Carl Stalling: This is ACM ByteCast, a podcast series from the Association for Computing Machinery, the world's largest education and scientific computing society. Talk to researchers, practitioners, and innovators who are at the intersection of computing research and practice. They share their experiences, the lessons they've learned, and their own visions for the future of computing. I'm your host, Carl Stalling.

Our next guest is Xavier Leroy. Xavier Leroy is a French computer scientist and programmer, best known for his role as a primary developer of the OCaml system and programming language. He's a professor of software science at the Collège de France since 2018. Before that, he was a senior scientist at the INRIA. He studied mathematics and computer science at the École Normale Supérieure in Paris, receiving a PhD in computer science in 1992. He's an expert on functional programming languages and compilers. In recent years, he's taken an interest in formal methods, formal approved and certified compilation. He is the leader of the CompSUR project that develops an optimizing compiler for the C programming language. Formally verified in COQ. In 2015, he was named fellow of the association for Computing Machinery for contributions to save high performance functional programming languages and compilers and to compile a verification. He has also received the 2016 Milner Award by the Royal Society, the 2021 ACM Software Systems Award, and 2022 ACM SIG Plan Programming Languages Achievement Award. It is a very great pleasure to have you on [foreign language 00:01:41] podcast.

Xavier Leroy: Thank you for this invitation to talk with you and for the nice introduction. It's a pleasure to be here, too.

Carl Stalling: Of course. I'll have to ask about OCaml. I've been using ML style languages a fair bit myself back in my day when I was still programming, but also for teaching. To me, OCaml always felt a lot more like a practical tool for working programmers than SMLC. There were all these small details that really made a difference to me when I was actually working. That made me think when they created this, they were really thinking about a working programmer. They really cared for the usability for the practical tools. I wonder how much of concern was that when you designed OCaml, this practical perspective of it?

Xavier Leroy: Well, it was certainly on my mind. Actually, let me go back to my first encounter with ML-like languages. In France, that area in the book where I ended up doing my PhD thesis, there was already some work on an ML language called Caml, which was very much inspired by Robin Milner's work like Standard ML of New Jersey. Where it was a heavyweight was running only on workstations, but still it was a beautiful language. When I was exposed to it as a grad student, I immediately fell in love with it. My first object with another grad student called Daniel Delije was to do a lightweight implementation of CamL. Simplifying a bit the language and then having a very simple runtime system written in C, by code compilation, really a minimal implementation. The first goal was to teach ourselves how it works, and that was very instructive, but it ended up being also

the first open source implementation. When we distributed it, it worked well for teaching and popularizing the language.

One of the design criteria was that to run on the PCs and the Macs of the day in one megabyte of ram. That was some of the practical constraints that we took into account. Also, since it was pretty small, it was a very good vehicle for research and experimentation with the language. It's true that even at that time it was obvious that making it a good unique citizen with a common line compiler, something that no IDE, just maybe a little bit of integration into E-Macs would be just easier because we had very limited resources.

A few years later after I did my PhD, after my post-doc at Stanford, I was back to France in a dominant position. Then I did the OCaml system with some other colleagues. Then we really tried to do a state-of-the-art compiler, something that would really be a state of a state-of-the-art ML with high-performance compiler with some language extensions like new module system and things like that. Some investigation into object-oriented programming. It's true that we kept this efficient runtime system, relatively lightweight command-line approach producing standalone executables and so on. Indeed, it paid a few years later. The initial uses for OCaml and the initial goals for it were for symbolic processing of course. Improving program analysis for transformation, compilations. Then we got our first users in systems, basically.

Things like distributed systems at Cornell and later IBM. Also. Let me remind you, some early web experiments like web crawling, then some real-time trading at Jane Street Capital, which is still a big actor in the OCaml community these days. Unikernels, applications that boot straight on top of a hypervisor, the Mirage system, blockchains like the Tezos blockchains. All those systems applications that were absolutely not planned initially. It turned out that OCaml is a pretty decent language for that, because it's relatively lightweight in terms of resource usage, he memory management, for instance, has low latency, there has no big pauses, so it's almost softly all the time. Then of course you get the benefits of a M Lite language, so functional power type safety and all those things. I think we found a pretty good niche in the world of functional programming and programming languages in general between relatively low level applications and the safety that you get from high level functional languages.

Carl Stalling: Right. Now you mentioned Jane Street Capital and blockchain applications. Well those are obviously financial applications. Are you aware of other areas besides that where OCaml is being used in industry today? Also why is it that the financial world to Ocaml ... I mean soft, real-time capabilities, that's not unique to OCaml, right?

Xavier Leroy: That is true. Yeah. Maybe I'll answer the first part of the question. For Jane Street, it was because they have, CTO did a PhD at Cornell and this project

[inaudible 00:07:06] distance project. He was very keenly aware of what you can do with a language like OCaml.

Carl Stalling: Serendipity basically, right? A lucky accident in a way?

Xavier Leroy: I think Jane Street liked the language perhaps more than lower level languages, because the code is fairly readable and they could have it reviewed by non-computer scientists, also by financial engineers. That's very important for them. I think in this aspect it was a good deal and maybe that's not a very good reason, but it helped them set the bar higher for hiring. When you're a Java shop, you have thousands of resumes that come from people who have a standard training in CS.

When you say, okay, you must be fluent in OCaml, you immediately get fewer resumes but have higher quality and often people who have Masters or even PhDs. It's also a way to select your program of population. As a regard other uses of OcamL, industrial uses where there are some static analysis tools like the Absinthe company in Germany, they do the Astray Static Analyzer, which shows the absence of undefined behavior in C code and it's used at Airbus and in the automotive industry. Yeah, a formal methods tool, basically. A verification tools, code generation tools like the SCADE compiler that is used to produce a lot of embedded code in cars and airplanes and trains like the Eurostar train. The compiler, which is a very critical piece of software. It's also written in OCaml.

Then, as I said, there's this such project which is kind of interesting. I'm not sure whether it's still academic or already kind of industrial, but the idea is really that, for many applications, you don't need a full operating system. You can just take your application statically linked it with some low-level systems libraries like a TCP stack for instance, and some basic memory management. Then you get something that can boot on typically on virtualized hardware. It's cool, because it's very secure, it boots very quickly so you can boot your mail server every time you get a new email and stop it in between mails. That's fine. It's really, I think, a new way of looking at systems and systems infrastructure that maybe will have some big impact in the future. I don't know.

Carl Stalling: Sounds like some kind of microservice at a cloud environment.

Xavier Leroy: It's the same kind of service, the same kind of ideas.

Carl Stalling: Right. Right. Now I am interested in the role of formal verification. If I ask my industry colleagues, they'll probably scoff at me and say, well formal verification, who's interested in that? We don't need that. The client doesn't ask for it. Then again, there are some applications, for instance cloud infrastructure, where reliability is at a completely different level than regular information systems. You just mentioned a couple of examples. Verification basically started in the 1960s as a vision of a mathematician and so many people have contributed to it over these decades and so many tools have been created. I

know of a couple of applications. I wonder what is your view? Are we very closely making this a practicality so that a verification can be done where it's worth? Is it always going to be confined to a niche of high risk applications or do you see a path where this becomes much more common?

Xavier Leroy: All right. You add that formal verification was born in the ‘60s on the work of pioneers like Floyd and Hoare, and I think it made slow progress in usability, things like SMT solving. Progress is in automated improving helped make the approach of or much more usable in the 2000’s. There have also been some more automated techniques that have been developed in the meantime in the ‘80s, like model checking or abstract interpretation. The field progressed slowly from the academic side, but you're right that even now it's still a niche. Many applications don't want to do or think they don't need for more methods. The niche in question is very important and I think it's slowly growing. Typically, it's life critical software in transportation like fly-by-wire airplanes or other less metros all in Paris and are verified. Well, the clinical functionalities have been formally verified. Controlling nuclear plants or chemical plants that can also be important or big infrastructures like the electrical grid.

As you say, verification is finding its way into a few more niches like security. I don't know if you know that or our audience knows that, but the cryptographic libraries they used by Chrome and by Firefox, which actually have been formally verified, one at MIT and the other at Microsoft Research. This is finding its way into very popular tools, but only for the most very tricky pieces of code. Okay. Well, there's a lot of work lately at Amazon AWS where using formal methods for more security, for other cloud infrastructure. Where there are some infrastructure software that is slowly being verified. Not many, but where you mentioned my Concert-C compiler, that's an example of a fairly widely usable piece of infrastructure that is formally verified. It gives a little more guarantees about the correctness of the generated code.

There has been this SCL-IV [inaudible 00:13:06] that's been verified in Australia, which is also quite an achievement. It's not a full operating system, but it's really the core security features of an OS. You can do a supervisor pretty easily using their code. These are some examples I think of an important niches where formal verification has found a place. Where would wider usage? Well, of course there are a cost to using formal methods. It takes more time. It takes time to do the formal verification. You need to hire people or train people into using it, which is not that easy. You need to select the right tools, et cetera. On the other hand, you also save some time debugging and testing. When I did this Compserve verified compiler, I was very relieved that I did not have to spend much time debugging the output. Debugging a compiler is-

Carl Stalling: I vaguely remember.

Xavier Leroy: I think the main difficulty is that there's a prerequisite for four methods. You need mathematical specifications it. Okay? Otherwise, you don't know what to verify against.

There are application areas where you have extremely precise specifications. Cryptography, for instance, all the math are written and standardized. Control common code like this fly-by-wire system, these are all partial differential equations. The math is there. Databases have also pretty neat, pretty clean mathematical specifications and et cetera. In many applications there's many areas. There's just no kind of mathematical specification like a website or artificial intelligence applications. What is a good tool to classify a correct classification of your photos? A correct answer by Chat-GPD. Okay. This is not mathematically well defined. To me, this is the ultimate frontier. What I would really like everyone to do is try to think of formal specifications for their problems. Even if they don't do full formal verification, there's a lot you can do when you have a spec.

You can do random testing, you can do all kinds of runtime verification to find undefined behaviors, etc. You can do static analysis that will show some correctness guarantees, but not all. That would be much more lightweight than a full formal verification. Now we need to think in terms of formal specifications, not so much in terms of formal methods.

Carl Stalling: Well, you speak to my heart. My day job, I'm a product owner and I write specifications all day long.

Xavier Leroy: Excellent.

Carl Stalling: I'm afraid it's more on the story and gyro ticket side, than formal specifications, I would imagine, or I would like to think, that part of my effectiveness derives from the fact that I have a formal methods background that makes it clear in writing.

You made an interesting point there about AI. Of course, AI is the big topic today, right? Everybody's talking about AI. It's the big news. Many people have gone as far as to say programming, as we know it, will disappear because the co-pilots are so good now, they provide you with lots of code. I must say, I tried that out and unfortunately I wasn't able to get Chat-GPT to write a CamL code for me. It happily wrote code in Python of course, and Java, but no way it would write CamL or Prologue or any of these exotic languages. I guess that makes it an equal teaching tool because students can't cheat, because they have to write the code themselves.

Xavier Leroy: Yeah, I don't know if that's a benefit or disadvantage for them.

Yeah. You are right that those co-pilot generators are very dependent on the training data where they have a lot of training data, Python and Java and

JavaScript. There's a lot less with CamL and other niche languages. Actually that's an interesting point because those systems are really very good at synthesizing code examples that they've already seen. You see that, for instance, on evaluations, if you give them problems from programming contests and so on. When solutions have been published, they will give some pretty good solutions. Then if you give them the latest edition of the contest, those solutions have not been published yet, they do pretty badly. They are really synthesizing existing knowledge more than synthesizing programs for specifications. Then sometimes the result is quite good and sometimes it's completely wrong. It's not even syntactically correct. Okay. That's easy to check. The part that makes me nervous is when the result is slightly wrong.

Carl Stalling: Just slightly wrong, right?

Xavier Leroy: Absolutely. Well, there was an example from I think the human evaluation where one of the ChatGPTs produced a Python code that looked good that passed the four or five tests that were part of the specification. If you looked at it very closely and tested it on other inputs, you saw that it was incorrect and it was fairly subtle. The first four lines were perfect. They were doing exactly the right thing, which was to compute some lists in sorted order without repetitions. Okay. Very good. Then the fifth line was destroying the result by applying a Python trick ... that well-known trick, you go to sets and then take to lists that eliminates duplicates, but can change the order, so the result is not necessarily sorted. That doesn't happen very often, but it happens. That's a Python idiosyncrasy. If you have the same code in OCaml, the fifth line would produce a correct result. It would be useless, but it would produce a correct result. When you convert from a set to a list in OCaml, it's always sorted.

That got me thinking. Yeah. This is good looking code, but it's harder to review than if it were written by a human. Humans generally don't add bad code after the good code. They just produce bad code. It's harder to review for a human, and it still needs testing, obviously, or some other kind of validation. My feeling is that Copilot and other things, they're automating the pleasant part of software development, which is writing code, running small functions given a specification area. If you're competent and it's rather pleasant and not a big deal. What is difficult is, first of all, architecting the whole system. I don't think that GPT or Copilot is of any use there. Then validating, reviewing code, running test suites and so on. Really, I would prefer Copilot to write test suites for instance, or do some of the reviews for me. I don't want to give up on programming. I would like help with other tasks. Really, I'm not quite sure what to do with those AI generated pieces of software, especially in high assurance applications of the kind we mentioned earlier.

Carl Stalling: Yeah. I definitely don't want to have a high assurance piece of code written by GenAI for sure.

Xavier Leroy: There may be a good way to use those systems, which is, think of a mathematical proof. Okay. If you asked ChatGPT to proof some mathematical statement, you will get a proof in English that we need to be reviewed. Let's assume the AI learns an actual interactive theorem prover where you can write formal proof that can be checked posteriorly. Then where the AI can try to produce a proof and then if it passes a checker, it's a good proof; if it doesn't pass the checker, you ask for a new proof. Something similar could happen with software. If you produce a software plus all sorts of assertions, for instance, enough logical assertions that the program prover can verify it, then you're good. I think there's some interesting things to do in this direction. It's just that it will be very hard because, again, training data ... assertions or programs or mechanized proofs, training data is pretty scarce and there's no wahoo effect that you get with is just synthesizing code. Hopefully we'll have to wait for a while before getting that.

Carl Stalling: Yeah, it certainly doesn't sound like low-hanging fruit that we just have to reach out and grab for.

ACM Bytecast is available on Apple Podcasts, Pod Beans, Spotify, Stitcher, and Tune In. If you're enjoying this episode, please subscribe and leave us a review on your favorite platform.

Another topic that I would like to cover with you, since I think this is the first time on ACM's podcast that we have a French interviewee and a German interviewer on this thing. I'm not sure whether you're aware, but the ACM has been trying to become a more global or international organization over the past 15 years and a lot more diverse than they used to be, like 30 years ago or so. I'm, of course, part of that. I wonder whether you have any idea as to how we could promote the diversity or the globality of the reach of ACM out of this little corner where a lot of ACM sits, but they definitely want to go out there. Do you have any idea of how we could spread this into Europe, to Asia, Africa, the whole world?

Xavier Leroy: I'm not sure. Viewed from my perspective, I think ACM is doing pretty well. My perspective being mostly conferences and journals and the more academic part of ACM. I think they're doing pretty well at opening all that to Europeans. In my area, programming languages, research, there's a lot of research in Europe and things like conferences, for instance, alternate between the US and Canada, between North America and Europe. I think we feel pretty much recognized by ACM as academics. I'm not sure this is case for Asia, for instance. There's probably more to be done to connect with Japan and South Korea, which has pretty good research, at least in that area. China is closed anyway or closing anyway, so that's yet another issue. You are right that there's also more to ACM than just conferences and journals. I'm not quite sure actually what ... so I know there are some initiatives towards reaching out to our students, for instance, our potential students in CS. Maybe we could have more of that in Europe or more of that in Asia.

It's true that I don't think any of my students have ever heard about ACM, except as a conference organizer basically and publishing house. What could be done? I'm not quite sure. No, I'll take that back. Well, there's the planning competitions which have ... well, there's a southern European one and a Northern European one. Those don't reach many students, but they reach a few. Perhaps more of that could be done and some mentoring sessions, some conferences, but still pretty good. Not quite sure what we could do specifically for Europe.

Carl Stalling: I have another difficult question for you, and that is my own career oscillated between academia and industry as I went back and forth a couple of times. I can't see that really benefited [inaudible 00:24:55] much, but it was an interesting ride, let's say. My feeling is that there is very little ties between the academic world and the industry world. We have been speaking about formal methods and how they are not very commonly employed in day-to-day operations and industry. That's certainly a fact. Probably for many problems it's not the right tool.

I wonder whether there's not more opportunity to interact and to transfer on the one hand ideas and results from academia and on the other hand, problems from industry. We have plenty of interesting problems in industry and I have a feeling that there are so many contributions in academia that just get lost that would need some application. I sense a gap between these two camps. This always bothered me tremendously, and I've tried what I can to close that gap or to bridge it. I wonder, from the academic perspective that you represent here, what we could do to get closer together, academia and industry?

Xavier Leroy: Well, I think I've had some good contacts with some industries. The point I would like to make first that where the computing industry is wide. The web shop or a company that makes mostly end-to-end software, ERPs and the like, probably doesn't have much contacts with academia. That much is true in my experience. Some other companies like Wherefore, Critical Embedded Systems, like I mentioned, or even some of the cryptocurrency and blockchain industries that popped out of nowhere recently. I have a lot more contacts with academia. There's also other industries that are not primarily computing industries, but they're also pretty demanding, like airplane manufacturers, for instance.

Airbus is a really big name in high assurance software, and they have lots of academic contacts in France and in Germany. What I'm trying to say is that I feel that I've had good contacts with some industrial users. I think about half of my PhD students went to industry, but I'm including also more research-oriented industries, or we mentioned Amazon AWS for instance, or Microsoft or Google, et cetera. We still have some contacts with them and indeed there are some good research issues that resolve problems that promote of them. What could we do to go further than that? It's a good question, where I think our students really deserve and ask for industrial internships throughout their curriculum.

That's a very good opportunity to get to know what's going on in the industry. I would like industry, at least in France, to hire more students with a PhD.

Carl Stalling: I can certainly empathize with that.

Xavier Leroy: The system masters is optimal for going to work in industry. PhD, they tend to say, no, you are too educated. You should stay in academia and so on. I think it's a big loss. I feel that in Germany it's a little better. The PhD is more recognized by industries and that's great. It's also important to have personal contacts within an industrial group, especially in big companies. Sometimes they have a couple of persons whose work title is academic relations, and usually those people are far off from the actual production groups and the actual groups where the real problems occur. Sometimes it can be hard to talk through them and hear the real problems they may be having.

Using from students or other personal contacts, you can get in touch with the actual R&D groups. Then it becomes a lot more interesting because they really love to talk about their problems and they have lots of interesting stories to tell. Yeah. I think there's maybe a little bit of a barrier to cost to just establish those kinds of contacts. I think there are some demands on both sides, the academics like me and the industrial people.

Carl Stalling: It's basically a call to improve networking between the two camps.

Xavier Leroy: Absolutely. Yeah, that's true. Maybe we need more opportunities to do that. Living in Paris, I get lots of email from networking events in the Paris area, but it's true that most often I don't go, because I'm not quite sure there's something relevant for me there. Maybe I should try harder.

Carl Stalling: Well, I would be very happy to invite you to the next networking event, if that should happen in Paris, but that's probably not good for me. Anyway, one last question, and you kind of steered toward that all by yourself. That is, whether you would have any advice for students as they make the decision of what topic to study, what field to pursue, or maybe when they're already in CS and wonder where to go. You made a negative advice earlier by saying, don't do a PhD if you want to get employed in industry.

Xavier Leroy: Let me quantify that. You can do a PhD if you want to be higher in the industry. It's just that it will not really be taken into account. You will get the same salary and the same career, most of the time as if you stopped at the Master's level. You will still have learned things in a PhD. Yeah, well, I guess kind of connects with some advice I can give. Whether to choose computer science or some other big ... I think it's kind of a personal choice. For me, well, I was exposed to a bit of computer programming early. It was fascinating, but also frustrating. Actually, I studied mathematics and physics initially and switched to computer science quite late, after getting my first theoretical computer science courses. That was absolutely fascinating. I've always been mathematically inclined.

What I like in computing is that it's also experimental. You can do concrete things with a computer, you can experiment. There's a practice that is informed by the theory and that is wonderful. I think if you are mathematically inclined, computing is also something to be considered. On the other hand, if you're packing and tinkering with computers and so on, maybe computer science will teach you the basics, the fundamentals. I think it's important. When I was tinkering with my Apple II in Basic, I didn't go very far because I had no guiding principles. I just had a few magazines with a sample of code that I was trying to imitate. CS will also give a good culture, and CS will also give you a lot more assurance and confidence in what you can do as a computer professional.

Now, if you've already into computing, I think we are living in a good time where there's a lot of opportunities for learning. Learning during the university in a CS degree, of course, but there's also learning by yourself. There's lots of online courses, tutorials. Wikipedia is a pretty good starting point for many questions. Stack Overflow discussions can take you pretty far, and there's huge amounts of good code that you can read and learn from. In my time that was less true, but I remember learning a lot. First I basically learned how to program in C by reading some very good soft code written like the E-Mac's source code, which was very interesting and quite comprehensive. There was an interest in some specific data structure, some systems code, et cetera. It was really interesting. The early Linux kernel, which was a thing of beauty. Where now it's a little too big, I guess, for discovery. Back in the, it was pretty small and really illuminating.

Yeah. Reading source code, maybe also participating in open source projects. Well, that's a good way to give back to the community, but there's also a good way to learn how it works, how to interact with other developers, how to do code reviews, how to do work with pull requests and so on. I think this is also a very formative experience. There's many ways to train yourself continuously as a computer professional, and this is just great.

Carl Stalling: Well, thank you so much for these thoughts of yours and for sharing them with us. It was fantastic talking to you. We covered so many topics, and I would like to go on for hours and hours, but unfortunately we can't. I'll wrap it here and say thank you [foreign language 00:33:51] and goodbye.

Xavier Leroy: Thank you.

Carl Stalling: ACM ByteCast is a production of the association for Computing Machinery's Practitioner Board. To learn more about ACM and its activities, visit ACM.org. For more information about this and other episodes, please visit our website at learning.acm.org/bytecast. That's learning.acm.org/ByteCast.