EPISODE 1033

[INTRODUCTION]

[00:00:00] JM: Programming languages are dynamically types of statically typed. In a dynamically typed language, the programmer does not need to declare if a variable is an integer, a strong, or another type. In a statically typed language, the developer must declare the type of the variable upfront so that the compiler can take advantage of that information.

Dynamically typed languages give the programmer flexibility and fast iteration speed, but these languages also introduce the possibility of errors that can be avoided by performing type checking. This is one of the reasons why Typescript has risen in popularity giving developers the option to add types to their JavaScript variables.

Sorbet is a type checker for Ruby. Sorbet allows for gradual typing of Ruby programs which helps engineers avoid errors that might otherwise be caused by the dynamic type system.

Dmitry Petrashko is an engineer at Stripe who helped build Sorbet. He has significant experience in compilers, having worked on Scala before his type at Stripe. Dmitry joins the show to discuss his work on Sorbet and the motivation for adding type checking to Ruby. We are in the midst of the COVID-19 pandemic, and a group of developers has created a hackathon called COCEVID-19, which is a pandemic hackathon. The goal is to create solutions that help people manage and survive during the COVID-19 pandemic and they're using the hackathon platform that I've built called FindCollabs. If you're interested in hacking on ideas related to COVID-19, you can go to codevid19.com or you can go to findcollabs.com and enter into the hackathon there. There are projects that are looking for volunteers and also there are volunteers looking for projects.

[SPONSOR MESSAGE]

[00:02:04] JM: I've recently started working with X-Team. X-Team is a company that can help you scale your team with new engineers. X-Team has been helping me out with softwaredaily.com and they have thousands of proven developers in over 50 countries ready to

join your team and they can provide an immediate positive impact and lets you get back to focusing on what's most important, which is moving your team forward.

X-Team is able to support a wide range of needs. If you need DevOps, or mobile engineers, or backend architecture, or ecommerce, or frontend development, X-Team can help you with what you need. They've got a full-range of technologists who can help with AWS, and Go lang, and Shopify, and JavaScript, and Java. Whatever your engineering team needs to get to the points of scale that you want to get to, X-Team can help you grow your team. They offer flexible options if you're looking to grow your team efficiently, and their model allows for seamless integration with companies and teams of all sizes. Whether you're a gigantic company like Riot Games, or Coinbase, or Google, or if you're a tiny company like Software Daily. You can get help with the technologies that you need. If you're interested, you can go to x-team.com/sedaily. That's x-team.com/sedaily to learn about getting some help with your engineering projects from X-Team.

Thank you to X-Team for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:03:50] JM: Dmitry Petrashko, welcome to Software Engineering Daily.

[00:03:52] DP: Hi, Jeff. Pleasure to meet you.

[00:03:55] JM: I want to talk to you today about Sorbet, which is a system for gradual typing in Ruby. But let's first talk about the usage of Ruby at Stripe where you work. Why is Stripe mostly built around Ruby?

[00:04:09] DP: The reasons are arguably most like historical. The time when Stripe was getting started, Ruby was a language that was commonly used by startups in the area to get to the market the fastest, which was powered by the fact that Ruby is a very expressive language which allows faster creation towards a prototype. [inaudible 00:04:30] since and that we didn't really have a good reason to replace it since build a lot of tooling that makes Ruby work better for us and we went pretty heavy with it.

[00:04:42] JM: You used Ruby, but you don't use Rails. Why not?

[00:04:46] DP: That's correct. Early Stripe had envisioned some use cases that Rails didn't support well such as maintaining persistence connections to clients pretty much forever. It didn't end up being used at Stripe of today much. The stag that Stripe uses Sinatra and some costume frameworks rather than Rails.

Another benefit that we get by not using Rails is that our company values explicitness and this is mostly driven by the fact that usually the business that Stripe has the code [inaudible 00:05:17] frequently moves money and thus having explicitness into what happens there, and a good understanding is of big value Stripe.

[00:05:25] JM: The explicitness, I think that carries forth well into a subject which is related to being explicit, type checking. Can you explain what type checking means?

[00:05:36] DP: Yeah. Let's say you were to write a program. As you're writing the program, if the language is built in a way that there's some invariance that can be verified even before you go to production, it can allow you a faster iteration cycle such as, for example, if you could know that as you're using some method or some functionality, this method is expected to take a numbers in argument and return a string. So then you can then verify whether you're actually using this method the way it's supposed to be used in a way that the thing that you're passing it as an argument is actually a number and the way you're using the result is used as if it was a string.

The type checking is a process in which those kind of invariance are checked about your program, and most commonly those kinds of invariance are checked for all possible evaluations of your program so you don't even necessarily need to run the test or have production traffic to verify those variance. Does it make sense?

[00:06:38] JM: It does. Ruby is an untyped language. So when you add some type checking to untyped language, what benefits do you get?

[00:06:49] DP: Most of the benefits that you get can be separated into pieces. One of them are entirely technical and that there are some kinds of errors that can be entirely prevented such as referring to a class with a typo in the class pane. It can no longer mistype the word integer as Stripe, for example, or using the result of a method in the way that's known to be safe.

Those are technical reasons why you would like to have a type system, but there are also people reasons for why type system can be beneficial, and that now you have stronger and spelled-out intentional agreements about what does this method do. For example, you can say that this method takes some kind of argument such as it takes a user and this user is expected to be a string that represents the idea of the user object, whereas if you didn't have a type for it, the named user could have been treated two ways where one way would be the user ID. Another is the actual database user object. It allows teams and allows people to have explicit spelled-out contracts, which is of huge help when you have a big engineering team.

[00:07:55] JM: Sorbet, which we're going to get to shortly is an optional typing system or a gradual typing system. If people are using Sorbet, it does not force them to use types. Why is that? Why not force people to make their code entirely be typed?

[00:08:17] DP: That's a great question, Jeff. We had to do this from the start because we were looking to type a preexisting huge codebase that Stripe had that did not have types yet. Our type checker had to be able to work in the world where substantial pieces of the codebase and initially even the majority of the codebase is not typed. Thus, it was a necessity for adaption path that said in the today world we also believe that this is a value into that.

Sometimes some users prefer to not yet type their code. In many cases this is because they don't know yet what they want this code to do. They're super early in their prototype and they don't want the rigidness imposed by types. They want more flexibility. They want to have less boundaries so that they're easier to break because they're so far still figuring out what they're doing. Thus, currently at Stripe, depending on how mature is your project, different people would use different amount of tightness. Some of them will go to extreme tightness for areas. They're critical and are used in production. Some of them will start with early prototypes where they may or may not use Sorbet at all.

[00:09:27] JM: Can you tell me about the initial process for creating Sorbet? Was there a certain point you reached where there too many errors being thrown in the unchecked Ruby language that created the impetus for wanting to have some type checking?

[00:09:46] DP: There are multiple reasons that brought to this project being funded. The team who I'm currently a pillar tech lead for at the time had a different pillar tech lead, Paul Tarjan, a bunch of questions coming in such as our users at Stripe were asking to provide then better ability to describe the intentions of the code so that the users of the library and authors of the library can better communicate with each other on how you're expected to use a library and what's a valid use of the library.

They were mostly asking about this in terms of asking for documentation though. At the same time, we're seeing similar problems in production where some code may not be as well tested as we wanted. They're all combination of potential behaviors, like testing all the branches of a complex method could be pretty hard, in particular, correctly testing error handling. Both of those asks we believe could have been achieved by a type system.

Additionally, as we're expecting from experience of other companies and Stripe itself, our codebase to continue growing at least quadratically, we believe that recently engineers will be having hard time having an understanding of Stripe as a whole where we believe we'll need to introduce natural boundaries, natural terms for them to think in so that they can stay productive.

Not only introduce the terms, but also tools, such reasons and such terms, such as IDEs, like enabling things like auto completions. Enabling things like jump to definition. Enabling things like find all references, and thus building Sorbet was a project that was moving towards this grand vision of improving productivity at Stripe and making the Stripe use of Ruby sustainability in a humungous codebase that we've been growing towards.

[00:11:36] JM: If I think about Typescript, that's a typed dialect of JavaScript that people might be familiar with. When you compile a TypeScript file or perhaps maybe interpretation is the word you might want to use. You change a TypeScript file to a JavaScript file before it's actually ready to run. How does that compare to the model for Sorbet? Is it a different file format that gets converted into Ruby files?

[00:12:09] DP: That's a great question. In Sorbet's case, we chose a slightly different path that the one that TypeScript chose. Sorbet files are Ruby files. We did not use a different syntax. We do not use a file extension, and Sorbet files are run with a normal Ruby frontend, with a normal Ruby interpreter. We modified some behaviors and introduced some methods into super classes such as sig. The method that's used to specify the type signatures. Ruby VM is so expressive that we didn't need to build the customer on time to power things like this.

We've been able to benefit from the Ruby VM without needing to reinvent things like IDE support initially where the standard IDE support worked with Sorbet. We didn't need to reinvent the runtime and we didn't need to reinvent integration with, let's say, GitHub. All existing tools that work with normal Ruby also work with type Ruby.

[00:13:07] JM: How does Sorbet run? When I have one of these Ruby files where I've added typing to it, is my code getting transformed on the fly or do I run some command line function to do the necessary type checking? How does the Sorbet analyzer actually work?

[00:13:30] DP: Sorbet has two components. One of them does the former that you're describing. Another one does the later. To dive deeper, one of them allows you to run an initial common line command that will take all Stripe's codebase and spit out errors where you would use some – Where we believe you were using method that either aren't guaranteed to exist or for things that aren't guaranteed to exist or using the methods in a wrong way. This is what we call static type checking, and that it allows you to statically verify that the codebase doesn't have some classes or verse. If the type checker is happy with it, you have much higher guarantee that those kinds of errors will not be happening. For some of the, that's 100%. It also enables faster iteration time because this something that's integrated into the IDE and it now has sub-500 milliseconds response time in our humungous codebase.

The second component though is the runtime component where we're verifying that the environments that static type system was promised by a user actual hold in runtime. We need to have this for two reasons. First of all, the un-typed code still exists, and thus un-typed code can violate those promises and thus lead to operations that we believe shouldn't be possible in runtime. Thus, it now allows us to introduce invariance both from correctness perspective, which

then translates into availability and security perspective. Again, in a company that moves money, both of those very important.

[00:15:03] JM: Before you started working on Sorbet, there were other Ruby type checking systems. Why did you need to create a new one?

[00:15:11] DP: Before we've kicked off the project to implement our own. For around 3 or 4 months, members of the team, Paul Tarjan and Nelson Elhage have been evaluating other type systems. Notably, RDL by Jeff Foster from, at the time, University of Maryland who's now working at Tufts, and typed Ruby from [inaudible 00:15:35] who worked at GitHub.

We've evaluated how they work on the Stripe codebase, and unfortunately we learned that they will require substantial modifications in order to work well in our codebase. Most commonly the reason being the size. To the best of our knowledge, our code is one of the three biggest codebases, if not the biggest in the world. Thus, getting those projects to work fast enough in our codebase seemed like they will require substantial redesign. Thus, rather than trying to modify them, we started on our own experiment to see how far we'll be able to get in designing this from first principles. We've got pretty far under two-month period, and this was our experiment that had been declared a success. From there, we ended up implementing our own type checking.

Additionally since then, we've build a good relationship with the people who are standing behind both of these type checkers and other type checkers, notably Steep, type checking coming from Sautaro Matsumoto, who's from Japan. All of us are members of a working group on Ruby 3 types. We work together with Ruby Core, a team and [inaudible 00:16:48] Ruby to bring types into Ruby 3.

[00:16:53] JM: One of your colleagues worked on HHVM and Hack at Facebook and I believe that was a project to create types on top of PHP. How do the motivations for Stripe building Sorbet compare to the motivations that Facebook had when they were building Hack?

[00:17:15] DP: That's a great questions. Indeed, Paul Tarjan, who was pillar tech lead of the team at time and the biggest sponsor of the project was believing that majority of the problems

that our team was looking to address based on experience of our users could be held by [inaudible 00:17:32]. In retrospect, he was right. He was leaning towards this direction, because Hack was a project that address similar needs at Facebook.

That said, the Hack is built differently from Sorbet at Stripe mostly for reasons of PaaS dependency. At Facebook, Hack followed HHVM. By the time Hack was build, Facebook already had a runtime that they've build before this to address performance concerns. Hack was built after it. Whereas at Stripe, we weren't looking to address performance concerns, rather we're looking to address productivity and correctness concerns. Thus, Sorbet is much closely integrated with Ruby, and that we didn't see a value of building a runtime, because we didn't have problems that would get solved by building a runtime.

[SPONSOR MESSAGE]

[00:18:28] JM: DigitalOcean makes infrastructure simple. I continue to use DigitalOcean because of the low friction and attention to user experience. DigitalOcean has kept the experience simple and I can spin up a server in less than a minute and get high quality performance for a low price. For an application that needs to scale, DigitalOcean has CPU optimized droplets, memory optimized droplets, managed databases, managed Kubernetes and many more products. DigitalOcean has the flexibility to choose the right instance for the right workload and he could mix-and-match different configurations of CPU and RAM.

If you get stuck, DigitalOcean has thousands of high-quality tutorials, responsive Q&A forums and a customer team who treats customers respectfully. DigitalOcean lets developers focus on what they are building. Visit do.co/sedaily and receive $100 in credit over 60 days. That $100 can be put towards hosting or infrastructure and that includes managed databases, a managed Kubernetes service and more.

If you want to get started with Kubernetes, DigitalOcean is a great place to go. You can use your $100 to start building your distributed system and you can get that $100 in credit for free at do.co/sedaily.

Thank you to DigitalOcean for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:20:03] JM: Let's say I'm a developer at Stripe. I've been writing Ruby code for many years and then I get told, "Okay. We've got this Sorbet thing. Start using it." How is my experience of writing code going to change one I have Sorbet?

[00:20:22] DP: Awesome question. We see a lot of engineers who join Stripe from other companies where they wrote Ruby, and most notably, GitHub, Shopify, and we some of the techniques that they use to be using are the ones that Sorbet doesn't necessarily like and that it cannot verify that they're safe.

Most commonly, this means that people have to get to learn the way how Stripe does those things, which maybe is lightly more verbose, but then they work better without tooling. For example, it's pretty common in Ruby to meta program classes and methods into existence. Whereas at Stripe, it means that you cannot describe types for them. Thus, a lot of our tooling will not work well with them and that if one being able to, for example, find the definition of these methods or find the usages of these methods.

Thus, you get to choose. Do you want to get majority of the tooling that Stripe in existence Stripe and Stripe has built that is built on top of Sorbet, or do you want to take a shortcut in meta programing thing to existence? There are cases where meta core programming use the right approach, but with a value proposition at Stripe of all the tools increasingly we use meta programming less and less. Thus, the tools that you as a developer at Stripe will most commonly see are things like auto complete, where you start typing in methods and you see all the methods with the same name. As you're finished typing the method, it will also tell you the signature of this method where it will you how many arguments does it take and what types you're expected to pass there?

Actually, where to go through Sorbet [inaudible 00:21:55], you can see a demo that shows experience which is very similar how it work at Stripe with a big difference that Sorbet [inaudible 00:22:02] works on a single file. While at Stripe, we're working on tens of thousands of files, if not hundreds or thousands of files.

[00:22:10] JM: Tell me more about the tooling that you're able to build around a gradually typed check language that is not possible with untyped code? How much infrastructure and support can you give to developers that are working with Sorbet that they might not have had with Ruby?

[00:22:30] DP: The biggest guarantee that we can provide that's much harder to provide with Ruby, if even possible, is guarantees in terms of confidence. For example, let's say that you were looking to rename a method. If you're renaming a method that happen to have very common name, arguably, it'd be very hard in a big Python codebase or a Ruby codebase or a very big [inaudible 00:22:54] type language codebase turning this method, because from all the call sites, you'll need to figure out could it be calling the actual method that you want to rename, or does it happen to be calling a name with a similar or a method with a similar name that's defined somewhere else?

At Stripe, because we have a huge typing percentage where recently reached 90% [inaudible 00:23:14], our IDE tool can tell you exactly all the locations where the method is used in all the type code and also tell you all the locations where the method with a similar name [inaudible 00:23:28], which brings people into willing to type it even more and that they can verify whether this is the same method or not. This is example the thing that what's close to impossible at Stripe before, Sorbet and now is pretty commonly done with a tooling that we have.

[00:23:44] JM: I'd like to know how this occurs or how this is useful in practice. Maybe I think one way to exemplify it is just how different teams interact with one another and how you can provide guarantees in the inter-team communication. I know Stripe has a number of kind of big monoliths. I think there are several big monoliths. There're a lot of microservices as well, but it's sort of a set of monoliths and then a set of microservices kind of codebase. It's not like entirely monolithic or entirely these tiny services. But in any case, you have teams that are interacting with each other's services. You might have infrastructure teams that are going to make and update to something relating to GRPC or some kind of method definition where they have to go in and change the code of a bunch of other teams. But in any case, you have teams working on each other's code. So I just want to understand how type checking can help to improve communications and guarantees between teams.

[00:24:52] DP: Let me give you an illustration of a problem that used to be very common at Stripe and now rarely existed ever. There is a common class at Stripe that's very pervasive. Let's call it user. Stripe codebase happens to have a lot of local variables or method arguments that are called user. Some of them mean that you should be passing the actual database class that represents the user object from our internal others, meaning that you should be passing the user ID. But the author of the method didn't write the underscore Ide because they were trying to be sure.

Before Sorbet at Stripe, it was frequently hard to understand as a user of a method. Should I be passing the object to the argument that's called user, or the string that represents the object ID into it? Thus, there was a lot of confusion where people need to go read the code and frequently go deep into a lot of forwarders to see how the thing is used. The reverse was also true. Sometimes infrastructure teams found that the method was misused. It was very hard for them to find all the places that misused it, and they grew some methods that were actually agonistic and they can work with either user object or the user as a string. This was creating even more confusion, because then it's very hard to state in variance. It's very hard to tell whether you can build all the cases.

Today, in the world, where this method will have a signature, it will be explicit in the code that either a user ID or a user object itself and it will be checked both statically before you commit your code and in production. It will verify that this promise of there's only users IDs or only the users, the object are getting passed here, will be held true in both tests and production.

[00:26:45] JM: Makes sense. Now I'd like to talk about the actual development of Sorbet, and I think it's worth talking through a bit like what Sorbet actually is. You corrected me before the show started that this is not a compiler. When I think of a system like TypeScript, I think of a compiler. I think of a language that is built on top of JavaScript that compiles down to JavaScript. If it's not a compiler, what is it? What is Sorbet?

[00:27:22] DP: If you were to think about Sorbet, it's more like Hack, the original Hack, in a sense that its output is error messages. It runs over your codebase and it starts complaining

about your code saying that the way you're using your code makes Sorbet uncomfortable in a sense that it cannot verify that some of the usages or some expressions of your code are safe.

In some cases it will say that you're calling a method that since we have a typo, it will suggest a corrected method name. In some cases, we will tell you that you're passing the argument. But in the end, it's output is error messages rather than some kind of executable file or some kind of a different program written in a different language that it transformed it into.

That's we call the type checker other than compiler in a sense that we don't actually have the compilation steps inside it. We don't have the last steps that are necessary to implement the compiler because we didn't need to build them.

[00:28:24] JM: Got it. The code, would you call it maybe a code scanner or I guess you just call it a type checker? What are the different components of the type checking process program?

[00:28:38] DP: The tool as a whole, we call it type checker. Internally, public name is called Sorbet. Internally it's called Ruby typer, and that we try to call things what they are at Stripe rather than the code names, and Sorbet is the public name because there are more than one external type check of Ruby.

Internal structure has a bunch of phases. The very early phases of [inaudible 00:28:57] where we take a string representation of Ruby as read from disk and we convert them to a tree-like representation that's most commonly used to represent programs. It's called abstracts and extreme. These abstracts and extremes goes through a bunch of transformations. Most notably, the very first ones are syntactic transformations that transform it to a simpler language and allow us to implement a much smaller subset of Ruby that will be more uniform.

For example, Ruby has prefix and postfix if, similar prefix and postfix while and a bunch of these. We're transforming Ruby language to be simpler to reason for future passes of Sorbet so that we can handle it more uniformly and in more systematic way. Later it followed by something that we call namer that discovers all the definitions that exist in your codebase and registers them in something that we call global state. After this, it's followed by resolver that finds all

usages of those definitions, all references to classes, all references to module, all references to constants.

Finally, resolver is followed by something that we call inferencer that runs type inference on your program and figures types of every local variable, type of every expression in your program and do they work well together and start raising errors if they don't. Does it make sense?

[00:30:23] JM: It does. But what about the fact that at the end of it, the code has types in it. I mean, the types, those are not going to be proper Ruby code? Isn't that just extraneous code that you have to remove before you actually execute the Ruby code?

[00:30:43] DP: The Sorbet types are actually proper Ruby code. They're DSL written on top of Ruby where at buffer methods you say sig, and insight the [inaudible 00:30:53] release you say that this method has specific parameters and return type and the entire thing is valid Ruby. It's evaluated in runtime both in test time and in production. The knowledge that you wrote in the signature is used in runtime to wrap the methods via a wrapper that will enforce types on the way in and out so that it will check that your arguments are the things that you promised. People will use you and that your result is the thing that you promised the thing that you will return. Again, it's valid Ruby. Sorbet does not use something like non-Ruby syntax for comments for coding types. This is what allows us to validate them in production.

[00:31:39] JM: Got it. I could just run my Sorbet code as normal Ruby code without running it through the type checking system.

[00:31:48] DP: Precisely, and you'll get some of the value even without the static type checker, because you'll have the runtime enforcements.

[00:31:54] JM: Cool. Now, the process of what you discussed there, you basically treat – Look at the string representation of Ruby, build an AST and then do a lot of work on top of that AST. That sounds like a lot of work to build even just the construction of the AST part. Is there anything you can take off the shelf there? Just talking about building the abstract syntax tree for Ruby, is that all stuff you had to write from scratch?

[00:32:28] DP: Actually, no. For taking Ruby source code and parsing into AST, we reused Parser that was written by [inaudible 00:32:38] GitHub for his type Ruby codebase that itself was a conversion of the wide course Ruby parser from Ruby into C++. That said, unfortunately, it ended up being not as fast as we wanted for us. But this is something that we solved pretty easily by introducing a bunch of layers or caching where we can verify that between the prior run of Sorbet and the new run of Sorbet, the file has not changed and thus being able to reuse the very initial parsed AST from it.

[00:33:11] JM: Well, that's pretty clever. You're basically saying the developer experience, the first time I run my abstract syntax tree generation thing, it's going to be kind of slow, but in future instances, it's going to be faster because you're going to be able to cache and reuse most of that abstract syntax tree.

[00:33:29] DP: Exactly, and that's also the trick that we used first in our library. Sorbet internally has burned in definition for Ruby standard library, which you don't need to pastor it. As Sorbet starts, there is one cache that's part of the Sorbet binary itself that contains the cache representation of standard definition of like integer, string and such. Because we don't need to reparse on every start, we can start as fast as single digit milliseconds. Whereas if we were to parse them, it will take us an essential amount of time.

[00:34:04] JM: As I was going through the Sorbet work, I noticed you used a project from Google called ABSale. I hadn't seen this before. What is ABSale?

[00:34:15] DP: ABSale is the project that Google open sourced where they're sharing some of the common building blocks that Google uses for C++ and Python. We use it for a very specific class called in-line vector. With sorbet, being a type checker for a big codebase of Stripe, the biggest constraint that will define our performance properties is memory and cache locality.

Inline vector is an implementation of vector for C++ where you can ask it that if a vector is slow enough and the value is small enough, you specify it as an argument for the type rather than having the vector be allocated on Heap. It will be allocated inline in the data structure itself, and thus substantially improve cache locality.

We use this data structure a lot in Sorbet for pretty much everything that of importance, where we profile what are the common sizes for, let's say, how many arguments does your method have normally? Thus the vector, the data structures that stores your argument list will be tuned that for common arguments length, the argument will be stores inline. That's substantially improving cache locality.

That's what we originally introduced ABSale for to be able to use this data structure. Similar data structures exists in other codebases such as the Facebook's common library. [inaudible 00:35:52] also has similar one. It's a pretty common trick, but we decided at the time to use ABSale one for no particular preferential reason [inaudible 00:36:01]. We just ended up choosing that one. Since, we've introduced some other helper methods from ABBsale, but the biggest reason while we introduced it originally was the inline vector.

[00:36:11] JM: Okay. The steps that Sorbet takes, after you make the abstract syntax tree, what's the next step after that?

[00:36:22] DP: The next step is namer.

[00:36:24] JM: Namer. What does namer do?

[00:36:25] DP: It discovers all the definitions. It finds all of your classes and all the methods that you find in them.

[00:36:32] JM: What does it do with that information?

[00:36:35] DP: It just registers something that we can later find them. We don't yet know the relationship between them, but we know that they at least exist.

[00:36:43] JM: Okay. Then what's the next step after that?

[00:36:46] DP: The next is resolver, where we find all the references to those classes and methods and we establish the relationships between them. For example, at this point, we will know in a class hierarchy or which class inherits the other class, or which interfaces does

implement, or which signature does your method have? But in order to be able to do this, we need to know what is integer at all. Thus, let's say namer, will register a nation there is such a thing as integer and resolver will find all references to the word integer after register have discovered that there's such a thing as integer.

[00:37:22] JM: What comes after the resolver phase?

[00:37:24] DP: After the resolver, the main phase is the inference phase, which knowing now all the use sites and all the definitions can verify that all the actual code is correctly type checkable. It will run a type inference over your program which is [inaudible 00:37:41] dependent, and thus the majority of complexity of it is the fact that we convert your methods into dependency graphs, into the data graph, and we'll run through this graph in order to verify that however you were to task variables around, all the ways you pass into other methods or call methods on them would succeed based from the promises that you gave us from types.

[00:38:06] JM: Okay. The dependency graph, is that where you're going to start to find actual type errors in the code because the dependencies are going to be mismatched?

[00:38:19] DP: Each of those phases discover some kind of errors. For example, the resolver can find that you have a reference to something that we haven't been able to find, and it doesn't exist [inaudible 00:38:29]. But the most common errors and most interesting errors are found by the inferencer, where it can say, for example, that this method was expected to return an integer. One of the many branches forgets to return a value, and thus it returns a default value of [inaudible 00:38:45].

[00:38:46] JM: Got it. That is after you build that dependency graph between the different methods?

[00:38:53] DP: Exactly.

[00:38:54] JM: Very cool. This is all written in C++. Is that right?

[00:39:00] DP: The static components in written in C++. The runtime component is written in Ruby.

[00:39:05] JM: What's the reasoning behind that language choice?

[00:39:08] DP: That's a great question. As you know, Stripe is pretty opinionated about the language choice, and C++ is not one of the languages which is supported at Stripe. In order to use C++ for this project, we went through a process at Stripe called design review [inaudible 00:39:24] more specifically where we were presenting why do we believe this project is special compared to all other projects at Stripe. The gist of it is that from prior experiences we're building type checkers, I've build Dotty that's later to become Scala 3. From prior experience of Paul Tarjan at Facebook, the thing that defines performance of a type checker is memory locality.

If you think about this, majority of type checkers are just building a huge hash map that's representation of all of your program and they're verifying whether all the things there work together correctly when you'll be looking them up. As you pass them around, that they can – When you're calling a method foo and we want to verify if they're using foo correctly, we need to look up the method foo first. Here you have this hash map like access.

The thing that that ends up defining the type checker performance rather than being just CPU utilization is most commonly whether you need to go into your RAM to look up this definition of methods. If there's a multiple tens and dozens difference between various cache levels. If something is in your registers, access is pretty instantaneous. Again, something is in your caches, the access will be much slower. If it's near memory, the access will be pretty ridiculous slow. By using language such as C++, we get to control which things are located together in memory. We'll get to control our performance properties. Does it make sense?

[00:40:54] JM: It does. I didn't actually know that type checking infrastructure could be so resource-intensive.

[00:41:01] DP: If you think about this, majority of the type checkers are non-linear on your codebase in a sense that they need to verify some environments that can be quadratic in some

pieces. For example, it's very common when you're completing checking whether the method overrides another method correctly. That this check will need to do scan all of your super classes and see which methods are overridden.

This operation is worst case quadratic by the size of your codebase, and thus you need to find the good algorithms that will be able to support it. Some pieces of Sorbet such as counter flow dependency are potentially cubic in the worst case, and thus it's very important for us to make sure that they multiplier before the cubic function is small enough that users don't run it for the functions that they will commonly write.

I know how to write a function in Sorbet that will take a few tens of thousands of line of code that will take longer than a lifetime to type check, but then people rarely write those long functions and the easy solution is write smaller functions.

[00:42:09] JM: Wait. I'm sorry. Did you say that people actually do write these kinds of code snippets that cause Sorbet to basically time out?

[00:42:17] DP: People rarely write this emphasis themselves, but they can write a program that will generate such a snippet. Stripe increasingly uses a lot of code gen, and if you were to write a method that was generated by a computer that has very complex counter flow and the method effectively encodes a state machine, where we'll now need to consider all the states. In some cases, data explosion can be substantial.

Sorbet's current algorithm is cubic in this regards, whereas things like Hack actually had a fixed point computation there. Thus, they don't have a guarantee when they will converge, if ever. We've learned from them, but still similar to how many other type checkers are affected by this [inaudible 00:43:01].

[SPONSOR MESSAGE]

[00:43:11] JM: Gauge and Taiko are open source testing tools by ThoughtWorks to reliably test modern web applications. Gauge is a test automation tool that makes it simple and easy to express tests in the language of your users. Gauge supports specifications in markdown, and

these reusable specifications simplify code, which makes refactoring easier and less code means less time spent maintaining that code.

Taiko is a node library to automate the browser. It creates highly readable and maintainable JavaScript tests. Taiko has a simple API. It has smart selectors and implicit weights that all work together to make browser automation reliable. Together, Gauge and Taiko reduce the pain and increase the reliability of test automation.

Gauge and Taiko are free to use. You can head to gauge.org to know more. That's G-A-U-G-E.org to learn about Gauge and Taiko, the open source test automation tools from ThoughtWorks.

[INTERVIEW CONTINUED]

[00:44:27] JM: If I write some Sorbet code that has some non-typed variables, are you giving me any guarantees around the untyped variables or are those simply areas of the code where I may be liable to have problems in the code because I have not done the work to actually type that code?

[00:44:50] DP: Sorbet will work hard to try to infer the type of your variable. For example, if you were to assign something that's as known types such as [inaudible 00:45:00] let's say A equals q. We'll know from there that A is an integer. Similarly, if you were to call a method that whose type we know, from there we'll know the value that let you assign to a variable, but there are cases where we we'll know the type of the variable and things like IDE will allow you to discover this. Where if you were to hover over this variable at Stripe in the IDE, it will tell you that the variable is untyped. Similarly, some features such as auto complete will not work in this variable.

[00:45:30] JM: Tell me about the process of testing Sorbet.

[00:45:33] DP: Actually, Nelson has written an awesome blog post about this, but the gist is Sorbet has a bunch of internal representations between phases such as after the parser will have the parse tree. After namer, we'll have the this global stage, which contains list of definitions. After resolver, that two will become results.

Sorbet has a way to print all of those intermediate states, and the way we test Sorbet is by verifying on a test suite that the intermediate states have not changed or if they have changed, the code review will include reviewing the changes that happen to them to make sure that all of them were intentional and all of them are not regressions.

[00:46:22] JM: What else have you done to improve the speed of Sorbet overtime?

[00:46:26] DP: Sorbet internally has a lot of parallelism. The very early phases, the parser, the early [inaudible 00:46:35] phases are massively parallel per file and they're also cached per file. That's if you were to move by it a single file in our repo, we won't actually need to reparse the entire codebase. We will only need to reparse that file. Similarly, we'll only need to do reformation to this file.

Namer is the only phase that's fully single-threated, because we need to discover the definitions and we're [inaudible 00:47:01] global state, mutations of which if were done concurrently would be unsafe. We currently actually have a project in fly that it senses ability to parallelize this, but this will introduce a potentially more complexity into namer.

Today it's warranted, because traffic has grown substantially that this has become a problem. While in early days of Sorbet, more than three years ago, having a sequential namer was not open. Inferencer similarly has been parallel from pretty much day one, and that by the type inference runs, all the knowledge about the codebase has been discovered and is immutable. Thus, the algorithm is just as parallel sharding across whole files while keeping a single copy of the global state and only reading through it.

All of those invariance in maintained by the C++ type system where we can't verify ownership with unique pointers and C++ [inaudible 00:48:00], which is transitive, allows to verify us that things like global states, if you have a constant inference rate, cannot be mutated in a way which will internal safe.

[00:48:11] JM: Before you worked on Sorbet, you spent some time working on Scala. Can you tell me what you learned from your time working on Scala that was applicable to your work on Sorbet?

[00:48:22] DP: Before joining Stripe to work on Sorbet, I was working together with Martin Odersky on the project called Dotty that today going to be called Scala 3. Together with Martin, I wrote a Ph.D. thesis on pretty much how do you write a fast and maintainable compiler? My area of research end up being very closely related to the project of Sorbet, and thus has benefited substantially and that a lot of things that were done on Sorbet essentially inspired about the solutions and the problems that we've seen in Scala.

[00:48:59] JM: Stripe itself uses some Scala and it also uses some Go. What are the places where Stripe uses those backend languages? Languages that are not Ruby?

[00:49:10] DP: Recently, we've also seen some other languages be commonly used at Stripe, such as Python and Java, and all of those languages have their own pretty specific place at Stripe. Scala is most commonly used for big data processing, things like Hadoop and Spark. If you were to do any kind of big data computation at Stripe, the recommendation is you use Scala for it.

Go at Stripe is used for things that need to handle a lot of connections, such as we have a Veneur, which is an open source project, which is a project that we open sourced that implements over the ability system of Stripe, which forwards metrics at Stripe.

Go is really good about handling a lot of connections and things like Ruby with a global interpreter log are much worst in this regard. Python in Stripe is used for machine learning with, so far, PyTorch and Tensorflow, and Java is used for some [inaudible 00:50:04] things. All of those languages have specific area of usages. We're trying to have a very opinionated position so that we can rip the benefits of synergy by using the same language to use similar problems.

[00:50:19] JM: Stripe is not the only company that has used Sorbet. As you've communicated with other companies, how does their usage of Sorbet compare to how it's used at Stripe?

[00:50:30] DP: That's a great question. Before Sorbet was open sourced, we actually had a close beta where more than 40 companies got access to Sorbet and we open sourced it after the experience of those companies became pretty good. We're verifying that Sorbet would be used not only at Stripe, but also in other companies, and this was the condition that we put before ourselves as a precondition for open souring.

Since then, we know of hundreds of companies who have adapted Sorbet. Most notably, big players such as Shopify, Coinbase and many others. [inaudible 00:51:05] has wrote a blog post about their experience adapting Sorbet. Things that we found that are different in the way how they use Sorbet are many of them disable their runtime enforcement. Runtime enforcement has some runtime overhead. At stripe, we had a metric that controls it and if it was to be hired in 7% by [inaudible 00:51:26], and I'll get paged.

In some companies, paying cost of 7% of performance might be considered too high for the guarantees additionally provided by runtime type system given that you'll already have substantial guarantees provided statically. Another difference is many of those companies use Rails, and I want to give a callout for the Sorbet Rails project built by Chan Zuckerbert initiative that makes it much easier to adapt Sorbet in the Rail codebase.

[00:51:57] JM: All right. Well, just to close off, what aspects of Sorbet are you working now? What are your projects for the future of the project?

[00:52:06] DP: At Stripe at this point, Sorbet is considered a success. The areas where there's active work in Sorbet area are further improvements in the IDE where we want to support more features and support them faster and being able to provide faster trade iteration in our codebase, in our particular, as our codebase grows.

In the core Sorbet in which [inaudible 00:52:29] the ongoing work is about making sure it continues scaling together with our codebase where there some areas such as namer which at the time made sense to write in a single-threaded way, but now as our codebase has grown, we want to make them faster.

At this point, at Stripe, Sorbet is a success and thus the project is mostly on maintenance mode and we're working to deliver value elsewhere. Similarly, the adaption of Sorbet, they're on 90%. We're not that [inaudible 00:53:02] it anymore.

[00:53:04] JM: Actually, one more question, just because I'm curious. What have you moved on to focusing on within Stripe now that you're work on Sorbet is somewhat complete?

[00:53:13] DP: My work personally has changed substantially since then. Since then I became a pillar tech lead. So I'm helping the entire wider team of [inaudible 00:53:21] of people have alignment with a wider org, but the biggest project that I had since Sorbet is like a test execution, where we're intercepting file reads on the [inaudible 00:53:34] level in our tests to see which files can be impacted by a diff that you're sending into our CI. Thus, which tests conservatively need to be rerun. This has substantially sped up our CI time and brought us to the lowest CI time in years and saved a lot of engineering waiting time on CI and also a lot of money on just CI infrastructure.

[00:53:59] JM: Awesome. Well, Dmitry, thank you for coming on the show. It's been really great talking to you.

[00:54:03] DP: Thank you, Jeff, for hosting this podcast. It was a pleasure to talk to you. Have a great day.

[END OF INTERVIEW]

[00:54:17] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team.

These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[END]