EPISODE 327

[INTRODUCTION]

[0:00:00.3] JM: Every program gets complied down to ones and zeroes before it can be executed against hardware. Before being translated to that machine code, program that are written in a language like Rust, or Swift, or Java, they spend time in an intermediate representation.

In java, this intermediate representation is Java bytecode. Many different languages, such as Scala, translate to Java bytecode, because there has been lots of optimization written to speedup that Java bytecode. Java bytecode runs on the JVM, the Java Virtual Machine. LLVM is a project that draws inspiration from the JAVA Virtual Machine. LLVM originally meant low-level virtual machine, but today it is just called LLVM and describes as set of compiler tools.

In today's interview with Morgan Wild, we explore how compilers work, how different process or hardware architectures present a problem for those compilers and why LLVMs intermediate representation creates a layer of interoperability for any language that compiles down to that intermediate representation.

Whether you are new to compilers or you have some experience, this episode will appeal to you. Morgan is an excellent teacher and his enthusiasm for the subject comes through. He has a 30-minute YouTube video, *A Brief Introduction to LLVM*, that's in the show notes. I highly recommend it if you're curious about this topic. It's one of those really succinct explanations that nonetheless has a ton of depth. Thanks to Morgan, and I hope you enjoy this episode.

[SPONSOR MESSAGE]

[0:01:44.3] JM: To understand how your application is performing, you need visibility into your database. VividCortex provides database monitoring for MySQL, Postgres, Redis, MongoDB, and Amazon Aurora. Database uptime, efficiency, and performance can all be measured using VividCortex. Don't let your database be a black box. Drill down into the metrics of your database with one second granularity.

Database monitoring allows engineering teams to solve problems faster and ship better code. VividCortex uses patented algorithms to analyze and surface relevant insights so users can be proactive and fix performance problems before customers are impacted. If you have a database that you would like to monitor more closely, check out vividcortex.com/sedaily to learn more. GitHub, DigitalOcean, and Yelp all use VividCortex to understand database performance. At vividcortex.com/sedaily, you can learn more about how VividCortex works.

Thanks to VividCortex for being a new sponsor of Software Engineering Daily, and check it out at vividcortex.com/sedaily.

[INTERVIEW]

[0:03:04.7] JM: For Software Engineering Radio, this is Jeff Meyerson. Morgan Wild is an iOS developer at Shoplandia. He is the creator of a YouTube video called *A Brief Introduction to LLVM*, which I found to be a very helpful resource.

Morgan, welcome to Software Engineering Radio.

[0:03:20.6] MW: Hey there, Jeff. Nice to be here.

[0:03:22.2] JM: Today, we're going to talk about the LLVM, and I want to have a general introductory discussion of some compiler stuff first before we get into what LLVM is, because it's related to compilers. For people who are needing a refresher, what are the high-level steps that a piece of code goes through when it gets compiled down to binary ones and zeroes?

[0:03:46.7] MW: Right. In a general sense, any kind of programming language has to end up looking like a machine code that a CPU can execute. Obviously, there's a whole bunch of CPUs, and there's a whole bunch of programming languages. There has to be something in the middle there to translate, so to speak, or compile the source into what a CPU would understand.

A big step up from what we had before compilers, which was assembly language, which essentially was a human-readable binary code. We would just issue instructions and they were

non-portable instructions. If you say worked on Intel, you can do ARM, or vice-versa. The compiler sort of created this abstraction layer over the machine language, or CPU instructions, by allowing us to program in a sort of more abstract programming languages where we worry about the logic, sort of the things that we want to see happen, rather than thinking about what to instruct the computer to do to get to that point.

Any kind of source that you write in a language that is compiled, because there're languages like Python, in PHP, and Ruby, that they are not compiled. They're interpreted light. Any kind of compile language does have a necessity of having a compiler to translate that into machine language.

First step is you take the source file and you basically just put everything into tokens. You just assign, do a syntax check whether the tokens check out and whether it's legitimate piece of language, and you compile trees into trees from that and you get, essentially, an abstract non-readable format of your code that just scrapes all of your comments, your variable names, and all of that stuff.

Depending on your architecture from that abstract form, sort of cleaned up form, there can either be a direct translation into one architecture or another. The term here is target. Any kind of CPU architecture you're looking for is essentially a target. If you're targeting an iPhone, you have to worry about instructions that are different when you would be targeting a Macbook, for example.

If you're directly compiling, like GCC, which is the new compiler collection, takes your C, for example, and just spits out machine code. Structures like LLVM create an intermediate representation of that code. It can optimize that code and deal all kinds of smart stuff that allows it to speed up your existing code. If you compile it, say, three years ago, you would compile it now and you get the speed up. That finally gets translated into machine-readable code.

[0:07:15.7] JM: Absolutely. Let's talk about the pre-LLVM world before we get into LLVM. You touched on this idea of the N by N language to process problem, where in a world without LLVM, you potentially can have this scenario where for each language and for each processor,

you need an individual way of getting that language into a form that that processor can understand. Is that accurate?

[0:07:49.1] MW: Yeah. It is a very big problem from, I guess, a few decades back, that prevented and stagnated the industry in that portability was pretty much nonexistent before Unix came along and created the POSIX standard. Before that, before POSIX, you had people building all these architectures and they didn't share a whole lot in common. If you, as a programmer, required a simple task as reading from desk, you would have to abstract that part away yourself and take care of any number of architecture, because Unix does file read in one way, there's one command, Windows does it differently and some architecture acts would do it, would do it differently.

These things that have stagnated the industry to a point where Java, when it came out, was this massive savior for the whole market and that programmers started dreaming about the build one once and run everywhere launcher which is pretty powerful when you're stuck in the world where you're limited in that regard.

[0:09:10.4] JM: Now, we're going to get into LLVM in a sec. You mentioned Java, and you mentioned the degree of portability that Java added. Can you explain what that portability was and what were some of the result of people being inspired by the JVM model?

[0:09:28.5] MW: Java is sort of this poster child of that build once and run anywhere movement that sort of later was transformed into the web architecture, or at least inspired the web architecture where, pretty much, you have this experience where you develop a website over a web service and you have the expectation of being able to run those anywhere. That came from Java, and JavaScript was sort of directly influenced by Java.

What Java does is it takes your source, and LLVM and Java has this part in common where LLVM has the IR, or the intermediate representation, and Java has the bytecode. From that point on, they act different upon those two things, because what Java does is it distributes the bytecode and you run and you execute the last part of what a compiler would do, which is translate bycode into machine-executable commands runtime.

There is an inherent lag in any kind of Java program where there's like a two-second thing that needs to happen for the bytecode to be loaded on to what's required for that architecture. Yeah, that simple fact of being able to carry that abstract to the form of your code in a form of bytecode and then compile it only when you run it, certainly does have its benefits.

[0:11:04.4] JM: Right. I think where you're getting at with this is that in the JVM ecosystem, you've got this bytecode, and the bytecode is this representation that's not super low-level, it's not super high-level, it's somewhere in between, and because we have this bytecode representation, we can move it between different architectures and we can have the same predictable runtime experience on each of those hardware architecture.

LLVM borrows from this in some ways, because of this intermediate representation that we're going to get to. Let's start simply. What is the LLVM?

[0:11:49.0] MW: Back in the day when it was first formed, it was a step towards emulating the bytecode intermediate language between an executable code and source code, and it turned into this pretty massive effort that's open-source and it's a whole ecosystem of compilers and their tools and the thing that joins them together, the thing that all of them share, is the intermediate representation, or IR.

Obviously, the man, or one of the two men that created LLVM is Chris Lattner. He later joined Apple specifically on that achievement. It was a graduate project. Most of us would dream of a graduate project that's successful. Yeah, it turned into something that I've heard recently it's able to C faster than GGC is able to do.

They started off a lot slower than C, but their whole mission is to create an optimizer for their custom intermediate representation that is able to continually optimize the code to the point where it just gets better and better and better.

[0:13:11.2] JM: Yeah. We'll get into as much of the LLVM as we can, but this is — For listeners who don't. This is a really big subject. Basically, before LLVM, compilers tended to have this monolithic structure. As you talked about in your YouTube, which I'll have in the show notes, LLVM breaks the monolithic compiler structure into a series of steps. It breaks this structure into

scanning, parsing, intermediate representation, the intermediate representation optimizer, semantic analysis, target code, and finally, compile code.

We could get into any number of these steps in the compilation structure. As you said, the intermediate representation is really the area that we should focus on. What is meant by that intermediate representation? What does an intermediate representation mean and how do we get there?

[0:14:08.5] MW: Sure. Before that, I do want to clarify that. You've mentioned seven steps here. They're not unique to LLVM. In general, when you're thinking about compilers, pretty much, all of them have the sequence of those six or seven steps. I want to stress that that's not unique. LLVM did not invent that sequence. What it did was is it decoupled the IR from, essentially, what is the frontend. Meaning, parts of the compiler sequence that are concerned with a specific source, source code, source files, and the backend, which essentially is concerned with optimizing specific targets and then creating executables for those targets.

The key unique thing that it did was it sort of separated IR from the sequence here. Pretty much, every compiler has that sequence, and pretty much, every compiler has that sequence in a modular-ish way within their own world, but none of them — Yeah, I feel like none of them — At least not the big ones. They don't have the intermediate language as separate as LLVM has.

[0:15:29.5] JM: Okay. Thanks for that clarification.

[0:15:31.8] MW: Yeah, because any computer scientist that goes to the compiler class, you'll sort of see the sequence, and it's there. It's not an invention that we can credit Chris Lattner with. To your question, IR is — It depends. Every system has its own IR, and it's just a step that uses the optimization steps, which create the — The executables are speedy and affirming.

What you have — In regular source code, as I mentioned in the beginning here, is soft of high-level structures. Let's take Swift, for example, because that's my favorite language. You have your classes, you have structures, you have functions and methods and all kinds of high-level stuff. Now, no processor operates on that level. Obviously, they don't understand that.

In between machine code, executable code, and source code, you have this assembly-like intermediate representation. When I say “assembly-like”, a lot of the people that deal with assembly languages can pretty much immediately feel accustomed to what’s going on in specifically LLVM IR. What you do there is — First of all, you don’t work with variables, you work with registers.

To that point, so that the biggest difference from any kind of assembly language is registers are mutable. By that, I mean, when you have your REX register, you just put stuff into it or you take stuff out of it. You read from it. It’s a constantly changing state. It’s a physical register that just constantly is mutating. Any number of parts within your code can change that. Being able to optimize any kind of dynamic data is a lot harder than optimizing something that’s immutable. IR for LLVM, has immutable registers, which is essentially just an unlimited sequence of let or const declarations that you do.

[SPONSOR MESSAGE]

[0:18:09.3] JM: You are building a data-intensive application. Maybe it involves data visualization, a recommendation engine, or multiple data sources. These applications often require data warehousing, glue code, lots of iteration, and lots of frustration. The Exaptive Studio is a rapid application development studio optimized for data projects. It minimizes the code required to build data-rich web applications and maximizes your time spent on your expertise. Go to exaptive.com/sedaily to get a free account today.

The Exaptive Studio provides a visual environment for using back end algorithmic and front-end component. Use the open source technologies you already use, but without having to modify the code, unless you want to, of course. Access a k-means clustering algorithm without knowing R, or use complex visualizations even if you don’t know D3. Spend your energy on the part that you know well and less time on the other stuff. Build faster and create better. Go to exaptive.com/sedaily for a free account.

Thanks to Exaptive for being a new sponsor of Software Engineering Daily. It’s a pleasure to have you onboard as a new sponsor.

[INTERVIEW CONTINUED]

[0:19:42.0] JM: Okay. Let's go through a simple example. Let's say we had a program that we wrote in Swift. As you said, this is one of the favorite programming languages of today, of 2017. I think there was a Stack Overflow survey recently that said this was the favorite language of programmers on Stack Overflow. This is a language that is used to develop most iOS apps these days.

Let's say we wrote a Swift app that was just X = 5, Y= 2, and Z = X + Y.

[0:20:16.1] MW: Sum — Yeah.

[0:20:16.9] JM: Yeah. Then, print the sum, or something. If wanted to say, "Let's compile that and run it." Maybe you can explain what happens to get to the intermediate representation point, and what's going to happen with those registers in the intermediate representation and optimization step.

[0:20:36.6] MW: First of all, when you're tokenizing your source, say that X = something like or something. Swift does a beautiful thing, which is called type inference. You don't need to write int, or float, or whatever. That step is required for the complier to allocate the correct number of bytes and sort of be able to deal with the underlying data.

What the first few steps do is they position everything that you've declared into sort of stack frames, and the stack frame is just concerned what tokens are you able to access within that stack frame and what tokens will you be able to access from shielding stack frames. At that point, what you do is you get the token identifier, which is X, or Y, or Z, and then you get the type, and then you get the current value for that thing.

Something as simple as sum — And this is one of those points that I showed in the presentation, is you take those types and you convert them into IR types. Obviously, you don't have structures or classes, but you have pretty advanced types like function pointers, and all kinds of numeral types, and functions, and all of that stuff, which is definitely a lot higher than what you would get with assembly-like languages.

Every single declaration of a register has to be preceded by a type, so there's no type inference. Yeah, you got your X and your Y and they can be translated into anything, but all those identifiers will start a percent sign and let's say percent one, percent two, would be your X and Y and they will probably have an integer type, let's say integer 64, depending on the architecture, of course, or 32.

Like with any register, you'd be able to issue the ad command. The only difference in terms of commands here, since your registers are immutable. It's very much a functional language. Any kind of result from an operation that mutates data or potentially would mutate data, it's just a return value. When you would add stuff in assembly-like languages, you do like ADD and then one register and another register. You won't be able to do that here. You wouldn't have to assign that value to another register, to a new register. Yeah, that's addition.

The only difference here would be you take the plus sign, you turn that into ADD. You add the type identifier, and your two sort of values — Not too big of a deal when it comes to simple math.

[0:23:44.2] JM: It sounds like there's not much optimization going on in this example. Is that right?

[0:23:48.4] MW: There's zero optimization.

[0:23:51.3] JM: What kind of things will we need to add to this code in order to illustrate the optimizer?

[0:23:57.8] MW: Let's say Y would be zero. Your X is two, your Y is zero, and you're adding these two numbers and you're using the sum. If you're defining value with just a numeral, you could optimize that out yourself. Often times, when value is defined in a static let somewhere in a struc that is far away, you don't see the value. These things easily happen. It's one of the most popular optimizations here.

What a processor would do is you'd waste a couple of instructions, at least, to do that non-operation, essentially, because adding zero to anything does not change the value. To save those couple of instructions, a couple of cycles, IR looks for common patterns. The sequence of patterns is constantly updated and it's a growing list. It obviously differs by the types that you're dealing with.

Say, any kind of any numeral type would have plus zero equals the same value. You can optimize that out. Times one, the same value, you can optimize that out. All that mathy stuff, you can optimize out. The way it works is deep in the C++ that's dedicated for LLVM, there is a split, a branch, depending on what kind of operation you're doing, that takes you to a for loop that iterates through all the possible optimizations for a specific — I forgot what the sequence is called. But for a specific sequence of code, which his essentially any kind of operation, that add would be one example, move would be another example.

It loops all the possible options there, and it applies the optimization and assigns the result to a temporary variable, and then it checks, first of all, whether there's no errors that are introduced by that change and whether there's any value gain from that operation. Then, if that's true, that gets substituted into the original code.

[0:26:26.4] JM: You're saying that when I compile my Swift code, after it gets translated into the intermediate representation, there is a tree of possibilities that are created to attempt to see. Is there a way to represent this same code in a more concise fashion than the original naïve intermediate representation that we've created? Is that right?

[0:26:58.1] MW: Yeah. First of all, you should probably never deal with IR directly, and — The first stop is sort of the naïve translation, a verbatim of what you see in your source file into IR. Only then can you do the optimization, because then you're dealing with immutable registers and a structure that inherently is more accessible to scans like that.

[0:27:28.8] JM: If we're looking at all these different potential configurations of our code, that sounds like something that would be really, really time intensive, especially for complex programs.

[0:27:39.1] MW: If you remember, when Swift came out, compiled times for something that was bigger than a few view controllers, they were pretty big. You're right, it does take time to optimize the optimizer for a specific frontend. The main thing here is that any kind of performance gains from, say, someone working on optimizing C, and obviously, to optimize C, you have to go through LLVM IR and do some clever optimizations over there.

Guess what? Those same optimizations translate to any frontend that gets compiled down to IR. People that are working on these — I don't know what's the number of languages that have frontends to LLVM IR, but it's pretty big, and some of the languages are very well-used. Those optimizations get shared.

[0:28:37.2] JM: Right. I want to eventually get to the modular part, but a little more on this intermediate representation optimization stuff. I am reminded of the Deep Blue, the IBM Deep Blue stuff, because this was the chess computer, where given any chess scenario, Deep Blue would look at all of the potential decisions spaces that you could potentially go down and play out the entire chess game down that part of the decision tree.

It sounds like the intermediate representation optimization is that level of complexity, because if you're looking at the entire tree of possibilities for different ways that your code could be arranged, that to me, in a complex program, that sounds like it is just too much work. Is that accurate, or is it just — Is it possible to break down?

[0:29:35.3] MW: I wouldn't say — In a way, it is just and that you end up looping over the entire sequence of this specific optimization, but you're not looping for every existing optimization for each line of code, or each operation, and operation is the atomic unit there.

First of all, what you're doing is you're putting that existing operation and the existing variables into a specific slot and niche. That alone reduces your problem space significantly. Obviously, when you reduce your problem space, you're not dealing with what's the chess board total probable moves when you start a game — Trillion? I don't know what that is.

You're not dealing with anything of that scale, because you have this inherent step of you take an operation and then you take out the existing identifiers, the values there, and then you only

go through the appropriate optimizations for that. There's, obviously, predictive stuff going on there which I'm not in depth on. From what I know, it is certainly not enumerating everything that you can possibly think of.

[0:31:05.7] JM: This certainly sounds like a problem that could be assigned across multiple threads, because if you're testing these different —

[0:31:13.1] MW: These are super parallel. When you're dealing with optimizations, you're optimizing each individual operation. You're not optimizing in terms of multi-operational sequences. The parallel nature of this whole thing is very great. Yeah, absolutely.

[0:31:32.6] JM: When you say operation, maybe this is a naïve question, but what do you mean? What is an operation?

[0:31:38.8] MW: For example, add one register to another.

[0:31:42.8] JM: Can you explain some other examples of operations? I guess I'm having trouble understanding where — Add one variable to another. That doesn't sound like something where there would be that many different configurations to test. Maybe you can help me clarify that.

[0:32:01.2] MW: There can be a whole bunch of them. Obviously, you have your load and store. You have your control transfer ops. You have your type conversion ops, like when you're adding —

[0:32:15.0] JM: You're illustrating add as a fairly high-level operation that gets translated into a bunch of low-level operations.

[0:32:21.5] MW: No. Add is as low as it gets.

[0:32:24.5] JM: Okay.

[0:32:25.3] MW: Yeah. There's nothing that —

[0:32:27.4] JM: Oh, I see. You're talking about other operations that add —

[0:32:30.4] MW: Other sort of equivalent parallel operations.

[0:32:35.0] JM: Right. Load and store — Maybe explain what a load and store operation is.

[0:32:39.9] MW: Loading is reading from a register, and storing is a writing type register.

[0:32:45.6] JM: Okay. That's an example of something that you could see being optimized if it was tested in different —

[0:32:52.8] MW: I will say you're reading a value and assigning that to another value without any kind of mutation. An optimization would be anywhere you're using the variable that you're — The identifier that you're assigning to. You can just replace with the identifier that you will be assigning in. If it's a constantly, obviously, there's — If it's a variable that can mutate the underlying value, you can't do that. If it's a constant and you're assigning just that, basically, creating an alias, you can optimize that out.

[SPONSOR MESSAGE]

[0:33:40.3] JM: Every user request is unique. Different requests come from different geographic locations. Some requests should hit a cache, or a CDN, and conditions change overtime. Your application might get a spike in traffic. You spin up new servers automatically, and then the traffic dies down, and you spin down the server.

Modern application infrastructure is changing all the time, and your routing is changing all the time. Dyn provides DNS that is as dynamic as your application. Dyn's intelligent routing gets your users to the right cloud service, the right CDN, or the right datacenter. With 10 years of experience, Dyn is trusted by Netflix, Twitter, and Salesforce, to maintain uptime even in the face of unreliable networks and malicious DDoS attacks. Dyn is a DNS solution for products and companies of all sizes.

[INTERVIEW CONTINUED]

[0:35:13.1] JM: Okay. You're giving an example where you take a load and store operation and you test different ways of doing it. You might get memory savings. You might get time savings based on variable reuse, for example.

[0:35:28.1] MW: Yeah. Any kind of variable that you don't have to assign is a variable that you don't have to get space from you stack for, is a variable that you don't have to de-allocate the memory for. It's always a plus to having fewer —

[0:35:45.0] JM: Cool. Before we hop out of the super lower level stuff and talk about some of the flashier characteristics. We talked about this load and store. Just to close the loop for something people who may not like load and store. That doesn't sound like anything I use in my programs. Can you just illustrate —

[0:36:00.7] MW: That's enough.

[0:36:01.3] JM: Yeah. What will be a higher level line of code that might end up translating to a load and store operation?

[0:36:07.4] MW: Assigning a variable from another variable would be an example of a load and store. Any kind of an assignment, it's a single operation in Swift. You just do A equals B, but there's an abstracted command, even in assembly, called move, which essentially does two things. You can use move from memory to memory. Anything that's on the stack, you can move things from the stack, or within the stack. If you're dealing with registers, there are certain instances where you would have to sort of read from a register, put that into memory, then take that memory and read it into — And write it into another register where you can do a flip. You would need to load and store that type of thing.

This is something a programmer would never interact with in their daily life, because, thank God, these things are abstracted away and all we deal with is a few controllers and transitions and all that beautiful stuff.

[0:37:12.7] JM: You've given us a beautiful, I think, from the top to intermediate representation explanation, and I think your earlier conversation sort of gave people a picture of what happens after the intermediate representation. It goes down to machine code, and there a process or stuff that happens. If people want to know more about that, they can look into how compilers work, or basically what happens after the intermediate representation.

I want to talk more about why this intermediate representation is a big deal, because, essentially, if you can get any piece of code into the intermediate representation, the intermediate representation can do a lot of optimizer work, and we can have an open source collective working on the intermediate representation and driving gains for any of these other languages that can compile down into the intermediate representation. We're talking about Swift, and Rust, and anything else.

Maybe you could talk about what are the challenges that a language has to overcome in order to be able to convert into this intermediate representation?

[0:38:25.0] MW: That's all beyond what I can contribute. It's a topic that I know a lot of people go through when they think about doing a hobby project of a sort of language, and most of them pick LLVM IR as a way to compile it into executable code, toy languages.

[0:38:48.3] JM: Fascinating.

[0:38:49.8] MW: From what I heard, it's one of the easier, or probably the easiest, or maybe even the only viable option there when creating a new language. Whether the audience would be just you or even if it's bigger.

[0:39:04.6] JM: Maybe eight years ago, people might have said, "I want to write a new language," and they write Scala, and then they say, "Okay. I want this language to compile down

to a machine, so I'm going to write it for the JVM, or I'm going to write it to compile down to bytecode."

[0:39:18.0] MW: Most likely, what they would do is translate that into C, then just use GCC to their compilers to get the machine code. I think Haskell has this attribute where their functional programming source gets translated into C, super unsafe C, and that gets transferred on to a machine code.

[0:39:43.6] JM: Ah! You're saying, that in the past, people might have used C as the interoperability layer. Whereas today, we can do LLVM.

[0:39:53.7] MW: It's very important to say that GCC is the bid dog here, and I think it's still is, since it's been going for, I don't know, 50 years now, 40. Who knows? It's a super massive project, and inherently, it has a lot of optimization. Whether they're good or bad, whether they're modular or not, you can produce super quick executables with GCC. All you need —

When you think about intermediate languages, you think low-level memory access, CPU instruction access, and you get a lot of that with C. Tweak it a little bit and you can get to run memory. A lot of these embedded systems use C as their sort of interface language.

C, inherently, has a lot of properties that IR has, and so it was sort of a natural use for the language until something came out that have the sole purpose of being in the middle. C does still take up that a little bit, but, obviously, it's being phased out by tools that were made specifically for that task.

[0:41:09.5] JM: Safety is something that I have heard. You're not the first person who I've heard mention C is somewhat unsafe if you want to build a systems level language, and Rust is something that I've heard is a language that has some safer properties than C. Perhaps, some of that is due to being able to be built on top of the LLVM, on top of the intermediate representation. Is there some notion of safety that is built into the intermediate representation?

[0:41:43.6] MW: No, because when you think about safety, you don't think about the language itself most often. What you think about is things that people build with the language. You have a human component there to abuse safety and turn it into something that's unsafe.

For example, a lot of people are taught C and when they go through sort of the initial tutorials, they see how to read and write data from the terminal. Obviously, that's been outdated for God knows how many years, and it's no longer safe to do it that way.

When it comes to safety, either you're using a framework that someone wrote that's a systems engineer, or a compiler designer, and it's unsafe, because it's using some part of the language unwisely. You're responsible for your own code and you don't know something, or you have a leak, or you don't check input, or something like that, and you create unsafe programmers. The intermediate representation does not, in and of itself, create safety issues, because the loop is pretty much closed there. You have that with higher level stuff.

[0:43:01.9] JM: I jumped the gun on talking about Rust. What is Rust able to leverage from the LLVM?

[0:43:10.2] MW: That's as far as I know in terms of Rust. It's using the backend for the source. It's created by the open-source community and they worry about the definitions there and how to organize the language. The part that is the most difficult in optimizing with that would require the most resources is the backend, or how to make code run quickly on Intel, on other architectures other than X86.

That's where you get the build-in benefits from LLVM by using their backends for — I think they have backends for pretty much every single platform that's big, definitely ARM and x86. That's pretty much allows you to reach that whole market.

If you're developing any hobby language — And Rust is still at a point where — In most cases, I might be ignorant to any kind of big existing project that uses Rust as their main language. I haven't heard of any at this point.

[0:44:20.3] JM: Dropbox.

[0:44:21.0] MW: Really? I heard they're Python-driven mostly.

[0:44:23.9] JM: I think that Dropbox — I think Dropbox is Rust. Anyway, sorry to interrupt you. Yeah, I'm pretty sure that — They moved off of the cloud recently. When they moved off the cloud, they rewrote a lot of stuff. I think that they used Rust. Anyway, I'm sorry that I interrupted you.

[0:44:43.7] MW: There's probably parts — Any kind of startup. Even individual developers have tools that they write. I think I deal with five different languages on a daily basis pretty much, and that's just because you use different tools for different purposes.

Regardless of that fact, it's still a very nascent language, and you either dedicate your resources, and that's probably the whole point of having these beautiful projects like LLVM, is that people can now work on things that there are the best at, or at least try to be the best at one specific part.

With LLVM, Rust people don't have to worry about being good, Intel architecture people, or ARM architecture people. They just need to worry about how to deal with memory and how to deal with a good structure, and so that it focused on the frontend. That's a big picture commentary. I wouldn't be able to do more than that.

[0:45:48.9] JM: We talked about JVM versus the intermediate representation layer a bit, and Java Virtual Machine. LLVM stands for low-level virtual machine. I probably should have mentioned that earlier.

[0:46:02.4] MW: No. It doesn't stand for that anymore.

[0:46:04.4] JM: Oh! That's right. It used to. Yeah, now it's just LLVM.

[0:46:08.7] MW: It's like BMW is BMW. You don't enumerate what each letter means.

[0:46:16.7] JM: Anyway. Where I was going with that was that there is an analogy to be drawn, because in the Java bytecode ecosystem, you have this shared layer, the JVM bytecode layer

where people — People will have a big Ruby on Rails application. When they hit a performance bottleneck, they will switch to JRuby, which is a version of Ruby that runs on the Java Virtual Machine. Basically, as I understand, JRuby is Java, so you can compile Ruby to JRuby and then JRuby runs on the JVM, the Java bytecode, so it runs faster. You get to take advantage of those bytecode optimizations.

Similarly, we've got this intermediate representation that we've been talking about where Rust can compile down to it. Swift can compile down to it, and we have these optimizations that are shared on a platform. How do the optimizations of — How does the Java — I should ask a bigger question. How does the Java bytecode ecosystem compare to the LLVM intermediate representation ecosystem today? How would somebody choose between writing their language to compile to Java bytecode, versus compiling to intermediate representation?

[0:47:29.8] MW: To some extent, they are different animals, you can do an apples to apples comparison, because they take a different niches, I think. IR is substantially lower level than what you would get with bytecode. Meaning, even though IR is isomorphic and that you can translate it to bitcode, which is a super tight version of what those operations and commands look like to a human-readable IR, and memory storied IR.

Human-readable IR is not source code that a person would be able to tell just skimming through it what the program does. Whereas JVM bytecode, you can actually take it and — It's isomorphic to the source level. What you have is even though you lose the original identifiers, and that can take away some part of the meaning, or in some instances, a lot of the meaning. You still have a line-for-line representation of what the source code look like.

That is probably the major different there. By the way, you can check that difference by just inquiring into — I know this for a fact. You can do that on Android Studio, so any kind of Itelli**JM:** IDEs, where you can open up libraries that are compiled into bytecode and you can look through the actual Java code without the identifier names and all of that, but you can still look through it. Whereas IR does not have that quality.

[0:49:16.2] JM: I want to begin to close off and just talk about where the LLVM ecosystem is at today. As you mentioned, LLVM was originally just called the low-level virtual machine. That's

what it stood for. Now, today, it's called LLVM. It's literally just called that, because it's representing a larger ecosystem of compiler tools. Can you explain —

[0:49:38.4] MW: Specifically — Sorry. The bigger parts would probably be the linker, LLD, and LDB, which is the debugger. If you've worked on Xcode at all, you would see LDB messages pretty often. When you add break points and stuff, that's when you're dealing directly with another piece of the LLVM architecture. That's the debugger. Those three parts are probably the biggest, if you abstract away IR.

[0:50:09.0] JM: Is that debugger — I guess that's running — What happens is you compile your code, it goes to the different — It goes to the intermediate representation. It gets optimized, and then it runs, and then you are debugging against the intermediate representation that was settled on after the optimization. Is that right?

[0:50:30.8] MW: You're debugging against live code, but that code is obviously annotated, so that you can have access to the code that you've wrote in Swift, for example. It's not like a stripped-down version. Any kind of debugger adds a layer to the intermediate executable code that can transfer some data.

When you exhaust that, you often times are taken to IR, or some form of IR. Then, you just see a whole sequence of registers within a specific stack frame, and then God help you finding what went wrong there. The LDB debugger takes care of that specific part.

In the Apple ecosystem, you probably don't veer too far from the entire sequence there, because you're linking your stuff with the LLVM linker against your static dynamic libraries and you're debugging with your debugger. Yeah, Xcode is very much reliant on LLVM.

[0:51:39.6] JM: Okay. I want to wrap up with one — I want to make one quick point, I guess. That people who are, maybe, new to thinking about compilers, the reason we're spending so much time talking about this optimization stuff and why this is so important is because if you can spend a lot of time automatically optimizing your code, then every end user who is executing that code gets to take advantage of the optimization. Even though the developer who is compiling the code might have to wait around while the different intermediate representations

are being tried. Once it settles on an optimal one, all of the end users who are actually executing that code gets to take advantage of the optimizations.

With that in mind, is that the main thing that the ecosystem is focused on? When we're looking at the future of the LLVM and the things that are going to develop out of this ecosystem, what should we expect? Should we just expect more optimizations that are going to be made on the intermediate representation level, or better debugging tools, or is there something bigger that is being worked towards?

[0:52:43.5] MW: I'm pretty sure what you've mentioned would be a good characterization for it. Essentially, what LLVM allows you in the future, I guess, you can probably see more and new exciting different languages being created, because you don't have the inherent bottleneck of worrying about sort of how that thing works on the different systems. That's one possible outcome.

You'll have a lot of frontend sharing their optimizations between each other. That's a big boom for the industry, because those resources are very valuable. Whenever you can share or collaborate with other people, that is a great, great benefit.

Yeah, it's going to be interesting to see what happens with the collection of tools here when Chris is no longer with Apple and no longer working actively on maintaining, as he was the primary person there since he moved to Tesla. It's going to be interesting how they're going to deal with the future related stuff come WWDC this summer.

[0:53:55.5] JM: Okay, Morgan. Is there any social media plugs that you'd like to give for people who would want to follow you? What are the best ways to follow you?

[0:54:04.8] MW: I use Twitter, and you can fine me @wildmorgan, last name, first name, on Twitter.

[0:54:11.7] JM: Okay. All right. Morgan, thanks for this illuminating discussion of LLVM. I think you gave a really clear explanation. I liked it that we were able to dive deep end, go higher level, and I recommend to anybody who's looking for further information on this check out your

YouTube video, which is only about 30 minutes long. That's why I liked it so much, is that it was concise and yet deeply informative. Thanks for producing that.

[0:54:37.4] MW: Yeah, I was a love child, I have to say. It took probably three weeks to get the thing together. 20 minutes. Yeah, you can have a pretty good refresher on not only LLVM, but the whole ecosystem for compilers.

[0:54:53.5] JM: All right. Morgan, thank you for coming on Software Engineering Daily.

[0:54:56.6] MW: Thanks for having me, Jeff. It's been a pleasure.

[END OF INTERVIEW]

[0:55:02.7] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. Thanks again Symphono.

[END]