EPISODE 439

[INTRODUCTION]

[0:00:00.3] JM: Scala is a functional and object-oriented programming language built on the JVM. Scala-Native takes this language loved by many and brings it to bare metal. Scale-Native is an optimizing ahead of time compiler and lightweight managed runtime designed specifically for Scala.

Denys Shabalin is a research assistant at the EPFL and the primary creator of Scale-Native. In this episode, Adam Bell interviews Denys about the motivations behind the Scale-Native project. How it was implemented and future directions. He also briefly touches on how Scala-Native made cold compilation times of Scala code twice as fast. If you're interested in functional programming, compiler design, or you want to learn some interesting tidbits about garbage collector design and tradeoffs, then you will like this episode.

In past shows we have covered many of the newer programming languages and the new twists on old programming languages. We've talked about Scala.js, which is related to this episode. We've talked about Swift on the server. We've also talked about Rust at Mozilla. If you want to find these episodes and many others, download the Software Engineering Daily app for iOS or android to hear all of our old episodes. They're easily organized by the categories.

As you listen, the Software Engineering Daily app gets smarter and it's going to recommend you content based on the episodes that you're hearing. If you don't like an episode, you can easily find something else, because our recommendation system knows what you like. These mobile apps are open sourced at github.com/softwareengineeringdaily. If you're looking for an open source project to hack on, we would love to get your help. We're building a new way to consume software engineering content.

We've got the android app, the iOS app, recommendation system, the web frontend. There's going to be more project coming soon, and if you have ideas for how software engineering media content should be consumed or if you're interested in contributing code or just hanging out in the Slack channel and seeing what transpires, come to github.com/softwareengineeringdaily or join our Slack channel. There's a link on

softwareengineeringdaily.com. You can send me an email, jeff@softwareengineeringdaily.com. I'd always love to hear from you.

Let's get on with this episode.

[SPONSOR MESSAGE]

[0:02:28.1] JM: Who do you use for log management? I want to tell you about Scalyr; the first purpose-built log management tool on the market. Most tools on the market utilize text indexing search, and this is great for indexing a book, for example. But if you want to search logs at scale fast, it breaks down. Scalyr built their own database from scratch and the system is fast. Most of the searches takes less than a second. In fact, 99% of the queries execute in less than a second. That's why companies like OkCupid and Giphy and Career Builder use Scalyr to build their log management systems.

You can try it today free for 90 days if you go to the promo url, which is softwareengineeringdaily.com/scalyr. That's softwareengineeringdaily.com/scalyr.

Scalyr was built by one of the founders of Writely, which is the company that became Google Docs, and if you know anything about Google Doc's history, it was quite transformational when the product came out. This was a consumer grade UI product that solved many distributed systems problems and had great scalability, which is why it turned into Google Docs. The founder of Writely is now turning his focus to log management, and it has the consumer grade UI. It has the scalability that you would expect from somebody who built Google Docs, and you can use Scalyr to monitor key metrics. You can use it to trigger alerts. It's got integration with PagerDuty and it's really easy to use. It's really lightning fast, and you can get a free 90-day trial by signing up at softwareengineeringdaily.com/scalyr.

I really recommend trying it out. I've heard from multiple companies on the show that they use scalyr and it's been a real differentiator for them. Check out Scalyr, and thanks to Scalyr for being a new sponsor of Software Engineering Daily.

[INTERVIEW]

[0:04:57.2] AB: Denys Shabalin is a research assistant at the EPFL and the primary creator of Scala-Native. Denys, welcome to Software Engineering Daily.

[0:05:08.4] DS: Hey.

[0:05:09.6] AB: Scala is a language built on the JVM. Could you give a brief overview of Scala the language before we get into Scala-Native?

[0:05:19.8] DS: Scala is this pretty cool language originally designed for JVM. It really can be described as a mix of a functional and object-oriented programming. It really doesn't bias towards one or another style. It really tries to blend both together, because there's both good and bad on both ends. For example, functional programming is basically considered like the [inaudible 0:05:44.2] type of Scala we have, and objected-oriented to be more of the Java style, old school, less popular side, but still language doesn't bias towards one side or another. You can perfectly do object-oriented, Haskell object-oriented programming and fancy functional programming at the same time, which is pretty unique, because most languages are heavily on one side or another, which is often considered to be a negative side of Scala because it's very [inaudible 0:06:09.8] — Anyway, that's what it is.

[0:06:13.2] AB: Yeah, make sense. Scala lets you kind of — You can combine sort of a Java style OO with ML or Haskell style functional composition.

[0:06:22.8] DS: Absolutely, yeah. That's it.

[0:06:24.4] AB: What is Scala-Native?

[0:06:26.6] DS: Scala traditionally has been a JVM-centric language. It used to compile only to JVM bytecode as the only target. What it means is it's really a plug and play on JVM and just can value Scala courses to bytecode and then you can run it alongside your Java application. It was the original backend, it's the original platform for Scala, but since then we got way more things.

I think the first major experiment to Scala outside JVM was .NET backend. That didn't work so well because of the differences between how JVM and common language runtime handle generics. It was a bit difficult. I think the first successful alternative platform for Scala is Scala.js by Sebastian Doeraene.

Basically, it's really a major difference in terms of how you run your Scala apps because you compile them to JavaScript through this very elaborate, advanced tool chain they have. Scala-Native is very much a similar project to Scala.js, but instead of compiling to JavaScript, it compiles natives codes. When I say native code, I mean more like a C, C++ than the long binaries that completely just don't require any virtual machine to run it. You get x86 or ARM binaries that you can just copy-paste into any machine with the same architecture and just runs. Of course, it has the good and bad of a native style development, but it's really kind of — The core idea of the project is very simple, it's to compile Scala to native binaries.

[0:08:00.7] AB: Makes sense. Scala.js gives you the ability to run in the browser. What problem does being able to run as a native application solve?

[0:08:10.5] DS: One of the issues with JVM as I see it is that JVM is a really heavy machinery. It's really requires quite a bit of a footprint just to run the VM. You can see it in terms of memory use. You can see it in terms of application startup time. You can see it sometimes in terms of overhead of the whole services, like just compilation behind the scenes. It's really because JVM is very advanced multi-stage, multi-tier VM and it's really hard to support all these functionality without incurring some overhead.

With Scala-Native and the way Scala-Native is different is that we do most of the expensive parts, like compilation ahead of time. It means you already have a pre-compiled and pre-optimized binary, so when you'd start it, it just runs your app. It doesn't do the whole multitier VM thing. We don't have an interpreter. We don't have multiple tiers of compilation, and whenever we emitted binary, that's it. There is no recompilation at runtime. There's no tricks we do. It's a simple binary, which means we have way lower footprint both in terms of memory and in terms of startup time.

This can be useful for a number of use cases of applications, like with ample common line tools. For common line tool, it's extremely important to start up quick, do your job and then die. This is an area where JVM is really managed right now, because just start the VM is extremely expensive operation. You definitely see this kind of like initial slowdown if your app is not long running.

[0:09:49.5] AB: That is what people refer to as like the JVM warm up. Is that right?

[0:09:53.6] DS: Yeah, exactly. It's like git warm up. It also has to do with the fact that when you run code in JVM, it actually goes to a number of stages. First, you go through interpreter. Interpreter is really slow. It's not meant to be there for a long time. It tries to go to compiled mode as soon as possible, and there are at least two compilers right now which are used in production, JVM C1 and C2.

C1 emits simple code to avoid interpretation cost, and C2 does a very, very — It's a very advanced optimizer that only optimizes heavily used parts. Basically, you have a very elaborate machinery which means that you don't get to optimize code only until your application is warmed up.

In native, we have basically — something in between C1 and C2. We already heavily pre-optimized your code before you're on it. At the same time, it's not quite the same as the VM, so there are some pros and cons.

[0:10:55.2] AB: Yeah. If you have a long running Java app, or a Scala app, I guess, then the cost of this warm up may be doesn't matter so much, but if something has to start up frequently, then re-optimizing it is a lot of overhead.

[0:11:10.3] DS: Absolutely. Yeah, that's it. Another area apart from common line tools is different types of user-facing apps where you can observe and perceive the startup time, which often are simple graphical apps, like for example like starting eclipse takes minutes. Before, it's like — You never close it, right? Because you're afraid to close it, because once you close eclipse, you'll have to go through the same thing again. It's basically not eclipse problem as far as I can

tell. It's largely a JVM problem, because, for example, other IDs started much faster than eclipse.

[0:11:42.8] AB: Yeah. I use IntelliJ, but I find the same thing. Yeah, I try to keep it running.

In your talk, Scala Goes Native, you were describing the JVM as a golden cage. Is this sort of what you mean by that, or you could describe this concept?

[0:11:58.5] DS: This metaphor was basically tried to motivate why we don't try to artificially limit what you can do in Scala-Native like JVM does. In particular, we don't try to send box codes so that you can only write a very safe code, which never escapes its cage basically. That's what I meant by the cage, is that we let you use low level, system level tools like raw access to memory, raw pointers and stuff like that, which is potentially unsafe but it's actually necessary in some domains, like systems programming where you want to have a very low level control of your memory.

The benefit is that you get more controls developer. You are not limited by its language and the VM, because you can do whatever you can do in C and C++. On the other hand, you do lose some of the safety you get of JVM, but this is kind of a tradeoff we take.

[0:12:53.5] AB: That makes sense. Is this strictly for memory management or how about interopt?

[0:13:00.3] DS: Generally, interopt is basically — Scale-Native exposes a number of language extensions aimed primarily at interop with C code. In a way it's a bit like writing C code in Scala. We expose things like pointers and structs. You can do [inaudible 0:13:17.1], but it also means it's extremely easy to call C code. You don't need to go through a number of layers and bindings to get there. You can just do it all in Scala without any C and C++ code. [inaudible 0:13:30.6].

It also means if you can call arbitrarily C code, you also can get arbitrarily issues that C code has. For example, different types of safety issues around buffer overflows and so on at first. It's definitely tradeoff. It's not free, like calling C code, it's not free in terms of like safety guarantees

you'd get. Again, this is kind of a tradeoff we make. We don't try to be as safe as possible. We try to be as flexible as possible.

[0:13:59.3] AB: Makes sense. You want to give people that tool even if it could go wrong. How about memory management on the JVM?

[0:14:09.8] DS: JVM is basically JC only platform. There are some very well-hidden but extremely well-known areas, like [inaudible 0:14:18.2], which lets you do unmanaged memory and manage it yourself. For example, you can allocate unmanaged memories with [inaudible 0:14:26.5] and then do row memory access on it. It's basically [inaudible 0:14:30.7] as pointers in C. Only JVM people try to hide it from you, even though every major performance centric framework actually uses it. Like for example, Spark does off-heap memory to manage most of its data, because it's just too expensive to allocate everything on [inaudible 0:14:47.7].

JVM people like to believe that you only have JC, which is the main paradigm. It's actually — JC on JVM is really good. Very often, it's good that you just use that and not do any unsafe memory management.

At Scala-Native we say those coexist and you have APIs for both and they're both easy to use. Definitely, unmanaged memory is danger thing. You can definitely shoot yourself, but for example if you want to do some domain specific thing to optimize in [inaudible 0:15:21.6] and so on. First, you can do that without jumping through to hoops, like with JVM.

[0:15:28.0] AB: Makes sense. It reports to be a memory managed, but if you look at these high performance apps, they're using backdoors to sort of actually do manual memory management.

[0:15:40.7] DS: Yeah. It's like most of them really go far away to try to [inaudible 0:15:46.3] to find this small door outside to get more freedom, but on JVM it's really hard to do these kind of things.

[0:15:53.7] AB: That's where the cage metaphor comes in.

[0:15:55.4] DS: Yeah.

[0:15:56.5] AB: Okay. Now that we understand some of the motivations for getting Scala to run natively, not on the JVM. Maybe let's discuss some of the implementation. Could you describe the compilation steps that it takes to get from Scala source to a native application?

[0:16:16.9] DS: Oh, sure. It actually a bit involved, and there is a good reason we have every single step on the line, but it's actually a really multi, multistep for us. First thing you do when you go from a Scala source, you always start with Scale source, which is a source of truth. In case of Scala-Native, you end with maybe a binary. It's not the one step [inaudible 0:16:37.3] from Scala source native binary.

The first things you do is you do [inaudible 0:16:42.1]. Basically, you do the pipeline from the main Scala compiler form JVM. This contains a number of things, but the most important one I guess is time check-in, because type check-in in Scala is very involved. We don't re-implement type check-in or language. We keep the same core, the same language on Scala and JVM, and then later once Scala compiler is almost on, we branch off and we emit something called NIR. NIR is short for native IR, which is our own [inaudible 0:17:16.6] presentation.

This NIR is the format we work with in our tool chain. When I say our tool chain, I mean Linker, Optimizer and Codegen. They all speak this language as if it was like a real language. To get from NIR to binaries, now we have one step closer, because NIR is already quite more low-level than Scala. For example, many things are gone and are like [inaudible 0:17:44.9]. Type system is much more simpler and actually very close to Java bytecode rather than Scala language.

The main difference from Java bytecode is that it's a safe worm, which makes it very easy to emit LLVM later. As I said, this worm for coder presentation, which is very nice for optimizing compilers.

From NIR, we need to get native binary and two major steps, extreme major steps on this way is, first, linking. Linking loads a minimal subset of your class path to satisfy your application requirements. Like for example, an app that doesn't use regular expressions should not pre-compile regular expressions in the binary and so on and so forth.

We try to really limit an amount of code we put into the final binary, not to include every single class of the class path, because sometimes class paths get quite bloated even though you don't use some of those things. Sometimes people [inaudible 0:18:43.2] on a library even though they've used single function from it.

We do something called [inaudible 0:18:49.2] at link time. Then after that step you'd get a minimal subset of the cross pass, which we optimize our own optimizer which removes common [inaudible 0:19:01.3] which LLVM doesn't know how to optimize well. Then in the end, we emit LLVMIR, which is another IR, but now for LLVM. LLVM is this project for reusable compilers basically.

[inaudible 0:19:15.8] for a C line compiler and it's also used by many, many other open source languages and it's actually very well-documented, very nice to work with. From there on, it's basically LLVMs jobs to get from LLVMIR native code, which we typically don't touch so much.

[SPONSOR MESSAGE]

[0:19:39.9] JM: The octopus, a sea creature known for its intelligence and flexibility. Octopus Deploy, a friendly deployment automation tool for deploying applications like .NET apps, Java apps and more. Ask any developer and they'll tell you that it's never fun pushing code at 5 p.m. on a Friday and then crossing your fingers hoping for the best. We've all been there. We've all done that, and that's where Octopus Deploy comes into the picture.

Octopus Deploy is a friendly deployment automation tool taking over where your build or CI server ends. Use Octopus to promote releases on prem or to the cloud. Octopus integrates with your existing build pipeline, TFS and VSTS, Bamboo, Team City and Jenkins. It integrates with AWS, Azure and on-prem environments. You can reliably and repeatedly deploy your .NET and Java apps and more. If you can package it, Octopus can deploy it.

It's quick and easy to install and you can just go to octopus.com to trial Octopus free for 45 days. That's octopus.com, O-C-T-O-P-U-S.com.

[INTERVIEW CONTINUED]

[0:21:11.0] AB: Yeah. If I understand this, you have your Scale code, you're using the frontend Scala compiler to get some intermediate representation and then doing some transformations and then passing that through to LLVM. Is that the big picture?

[0:21:30.2] DS: That's pretty much the gist of it.

[0:21:33.4] AB: Because there could be a lot in your class path, you're making sure that you only include things in that binary that are actually called within the program.

[0:21:47.8] DS: We don't only [inaudible 0:21:48.1], but we'd try to analyze it and kind of do our best guest of what it's going to be called, yeah.

[0:21:53.9] AB: Okay. Yeah, makes sense. One of your frustrations with the JVM was that its garbage collector doesn't fit to every use case. Many listeners may know that there are different types of garbage collection strategies. I was wondering if you could describe a couple of strategies for performing garbage collection.

[0:22:15.1] DS: On JVM you actually have a number of built-in garbage collector. As far as understanding, [inaudible 0:22:21.1] one is called G1. G1 is the latest collector from Oracle which is optimized for latency-centric workloads. Typically GCs are often kind of put in either latency sensitive or throughput sensitive buckets. What latency sensitive means is that the GC is optimized for shortest pause that GC can take to collect garbage, but this pause can be extremely frequent, but every single pause is small.

Throughput-centric collectors care about [inaudible 0:22:54.2] lengths of a single pause, but rather the total sound of time spent in GCs. For example, [inaudible 0:23:00.3] collector can take less pauses, but makes it much longer. Basically, on JVM right now, the official ones is G1 for latency sensitive and [inaudible 0:23:10.3] as far as I understand.

CMS, which was previously the default is deprecated as of Java 9, which is a bit sad because some of our workloads, like Scala compiler, I think CMS is still the best one, but otherwise it's basically three main collectors we have right now with CMS being deprecated.

The general seems for all of these collectors is that they're typically generational. They typically at least parallel, often concurrent. Concurrent mean is a collector runs alongside your application and tries not to stop your application as much as possible. The basic way it does garbage collection, not just in parallel, as in doing multiple shreds of garbage collection, but also concurrently to your application.

Compared to all of these, so where does Scala-Native send? Right now, we have a rather simple garbage collector called MX. It's inspired by a paper. You can see more information on our website if you're interested, but the general idea is it's a single generation collector, which is right now optimized for predictability. It's not concurrent today. It's [inaudible 0:24:21.5]. We currently optimize [inaudible 0:24:25.8] throughput and latency sensitive is our next big milestone, which we haven't reached yet.

[0:24:33.1] AB: Okay. That makes sense. You also have — As I understand it, you have more than one GC available in Scala-Native. Maybe you could describe what they are.

[0:24:46.4] DS: Right. That was a default one. It's actually not MX. It's called [inaudible 0:24:49.1]. GC is this super easy to use plug and play garbage collector, which was designed originally for a C and C++. The reason why it's even possible at all to make it work in this environment is that garbage collection is conservative. What does it mean? It means that garbage collector doesn't really require your app to declare ahead of time kind of [inaudible 0:25:11.4] objects. It will [inaudible 0:25:13.5] to a guest what objects are based on their size layout.

For example, if a field [inaudible 0:25:13.5] offset looks like a pointer, it can't consider the pointer even it's not as long as [inaudible 0:25:25.8] a bunch of properties that she wants to see from pointers. This is more expensive than precise garbage collection. Precise garbage collection knows exactly at which offsets you have pointers and which offsets it just stays at. It needs to do less work.

The main reason why our current new collector called MX is faster is because it's precise. We do use information about the whole object layout and it's way easier to collect the garbage. It's

still conservative in one small aspect, but it typically doesn't matter much. The stacks are conservative. It typically is not a problem.

Another cool thing about MX convert [inaudible 0:26:08.8] is that it actually uses a very smart data structure for allocation and collection which lets it bump allocate most of the time, which is really important, because bump allocation is the fastest way to allocate. Why we still use [inaudible 0:26:27.4] from time to time, [inaudible 0:26:28.8] to be quite expensive in our own experience.

Apart from these two, we have another collector called none GC, or setting is called native GC: = none. That one lets you completely disable the garbage collector. The idea behind that one is to kind of have a rough understanding of how much time is spent garbage collection and what's the baseline performance, what's [inaudible 0:26:57.0] perfect garbage collector, because essentially allocating and never [inaudible 0:27:02.4] is actually extremely close to perfect garbage collection. It's not perfect, because it will still allocate object to fall apart, if objects [inaudible 0:27:11.4] at the same time. It can still cause problems with memory locality. Most of the time it basically spends zero time in garbage collection, so it means it's as low overhead most of the time for most application. We use it as baseline to benchmark our GCs. Basically, main purpose is benchmarking.

Apart from that, [inaudible 0:27:33.0] use case like extremely short lived application, which really you don't need to manage memory, because they like run for less than a second and they don't allocate gigabytes of memory, but maybe hundreds. For those kinds of apps, it's actually beneficial to be able to disable garbage collector, because it means they will run at best performance as possible.

[0:27:54.9] AB: Makes sense. None exist as sort of for performance testing, but in actual fact it can be used for like a command line app.

[0:28:06.6] DS: Yeah.

[0:28:07.1] AB: You have none, so you can test what you're calling a perfect GC against the two that you have. When you do this type of testing, how do they perform compared to a perfect standard?

[0:28:21.1] DS: Compared to our friends, typically, MX is somewhere around 20% overhead. This means if you add MX, your app will run 20% slower. In comparison to [inaudible 0:28:31.8], is somewhere around 100%. It basically enables GC, slows down the application factor of 2X, which is pretty bad and it mostly has to do with conservative nature of the collector. MX is 20%. It's actually still higher than we want it to be. I think we can get 10 or maybe even less without changing the design of the collector too much.

[0:28:58.0] AB: Interesting. Do you happen to know — 20% away from absolutely perfectly doesn't sound too bad.

[0:29:05.5] DS: Yeah.

[0:29:08.0] AB: Do you know like where the JVMs generational garbage collector would fit on such a measure?

[0:29:15.4] DS: It's a little bit hard to compare with something like CMS or G1, because they're [inaudible 0:29:18.8]. It's typically under 5% and that would probably say even — Probably even less than that, because for current garbage collector, you never perform the garbage collection on the actual application stack. You have a separate stack, which only pauses application to do simple things, like scans a stack or wait for this condition to halt. It's typically short pauses of five milliseconds or less.

They can be frequent, but typically [inaudible 0:29:43.9] as like under 5%. Basically, this is our goal performance to be on par with JVM. Right now we don't guarantee [inaudible 0:29:52.7] with JVM in terms of performance. There's still quite a bit of work to be done there.

[0:30:00.1] AB: Makes sense. Some people's complaint with the JVM is sort of the stop the world garbage collection, but you shouldn't go to Scala-Native to get away from that, because that's all you have at this point.

[0:30:13.9] DS: Yeah. At the moment, we don't solve [inaudible 0:30:16.3] problem. We're researching ways to refine our GC further, but right now, as of released version, only No GC has [inaudible 0:30:27.6] problems, because it does in GC.

[0:30:31.2] AB: Yeah, makes sense. Now, I think I understand how the GC works. I'd like to look at little bit at Scala-Native usage. Is Scala-Native the same language? Is it Scala or is it something, like a superset?

[0:30:49.5] DS: Scala-Native at its core is one-to-one. There are very few differences in terms of how we treat normal Scala language features. They mostly are around edge cases, like what happens when you call a message on a null or what happens when you do a cast which doesn't make sense?

On JVM, those cases are defined to show exceptions. Some of those are just undefined behavior on native, so it means anything can happen if you do this. Typically, it means it just crashes with [inaudible 0:31:19.2], which is basically a bit worse than JVM, but still, it's easily debuggable through native tools, like [inaudible 0:31:26.9] effectively show you as much as null pointer exception.

We don't currently guarantee one-to-one parity into edge cases, it's likely we will never this because it's typically been a none issue for us. It's a bit more annoying to debug some of these, but essentially it simplifies our [inaudible 0:31:47.3] quite a bit.

Apart from the core language, we're just almost exactly the same, like 99% the same. We have a bunch of extension for interop. Interop extensions are very different from Scala JVM. We don't have anything similar. We do have [inaudible 0:32:04.6] and things that go with them, like memory layout types, like structs. You can [inaudible 0:32:11.2] structs and they test meaningful [inaudible 0:32:13.9] which is the same as in C. We also have function pointers and a bunch of other things to basically make it easier to call [inaudible 0:32:13.9].

Generally, you don't have to use these kinds of extensions all. Actually, they're only for interop. Pointers are also extremely useful for kind of having a lower level GC free subset of language

that you can use for extremely performance sensitive applications. Again, you don't have to use any of these. The core Scala is truly as close as we can make it to be [inaudible 0:32:44.8] in JVM.

[0:32:46.7] AB: Makes sense. I guess with the pointers, then you can kind of approach that perfect GC we were talking about. If you've added a concept like structs, like structured types, functional pointers, doesn't that make it the language like a superset. Are these new keywords in the language? New syntax?

[0:33:06.3] DS: We don't add any syntax, whatsoever. [inaudible 0:33:09.6] check and without any problems, but by normal [inaudible 0:33:14.7]. It might not make sense, but essentially all of our extensions are tied to like magical intrinsic methods or magical annotations, which modify how we compile things. At the same time, it's still type check one-to-one [inaudible 0:33:29.7] compiler without changes. From a text point of view, it's the same language. From a runtime [inaudible 0:33:36.4] point of view, it's quite different, but they are still the same.

[0:33:42.9] AB: That makes sense. I think that's a nice way to do it. Because you're using annotations, does that mean that you can actually cross compile so the same source can be a native binary and a jar?

[0:33:58.6] DS: Absolutely. We do support for cross compilation. Cross compilation is done through as [inaudible 0:34:05.3] project plugin. It's [inaudible 0:34:06.9] plugin that lest you cross compile against three major targets, which is JavaScript, JVM and native. These targets are basically treated as a separate sub projects of one mega project, which is called cross project.

[inaudible 0:34:21.4] point of views are kind of like separate projects with separate jars, but we try to streamline and do experience so that it truly feels more like one single project, which you really just manage through this cross project API.

[inaudible 0:34:35.4] or a cross compilation is you can't create a cross project with one or more platforms, and then when you compile and publish, you probably use one jar for every platform you want to support.

[0:34:46.5] AB: How about libraries, like the Scala standard library? I think it's kind of very important and kind of gives the language a lot of its feel. Do you have the standard libraries available natively?

[0:35:00.7] DS: Standard library story is a bit involved, but generally [inaudible 0:35:03.3] things like collections and standard types which are [inaudible 0:35:10.2] just work. The way it works is Scala library is implemented in terms of Java APIs very often. Instead of trying to rewrite the whole library and have like compatible but different library, we do a bit more involved thing, which gets us better compatibility story, is we implement subsets of JDK APIs which are used by Scala standard library and popular third party projects to be able to have the same code and both JVM and native completely unchanged.

For example, projects like uTest and FastParse to cross compile to native, they had zero changes in the source. They only had to change the build to support cross projects, and that's it.

[0:35:52.7] AB: What about the JDK? I assume that's underpinning a lot of these Scale standard libraries JDK calls.

[0:36:01.7] DS: Yeah. Basically, those are the Java libraries we care about. Typically, what it means, is we have our own [inaudible 0:36:08.9] Scala implementation of Java lang, Java util, Java IO, Java NIO and a bunch of other things which are essentially the core APIs most people rely on in open source projects and Scala library.

We try to implement those as faithfully as possible to their reference foundation on the reference JVM, but we don't look at the source of the reference at JVM, because we try to kind of stay away from the JPL code as much as we can. Essentially, Scala is these [inaudible 0:36:40.6] licensed and our implementation is [inaudible 0:36:43.4] licensed.

One of the only inspiration for some of the parts of APIs we implement, was Apache Harmony project, which is a reimplementation of Oracle APIs without GPL, but under Apache license. We sometimes use it for some cases where it's hard reverse engineer [inaudible 0:37:06.2] behavior of the JVM and we need some help there.

[0:37:09.7] AB: Interesting. I hadn't heard of that project. If you're recreating the — I'm just thinking there could be the case where an implementation detail of some aspect of the JDK actually become something that becomes dependent on, and then when you have a new native implementation and somehow that varies and things break. Have you come across any cases like these?

[0:37:35.3] DS: We already experienced some of those. Technically, every time we see some of those, it's a bug in native and we fix it as soon as we can. There are differences between off. Some of them seemingly minor, but they can still cause accidental breakage. For example, our float to a string, like Java lang float, box type to string. It's slightly different output format, which still was the same number, but sometimes more [inaudible 0:38:00.3]. One on JVM, and it has caused some open source test projects which rely on two sting output to be exactly the same as in JVM to fail.

We try to fix those as fast as possible. The first time [inaudible 0:38:13.6] it's a big hard, but our philosophy is if you can observe the difference from the reference foundation, it's a bug, and we have it [inaudible 0:38:23.1].

[SPONSOR MESSAGE]

[0:38:32.5] JM: Indeed Prime flips the typical model of job search and makes it easy to apply to multiple jobs and get multiple offers. Indeed Prime simplifies your job search and helps you land that ideal software engineering position from companies like Facebook, or Uber or Dropbox. Candidates get immediate exposure to top companies with just one simple application to Indeed Prime, and the companies on Prime's exclusive platform message the candidates with salary and equity upfront.

Indeed Prime is a 100% free for candidates. There are no strings attached. Sig up now and help support Software Engineering Daily by going to indeed.com/sedaily. That's indeed.com/sedaily if you're looking for a job and want a simpler job search experience. You can also put money in your pocket by referring your friends and colleagues. Refer a software engineer into the platform and get $200 when they get contacted by a company and $2,000 when they accept a job

through Prime. You can learn more about this at indeed.com/prime/referral. That's indeed.com/prime/referral for the Indeed referral program.

Thanks to Indeed Prime for being a new sponsor of Software Engineering Daily. If I ever leave the podcasting world and need to find a job once again, Indeed Prime will be my first stop.

[INTERVIEW CONTINUED]

[0:40:13.7] AB: That's a hard standard to hold yourselves to. To me, it almost seems like their tests shouldn't be dependent on the number of zeroes that a two string implementation does.

[0:40:24.2] DS: Yeah, I know.

[0:40:26.6] AB: I'm interested to hear if any like of the large Scala frameworks can run on native. I'm thinking like Spark or [inaudible 0:40:35.8]. I don't even know if the play framework has any large project been taken over?

[0:40:41.6] DS: As far as I know, nothing major has happened yet. Probably the biggest codebase [inaudible 0:40:47.8] has been done as part of our recent experiments. Technically, it's not hard to compile the source to NIR, like the first step. What's hard is to satisfy all of the Java dependencies, all of the Java library assumptions, which are expected by this project. For example, to run [inaudible 0:41:05.1] you need like good IO support, like from [inaudible 0:41:09.0] into complete circuit support.

Some of these parts are still working for [inaudible 0:41:14.9], circuits has been just merged [inaudible 0:41:16.2] has been just merged in the previous release and we're still working there. It's a bit early for major [inaudible 0:41:26.6] expert to just happen out of the box, but we are constantly looking at basically what's blocking people in terms of Java library coverage and in terms of API support. In fact, we're often implementing things just based on reports from people trying to port libraries.

Typically, right now, it's a smaller scale open source projects like Utest and FastPrase, but still even for those to run cross compile and test it, it often like a little bit of small differences in the library semantics are important.

[0:41:58.5] AB: You mentioned a scale of C, that the compiler has been ported over. Could you describe why and how that went?

[0:42:07.3] DS: We have this like still private, kind of mostly private experiment to port the Scale compiler to the Scala-Native. It is right now on JVM because of the startup issues. You kind of have to have SBT always as a background, because otherwise compiler is just unusably slow. It's only usable after it's warmed up, after a few compilations. If you have native dump, we have to have this problem, because a very first round is overly optimized. You can already run optimized code immediately.

What we observed in our very early experiments right now is we offer significantly faster performance on cold compilations. On simple projects like [inaudible 0:42:48.5] can be like times faster. Basically, cold builds [inaudible 0:42:53.5] on JVM can be like two to three times slower than cold built on native.

[0:42:59.0] AB: That's amazing. One of my frustrations with Scala is, yeah, the cold compilation time can be longer than any other language that I can think of. What were some of the challenges of moving it to native?

[0:43:15.9] DS: Probably the major challenge was to have enough IO. We had the long story of doing file IO and different types of file IO, because [inaudible 0:43:23.7] uses almost every single type of file [inaudible 0:43:27.5]. Don't ask me why, I don't know, but it basically uses NIO, it uses Java IO and a bunch of other things. Also things like jar and zip APIs.

Most of those have been contributed by Martin Duhem from Scala Center, and it's been extremely helpful to make this even possible, because essentially without these libraries a project depends on, it's hard to run it on native. Basically, [inaudible 0:43:54.5] probably hardest parts. We also have a [inaudible 0:43:58.4] of Scala ASM. Scala ASM is a fork of ASM library,

which is Java bytecode generation toolkit, which basically lets you [inaudible 0:44:09.9] emit Java bytecode. That's what Scala C does all the time.

We have a limited subset offset library ported to native to have enough APIs to make Scala C compile and to make class files. Otherwise, those basically were the only challenging parts. We only kind of — Library problems, we haven't actually discovered any major bugs in Scala native this way. As soon as we had enough Java libraries that run. Basically, it's a typical story of porting stuff form native.

[0:44:42.5] AB: Once those libraries are in place, then it works great. If I'm in Scala-Native and I have access to see as well as to Scala and JDK libraries, what is the string? When I create a string, is that a native string? Is that a Java string? Is that immutable?

[0:45:04.5] DS: Scala string is an instance of type Java lang string, which is immutable string [inaudible 0:45:11.3] by Scala array, which is also garbage collected, which is quite different from what C has for arrays. C has just basically a sequence of bytes in memory and [inaudible 0:45:24.1] zero. This memory can be really anywhere, because it's C and it's [inaudible 0:45:29.3]. When you call an API which expects the C string, you need to convert Scala strings to C strings.

In some cases where you know you have the same bit representation in both Scala and C side, you can share data structures. Often, you have to [inaudible 0:45:46.9] data over if they're different formats. For example, for file IO, when you read or write bytes, if you can just share memory with Scala [inaudible 0:45:57.8] arrays result copy. It's not often the case that you have to copy data over.

[0:46:04.0] AB: You can use either and you get to choose and there are some helpers for going back and forth.

[0:46:09.3] DS: Absolutely.

[0:46:09.9] AB: Yeah, I can see why that would be very useful. What hardware architectures? What platforms can Scala-Native run on?

[0:46:17.7] DS: Technically, we have a very little requirements, but right now we only test on 64-bit architectures. Our CI, all time CI is Mac and Linux 64-bit Intel. People have reported and it seems to work on 64-bit ARM unchanged also. We don't officially support ART at the moment. As in we don't have CI for it. Generally, just about any 64-bit architecture should just work out of the box.

We only had reports about ARM and Intel, but maybe more [inaudible 0:46:53.5] purpose you would work to, but we don't know for sure, because we don't have this kind of hardware. Basically, anything with 64-bit [inaudible 0:47:01.0] should just work.

[0:47:03.5] AB: I think now I kind of understand a lot of the usage around Scala-Native. What interesting projects have you seen making use of this project?

[0:47:12.4] DS: There have been a number of experiments going around. One of the more interesting ones is this experimental [inaudible 0:47:19.9] development called Dinosaur. It's actually very, very early stages, but it tries to be like native first framework, which kind of is built on simple stuff like [inaudible 0:47:32.5] and it seems like it's an interesting place to be, because up until the point we have a stable [inaudible 0:47:43.1] basically, first framework to market will be this main framework for Scala-Native probably. It seems like [inaudible 0:47:48.7] has the biggest lead to market so far. There's already probably a bit of code working and quite a bit of experiments. You can check of Richard's blog posts. I think we had some of them retweeted from Scala-Native Twitter. Basically, [inaudible 0:48:07.0] to do a native first web framework, which is pretty cool.

I've always seen people do different types of common line tools, and this is basically the area where we excel and the area where JVM is often [inaudible 0:48:20.2] and usable performance-wise.

[0:48:22.3] AB: Just because of the warm up time, yeah. If you write a command line tool, it just is slow.

[0:48:27.2] DS: Yeah.

[0:48:27.9] AB: Because of that quick startup time, I'm interested if anybody has thought of or if you think it'd be a good idea to use Scala-Native for things like Amazon Lambda, like serviceless computing.

[0:48:41.5] DS: It's probably an interesting idea. I've never seen anyone try it on. It would be interesting to see how it works out.

[0:48:48.5] AB: I think I saw some talk on your website about compiling down to iOS to make an iPhone app. Is that a real thing?

[0:48:59.2] DS: People tried to compile it into iOS and it seems to work in principle. The main challenge with iOS is interop with subjective C. Right now, we don't support objective C, it's basically are a bit in an uncomfortable place. Right now, as far as I know, nobody is active in trying that. It's possible in principle, but it's not directly on our [inaudible 0:49:22.6] list in terms of things we want to do now.

[0:49:25.5] AB: I think that you mentioned earlier that you were inspired by Swift with the LLVM intermediate language. Is that right?

[0:49:33.4] DS: Yup.

[0:49:35.1] AB: How did it inspire the implementation of Scala-Native?

[0:49:39.8] DS: Swift has called [inaudible 0:49:40.4], but the major inspiration for Scala-Native was Scala.js, because before Scala.js it was basically considered general truth is it's too hard to implement Scala outside JVM. Essentially, major inspiration for Scala is Scala.js and not Swift.

It's a very Swift influence Scala-Native is mostly in terms of compiler technology in terms of what we do under the hood. Swift has this intermediate language called SIL, which is short for Swift Intermediate Language and it's kind of like higher level LLVMIR. Basically, the area we're also aiming for was NIR, like higher level LLVMIR, like same.

The main difference between SIL and NIR is that SIL is reference content and NIR is garbage collected. Basically, this probably is the major difference between the two. Otherwise, they're trying to solve the similar problem. Those are presentation for high level optimize in a compiler or high level language and they both try to optimize parts which LLVM cannot do well, because LLVM is actually a very low level API and very low level representation, because, for example, some things are just simply gone by the time you emit LLVMIR.

One of our longstanding issues is the promise of virtual dispatch. We've already did a lot to make it pretty fast, but still an LLVM, when you compile virtual dispatch, you typically end up as closer to a function pointers. Basically, this is what you compile down to. When you at that level of abstraction, it's really hard to optimize this away. LLVM typically does very little, close to nothing to optimize virtual dispatch. This is what we do ourselves.

SIL also solves similar problem. Basically, it's a format for pre-optimization before LLVM optimization happens. You try to make LLVM job as easy as possible and to emit high quality coding.

[0:51:41.1] AB: Was there any challenges with having a language that has two paradigms, like Scala, and kind of having this compile to LLVM?

[0:51:51.7] DS: Actually, I don't think this two nature thing was a big problem. Probably the main reason is that essentially Scala compiler already does functional to object-oriented part of compilation. Essentially, all of the high level features are — All of the high level functional features are replaced by equivalent object orient features.

Typically, what you end up by the end of the Scala compiler is very object oriented code. Essentially, most of our challenges to make function code work well are the same as to make object orient code well, because at the end of the day, for example, closures are just object [inaudible 0:52:29.6]. Just the same as any other object oriented thing.

Basically, it all compiles down to the same representation where it has the same [inaudible 0:52:40.3] object oriented and functional features.

[0:52:43.2] AB: That makes sense. That kind of part is taking care of for you. What features are up and coming in Scala-Native?

[0:52:51.6] DS: Right now we are pretty much complete in terms of language support. We don't know of any major semantic difference which will be a break in changes and we would like to fix it as soon as possible. Most of the innovation right now is happening into libraries. We are slowly working towards bigger and bigger coverage of our reimplementation of Java APIs.

One of the major things which we're trying right now are [inaudible 0:53:17.2] APIs [inaudible 0:53:20.3] concurrent [inaudible 0:53:20.8] primitives and so on and so forth. Apart from that awesome networking and things like that. Basically, those are typical APIs you would need for backend micro-service kind of app. This is kind of the area which we see Scala being used more in the future.

Apart from library innovation, we do lots of work on the compiler code quality and runtime code quality. Basically, those are small iterative changes of the common patterns we see basically to improve performance to reduce overhead, to reduce footprint, to make it even more lightweight and so on and so forth. I guess that's pretty much what it is.

One of the areas we'll probably see the biggest changes, which are like non-iterative incremental slow convergence towards better performance are changes to the garbage collector. It's probably the area where we could do things [inaudible 0:54:16.5] better than what we do now.

[0:54:19.7] AB: If people would like to learn more about Scala-Native, where should they go?

[0:54:24.0] DS: Places are websites, scale-native.org and our Twitter, twitter.com/scala_native. Those are two central places for announcements, latest reviews and so on and so forth. You can also get to Gitter, as Gitter is like a nice cozy chat room. If you're just trying Scala and if something doesn't work or you have a problem, it's basically a place where you go to to ask questions. Of course, for all of the active development we use [inaudible 0:54:53.8] and GitHub issues, like WordPress and discussion on what's going on [inaudible 0:54:59.7]. Basically, if you

subscribe to Twitter and Gitter and GitHub, that's pretty much you will see everything that's going on.

[0:55:07.0] AB: I understand, since you first announced this project, you've had a lot of contributors. Is there are a lot of contributions coming in?

[0:55:15.3] DS: There's actually quite a bit of contributions. Right now we have a bit more than 60 contributors overall. It's really nice, because people often contribute sometimes small things, sometimes bigger things, but it's really, really nice to see people interested in the project and trying to help as much as they can.

[0:55:33.2] AB: Yeah, that's great. It's great to have a community involvement, that it's not just a couple of people working away on it. Thank you so much for your time, Denys. It's been great to learn about Scala-Native.

[0:55:44.9] DS: Thank you for having me.

[END OF EPISODE]

[0:55:48.3] 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. That's symphono.com/sedaily.

Thanks again to Symphono for being a sponsor of Software Engineering Daily for almost a year now. Your continued support allows us to deliver this content to the listeners on a regular basis.

[END]