CRAIG (00:00):

Hi, I'm Craig Smith and this is Eye on AI.

This episode is sponsored by [Paperspace](https://www.paperspace.com/), which provides the tools for end to end management of machine learning workflows. It comes with dedicated GPUs in the cloud. Check them out at paperspace.com.

I'm editing my audio with a tool created by [Ken Church](https://scholar.google.com/citations?user=E6aqGvYAAAAJ&hl=en), a pioneer in NLP, and now distinguished scientist at [Baidu](http://research.baidu.com/), the Chinese AI giant. It's a work in progress, but I want to thank Ken and Baidu for their efforts.

This week I talked to [Justin Gottschlich](https://www.intel.ai/bio/justin-gottschlich/), who founded the machine programming research group at [Intel Labs](https://www.intel.com/content/www/us/en/research/overview.html). As Justin explains, machine programming is a new field of research combining machine learning, formal methods, programming languages, compilers, software and hardware systems, in order to automate software development. The ambition is to make it possible for anybody to create software simply by describing what they intend the software to do. It's a pretty wild vision, but one that has already borne fruit. I hope you find the conversation as amazing as I did.

CRAIG (01:24):

Can you start by introducing yourself and telling me how you got into machine learning. I'm always interested in hearing people's journeys.

JUSTIN:

Absolutely. So my first introduction to artificial intelligence was really through watching movies. I remember being a very big fan of sci-fi; Star Trek, Star Wars, and, in particular, Terminator. So the idea of AI I think was around since I was a child. Then when I got into grad school, that's when I first started actually taking courses in AI and I fell in love with neural networks. I studied all the different types of machine learning, but it was really neural networks that I thought were really inspiring because they could do some really fascinating things.

CRAIG:

And you went to grad school specifically for computer science?

JUSTIN:

That's right, yeah, computer science. So I've always been interested in how computers work and the idea that we can create something that then can in some way learn on its own was really fascinating to me.

JUSTIN (02:28):

Actually, my undergrad was computer science and my master's and my PhD are in computer engineering and it's a very minor shift. But after doing an undergrad in computer science, I understood software really well, but I felt like I didn't understand hardware really well.

So machine programming is a little bit different than machine learning. What machine programming is interested in at the highest level is creating software that can create its own software. What we're trying to do with machine programming is democratize the ability to create software. So we envision a space where the global population can create software. And the way we're kind of thinking about this is through a lens of what we call the three pillars of machine programming. This is a [paper that we jointly wrote with MIT](https://people.csail.mit.edu/tatbul/publications/mapl18.pdf) and published last year. It's joint Intel Labs and MIT, and the three pillars are intention, invention and adaptation.

JUSTIN (03:31):

Intention is the space where a human is trying to express his or her ideas to the machine. Traditionally, how we do the creation of software is through programming. So we're writing code. The code is essentially capturing the intention. What we find though, our view is that this approach of providing intention through code can often blur though the intersection of those other pillars. So as I'm providing my intention through software, what happens is I then specify details of a second pillar, which is the invention pillar, which describes the data structures, the algorithms of the idea. And then on top of that I then provide details of adaptation. The adaptation is the space in which I'm saying this piece of software will run on this particular software ecosystem with this heterogeneous backend, that type of thing. However, what we want to do is abstract away those last two pillars from the humans.

JUSTIN (04:35):

What we really want to do is enable a space where any human can approach a computer and use whatever interface he or she chooses, whether it's natural language or in my case that's writing a little bit of code or maybe it's input, output examples, visual diagrams, hand gestures, whatever. Then the computer adapts to the user's preference of intention. Once the intention is captured probably through a thing called conversational programming, where, as the user is supplying some intention, there'll be gaps in the software that the user is trying to invent and so the computer will then come back and say, ‘Oh, well you're missing details.’ So the canonical example I use is I'll say, ‘computer, notify me whenever I'm near a Starbucks’ and then the computer will come back and say, ‘okay, so what do you mean by notify? Should I email you? Should I send you a text message? Should your phone vibrate?’

JUSTIN (05:32):

And then I'll clarify that and then it'll say, ‘okay, what do you mean by near. If you're on foot? If you're in a car? And if you're on an airplane, should I just not bother notifying you cause you might be running into a lot of Starbucks.’ And so we'll figure that out together through this conversation. And then lastly, it may say, ‘you said Starbucks, but is it just generally coffee that you're seeking? Can I expand the space and give you pizza or other places as well,’ that ‘you might have, may be over specified your intention and I can make it more general.’

Then once those details are figured out, at that point, the development of the software gets handed off to the machine. What we think is really nice about this approach is traditionally when we've tried to automate the development of software, part of the challenge we think is from the blurring of these three pillars.

JUSTIN (06:28):

As you can imagine, if I start to write a program and I start to specify certain details about invention, I say, you know, it should probably use this particular type of data structure or this algorithm. Then if the computer tries to automatically generate the rest of the code for me, it may think that the data structure and the algorithm is semantically important to the program, that it can't change that. So what I've done is I've actually over specified the constraints of the program. Whereas if we can keep the user specifically in that intention pillar, what happens is it actually allows the machine more freedom to explore the possible implementations through the invention and the adaptation pillars. Yeah, so our paper, for example, this year is on an automated technique to detect performance regression bugs in parallel software. Performance regressions are when you have a base line of code and then I do an update to that code and I accidentally degrade the performance of the software.

JUSTIN (07:32):

We focus particularly on parallel software. So parallel software is the idea that you have multiple threads that are executing concurrently. And we do this because what we find is parallel software, identifying the performance regressions in these codes tends to be much more challenging than if you try to identify them in serial code. And through our techniques that uses three different things - [zero positive learning](https://arxiv.org/abs/1709.07536), [autoencoders](https://en.wikipedia.org/wiki/Autoencoder) and [hardware telemetry](https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/7-x/programmability/guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_7x/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_7x_chapter_011011.html), which I can explain in more detail later if you'd like to learn about those - what we found is our techniques can discover these performance anomalies with greater accuracy than the state of the art. And on top of that, it can detect these performance anomalies that in some cases have eluded even expert human detection. So we're trying to, again, help the programmer and automate away some of these nuance things that maybe have nothing to do with the intention, but really help us expedite the programmer productivity.

CRAIG (08:35):

Right. The first layer of the interface with the computer, between the human and the computer, is there a substrate that that is a computer language that the speech or gestures is being translated into, that then is handed off to the machine?

JUSTIN (08:52):

That's exactly right. Yeah. For the different inputs, we have different systems that then we'll know how to interpret those things. So for example, there's early work now where we can take relatively simple natural language, send it through a natural language processor, then fuse that with other techniques like graph neural networks, and then once that translation is done, we can then spit out actual software. We can fully synthesize a very small program.

CRAIG:

There was a paper a couple of years ago about translating, auto- translating or automatically translating mathematical notation into code and that interested me because it looked like a step toward a layer that translates natural language into mathematical notation. Is there a layer in there that that translates speech first into mathematical notation so that you have the algorithms necessary then to code generally?

JUSTIN:

Correct. Yeah. Usually there's an initial system that's really meant to just to translate whatever the interface is to something that can be meaningfully processed.

JUSTIN (10:12):

Then by a second system or a third system that then translates into the actual code. In the systems that we're looking at, you don't go directly from natural language to code. That's pretty challenging. But instead you have these multi-phased approaches where you'll use existing techniques and things like natural language processing, natural language understanding, to extract out the semantic meaning. And then once you have a good semantic representation that is representable in a way that a neural network or something can process, then you can start doing things like automatic program synthesis.

CRAIG:

Right. Natural language is very imprecise and mathematical notation is very precise. So unless someone can speak in mathematical notation, which is a fairly small subset of people, how do you – do you have to train people to speak in a certain language?

JUSTIN (11:17):

Right. So this is an excellent point. We're enabling the global population to create software. And I can imagine people like my mom, for example, she's an entrepreneur, she's created several businesses. She has these wonderfully creative ideas, but she can't write a line of code. So the entire world of software is closed off to her. If we can lower that bar of entry, I can only imagine how many brilliant ideas we would start to create in software. So that's the first piece, right?

So then people may say, okay, with this model then do we no longer need to learn programming languages? And my thought on that is twofold. First, many of these systems are built using code. So without big code, we really have a hard time automatically synthesizing programs. So the need for programmers at the coding level actually increases to do machine programming. The second is for those people that are not coders, what do they have to do?

JUSTIN (12:12):

Going back to your original point, yes, there will need to be some training I think to help people understand how to think critically and then be able to present their ideas in a precise way. We don't think people will get this perfect. And thus, conversational programming, we do believe there will be a back and forth. But my hope is that over time, once people have practice doing this type of thing, their precision in language will improve.

I think you make an excellent point that spoken language tends to have lots of ambiguity and code on the other hand tends to have very little, if not no ambiguity. So there's definitely a chasm there that we need to cross in order to get that precision.

CRAIG: Yeah. And is there a specific, a particular language on the computer language level or is this agnostic? You could use Python or C++, Ruby on Rails or …

JUSTIN (13:18):

So what we've seen so far is the early solutions where machine programming has been really successful are in spaces where we use what we call domain specific languages. So domain specific languages are languages that are meant, just the name implies, to work in a very specific domain. So for example, SQL is a programming language that's meant to work specifically with databases. There's another programming language that was created by one of my colleagues who's a professor at MIT, [Jonathan Ragan-Kelly](https://people.eecs.berkeley.edu/~jrk/), it's called [Halide](http://people.csail.mit.edu/jrk/jrkthesis.pdf). And it's also a domain specific language, a DSL, and it's meant for image processing. So what we've seen is that when you have these domain specific languages, what they do is they very much follow the model that we've presented in our three pillars paper, is they focus on really capturing the programmer’s intention. And in some cases with both SQL and Halide, the programmer is actually not allowed to specify the details of invention or adaptation.

JUSTIN (14:21):

And because the programmer isn't allowed to go there, it opens up the opportunity for the machine to explore those spaces. And we're seeing early evidence that not only can the machine explore and find correct implementations, but those implementations in some cases are superhuman. One concrete example of that is the Halide work that came out in 2019 at [SIGGRAPH](https://www.siggraph.org/). The Halide team essentially showed for the first time that through automatic creation, using advanced machine learning techniques, that they can create these schedules, which are essentially optimizations of the implementation, that are better than the world's foremost experts in that programming language, which I believe is a pretty astounding achievement.

But keep in mind, one of the core challenges as I think you correctly pointed out is it becomes much harder as we start to target languages that are more general purpose. So for example, if we were to target language like Python or C++, which aren't domain specific, then the challenge of keeping the programmer in that intentional space becomes an obstacle for us to automate things.

JUSTIN (15:35):

So we're actually doing some work in that space as well. We've done what we call intentional C++ programming. And what we do is we actually ask a programmer to specify a block of code that they will then write the most naive implementation that they can, that just captures the semantics of what they want to achieve. From there, what we do is we use a technique called [verified lifting](https://people.eecs.berkeley.edu/~akcheung/) that then we'll formally lift out the code semantics into a higher level DSL and then after it understands this, it will do a transformation of that into a more optimized implementation. So we have early evidence that things like this can be done in general purpose languages for a small sliver and we're working essentially on trying to broaden that scope.

CRAIG:

And then the other layers, once you have invention and the adaptation layers, once you have the language successfully turning into code, I mean as you said, there are going to be gaps in code, you're not running the code in front of you as you write it. And then on the adaptation layer, is that machine learning then that takes over?

JUSTIN (16:41):

That's a great question and this is actually one of the things that I try to clarify when I talk about machine programming and how it's different than machine learning. We think of machine programming really as a spectrum. So if you think of sort of a spectrum of colors, we think of a spectrum of solutions in machine programming. On one end of the spectrum we have what we call precise solutions. These are solutions that use traditional techniques, things like formal program synthesis where you have some input and you need to generate some output and it will synthesize a program for you that will precisely match that. So you can imagine that in systems that might be safety critical, one would want to use a precise machine programming solution. On the other end of the spectrum, we see the things that are more in the machine learning space.

JUSTIN (17:38):

We call these approximate or probabilistic solutions. And this is because while we can have some level of confidence, maybe it's even 99.9% confidence, we can't actually guarantee, at least with today's techniques that this will be precisely correct.

Now there is work actually, we're doing a collaboration with Stanford and Hebrew University at Jerusalem that's trying to formally verify some of these probabilistic things so they can be a bit more precise. But right now the current state is they are a little bit probabilistic and so they can work well in spaces where you're comparing it against some known quantity, like an average human being and trying to ensure that your computer vision algorithm exceeds human level performance. But we know that human vision is not perfect, right? Otherwise I probably wouldn't be wearing these glasses. And so areas like that. In particular, other areas would be software optimization.

JUSTIN (18:35):

You know, in my mind, the idea of a perfectly optimal program is one that takes zero time to run, yet we will probably never achieve. I mean maybe with quantum, but so this is an area where it's very sort of nebulous and we can use these approximate solutions to increase things. So if you look at these two points together, what happens is we see a fusion of solutions all along the way where you may use an approximate solution and then you'll combine it with some formal methods to verify it. One of my colleagues, [Armando Solar-Lezama](https://people.csail.mit.edu/asolar/) out of MIT, who's a joint author on the three pillars paper, he's taking formal methods and fusing it with deep neural networks to help improve the computational time to find a synthesize program. So it's actually really fascinating the number of ways that you can try to solve this problem.

JUSTIN (19:32):

I think notion of what we're trying to achieve, at least in today's focus, is that the machine program generated code usually has to be then maintained by humans. So we're not yet to the point where we can do full automation. This is something that maybe pulls it away from some of the machine learning approaches that we have today.

So, for example, we can have these really great models that perform really well, but somehow they get stuck on one particular problem and it's really challenging then to debug them because we don't really understand how these neural networks work. So instead, rather than fiddling with the topology, we just try to provide it more examples. With machine programming, what we're hoping to do, and we have early evidence of doing this, is we will not only generate the solution, but then the solution will actually be human understandable. So then the human can then go in and say, ‘Oh, there's a bug here. I'm going to now fix that bug.’ And then the system can say, ‘okay, I understand the fix. I won't make that mistake in the future,’ and so on and so forth.

CRAIG:

And that sounds like machine learning, just that, that element that, that last element.

JUSTIN:

Absolutely. Yeah. Yup.

CRAIG (20:49):

Yeah. When we're talking about machine learning and, and this ambitious program of machine programming, there's a certain amount of machine learning in the process, right? For example, there may be at some point this debugging interface, but the system could be used to create machine learning software as well. Right?

JUSTIN (21:08):

Absolutely. In fact, so one of the really grand challenges that I see in this space, and this is something that we're just starting to explore, is how to use these machine programming techniques to actually enhance the state of the art machine learning techniques. So I'll give you just one example.

We have work that's jointly being done with Intel Labs and [Texas A&M](http://people.tamu.edu/~abdullah.muzahid/index.html) where we have a thing called a genetic algorithm. Genetic algorithm is a specific type of machine learning that basically follows these evolutions, which you're familiar with. One of the challenges of genetic algorithms historically has been building this thing called the fitness function. The fitness function is essentially the thing that grades each of the genes per evolution. And so traditionally what we have to do is we have to handcraft these fitness functions. Unfortunately, what we find is that the more complex the problem becomes, the more complex and more challenging it is to write that fitness function. Some people would argue that it becomes so challenging, why don't you just try to solve the problem yourself? So what we've done in the space of machine programming is we figured out a way to automatically generate the fitness function for genetic algorithms, actually using a neural network. And then the genetic algorithm itself is actually synthesizing complete programs. We then use another type of machine learning to automate the first piece and then together as a system, they automatically function as a [program synthesizer to create new software](https://arxiv.org/pdf/1908.08783.pdf).

CRAIG (22:43):

Hmm. In [AutoML](https://en.wikipedia.org/wiki/Automated_machine_learning), one of the things that it's being applied to is a neural architecture search or hyperparameter optimization. Within this stack, let's call it, of the machine programming that you're working toward, would there be bits of AutoML in there to do things like that? I mean the, the programmer expressing an intent that's being translated into code, but at some point the system has to decide architectures or things like hyperparameters. Would there be an AutoML feature or layer within that?

JUSTIN (23:25):

Absolutely. So this is great intuition. In fact, when we think of machine programming we think of AutoML or, for example, neural architectural search as one example of machine programming and the way that we cast it in the machine programming space is that the way AutoML works to my understanding is you essentially provide it with input output examples. The input output examples can be thought of as your intention that they're essentially saying, “here are some examples of what you should take in as an input, I've labeled these things to let you know what the output should be. Create that program.” So you can think of then the neural network that's being generated through that neural architectural search, which I believe actually is using genetic algorithms, that's actually in some sense synthesizing that program. So we definitely think that AutoML is part of automatic generation of various models.

JUSTIN (24:23):

It could be a genetic algorithm or a neural network or a Bayesian network or something of that nature as part of like one sliver of the machine programming space. But we would actually like to take that further in that what Google has been able to do today, which is fantastic, with AutoML. We want to be able to then take those models and have them be interpretable so the programmer may get back this model that works really well for the input-output examples maybe perfectly, but then when provided new data, maybe it misbehaves. And currently with the models that we're generating, interpretability is one of the core challenges. So this is one of the other problems that we're also working with at MIT and we have a couple of different things in place. In a system we built called [AutoPerf](https://www.intel.ai/zero-positive-learning-autoperf/) that automatically detects these performance regressions. One of the limitations that it has is that it can only detect the performance anomaly. It can't tell the programmer where the performance anomaly is or what's causing it. So what we'd like to do is be able to then interpret the model and then back trace essentially to the root cause based on the input that it receives. This is an open problem for us. We've started to explore it within Intel Labs and MIT, but we have yet to actually come up with a solution.

CRAIG (25:46):

Yeah. You were talking about the spectrum between essentially control theory and full machine learning, maybe unsupervised or reinforcement learning sort of at the outer edge of machine learning, within the machine programming space. It may be combining elements within that spectrum, is that right? I had conversations with a woman named [Aude Billard](https://www.eye-on.ai/podcast-028) in Switzerland who works on machine learning and robotics and she has some fascinating work where a robot arm can catch something thrown at it in a nonlinear fashion.

JUSTIN:

That's a hard problem.

CRAIG:

Yeah. Like a spinning tennis racket. And the robot arm can catch it. And she uses a combination of classical control and machine learning because to control the actual movement of the robot is control, but to figure out where the arm should be at a particular point in time is through a machine learning. So is that an example of the kind of thing that machine programming would be able to do in combining different parts of the spectrum into one software?

JUSTIN (27:03):

Absolutely. Yeah. One could actually think of the natural language processing as providing a solution in that capacity. So for example, to the best of my knowledge, the cutting edge techniques for NLP, natural language processing or natural language understanding, are really through machine learning techniques. If we then express our intention through natural language, we might want to use the machine learning approaches to solve that piece. Yet once we understand and extract the semantic meaning of what the programmer is providing or the software creator is providing, we then may want to use formal methods that will provide a precise solution because maybe this system is going to be deployed on a safety critical system where it's not acceptable for us to use a machine learning system that would give us an approximate solution but maybe not formally verified. So absolutely along the lines that you've just presented, I think that there's definitely many different ways to carve this up. And I'm seeing the emergence of fusion across both of these areas in that spectrum we were talking about to get some of the state of the art solutions.

CRAIG (28:15):

Yeah. So there's this grand vision and you've described some of the pieces of it that are being worked on. How much of this has been integrated into a single workflow?

JUSTIN (28:29):

This is an excellent question. So for the machine programming research group that I lead at Intel Labs, we have two primary charters. The first charter is around programmer productivity. The second charter is around programs synthesis, really the creation of software for the programmer productivity. Our blue sky vision is to improve programmer productivity by at least two orders of magnitude. One could imagine then if you look at programmers today and say, ‘okay, 100% of their time is used for programming, we want to have them be able to develop exactly the same software, only requiring 1% or less.’ So that's one of our blue sky visions. The second is around synthesis. We then want our systems, our machine programming systems to synthesize programs, create these new programs that are superhuman across four vectors. Those vectors are correctness, performance, security, and in the scope of machine learning, accuracy. So that it's generating, in the context of AutoML, automated models that have accuracy better than the best machine learning experts in the world could handcraft as a solution.

JUSTIN (29:42):

So this is sort of like the blue sky vision, but going back to your question of well, how much of this is being done? So we have our long-term goal, but then we have lots of short term goals. Yeah, there's actually several things that we're doing today that are practically being explored and used in industry as low hanging fruit. Even though we see this as a multi-decade problem, there's lots of small little footsteps we can make. So the first example is really the AutoPerf work. Manual regression testing is a pretty menial task. And I suspect as an engineer myself, before I became a researcher, I wrote a lot of non-performant code and most of the time the way I found out that it was non-performant is I would deploy my software and my customers would come back and complain.

JUSTIN (30:33):

So I created a startup when I finished my undergrad. It's this online game I wrote and I would always ensure that the program was correct. But performance is much more difficult to analyze because it can be hard to understand through a testing environment if you've actually slowed the program down. Unless it's like potentially an order of magnitude or something slower, it can be really hard for us to observe those things. With AutoPerf, we've shown that for every case that we've looked at – and we've looked at 10 different parallel bugs across I think seven pieces of real world software, so these aren't toy programs, these are things like MySQL or the Boost C++ libraries – what we've found is that, and these are all parallel programs, too, so these are sort of like the hardest types of performance bugs that we can imagine. What we found with AutoPerf is that it has zero false negatives.

JUSTIN (31:26):

So that basically means that for every performance anomaly we've thrown at it, auto perf has detected it. On the flip side, it does have some false positives. So one of the areas that I work on is anomaly detection. And usually what you find is it's very challenging to build a perfect anomaly detection system. A perfect anomaly detection system would be one that has zero false negatives and zero false positives. Usually there is a trade-off. So in this particular case what we did is we sort of biased the system so that it would place importance on having no false negatives and instead we incur some penalty on false positives. The reason why we did that is because we want to use this in practice. And, by the way, it's actually open source. So if people visit [my website](http://justingottschlich.com/) they can see a link and actually start using it today.

JUSTIN (32:16):

This is one thing that's nice about Intel is a lot of our collaborations result in open source software. It's open-sourced on GitHub. My website is [JustinGottschich.com](http://justingottschlich.com/) - so it's just my full name .com. So what we did is we biased the system so it would have no false negatives. And we think this is important because if you had a false negative, what that would mean is you missed a performance anomaly. And we think the developers would prefer to see a couple of false positive and say, ‘no, that's not a performance anomaly. You are wrong.’ Rather than saying, ‘Oh look, there's no performance anomaly, great,’ and then they deploy their software and suddenly all their customers say, ‘Oh my gosh, you've just reduced the performance by three X.’ Right? You've lost a customer for life now. So this was an intentional thing. And going back to your question about what can we do today, this is very much in line with the idea that there are techniques that we can start to explore that will enhance productivity, will give us things like, in this case, this is in the area I think of superhuman testing that we're able to detect performance bugs that have eluded some of the world's foremost experts in the parallel programming space.

JUSTIN (33:32):

Other work that we're doing right now to look at integrating into products is we're working on a thing that's essentially a code recommendation system. What it aims to do is, you'll have a naive implementation of some algorithm by a novice programmer, let's say an undergrad in computer science, and what we want to do is have that algorithm run really efficiently on Intel hardware. But the way that the programmer has written it currently won't suffice. So with our code similarity project, what we do is we try to lift out the semantic meaning of that algorithm and then map it to a Ninja level implementation. A Ninja is essentially an expert in hardware and software. And then return the programmer this snippet of code or potentially just an API call that says, ‘Oh, what you're trying to do is this, here's a piece of code you should use,’ or ‘just call this API. It's doing what you're doing except it has more correctness, more performance, and it's more secure.’ And this is something that we actually are aiming to put into an Intel product called, I believe it's Intel Inspector. It's in our roadmap.

CRAIG (34:41):

Yeah. And just what you described, because they're for programmers, programmers are not writing solutions, code solutions from scratch. Normally they are picking from a library. Is this interface at the level at which, then searching through all of the libraries and finding the optimal block of code and then plugging that in and essentially building the software through pulling from libraries and stitching them together.

JUSTIN (35:17):

Yeah, so this is an excellent question. One of the core challenges that we see with this software similarity project of trying to take naive code and transform it to Ninja level code is that if we were to return the Ninja level code the way that it's written today, a novice programmer would likely have a very hard time understanding that code because that code will do advanced things like tiling. It'll try to have padding for the cache lines. It'll use really interesting ways to use shared memory that might parallelize the algorithm. So from a maintainability perspective, we think that this is not necessarily a feasible solution. What we instead would like to do is, what you just described is that you don't actually provide them the actual implementation. Rather what you provide them are a number of API calls to libraries that already have those efficient implementations. So you provide them, for example, a call into one of Intel's libraries, like the math kernel library, MKL, that we believe is a solution that's maintainable because then the programmer says, okay, ‘I was trying to write this call but this function inside of MKL does exactly what I want it to do. I didn't know about it before. Now I'll just use this.’ So that's the direction that I think that we're heading in for this particular project.

CRAIG (36:43):

Yeah, and then the long-term vision of putting those all into one end to end system where as you said, your mother can talk into a microphone and magic happens and outcomes and iPhone app that does precisely what she imagined. You mentioned decades. I mean people hate to make predictions, but yeah. Do you have any sense of where on that timeline you are?

JUSTIN (37:12):

This, this is a wonderful question. I think there's really two pieces that I'd like to talk about on this. The first is when I presented the idea of machine programming to Intel's executives earlier this year, which really got us to have the result of spawning off our new research group around machine programming. What I presented to them is that the fully automated solutions we believe are at least two decades out at the very minimum. So that blue sky vision, at least two decades. But of course there's early work that we can do like AutoPerf that we can put into production today. And there's actually dozens of examples that I could walk you through that exist not just in academic settings but are actually being used in industry. Things like Facebook's Aroma system. Aroma is a system that they just published this year and it is a code recommendation system much like the one we're building, but it's focused more on correctness.

JUSTIN (38:07):

I believe it's not focused as much on performance. The high level answer to your question is, we know we've got a lot of work to do, but on the flip side, one of the things I think makes this so exciting to us today is we believe we're at an inflection point and the inflection point I think breaks down into really two subareas. The first area is we think we have the technology today to start to make this work. That the advances in algorithms, not just in machine learning but also in programming languages, in formal methods, in compilers and so on and so forth is essential to making this happen. The second are advances in compute. We're seeing tremendous advances there. Turing award winner Dave Patterson and John Hennessy are saying, you know, this is the golden era of architectures because no longer is it just the CPU.

JUSTIN (39:04):

There's a number of accelerators. And then the last is the abundance of data that we have, an enormous abundance of programming data and that's really needed to train a lot of these systems that even if you use the formal methods, they still need programming examples to look at. And if you look at Github, which is a repository for source code, what we see is that in 2008 Github had 33,000 repositories; in 2019, just slightly over a decade later, there are, I believe approximately 210 million repositories. It's almost a four order of magnitude increase, which is phenomenal growth. It is literally an exponential growth. So that's the first part of the inflection point. The second inflection point I think really stems or goes back to the work I was talking about with the [three pillars](https://arxiv.org/abs/1803.07244), is that the paper that we wrote with MIT, the point of this paper was really to try to help the community rally around a common goal that we think the separation of concerns, separating things out, particularly intention and pulling it away from invention and adaptation is really the key to sort of unlocking the future. And so this is an ongoing collaboration we have with MIT and other academics to try to drive that intentionality. There'll be many small footsteps that we make along the way to try to fulfill that multi-decade solution.

CRAIG (40:40):

A broader question still, this democratization of software, as software permeates society more and more, will require a cadre of very high level experts to maintain and keep pushing forward. How do you, just personally, how do you view society evolving. It feels as though the world is shifting from physical machines to code and you know there's this massive separation between the class of people that understand this and can operate behind the curtain, and then the rest of the world that's in front sort of dazzled by the big head or whatever. How do you envision that? Is there a hierarchy that's developing with a knowledge class and the useless class as Yuval calls it?

JUSTIN (41:34):

So I really like this question because I actually think that if we look at the spectrum of people, there are those that are at the front of the line racing toward this future, right? This is probably very clearly where I fall, right? But I also have people that I know that fall into the class that have a deeper understanding of the technology, my stepfather for example, and he's also very reticent about the advancement of this technology, is concerned about what will happen as these things evolve. Will jobs be taken away? That kind of thing. And I actually think it's good to have these groups of people. I think in some ways they help keep us honest, that they help keep us thinking about things. And that in the back of my mind, I always thought that machine programming for example, would have the capacity to generate a lot of jobs.

JUSTIN (42:32):

I didn't start really thinking about it deeply until I was talking with my stepfather and he said, ‘Oh my gosh, if this works, are you going to put all the programmers out of work?’ And then I thought deeply about it and I called him up and said, actually, no, I think this is going to create millions of jobs. It's just the way that we operate is going to be fundamentally different. So to answer your question, I think that this ideal future that we're envisioning, I think it will be much more ambient. That it will be sort of this virtual environment, that our interaction will be almost seamless. That the computer environment will constantly be around us and we can kind of already see hints of that in our smart homes, in, you know, other areas. And I think that as we start to advance with machine programming, we'll be able to take that even further.

CRAIG (43:21):

Yeah. But still there will be a class of people who understand it and then the rest of the people that use it.

JUSTIN:

That's right. And this is actually a really good point too, because some of the foremost ninjas that I know, they'll talk to me on LinkedIn or via email or something, and sometimes even in person, if you can believe it, and they'll say, I don't like this, right? Because I like writing this Ninja level code. I like building these systems and I constantly remind them, no, no, no, we need you. Right? Because as we are building these systems, they will fail. There will be problems, there will be maintenance and addition to that, it's data-driven, it's data-driven. And the data that we're actually using is ingesting in many ways the software that those experts are writing. So those people are going to be truly essential, I think even more so in this future than they are today.

CRAIG (44:13):

Obviously when you're working on a big project with a massive, ambitious vision, you, you invest yourself in it and you have a lot of confidence. I'm just wondering, in 20 years are we going to meet and - hopefully I'll still be around in 20 years - and I'll be like, it's amazing that we had that conversation and here it is. I can talk into my cell phone and build something. How confident are you that you'll reach that goal?

JUSTIN: Personally, I'm wildly confident, and sometimes I think as my colleagues will point out, perhaps too confident. But based on my 20 plus years of experience in industry and working in these spaces, it seems like not only are we at that inflection point, a larger community is evolving and really I think this is a communal effort that are all getting behind this. That we're seeing the emergence of dozens of startups in Silicon Valley specifically just for machine programming. We're seeing every leading tech company in the world starting to adopt machine programming in some capacity, how you use machine learning to solve the machine programming problem. So what I think is happening is people are starting to see this is realizable and because of that it's spreading like a wildfire. The fact that we have this community that's growing at really an exponential rate. I think that when we have this conversation 20 years from now, we're both going to be stunned at how much progress we've made.

CRAIG (45:58):

That's it for this week's podcast. I want to thank Justin for his time. If you want to learn more about Justin's work, you can find a transcript of this episode on our website eye-on.ai Also check out paperspace at paperspace.com. They have a terrific blog that tracks the latest AI trends and are doing amazing work. We love to hear it from listeners, so feel free to contact us with comments or suggestions.

The singularity may not be near, but AI is about to change your world, so pay attention.