MIT CSAIL Alliances | Armando Podcast Export 2

Welcome to *MIT'SComputerScienceandArtificialIntelligenceLabsAlliances* podcast series. My name is Steve Lewis. I'm the Assistant Director of Global Strategic Alliances for CSAIL at MIT. In this podcast series, I will interview researchers at CSAIL to discover what they're working on, and how it will impact society.

Armando Solar-Lezama is a professor of Electrical Engineering and Computer Science and Associate Director and COO of the Computer Science and Artificial Intelligence Lab at MIT. He is best known for his work on programmed synthesis and the development of the SKETCH program synthesis system. He is currently the lead PI of the NSF funded Expeditions project "Understanding the World through Code" and is also the founder of *playskript.com,* an online platform for creating interactive presentations.

Armando, thanks for your time today. So we're going to be talking a lot about program synthesis in this podcast. So why don't we start off with just what is the definition of program synthesis?

So that's a very good question. When we started working in this field, the goal was to develop technologies that could help people writing code. Ways of helping with some of the more challenging aspects of programming help to automate those away. And initially, our primary goal was to help software developers. However, as the field has evolved, it has really expanded in terms of its scope.

One of the things that we realized early on is that if you have this ability to generate small pieces of code that do specific things that you want them to do, you can use this, for example, for things like end-user programming. So people who are not software developers but who want to get their systems to behave in certain ways can leverage program synthesis in order to get the machine to do what they want.

There's been a lot of work on things like programming by example, for example, where somebody who is not necessarily a programmer but who needs to do some data manipulation task, they can provide a few examples, and have the system from those examples derive what they want to do. This actually, for example, the functionality that's provided by Flash Fill in Microsoft Excel.

One of our students from our group, Rishabh Singh actually participated in that project. Even going beyond that, once you have the ability to generate a little piece of code that is maybe consistent with some observation, it gives you the ability to, for example, formulate hypothesis about how something works, and frame those hypothesis as little pieces of code.

People in our field have done that to explore everything from biology and how regulatory networks work, to linguistics and how language works, to doing reverse engineering of computer systems. So it's actually a pretty expansive field, but the core of this field is all about, can we generate pieces of code that behave in very specific ways that somebody needs them to behave in?

So you mentioned Flash Fill with Excel. How close are we to automating major aspects of programming?

So there's programming in the small. The kind of oh, I need to write a little script to maybe manipulate some data somehow, maybe move some files around, maybe even solve a small short scale task. I think today we can actually do a really, really good job automating that level of programming, both the neural techniques that you're seeing now in the marketplace with things like copilot, for example, or even with things like ChatGPT. As well as more traditional techniques that are actually able to, for example, produce verified implementations for things like data structure manipulation.

So for that level of programming, we can achieve a fair degree of automation. However, once we talk about more programming in the large, right? Developing computer systems and really writing applications that people will use. We're talking about a very different set of concerns. We're talking about how do we actually even break up the problem into manageable pieces, where each of these pieces have well-defined interfaces, where each of these pieces has-- the pieces integrate well with each other, where you don't have a lot of repetition in different places?

And that level of software design is something that we're very much not close to automating. There's a lot of tasks that professional programmers do in their day-to-day work that are actually really, really hard to automate because they require navigating very different levels of abstraction. They actually require a lot of creativity. They're much more difficult to automate than some of the programming in the small kinds of tasks.

So is there any technology today that's required for program synthesis to work that we don't have? Is it the-- is it the AI that does those creative tasks, or what is it in your opinion?

Well, it depends. I think for individual problems, for example, when it comes to developing software, a modern application is going to be sometimes hundreds of thousands, sometimes millions, and sometimes even tens of millions of lines of code. The technologies that we have today to produce code just don't operate at that scale, and there's nothing I think really in the pipeline that would suggest that we're close to them operating in that scale.

And moreover, when you're talking about software at that scale, it's not just about I know what I want and then I need the computer to do it. When you're talking about an application of that scale, the complexity of behaviors and just the range of behaviors that the application might take is so large that a lot of times a big part of the role of software design is about figuring out what is the right thing for this piece of code to do. And that requires being able to interact with stakeholders, being able to talk to people and figure out how comfortable they are with different trade offs, for example, what is the functionality that they really, really need, how do the processes that they want to use software for actually work? These are tasks that really require humans and the loop.

Now, could we get machines that are able to do that? Sure. Someday. But we're talking about tasks that, at this point, given the current state of the art, I don't think we're anywhere close to being able to automate.

So we're far away from being able to say to ChatGPT, for example, write me a program that does x, y, and z and the output is this?

It depends on what x, y, and z is, right? So if x, y, and z is write me a program that computes the greatest common divisor of two numbers then yeah, it will do a great job at it. In fact, even there was a system called the AlphaCode that came out last summer, where they showed that you can actually train some of these very large deep neural networks to solve programming competition problems, and to score above the 50th percentile. Basically, above the median of people who participate in some of these coding competitions.

So that's pretty-- that's really impressive. And that is really exciting that we can do that today. So if your goal is generate me a program that sorts all of these numbers, yes, you can do it. If your goal is generate me Microsoft Word then no. We're talking about just the number of behaviors that a program like that has, and the number of decisions that have to be made about what this program is going to do in this case and in that case, and the design of what the user interaction should look like, is something that is just very far from what machines know how to do right now.

So can we talk about SKETCH now and what that is?

Sure. So SKETCH was actually designed in response precisely to this question of how can we get some automation that is helpful for programmers, but that lets programmers do the things that they know how to do well? And so the idea was that this problem of trying to go directly from let there be PowerPoint, and poof, outcomes fully-formed PowerPoint. That's clearly not going to be in the cards for a very, very long time, if ever.

But how do we help people who already know the task they're trying to solve? Who already know what it is that even how the code should look like, and they just need help with the low level details that for complex algorithms might get really painful, and really require people to take out pencil and paper and figure them out.

And so the idea was to have a programming system, where people could write a piece of code but then leave behind some holes in places where they already know that a complicated piece of code needs to be there, but they don't know exactly the details of how that code should look like. So they can leave a placeholder there for the synthesizer to fill in. And so what that does is it makes it easier for the synthesizer because it doesn't have to figure out from scratch how to solve your problem.

It only has to search for these very local pieces of code. And at the same time, it means that the programmer doesn't lose control over their code. They still have full control over the structure of their code, over exactly what they want the code to do, but they're getting help with these very local decisions, and it's leveraging verification technology to help make sure that the code that is produced satisfies certain correctness guarantees.

At one level, this project was extremely successful in that before we started this project, there was very little attention paid to program synthesis even in the research community. And because people when they thought of program synthesis, they thought of this unobtainable goal of let's just try to go and generate everything from scratch. And this provided an avenue that allowed you to make progress and do interesting things.

And so it was very influential in the academic community. In terms of serving an actual programming model, it also showed some of the shortcomings of the style of synthesis, and that the programmers still required-- still had to write a fair bit of code. And it actually takes-- it really took some skill to actually figure out exactly what parts of the problem to abstract away, what parts of the problem to leave to the synthesizer to discover, to get a feel for exactly how much the synthesizer could figure out on its own?

So that in terms of an actual aid to help you write code easier, it proved to be not as effective at actually helping you write code easier. But actually proved very useful as an internal engine for higher level tools that could actually do useful things for you. For example, places where you know that you're going to be solving very similar synthesis problems over and over again, you could actually leverage SKETCH very easily to engineer a domain specific synthesizer for a particular class of problems that then could actually be very useful.

In which domains do you think are best suited for program synthesis?

So in general, domains where you can do very useful things with relatively small programs, and where there are natural or well-established ways of conveying what it is that you want the program to do. So, for example, one domain that has proven to be very fruitful is data wrangling.

I have maybe a giant log file with lots and lots of information, and I need to write a little piece of code that extracts from these giant log file all the pieces of data that I need and organizes them, and maybe puts them in a database. This is the thing that is relatively easy to describe often even by example to say like yeah, from this little chunk of text, I want to get this piece and put it here and get this piece and put it here and this piece and put it here.

It's also the sort of thing that if you have a language with the right level of abstraction, you can do with a small program. You can do it with dozens of lines of code not thousands or-- tens of thousands of lines of code. And it's also the thing that sometimes people who are not computer scientists have to do.

Maybe it's a biologist, who has to curate some experimental data and organize it into a spreadsheet. Or maybe it's an accountant, who has to extract data in this way. Or maybe it is a social scientist, for example, who has to go on the web and scrape the web for particular bits and pieces of data from different web pages, for example.

And this is also another task that where program synthesis has shown to be quite effective at automating.

And are there tools that are commercially available today that can be implemented in these kinds of domains?

So Microsoft actually has a group led by Sumit Gulwani, who has actually been integrating these kind of technology into many of their-- many of their products. They have a system called PROSE that you can find. It's been integrated into a wide range of Microsoft products to provide exactly this functionality. It's doing program synthesis under the covers in a way that to the end user, it doesn't really look like program synthesis.

It just looks like-- from a few examples, the system is able to capture my intent, and is able to help me accomplish my task.

That's exciting. So let's talk about neuro-symbolic program learning. How has deep learning influenced and/or enhanced program synthesis?

I would say it's actually been quite game changing in that it's allowed us to bridge the gap between the formal world, where programs live, where everything has to be super precise, and everything has to be spelled out in full detail. And the informal world of natural language and whiteboard diagrams and pictures in a napkin. And all of a sudden, once you can bridge that gap, you can actually leverage much richer forms of interaction with the program synthesizer.

You can also learn from the vast amounts of code that are available in the world. So it's really transformed the field. At the same time, there are things that you cannot do very well with purely neural models, and for which symbolic techniques are very, very powerful. At the end of the day, when you're talking about code, the ability that you have to run programs, to analyze them, to verify them, makes producing code very, very different from, say, producing natural language text, or producing images.

If you produce natural language text, maybe it's right, and maybe it's wrong, and maybe it's a little bit wrong but correct enough that people know what you're talking about. And so that's OK. And it's very hard to get at the ground truth when you're talking about text. By contrast with software, you can run it. You can test it. You can analyze it.

And that really gives you an ability that you didn't have before, but also makes the problem harder because there's a lot less tolerance for noise and for errors.

So can you just talk a little bit more about neuro-symbolic programming learning?

So one of the things that we're really interested in is how do we get programming systems that can bridge these different modes of reasoning from the-- being able to recognize, hey, this problem sounds like this other problem that I know how to solve. Or this problem sounds like this class of problems that I know should be approached in this particular way. And how do you combine that reasoning, which neural networks are really, really good at with the very detail-oriented reasoning, where you have to be aware of exactly what line of code is doing, and what-- and how what this line of code is doing is going to interact with what this other line of code is doing.

So that you can really have some confidence that the code that you produce at the end is actually going to do what you wanted it to do. And neural networks are not so good at that, but the formal reasoning techniques that the programming languages have-- community has developed are. And so how do you bring those two together.

Can you tell us what first drew you into this field of program synthesis, and what excites you about the field going forward?

In terms of what brought me into the field, I really like programming. And it's just something that I do even for fun even when I don't have to do it on my free time. At the same time, there was this recognition that there was a gap between what the machine could be doing for you and what traditional programming languages usually are able to do for you.

Traditional programming-- existing programming languages already have a fair bit of automation within them. But when it comes to really the hard task of figuring out exactly how to solve a problem, you're on your own. And so I was really drawn by the appeal of having programming systems that still let programmers be programmers and still keep people in control of what they're trying to do. But are able to provide support at a much higher level than traditional programming languages.

More recently, the thing that has made me very, very excited is the potential of this field to have an impact beyond software development. In some ways, programming languages and code is such an expressive medium that today our scientific theories are often expressed as code and a simulation. That's when you have a theory that you can really take to the bank when you can write it into code, and you can use it to make really precise predictions.

So a lot of code is such a powerful and such an expressive mechanism for describing the world, for describing how things work that if you can generate code subject to observation, subject to constraints, it's all of a sudden gives you this mechanism for building this very expressive knowledge representations. And a tool for reasoning that is very, very different from other mechanisms out there that people have tried in the past, including even deep neural networks.

And so that makes the problem of how to generate pieces of code that do very specific things that you want them to do. Makes it have a much bigger relevance than just helping us write software more cheaply.

And is there anything else that you'd like to share with our audience about your research and where could they go to find out some more maybe about sketch or what you're doing in your group?

We actually have a web page *neuralsymbolic.org,* where you can find more information about this project.

Great. Well, it's been a fascinating discussion, Armando. Thank you very much for your time today.

Sure. You're welcome.

If you're interested in learning more about the CSAIL Alliance Program and the latest research at CSAIL, please visit our website at *cap.csail.mit.edu,* and listen to our podcast series on Spotify, Apple Music, or wherever you listen to your podcasts. Tune in next month for a brand new edition of the CSAIL Alliance podcast, and stay ahead of the curve.