conditions that determine whether it produced an acceptable result. Then you specify a primary implementation and a series of alternative implementations that might not be perfect, but would still allow the program to continue.[53]

Alpha: This could be quite difficult to implement well. If you run an operation that fails and want to run an alternative implementation, you will need to restore the system state back to a state before you started running the failing operation. This might not even be possible if the operation sends a message to some other system over a network...

Epsilon: Actually, there is a way of avoiding this problem altogether. You just need to think about the system differently! The Erlang language, which appeared in 1980s is based on the idea of lightweight processes that communicate with each other. When the process gets into an erroneous state, it just kills itself.

Alpha: I might be missing something, but how does killing a process make your system more fault-tolerant?

[51] [Goodenough 1975]
[52] [Goodenough 1975]
[53] [Randell 1975]

Epsilon: Oh, the key idea is that lightweight processes that can be killed encourage more fault-tolerant architecture. You will write systems that compose from a large number of simple processes and structure them in supervisor trees. A parent process will then monitor child processes and, if they fail, restart them or even trigger some other recovery mechanism or a simplified alternative implementation.

Tau: You started by claiming that having some errors in programs is inevitable and now you are intentionally making parts of your program crash! I wonder what comes next. Will you try to convince me that the more programs crash, the better?

Gamma: Why not? Biological systems adapt and learn from failures. That is how our immune system develops resistance and how some vaccination works. Maybe if software systems were exposed to controlled errors more frequently and had the ability to learn from the failures, they could actually become more fault-tolerant as a result.[54]

[54] [Tseitlin 2013]

NOTES: HOW SOFTWARE GETS RELIABLE

When Charles Babbage designed the Analytical Engine in 19th century, he did not envision programming of the engine to be a problem. In 1837, he even wrote that "if trials of three or four simple cases have been made, and are found to agree with the results given by the engine, it is scarcely possible that there can be any error among the cards." [Babbage 1982] The belief that programming does not pose significant difficulties prevailed through the early days of digital computers during 1940s. In early 1950s, the designers of the SAGE air defense system were still thinking of programming as a more versatile and flexible version of physical electronics [Slayton 2013, p.18].

The inherent complexity of software systems, caused by the fact that software is made of a large number of discrete components without a repetitive structure eventually overtook the problem of making discrete, but repetitive electronic components reliable. Maurice Wilkes, the designer of the EDSAC computer, well remembers a moment in 1949 when "the realisation that a good part of the remainder of [his] life was going to be spent in finding errors in [his] own programs." [Wilkes and Collection 1985] Wilkes might have been one of the earliest computer scientists to recognise the difficulties of programming errors, but it did not take long before problems related to programming became the main obstacle, threatening the "bright and rosy future" of the industry.

Many of the programming cultures argued for their own solution to the problem, be it the use of mathematical methods for program construction or the development of managerial practices as a way of controlling the unpredictable workforce. Half a century later, the computer industry has become one of the most successful and profitable industries of all time, [Ensmenger 2012, p.228] without adopting any of the fundamental methods advocated by some of the different programming cultures. This was a surprise to many. Tony Hoare, prominent member of the mathematical culture, aimed to answer the question of "How did software get so reliable without proof?" in his eponymous 1996 paper. He concludes that the answer is likely due to a mix of good engineering practices including "rigorous management of procedures for design inspection and review; quality assurance based on a wide range of targeted tests; continuous evolution by removal of errors from products already in widespread use; and defensive programming, among other forms of deliberate over-engineering." [Hoare 1996]

In this lecture, we delve into a number of the engineering practices mentioned by Hoare and follow a number of historical strands that contributed to making software mostly reliable in practice. The methods we focus on in this lecture originate largely from the hacker culture of programming, but they later mixed with other programming cultures and often evolved into sophisticated engineering and managerial practices.

1 ELIMINATING ERRORS AND DEBUGGING

Despite a popular tale, the term 'bug' was not introduced into the vocabulary of programmers when Grace Murray Hopper removed a dead moth from one of the circuits of the Harvard Mark 1 computer. Hopper has, in fact, removed a dead moth and kept it in her logbook, but the term 'bug' has been adopted from contemporary engineering slang. Perhaps the earliest recorded use of the term in the modern sense dates back to the end of the 19th century and reports on "Mr. [Thomas] Edison (..) discovering a 'bug' in his phonograph." By calling the flaws 'bugs', engineers also suggested that the issues are minor and can be easily fixed with a little bit of effort [Kidwell 1998]. This proved to be an ironic naming, given that the estimate cost of work done on fixing the "Y2K bug" was over $300bn.

1.1 Debugging early digital computers

The first non-trivial program that has likely been subject to debugging is a program to calculate the Airy function, created by Maurice Wilkes for the EDSAC computer. A detailed account of the debugging process has been given by Campbell-Kelly [1992] who analysed an early tape with the program, finding "approximately twenty errors in the 126 lines of the program." Many of those were simple punching errors, but some likely required notable debugging effort.

The fact that producing correct programs will be difficult was unexpected and so EDSAC did not initially have dedicated program debugging tools. The EDSAC computer did, however, have a "Single E.P." button that executed one instruction and a CRT monitor displaying contents of a portion of the computer memory. Some computers even provided switches for manually modifying the instructions in memory. This allowed operators to execute programs step-by-step and inspect the memory state. This tedious process was referred to as "peeping" [Campbell-Kelly 1992].

As Campbell-Kelly notes, peeping was soon recognised as "extravagant use of computer time" and was outlawed by the designers of EDSAC. Instead, a number of debugging tools were developed. First, "postmortem" dump routines were used to print the state of the memory when the program terminated abnormally, so that the inspection and debugging could be done offline. Second, "interpretative trace routines" developed in 1950 by Stanley Gill provided a way to instrument the execution of a program and print, for example, a trace of instructions being executed [Campbell-Kelly 1992].

The notion of debugging in the early days of computing was broader than today. Gelperin and Hetzel [1988] note that "the early view of programming was that you 'wrote' a program and then you 'checked it out'. For many, the concepts of program checkout, debugging, and testing were not clearly differentiated." There was no agreement on the difference between debugging and testing and they referred to a range of activities involved in discovering and eliminating program errors. Debugging and testing started to differentiate in late 1950s when the task of debugging was to "make sure the program runs" and the task of testing was to "make sure the program solves the problem" [Gelperin and Hetzel 1988]. We follow the history of debugging next and return to testing in a subsequent section.

1.2 Time-sharing and interactive debugging tools

Interactive debugging of programs, which was an extravagant use of EDSAC compute time, became possible with the development of time-sharing systems. Many of the early debugging tools were just virtual versions of what EDSAC and other early computers provided at the hardware level. Debugging tools allowed programmers to inspect the state of the memory, run programs step-by-step and modify programs on-the-fly during debugging.

The early debugging tools were a good example of the private arcane matter that was programming in 1950s. As noted by Evans and Darley [1966], "no discussion of [debugging tools] has appeared in the literature [and] they are far from completely described even in internal memoranda." Nevertheless, Evans and Darley present an extensive survey of debugging tools available at the end of 1950s and early 1960s. Tools with similar capabilities were developed for both low-level assembly languages and for higher-level (algebraic) programming languages.

Two examples of early debugging tools for assembly language were FLIT, developed in 1957 for the TX-0 computer, and DDT (DEC Debugging Tape) developed in 1961 for the PDP-1 computer. Both of the tools provided support for breakpoints. Setting a breakpoint in DDT inserted a jump instruction into user's program that transferred control to DDT, which then saved and printed the state of registers and allowed programmer to examine them. [Kotok 1964] The DDT tool also allowed programmers to patch their programs by inserting machine instructions, entered in a

symbolic assembly-language form, in some available memory space and inserting a control transfer instruction into the program.

When Evans and Darley published their survey in 1966, they noted that "a close analog of almost every principal assembly-language debugging technique exists in at least one debugging system pertaining to some higher-level language", but that "on-line debugging facilities for higher-level languages are in general less well-developed and less widely used." The higher-level language with most advanced on-line debugging features in 1960s was LISP. The LISP implementation for the MAC system supported tracing (adopted from earlier, batch-processing implementation), conditional breakpoints, but also an editor that made it possible to modify the list structure in which both code and data is stored in LISP. The homoiconicity of LISP made the development of such tools easier than, for example, FORTRAN. However, even for FORTRAN, debugging tools based on interpretation allowed programmers to inspect state and modify programs live during a debugging session.

One of the subtle points faced by both assembly languages and higher-level languages was keeping the original program in sync with changes made interactively during the debugging session. The DEBUG tool developed by Evans and Darley [1965] addresses this issue by keeping a table of edits performed manually during the interactive session and printing it out on an "alter tape" after the end of the debugging session.

1.3 The evolution of debugging

Throughout the early history, debugging tools were created by the hackers who were trying to get different generations of computers and computer programs to run. The techniques evolved from tinkering with a physical machine to inspecting the state of digital memory and, eventually, to working with increasingly high-level abstractions such as variables and lists. The methods for observing the state and modifying the program have changed, but the overall approach remains surprisingly consistent.

Debugging tools in modern IDEs (Integrated Development Environments) such as Visual Studio or Eclipse are, in many ways, very similar to what tools such as FLINT provided in 1950s. One notable change is the shift of focus from a command-oriented interface to an interface where most operations are done by somehow manipulating (a visual representation of) the program source code. Most notably, modern tools avoid the problem of keeping the program source in sync with the executing program by allowing the programmer to modify the program source code and then recompiling and hot swapping modified parts. The remark of Evans and Darley that debugging techniques are rarely documented in academic literature remains true for more recent developments and so tracing the origin of those changes is difficult, but it is likely that many of them appeared in 1980s in the Smalltalk programming system [Foote and Johnson 1989].

An interesting aspect of debugging and debugging tools is that they largely remain within the realm of the hacker culture of programming. Building a sophisticated debugger for a modern IDE might be an engineering challenge and some work done in the mathematical culture contributed ideas to debuggers [Weiser 1981], but debugging is still a way of tinkering with the state of the program. Consequently, debugging tells us important facts about the hacker culture of programming.

The most notable aspect is that knowledge in the hacker culture of programming is often not written down. As noted by Evans and Darley [1966], "much of the work [on debugging] has been described only in unpublished reports or passed on through the oral tradition." Even the internal memoranda documenting debugging tools often merely provide a reference manual for the tools, rather than documenting how they should be used. Debugging skills seem to be a kind of unwritten knowledge shared by practice that Polanyi [1958] refers to as *personal knowledge*. Polanyi documents the importance of personal knowledge in scientific practice and it seems that similar

personal knowledge plays a major part in how the hacker culture contributes to the elimination of bugs in computer programs.

The early proclamations that liken programming to "black arts" or alchemy seem to be right in one way. Just like the debugging knowledge possessed by the hacker culture, experimental knowledge of alchemists was also not widely reported and shared. In case of alchemy, this was for sectarian reasons, while in case of debugging, this seems to be for practical reasons. Debugging is a kind of skill that is learned through practice, rather than studying texts about it. However, this lack of shareable knowledge makes it difficult to further develop the discipline. It contributed to the demise of alchemy [Wootton 2015, p.337] and it might similarly be preventing the development of new debugging tools and techniques. If text is not the right form of sharing practical knowledge of debugging, then, perhaps, the hacker culture of programming will need to find a different format for sharing its knowledge, perhaps a more hands-on practical one such as a screencast.

2 TESTING AS PART OF A DEVELOPMENT PROCESS

Just like debugging, testing has origins in the hacker culture of programming. For some, testing and debugging were indistinguishable; for some, debugging meant any activity involved with getting the bugs out of a program and testing was one of the activities [Gelperin and Hetzel 1988]. Unlike debugging, software testing did not remain fully within the realm of the hacker culture. One reason for this might be that software testing can involve the production of technical artifacts that can be interpreted differently by different cultures.

In the era when debugging and testing were considered indistinguishable, testing was merely a matter of running the program with some random suitable input and verifying that it behaves as expected. However, testing can be scripted or even automated. A testing script can become a part of the social process of software development and a programmatic test can become a part of the engineering process. Through such scripts or automatic tests, the hacker culture of programming cross pollinated with the managerial culture and with the engineering culture.

2.1 Testing as a phase of a development process

Following the 1968 NATO conference on Software Engineering, numerous efforts were made to turn programming into a "proper engineering discipline." The managerial approach to the problem was modeled after industrial factories. The software factory would follow a development process that leads to reliable software production, without relying on particular skills of individual programmers [Ensmenger 2012, p.60]. Such development processes often included different kinds of testing as one of the steps or phases, however testing played a different role over time.

Gelperin and Hetzel [1988] identified a number of major shifts in how testing was used in the context of a development process. During the "demonstration-oriented" period until 1979, software testing was seen as a way of "making sure the program solves the problem". The positive phrasing suggests that testing was seen as a way of showing that the software satisfies its requirements. In late 1970s, a number of methods were developed for the selection of input data for testing purposes. Goodenough and Gerhart [1975] attempted to provide a theoretical foundation for testing and presented a method for test data selection that makes it possible to comprehensively test a program and show the absence of errors based on case analysis.

The next period of software testing that Gelperin and Hetzel refer to as "destruction-oriented" period started in 1979 with the publication of "The Art of Software Testing" by Myers et al. [1979]. Meyers sees testing as "the process of executing a program with the intent of finding errors". This way of looking at testing encourages a test selection method that is more likely to find faults and discover edge cases that might not be considered when the aim is to make sure that a program works as required.

The next shifts in the use of testing do not transform the methods and goals of testing, but instead, transform where it fits in the software development process. At this point, testing has become an important tool of the managerial culture of programming and the next developments reflect ideas about project management. The publication of "Guideline for Lifecycle Validation, Verification, and Testing of Computer Software" in 1983 [National Bureau of Standards. 1984] marks the beginning of the "evaluation-oriented" period where testing is used as a way of evaluating the progress made during the system development. Testing is used in different ways in different phases of the software lifecycle. Unit testing, integration testing and system testing is done at the end of the programming and testing phase while acceptance testing is done manually, by the customer, at the end of the software installation phase.

Finally, [Gelperin and Hetzel 1988] identify the year 1988 as the beginning of a new period of software testing that they refer to as "prevention-oriented" period. The development process they advocate sees test planning, test design and development and testing as an ubiquitous activity that happens in parallel with the entire system lifecycle, rather than being done in isolation during certain development phases. Gelperin and Hetzel still discuss their ideas in the language of the managerial culture, but they captured a shift that happened throughout 1990s and gave testing yet another meaning.

2.2 Testing as an engineering practice

As discussed in the previous lecture, the disillusionment with heavy-weight managerial methods of software development led to a backlash in 1990s. A number of new methodologies that aimed to reduce the overheads of heavy-weight software development processes and make software development more responsive to changing requirements. One of the earliest methodologies was "extreme programming" (Beck, 1999), which takes many of the good software engineering practices and takes them to "extreme" levels. One of the good practices that was taken to an extreme level was software testing.

Making good practices such as code reviews, customer feedback and testing a part of daily routine also means that activities which could previously be done by separate team members is now done by the programmers themselves. As a result, extreme programming is not just a new managerial methodology, but a change of cultures. The managerial culture that tried to eliminate the reliance of individual programmers is partly replaced by an engineering culture where programmers have joint responsibility for the project and for choosing or building appropriate tools.

One of the extreme programming practices centered around tests is test-driven development (TDD). In test-driven development, tests are used as a mechanism for providing rapid feedback and for controlling the development process. When implementing a new feature using TDD, you first add a failing test that documents and specifies the desired behaviour. Then you write code until the test passes and, finally, you refactor your code to improve its internal structure.

Testing as practiced by the TDD methodology forms an interesting blend of activities that were envisioned in a different way by different programming cultures. It keeps the quick feedback provided by program checkout that was always important for the hacker culture; tests become a form of partial software specification that the mathematical programming culture required; tests provide a way of controlling and structuring the development process which was important to the managerial programming culture. Finally, test-driven development is done by individual programmers who often see it as a way of turning programming into a solid engineering discipline.

3 RECOVERING FROM AND EMBRACING FAILURES

An important method for developing reliable software systems is to make them fault-tolerant. Even if there were no programming mistakes, a program can get into an unexpected state as a result of a

hardware failure or because of an invalid input. In practice, programming mistakes are another common reason why a program gets into an unexpected state. A fault-tolerant system should be able to recover from such unexpected states.

3.1 Hardware faults and programming mistakes

Despite the fact that hardware failures have become relatively infrequent, there are still many situations where hardware failures are an important concern. First of all, some hardware such as magnetic tapes or wireless networks is unreliable for physical reasons. However, even hardware that is very reliable can cause problems when we consider it at a scale. A data centre or a large telecommunications network needs to be able to account for hardware failures even if the individual components are so reliable that the potential for a failure could be ignored if they were operating as stand-alone systems.

In the perspective of the mathematical programming culture, hardware failures need to be handled, but other kind of failures should be ruled out by a mathematical proof. However, programming mistakes happen and they likely account for a majority of cases where a program gets into an unexpected state. Formally, consequences of programming mistakes may be of a different kind than hardware faults, but in practice, they have similar consequences and need to be handled in a similar way. This is mirrored in the programming language features that were developed to deal with errors. The same language feature is typically used for handling hardware failures, as well as programming mistakes.

In the lecture, we introduced fault-tolerance as one of over-engineering methods that programmers can use. This is an appropriate classification when handling programming mistakes. A fault-tolerant system that can recover from certain unexpected states will be able to accommodate a certain amount of mistakes. However, calculating a number akin to safety factor for software systems is difficult because of the nature of software systems discussed earlier. Safety factor can be easily calculated for an analog system with a continuous formal model, but software systems are discrete, which makes the calculation of a safety factor difficult.

3.2 Recovering from failures using exceptions

The early history of exception handling mechanism in programming languages has been documented by Ryder et al. [2005]. A programming language support for handling errors first appeared in LISP in 1950s in the form of the `ERRSET` construct [Steele and Gabriel 1996]. An expression wrapped in the `ERRSET` construct would evaluate to a `NIL` value when the computation failed to evaluate normally. The mechanism allowed ignoring errors, but it did not provide an indication of what kind of error occurred. Reliability and safety were important considerations for the design of PL/1, which introduced a way of handling unusual situations via the `ON` construct. This allowed handling 23 conditions such a numerical overflow or an end of file.

In LISP and PL/1, the error handling constructs likely appeared as convenient constructs for handling certain anticipated situations, without a more ambitious engineering motivation. In other words, they were a product of the hacker culture of programming. However, as noted by Ryder et al. [2005], the next development in exception handling constructs was more directly influenced by the work in software engineering and the engineering culture.

An exception handling construct that shares many aspects with modern exceptions as known from, for example, Java has been proposed by Goodenough [1975]. The work appeared first in the Principles of Programming Languages conference and later in the Communications of the ACM, but it was written by a software engineering researcher working on issues of software reliability [Ryder et al. 2005]. Goodenough introduced a way of attaching exception handlers to syntactic units of code. The exceptions included a number of language-defined exceptions (e.g. `OVERFLOW`),

but it also allowed programmer-defined exceptions. Goodenough's theoretical paper had a direct influence on the design of exceptions in the CLU language, which became available at the end of 1970s [Liskov 1993].

It is worth noting that a large fraction of exception handling mechanism ended up being used for other purposes than just handling of exceptions. Steele and Gabriel [1996] point out that programmers began to use ERRSET in LISP "not to trap and signal errors but for more general control purposes (dynamic non-local exits)"; Goodenough [1975] points out that exception handling mechanisms "are needed, in general, as a means of conveniently interleaving actions belonging to different levels of abstraction" and, finally, Liskov [1993] reflects that exceptions in CLU are "a general way of conveying information from a called procedure to its caller." It is a curious fact that a similar situation occurred in three distinct programming languages. One interpretation is that exceptions were motivated by a practical concern of the engineering programming culture, but their integration into the hacker culture of programming languages invited other applications that were unanticipated by the engineers and became apparent to the hackers with their arcane practical knowledge.

Exception handling might have initially been conceived as a mechanism for dealing with system faults such as numerical overflows or reading of data after the end of the file, but it soon became useful as a more general over-engineering method. Goodenough [1975] classified exceptions into two categories. "Range failure occurs when an operation either finds it is unable to satisfy its output assertion" and "Domain failure (...) occurs when an operation's inputs fail to pass certain tests of acceptability." In an ideal world imagined by the mathematical culture of programming, a domain failure would provably never happen. Triggering exceptions in response to a domain failure is thus an early form of an over-engineering mechanism known as defensive programming. The software engineering research on software reliability inspired a number of other over-engineering methods. Randell [1975] introduced the idea of recovery blocks, which is a programming language feature explicitly designed to build fault-tolerant systems that can recover from design inadequacies. A recovery block is similar to exception handler, but the body is guarded by an acceptance condition (a check for a range failure). If the condition is violated by the primary implementation, the block proceeds to execute one of the alternative, possibly simplified, implementations. Another over-engineering mechanism has been proposed by Yau and Cheung [1975] who advocate the development of self-checking mechanisms where a system actively monitors itself to detect range and domain failures.

3.3 Embracing failures

The developments described in the previous section accept that unexpected situations might happen, but they generally aim to recognise them and recover from them as early as possible. However, a number of programming languages and development methodologies take the idea of accepting errors even further. The Erlang language (Armstrong, 2007), which started in 1986, was designed as a language for programming switching systems in the telecommunications industry. As such, it had to accept that hardware failures will happen.

Armstrong [2007] reflects that "error handling in Erlang is very different from error handling in conventional programming languages", because "you cannot build a fault-tolerant system if you only have one computer." Erlang programs are structured as sets of lightweight processes, which may execute on separate machines. When a process gets into an unexpected state (as a result of hardware failure or programming mistake), the programmer should let the process crash [Armstrong 2003]. The crash will then be handled by a dedicated supervisor process that can choose to restart the process, propagate the error further or activate another recovery mechanism.

In Erlang, failure of a process is embraced. It is not an exception that we hope to avoid, but instead, a useful mechanism for building reliable fault-tolerant systems.

The idea of embracing failures has been used in a number of other engineering contexts. Modern large-scale distributed systems are often based on microservices, which are independent components that are developed and deployed independently. Microservices are expected to occasionally fail and the infrastructure should be able to restart them. The microservice architecture encourages redundancy and fault tolerance, but there is one more method needed to increase the resilience of the system. As argued by Tseitlin [2013] the resilience of such systems also relies on regularly inducing failure in order to reduce uncertainty.

An automatic system that induces failure at a large scale has been first implemented by Netflix under the name of "Chaos Monkey". A Chaos Monkey is an agenda that shuts down virtual machines in the production environment at random. This ensures the reliability not just of the infrastructure, but also the human element. The increased frequency of failures make it possible to develop better methods for coping with them and, consequently, the socio-technological system becomes more resilient through failure. The idea that a system should be able to learn from errors and become stronger thanks to errors has been termed *anti-fragility* and is a subject to active research [Tseitlin 2013].

In case of Erlang, failures are embraced merely by the engineering culture as a useful programming methodology. Chaos Monkey as implemented by Netflix adapts a concept born in the engineering culture to a more general managerial programming culture. Chaos Monkey does not just affect systems, but it also influences the structure and priorities of the developer teams that build the systems.

4 INFORMATION HIDING

The cases of testing and error handling are similar in that they both originated with a technical notion created within the hacker culture, but then acquired a new meaning as part of engineering or managerial methods. Their technical side becomes a component in a more general development methodology or a framework. This pattern repeats itself for a number of other technical ideas that contributed to making large software systems reliable. In this section, we use this perspective to take a fresh look at the idea of "abstract data types" that we introduced in the lecture on types.

In the context of software engineering, Parnas [1972] proposed a way of decomposing systems into independent modules that hide information about their internal data representation from other modules. Parnas already envisioned that this way of structuring systems would make the independent development of such modules easier. In parallel, abstract data types, which support information hiding as envisioned by Parnas were implemented in the Clu language by Liskov and Zilles [1974]. Liskov and Zilles introduced the idea of abstract data types more as a mechanism for extending the langauge with new data types, than as a language construct to support certain software engineering practice. However, it was soon recognised, for example by DeRemer and Kron [1976] that abstract data types can be used as "a means of communication between members of a programming team."

The influence that the ideas of modularity and information hiding had on programming languages during 1980s has been documented by Ryder et al. [2005]. Object-oriented programming languages adopted the idea of information hiding as one of the key design principles. One of the three characteristics of object-oriented languages given by Cardelli and Wegner [1985] is that "they contain objects that are data abstractions with a defined interface and hidden state." Information hiding in object-oriented programming languages also supports a certain managerial structure of software development. Systems can be decomposed into independent components that communicate

via a public interface between them. This means that components can be built by independent teams that only need to agree on a public interface between components.

More recently, the same idea resurfaced in the architecture of large distributed systems. The microservice architecture decomposes systems into small loosely connected services. Such services are developed and maintained by independent teams who gain more autonomy over how the service is built. Similarly to testing, the ideas discussed in this section originated within the more technical cultures of programming, but gained both engineering and managerial interpretations. The engineering culture is mainly concerned with a programming language mechanism that makes it possible to structure software systems better, while the managerial culture is concerned with a better way of structuring development teams. Abstract data types, object-oriented languages and microservices gained popularity because they allow these two ideas to coincide.

5 THE HOARE PARADOX

This lecture examines an interesting observation about the computing industry that MacKenzie [2004] referred to as the "Hoare paradox". Many proponents of the mathematical culture of programming find it surprising that "although software seldom has had proof applied to it (...), its dependability is often adequate to the tasks for which it is actually used." Moreover, "faulty software appears to have killed remarkably few people; security failures have generally been embarrassing rather than catastrophic; in areas such as telephony and avionics, real-time systems, containing millions or tens of millions of lines of program, function tolerably well." [MacKenzie 2004, p.301]

Each of the cultures of programming has their own, strongly held, beliefs on what methods should be used to build reliable software. The mathematical culture beliefs this should be done through formal proof; the hacker culture beliefs in the skills of individuals; the engineering culture in the development of appropriate tools and practices and the managerial culture favors designing a robust organisational structure. However, the cultures always mix and many ideas that originate in one culture are interpreted and used differently by another culture. In practice, the tools, methods and practices that make modern software "function tolerably well" are very often a product of mixture of cultures.

In an earlier lecture, we discussed the concept of types at great length. Types appeared, likely independently, in the mathematical and hacker culture, but they were soon influenced by the engineering culture and abstract data types even had managerial implications. As discussed, types in modern programming languages are very diverse and so they contribute to reliability of software in different ways, be it by providing formal guarantees or by providing a useful engineering tool. In this lecture, we extended our discussion to cover a number of other concepts and practices. We focused on those that can be used to find, eliminate or otherwise address bugs in programs, including debugging, testing and error recovery.

The first two strands discussed in the lecture start with "program checkout" as practiced by the hacker culture in 1950s. This later evolved into two separate ideas, debugging and testing. While debugging remained mostly a hacker practice, testing was adapted by the managerial culture in the 1970s and 1980s and became a part of the application development lifecycle. Later, it also became a key component of the engineering practice of test-driven development. The throed strand that we discussed was exception handling, which also originated with the hacker culture, but later acquired new engineering interpretations and, in recent years, also served as a source for ideas developed in the managerial culture.

Comparing the history of debugging and testing, raises an interesting question. Why do some of the concepts cross cultures and evolve more than others? We offer two potential answers. First, testing is done through tests, which can be seen as technical artifacts that are, to some extent, independent of cultures. This situation seems similar to the case of types which are also shared,

but seen differently by different cultures. As pointed out earlier, this makes them boundary objects in a sociological sense. The other possible explanation is that the different cultures favor different kinds of knowledge. In particular, the hacker culture often relies on personal skills that are learned through practice and are not (or even cannot be) written down.

CONCLUSIONS: CULTURES OF PROGRAMMING

The goal of this paper was to document the wider context within which programming languages are designed and developed. As such, we did not focus on a specific language, language feature or a family of languages. Instead, we aimed to understand the what is the substrate from which programming languages are born. To do this, we had to broaden the scope of the paper. To understand the design of some programming languages or language features, we not only have to discuss the technical choices, but we also have to understand the debates happening in the computing community and the computing ecosystem. For example, why did record types appear in Flow-Matic well before they appeared in Algol? Or, how did it happen that type systems of several recently developed programming languages are intentionally unsound?

The key idea presented in this paper is that many interesting programming languages, language features and other technical artifact, as well as interesting methodologies and controversies happen as a result of the mixing of several cultures of programming. Before computer science and software engineering became disciplines, most people working with computers came from a variety of backgrounds including logic, electrical engineering and business. However, these different cultures brought with them unique perspectives that have never been assimilated into a single all-encompassing culture. We believe that this is not a sign that computing is an immature field, but instead, a sign that it developed a fruitful multi-cultural identity.

As documented in this paper, when cultures meet, interesting things happen. We focused on two kinds of interesting things: technical artifacts and controversies. The most notable technical artifacts that we discussed in the paper were programming languages themselves, types, tests and exceptions. Over their history, all of these were influenced by multiple cultures of programming that adapted them for their own purposes and, often, extended them with a new meaning that ended up being useful in another context. The controversies that we followed in this paper focused on the feasibility of proving programs correct using formal mathematical methods and on the possibility of building reliable systems that satisfy certain properties.

1 CULTURES LEFT OUT FROM THIS PAPER

The cultures that we considered in this paper are the hacker culture, mathematical culture, engineering culture and managerial culture. We also occasionally considered an idea that came from outside of these, adding an artistic or philosophical perspective. To keep this paper focused, we restricted ourselves to cultures that played an important role in the controversies and technical artifacts we discussed. As a result, this paper has omitted certain perspectives.

In particular, there certainly is an artistic culture of programming within which computers are used as a medium for creativity. This culture has, so far, been somewhat marginal, perhaps because of the financial incentives in the computing industry. However, the artistic culture of programming had certainly influenced programming languages such as LOGO [Feurzeig and Papert 1969], Smalltalk [Goldberg and Robson 1983] and, more recently, work on live coding environments such as Supercollider and Sonic PI [Aaron and Blackwell 2013; McCartney 2002]. Another culture that has been influential in the early days of computing is the one that gave birth to cybernetics, a multi-disciplinary study of self-regulatory systems. Despite its prominence in 1950s and 1960s, it seems that cybernetics has had little influence on the design of programming languages as of yet.

2 CHARACTERISTICS OF PROGRAMMING CULTURES

The four programming cultures that we identified in this paper are well recognizable based on the principles they follow, methods they advocate and ways of working with knowledge. That said, historical actors do not necessarily belong to just one programming culture. For a number of

people quoted in this paper, their allegiance to a certain culture is clear and does not change over time. However, there is certainly not the case for everyone.

What are the key characteristics of the four cultures that we discussed? The hacker culture puts a strong emphasis on the skills of an individual. The skills are learned through practice and are of a form that is difficult to formalise or write down. Polanyi [1958] labels this as personal knowledge and discusses its importance in many areas including science and law. However, to an external observer, the personal knowledge of the hacker culture often appears as "black art". The hacker culture had a strong influence on programming in 1950s when working with computers involved a lot of tinkering, but it remains important for producing reliable software and plays a key role in practices such as debugging.

As the name suggests, the mathematical culture views programs as mathematical entities and advocates the use of formal methods and proofs for producing correct programs. As noted by Priestley [2011], the mathematical culture gained prominence with the appearance of the Algol programming language. The knowledge that the mathematical culture develops is written in the formal language of mathematics, which shifts focus on identifying common patterns and structures across multiple areas of programming, at the expense of informal aspects. To an outsider of the culture, it often seems that "authors in this field, who in the formal aspects of their work require painstaking demonstration and proof, in the informal aspects are satisfied with subjective claims that have not the slightest support, neither in argument nor in verifiable evidence." [Naur 1993] One interesting observation about the mathematical culture is that its proponents often held very strong beliefs in its principles. This is likely due to the intellectual appeal of mathematical knowledge that Lakoff and Núñez [2000] calls "The Romance of Mathematics".

The engineering and managerial cultures are perhaps the most closely related ones of the four cultures we discussed. They both believe in improving the quality of software by developing and following suitable methodologies. These often consist of processes that should be followed and tools that can be used to guide and automate some aspects of the process. However, the managerial culture focuses on developing processes that provide control over the workforce and often focus on training and structure of teams. The goal is to have a process that, on average, eliminates the reliance on individual skills. In contrast, the engineering culture focuses on finding small-scale processes that an individual or a team can follow to produce more robust and maintainable software.

3 THE NATURE OF PROGRAMMING CULTURES

We were able to identify four different cultures of programming, but so far, we avoided defining what a culture of programming actually is. The term *culture of programming* has been inspired by the "Cultures of Proving" paper by Livingston [1999], who describes components of one specific mathematical culture. However, just like this paper, Livingston relies on the common sense meaning of the term 'culture'. In this paper, cultures of programming consist of shared principles and beliefs, that are concerned with the nature and practice of programming.

We can better understand the nature of programming cultures by relating them to established ideas from philosophy of science, such as research programmes and research paradigms which are both highly relevant to our discussion about cultures. The concept of research paradigm was introduced by Kuhn and Hacking [1962]. According to Kuhn, normal science is governed by a single paradigm that sets standard for legitimate scientific work. A paradigm is implicit and is formed by assumptions, methods and practices that scientists learn during their training. Normal science is concerned with solving problems (or puzzles) within the scope of the paradigm. A paradigm shift is a slow process that happens when a paradigm accumulates enough problems that it cannot effectively solve using its own methods. A research programme introduced by Lakatos et al. [1980] is an attempt to reconcile Kuhn's ideas with Popper's falsificationism [Popper 1959]. The idea of a

research programme recognizes that some principles are more basic than others. Faced with an experimental failure, a scientist never blames such hard core assumptions, but instead addresses the issue by modifying some of the additional assumptions provided by the protective belt of the theory. Due to the different hard core assumptions, the work arising from different research programmes is to some degree incommensurable.

The idea of cultures presented in this paper differs from both research paradigms and research programmes. Unlike a research paradigm, multiple cultures can coexist and exchange ideas. A single culture does not govern all work done within normal science. Unlike a research programme, cultures affect a larger community than, say, a number of research groups. The principles and beliefs are broader and less strict than hard cores of research programmes. The principles of a programming culture can materialize in a number of more specific research programmes and a research programme can also combine ideas from multiple cultures.

Does this mean that the cultures of programming, as introduced in this paper, are an overly general concept that admits almost anything and does not, in fact, teach us much about the history of programming? I believe that this is not the case. The controversies discussed in this paper are a historical fact and seeing them as a clash between cultures of programming provides a constructive way of analysing and explaining them. Similarly, the idea of programming cultures allowed us to shed a new light at the historical developments concerning technical artifacts and programming langauge features such as types, tests or exceptions.

REFERENCES

Samuel Aaron and Alan F. Blackwell. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/2505341.2505346

Chris Aitchison. 2011. You are NOT a Software Engineer! http://chrisaitchison.com/2011/05/03/you-are-not-a-software-engineer/ [Online; accessed 10-September-2018].

C. D. Allen, D. N. Chapman, and Cliff B. Jones. 1972. *A formal definition of ALGOL 60.* Technical Report 12.105. IBM Laboratory Hursley. http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR12.105.pdf

Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph.D. Dissertation. Mikroelektronik och informationsteknik.

Joe Armstrong. 2007. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 6–1–6–26. https://doi.org/10.1145/1238844.1238850

Troy Astrate. 2017. Towards an Interconnected History of Semantics. In *Fourth International Conference on the History and Philosophy of Computing*. Brno, Czech Republic.

Charles Babbage. 1982. On the mathematical powers of the calculating engine. In *The origins of digital computers*. Springer, 19–54.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. 1963. Revised Report on the Algorithm Language ALGOL 60. *Commun. ACM* 6, 1 (Jan. 1963), 1–17. https://doi.org/10.1145/366193.366201

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and L. Ziller. 1956. *Fortran: Automatic Coding System for the IBM 704 EDPM.*

Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001).

H. D. Benington. 1983. Production of Large Computer Programs. *Annals of the History of Computing* 5, 4 (Oct 1983), 350–361. https://doi.org/10.1109/MAHC.1983.10102

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–130.

Bruce I. Blum. 1989. Formalism and Prototyping in the Software Process. *Information and Decision Technologies* 15 (1989), 327–341.

Alan Borning. 1991. Computerization and Controversy. Academic Press Professional, Inc., San Diego, CA, USA, Chapter Computer System Reliability and Nuclear War, 560–592.

Robert Bosak, Richard F. Clippinger, Carey Dobbs, Roy Goldfinger, Renee B. Jasper, William Keating, George Kendrick, and Jean E. Sammet. 1962. An Information Algebra: Phase 1 Report&Mdash;Language Structure Group of the CODASYL Development Committee. *Commun. ACM* 5, 4 (April 1962), 190–204. https://doi.org/10.1145/366920.366935

Fred Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20, 4 (April 1987), 10–19. https://doi.org/10.1109/MC.1987.1663532

Herbert Butterfield. 1965. *The Whig interpretation of history.* WW Norton & Company.

J.N. Buxton, B. Randell, and NATO Science Committee. 1970. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee; available from Scientific Affairs Division, NATO.

M. Campbell-Kelly. 1992. The Airy Tape: An Early Chapter in the History of Debugging. *IEEE Annals of the History of Computing* 14 (10 1992), 16–26. https://doi.org/10.1109/85.194051

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042

Ryan Cavanaugh. 2018. Type-checking unsoundness: standardize treatment of such issues among TypeScript team/community? https://github.com/Microsoft/TypeScript/issues/9825#issuecomment-234115900 [Online; accessed 10-September-2018].

A.F. Chalmers. 2013. *What Is This Thing Called Science?* Hackett Publishing Company, Incorporated.

Alonzo Church. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 2 (1940), 56âĂŞ68. https://doi.org/10.2307/2266170

Avra Cohn. 1989. The notion of proof in hardware verification. *Journal of Automated Reasoning* 5, 2 (01 Jun 1989), 127–139. https://doi.org/10.1007/BF00243000

R L Constable. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Thierry Coquand and Gerard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2–3 (1988).

Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social Processes and Proofs of Theorems and Programs. *Commun. ACM* 22, 5 (May 1979), 271–280. https://doi.org/10.1145/359104.359106

F. DeRemer and H. H. Kron. 1976. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* SE-2, 2 (June 1976), 80–86. https://doi.org/10.1109/TSE.1976.233534

Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM* 11, 3 (March 1968), 147–148. https://doi.org/10.1145/362929.362947

Edsger W. Dijkstra. 1972. The Humble Programmer. *Commun. ACM* 15, 10 (Oct. 1972), 859–866. https://doi.org/10.1145/355604.361591

M. Dyer and Harlan D. Mills. [n. d.]. Cleanroom Software Development. In *Sixth Annual Software Engineering Workshop* (1981). NASA.

N.L. Ensmenger. 2012. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise.* MIT Press.

Thomas G. Evans and D. Lucille Darley. 1965. DEBUG–an Extension to Current Online Debugging Techniques. *Commun. ACM* 8, 5 (May 1965), 321–326. https://doi.org/10.1145/364914.364952

Thomas G Evans and D Lucille Darley. 1966. On-line debugging techniques: a survey. In *Proceedings of the November 7-10, 1966, fall joint computer conference.* ACM, 37–50.

James H. Fetzer. 1988. Program Verification: The Very Idea. *Commun. ACM* 31, 9 (Sept. 1988), 1048–1063. https://doi.org/10.1145/48529.48530

W. Feurzeig and S. Papert. 1969. *Programming languages as a conceptual framework for teaching mathematics. Final report on the first fifteen months of the Logo Project.* Technical Report Technical Report 1889. BBN, Cambridge, MA.

P Feyerabend. 1975. *Against Method.* Verso.

Christiane Floyd. 1987. Outline of a Paradigm Change in Software Engineering. In *Computers and Democracy: A Scandanavian Challenge*, G. Bjerknes (Ed.). Brookfield, Gower Publishing Company, Old Post Road, Brookfield, USA, 191–210.

B. Foote and R. E. Johnson. 1989. Reflective Facilities in Smalltalk-80. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89).* ACM, New York, NY, USA, 327–335. https://doi.org/10.1145/74877.74911

Future of Life Institute. 2015. Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. https://futureoflife.org/ai-open-letter/

Richard P. Gabriel. 2006. Design Beyond Human Abilities. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development (AOSD '06).* ACM, New York, NY, USA, 2–2. https://doi.org/10.1145/1119655.1119658

Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012).* ACM, New York, NY, USA, 195–214. https://doi.org/10.1145/2384592.2384611

D. Gelperin and B. Hetzel. 1988. The Growth of Software Testing. *Commun. ACM* 31, 6 (June 1988), 687–695. https://doi.org/10.1145/62959.62965

B. Gold and R.A. Simons. 2008. *Proof and Other Dilemmas: Mathematics and Philosophy.* Mathematical Association of America.

Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing.

John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (Dec. 1975), 683–696. https://doi.org/10.1145/361227.361230

John B. Goodenough and Susan L. Gerhart. 1975. Toward a Theory of Test Data Selection. In *Proceedings of the International Conference on Reliable Software.* ACM, New York, NY, USA, 493–510. https://doi.org/10.1145/800027.808473

Mike Gordon. 2000. From LCF to HOL: a short history.. In *Proof, language, and interaction.* 169–186.

Michael J. C. Gordon. 1988. *HOL: A Proof Generating System for Higher-Order Logic.* Springer US, Boston, MA, 73–128. https://doi.org/10.1007/978-1-4613-2007-4_3

M.E. Gorman. 2010. *Trading Zones and Interactional Expertise: Creating New Kinds of Collaboration.* MIT Press.

I. Hacking. 1983. *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science.* Cambridge University Press.

Mark Halpern. 1965. Computer programming: The debugging epoch opens. *Computers and Automation* 14, 11 (November 1965), 28–31.

Wolfgang Henhapl and Cliff B. Jones. 1978. *A formal definition of ALGOL 60 as described in the 1975 modified report.* Springer Berlin Heidelberg, Berlin, Heidelberg, 305–336. https://doi.org/10.1007/3-540-08766-4_12

Alex Hern. 2018. Google's solution to accidental algorithmic racism: ban gorillas. https://www.theguardian.com/technology/2018/jan/12/google-racism-ban-gorilla-black-people

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

C. A. R. Hoare. 1972. Structured Programming. Academic Press Ltd., London, UK, UK, Chapter Chapter II: Notes on Data Structuring, 83–174.

C. A. R. Hoare. 1985. The Mathematics of Programming. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1–18.

C. A. R. Hoare. 1996. How did software get so reliable without proof?. In *FME'96: Industrial Benefit and Advances in Formal Methods*, Marie-Claude Gaudel and James Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

W. Howden. 1982. Life-Cycle Software Validation. *Computer* 15 (02 1982), 71–78. https://doi.org/10.1109/MC.1982.1653943

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Cliff B. Jones and Troy K. Astarte. 2016. *An Exegesis of Four Formal Descriptions of ALGOL 60.* Technical Report CS-TR-1498. Newcastle University School of Computer Science. https://assets.cs.ncl.ac.uk/TRs/1498.pdf Forthcoming as a paper in the HaPoP 2016 proceedings.

Jana Kasperkevich. 2015. Google says sorry for racist auto-tag in photo app. https://www.theguardian.com/technology/2015/jul/01/google-sorry-racist-auto-tag-photo-app

Stephen Kell. 2014. In Search of Types. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 227–241. https://doi.org/10.1145/2661136.2661154

Andrew John Kennedy. 1996. *Programming languages and dimensions*. Ph.D. Dissertation. University of Cambridge, Computer Laboratory.

P. A. Kidwell. 1998. Stalking the elusive computer bug. *IEEE Annals of the History of Computing* 20, 4 (Oct 1998), 5–9. https://doi.org/10.1109/85.728224

D.E. Knuth. 1968. *The Art of Computer Programming*. Vol. 1-4. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

A. Kotok. 1964. PDP-1 program library. http://www.computerhistory.org/collections/catalog/102750686 Computer History Museum, catalog no. 102750686 [Online; accessed 10-September-2018].

Douglas Kramer. 1999. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation (SIGDOC '99)*. ACM, New York, NY, USA, 147–153. https://doi.org/10.1145/318372.318577

T.S. Kuhn and I. Hacking. 1962. *The Structure of Scientific Revolutions: 50th Anniversary Edition.* University of Chicago Press.

I. Lakatos, J. Worrall, and G. Currie. 1980. *The Methodology of Scientific Research Programmes: Volume 1: Philosophical Papers.* Cambridge University Press.

I. Lakatos, J. Worrall, G. Currie, E. Zahar, and Cambridge University Press. 1976. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge University Press.

G. Lakoff and R.E. Núñez. 2000. *Where Mathematics Comes from: How the Embodied Mind Brings Mathematics Into Being.* Basic Books.

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

P. J. Landin. 1965a. Correspondence Between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM* 8, 2 (Feb. 1965), 89–101. https://doi.org/10.1145/363744.363749

P. J. Landin. 1965b. A Correspondence Between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM* 8, 3 (March 1965), 158–167. https://doi.org/10.1145/363791.363804

Peter J Landin. 1966a. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*. North Holland Publishing Company, Amsterdam, 266–294.

P. J. Landin. 1966b. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

Peter E. Lauer. 1968. *Formal definition of ALGOL 60.* Technical Report 25.088. IBM Laboratory Vienna. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Lau68.pdf

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

J. C. Licklider. 1969. Underestimates and overexpectations. *Computers and Automation* 18, 9 (1969), 48–52.

J. C. R. Licklider. 1960. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics* HFE-1, 1 (March 1960), 4–11. https://doi.org/10.1109/THFE2.1960.4503259

Barbara Liskov. 1993. A History of CLU. In *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, New York, NY, USA, 133–147. https://doi.org/10.1145/154766.155367

Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/800233.807045

Eric Livingston. 1999. Cultures of Proving. *Social Studies of Science* 29, 6 (1999), 867–888. https://doi.org/10.1177/030631299029006003

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

Donald MacKenzie. 2001. *Mechanizing Proof: Computing, Risk, and Trust.* MIT Press, Cambridge, MA, USA.

D.A. MacKenzie. 2004. *Mechanizing Proof: Computing, Risk, and Trust.* MIT Press.

Simone Martini. 2016. Several Types of Types in Programming Languages. In *History and Philosophy of Computing*, Fabio Gadducci and Mirko Tavosanis (Eds.). Springer International Publishing, Cham, 216–227.

P. McBreen. 2002. *Software Craftsmanship: The New Imperative.* Addison-Wesley.

Conor McBride. 2008. Clowns to the Left of Me, Jokers to the Right (Pearl): Dissecting Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* ACM, New York, NY, USA, 287–295. https://doi.org/10.1145/1328438.1328474

John McCarthy. [n. d.]a. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming* (1964). North-Holland Publishing Company, 1–12.

John McCarthy. [n. d.]b. Towards a Mathematical Science of Computation. In *Information Processing* (1963), C. M. Popplewell (Ed.). North-Holland Publishing Company, 21–28.

John McCarthy. 1959. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 26. Elsevier, 33 – 70. https://doi.org/10.1016/S0049-237X(09)70099-0

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68. https://doi.org/10.1162/014892602320991383

Erik Meijer and Peter Drayton. 2005. Static Typing Where Possible, Dynamic Typing When Needed. In *Workshop on Revival of Dynamic Languages.*

Robin Milner. 1972. *Logic for Computable Functions: Description of a Machine Implementation.* Technical Report. Stanford, CA, USA.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1979. Lcf: A way of doing proofs with a machine. In *Mathematical Foundations of Computer Science 1979*, Jiří Bečvář (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–159.

R. Milner. 1999. *Communicating and Mobile Systems: The Pi Calculus.* Cambridge University Press.

J. Strother Moore. 1989. A mechanically verified language implementation. *Journal of Automated Reasoning* 5, 4 (01 Dec 1989), 461–492. https://doi.org/10.1007/BF00243133

James H. Morris, Jr. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73).* ACM, New York, NY, USA, 120–124. https://doi.org/10.1145/512927.512938

P. D. Mosses. 1974. *The Mathematical Semantics of ALGOL 60.* Technical Report Technical Monograph PRG-12. Oxford University Computing Laboratory, Programming Research Group. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Mosses74.pdf

J. D. Musa. 1975. A theory of software reliability and its application. *IEEE Transactions on Software Engineering* 1 (09 1975), 312–327. https://doi.org/10.1109/TSE.1975.6312856

Glenford J Myers, T Badgett, and C Sandler. 1979. The Art of Software Testing. John Wiley & Sons, Inc. *New York* (1979), 22041–3467.

National Bureau of Standards. 1984. *Guideline for Lifecycle Validation, Verification, And Testing of Computer Software.* Technical Report. U.S. Dept. of Commerce, National Bureau of Standards.

Peter Naur. 1993. The place of strictly defined notation in human insight. In *Program Verification.* Springer, 261–274.

P. Naur, B. Randell, F.L. Bauer, and NATO Science Committee. 1969. *Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.* Scientific Affairs Division, NATO.

David Nofre. 2018. IBM and the Algol project: Exploring the challenges and dilemmas of early computer science research. (2018). https://www.shift-society.org/hapop4/abstracts/nofre.pdf Fourth Symposium on the History and Philosophy of Programming.

David Nofre, Mark Priestley, and Gerard Alberts. 2014. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture* 55, 1 (2014), 40–75.

Linda Northrop, Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, et al. 2006. *Ultra-large-scale systems: The software challenge of the future.* Technical Report. Carnegie-Mellon University, Pittsburth, Software Engineering Institute.

Hannah Jane Parkinson. 2016. Google translates Russia to 'Mordor' and minister's name to 'sad little horse'. https://www.theguardian.com/technology/2016/jan/07/google-translates-russia-mordor-foreign-minister-ukrainian

D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

David Lorge Parnas. 1985. Software Aspects of Strategic Defense Systems. *Commun. ACM* 28, 12 (Dec. 1985), 1326–1335. https://doi.org/10.1145/214956.214961

Tomas Petricek. 2015. Against a Universal Definition of 'Type'. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 254–266. https://doi.org/10.1145/2814228.2814249

Tomas Petricek. 2017. Miscomputation in software: Learning to live with errors. *The Art, Science, and Engineering of Programming* 1, 2 (2017), 14.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

B.C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

B.C. Pierce. 2005. *Advanced Topics in Types and Programming Languages*. MIT Press.

Michael Polanyi. 1958. *Personal knowledge*. University of Chicago Press.

Karl Popper. 1959. *The logic of scientific discovery*. Routledge.

John A. Postley. 1960. Letters to the Editor. *Commun. ACM* 3, 1 (Jan. 1960), 0.06–. https://doi.org/10.1145/366947.366948

M. Priestley. 2011. *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer London.

B Randell. 1975. System structure for software fault tolerance. *IEEE Transactions on Software Engineering* SE-1, 2 (June 1975), 220–232. https://doi.org/10.1109/TSE.1975.6312842

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*, B. Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–425.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

Daniel Rosenwasser. 2018. Announcing TypeScript 3.0. https://blogs.msdn.microsoft.com/typescript/2018/07/30/announcing-typescript-3-0/ [Online; accessed 10-September-2018].

Bertrand Russell. 1908. Mathematical Logic as Based on the Theory of Types. *American Journal of Mathematics* 30, 3 (1908), 222–262.

Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. 2005. The Impact of Software Engineering Research on Modern Progamming Languages. *ACM Trans. Softw. Eng. Methodol.* 14, 4 (Oct. 2005), 431–477. https://doi.org/10.1145/1101815.1101818

Stephen A. Schuman and Philippe Jorrand. 1970. Definition Mechanisms in Extensible Programming Languages. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference (AFIPS '70 (Fall))*. ACM, New York, NY, USA, 9–20. https://doi.org/10.1145/1478462.1478465

R. Slayton. 2013. *Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012*. MIT Press.

Brian Cantwell Smith. 1985. *Limits of Correctness in Computers*. Technical Report CSLI-85-36. The Center for the Study of Language and Information, Stanford, CA.

C. Solis and X. Wang. 2011. A Study of the Characteristics of Behaviour Driven Development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. 383–387. https://doi.org/10.1109/SEAA.2011.76

Susan Leigh Star and James R. Griesemer. 1989. Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science* 19, 3 (1989), 387–420. https://doi.org/10.1177/030631289019003001

Guy L. Steele and Richard P. Gabriel. 1996. History of Programming languages—II. ACM, New York, NY, USA, Chapter The Evolution of Lisp, 233–330. https://doi.org/10.1145/234286.1057818

Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in Information-rich Functional Programming for Internet-scale Data Sources. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming (DDFP '13)*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/2429376.2429378

J.P. Talpin and P. Jouvelot. 1994. The Type and Effect Discipline. *Information and Computation* 111, 2 (1994), 245 – 296. https://doi.org/doi.org/10.1006/inco.1994.1046

Ariel Tseitlin. 2013. The Antifragile Organization. *Commun. ACM* 56, 8 (Aug. 2013), 40–44. https://doi.org/10.1145/2492007.2492022

Raymond Turner and Nicola Angius. 2017. The Philosophy of Computer Science. In *The Stanford Encyclopedia of Philosophy* (spring 2017 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

Univac. 1957. Introducing a New Language for Automatic Programming Univac Flow-Matic. http://www.computerhistory.org/collections/catalog/102646140 Computer History Museum, catalog no. 102646140 [Online; accessed 10-September-2018].

Peter Varhol. 2018. To agility and beyond: The history—and legacy—of agile development. https://techbeacon.com/agility-beyond-history%E2%80%94-legacy%E2%80%94-agile-development [Online; accessed 10-September-2018].

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 310–325. https://doi.org/10.1145/2908080.2908110

M. Mitchell Waldrop. 2001. *The Dream Machine: J.C.R. Licklider and the Revolution That Made Computing Personal*. Viking Penguin.

Hao Wang. 1960. Toward Mechanical Mathematics. *IBM J. Res. Dev.* 4, 1 (Jan. 1960), 2–22. https://doi.org/10.1147/rd.41.0002

Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.

Wikipedia contributors. 2018. Formal proof — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Formal_proof&oldid=841416708 [Online; accessed 10-September-2018].

M.V. Wilkes and Eric Weiss Collection. 1985. *Memoirs of a Computer Pioneer*. MIT Press.

N. Wirth. 1971. The programming language pascal. *Acta Informatica* 1, 1 (01 Mar 1971), 35–63. https://doi.org/10.1007/BF00264291

D. Wootton. 2015. *The Invention of Science: A New History of the Scientific Revolution*. Penguin Books Limited.

S. S. Yau and R. C. Cheung. 1975. Design of Self-checking Software. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 450–455. https://doi.org/10.1145/800027.808468