EPISODE 508

[INTRODUCTION]

[0:00:00.4] JM: Functional programming can improve the overall design of an application architecture. Rúnar Bjarnason has been exploring how writing in a functional style increases modularity and compositionality of software for many years. He's the coauthor of Functional Programming in Scala, a book that explores the relationship between functional programming and software design.

In this interview with guest host, Adam Bell, Rúnar explains how writing in a functional style involves limiting side effects, avoiding exceptions and using higher order abstractions. Writing in this style places constraints on what a module and a software system may do, but by constraining modules in this way, the software modules themselves become endlessly composable.

I hope you enjoy this episode.

[SPONSOR MESSAGE]

[0:00:57.7] JM: Azure Container Service simplifies the deployment, management and operations of Kubernetes. Eliminate the complicated planning and deployment of fully orchestrated containerized applications with Kubernetes. You can quickly provision clusters to be up and running in no time while simplifying your monitoring and cluster management through auto upgrades and a built-in operations console. Avoid being locked into any one vendor or resource. You can continue to work with the tools that you already know, such as Helm and move applications to any Kubernetes deployment.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your applications off-line. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, all while increasing the security, reliability and availability of critical business workloads with Azure.

Check out the Azure Container Service at aka.ms/acs. That's aka.ms/acs, and the link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:02:24.2] AB: Rúnar Bjarnason is an engineer at Takt. He is the coauthor of Functional Programming in Scala. Rúnar, welcome to the show.

[0:02:34.6] RB: Thank you.

[0:02:36.0] AB: There's been a number of episodes, Software Engineering Daily, in the past about functional programming and functional programming languages. Your book is about purely functional programming and how it leads to more modular software. Before we get into specifics, what is purely functional programming?

[0:02:54.2] RB: Well, purely functional programming is programming with functions only, and so when I say functions, I mean mathematical functions. A function just takes an input and produces an output and it doesn't do anything else. Then purely functional programming is purely programming with such functions.

[0:03:12.5] AB: So there is no side effects in a mathematical function, like a mathematical function cannot write to disk for instance.

[0:03:20.3] RB: Right, exactly. In functional programming, instead of having a function write to disk, the function returns a little program that requests at the caller, instructs the caller that something might need to happen, like writing to disk, and then it's up to the caller to pass that back to the runtime environment to have that actually happen.

[0:03:38.9] AB: What problem does functional programming solve?

[0:03:42.5] RB: Well, functional programming solves the problem of modularity and compositionality. The sort of super power of functions is that they compose. If you have a function that takes some type A and produces some type B, and you can always compose that with some function that takes that type B and produces some other type C, and then you have a composite function from A to C and they will always work and since there are no side effects that never crash or go wrong.

[0:04:11.1] AB: Mm-hmm. So composition is the super power of functional programming.

[0:04:16.1] RB: Yeah.

[0:04:16.4] AB: If I write my entire program in this functional programming style, how does that affect the architecture overall of the program?

[0:04:24.9] RB: I think it affects it in a rather profound way. Your whole program will be a single expression, and then to evaluate that expression will be to run the program. So it's a fundamentally different way of instructing software.

[0:04:37.6] AB: What led you personally to this way of structuring software?

[0:04:41.1] RB: My background is in Java programming. So I did a lot of sort of enterprise Java for large systems in the past, and it always struck me that these systems were really difficult to manage, difficult to test. They had a lot of bugs and it was difficult to just achieve the kind of stability that I wanted. So that led me to investigate whether there was some way of having better static guarantees about software, and that led me to languages like Haskell, and Haskell is a purely functional programming language. From there, I started investigating functional ideas and I started importing those ideas into Java and I was a contributor to a library of functional Java for a while and then a similar one for Scala, which is called Scala Zed.

[0:05:26.9] AB: Your book, Functional programming in Scala teaches functional programming using Scala programming language. Is that required? Is Scala required to do functional programming or can another language work?

[0:05:38.2] RB: Oh, yeah, Scala is absolutely not required to do functional programming. I mean, the first words in the book are this is not a book about Scala. So like I said, we were doing functional programming back in the day in Java before it even had closures. So all you really need is some ability to [inaudible 0:05:54.4] functions, to talk about functions as your first class thing. The way that we were doing this in functional Java was just as an anonymous objects, so function is just an anonymous objects of a class that had a single method called apply, and that works perfectly well. I wouldn't say perfectly well, but it did work.

[0:06:13.2] AB: Maybe a little verbose, but gets the concept across.

[0:06:15.5] RB: Right, and it allows you to start through your programming in this way and reap the benefits such as they are of purely functional programming.

[0:06:23.5] AB: A pure function is a function in a mathematical sense and it has no side effects and you're stating a pure function is more modular and reusable than, say, an imperative procedure or a standard Java definition. Why is that?

[0:06:39.2] RB: Modularity is the property that you can take a system apart into modules and then reassemble them in ways that you didn't necessarily anticipate you're designing those modules. A function is sort of the ultimate module in a sense, because a function can basically always participate in a composition where its input type is provided and where its output type is expected, and that there's never going to be the case like when you pull a function out of the system, it's never going to have sort of like wires hanging off of it, like pulling the carburetor out of a car or something. It's always going to just have an input node and an output node and you're going to be able to sort of snap that in to wherever that makes sense.

[0:07:26.5] AB: This is what is meant by referential transparency?

[0:07:31.0] RB: This is what is meant by a modularity. Referential transparency is the property that when you evaluate the program, when you evaluate an expression, it's not going to have a side effect. That is the results of evaluating the expression is going to have the same meaning as the expression that you started with.

For instance, like 5 + 2 is referentially transparent, because it just means 7. If you replace that expression with 7, you're going to have the same program that you started with, and in functional programming, you have this property at every level.

[0:08:05.7] AB: This is often called the substitution model, right? I can take the result of my 5 + 2, which is 7, and I can replace it with 5 + 2 and get the same result.

[0:08:19.4] RB: Right, exactly, and this is what it means to execute a program in functional programming. You just keep doing this, to all of the other expressions of the program, you keep evaluating them and replace, substituting the evaluated results for the expressions. Once you have substituted for all expressions, then you have executed the program.

[0:08:38.5] AB: So I think that makes great sense in terms of math. I think it seems a little tricky when you are pulling in information from a file system. You mentioned before, the runtime system. What happens when there's io involved?

[0:08:53.8] RB: For instance, in Haskell, the main entry point into your program is going to be an expression, which is called main, and it's going to have a type which is called io. What ends up happening is that this expression is going to struck a value of this type io, so it's going to reduce to a single such value, and that value is then going to be returned to the caller, which is the runtime environment. The Haskell runtime environment is going to receive this sort of script of type io and it's going to execute that. So the io effects, like leading the files or whatever, that never happens inside of your program. They happen outside of your program and in the runtime environment. Does that make sense?

[0:09:38.5] AB: Yeah, it does make sense. I think in some ways it seems a little more complex than maybe we're used to in an imperative sense, but it actually matches the way like the computer works. Like your program isn't actually performing the io, right? Like the disk reading is done by some sort of io system within the architecture of the computer.

[0:09:56.2] RB: Yeah, exactly. I mean the way that traditional programming languages work, procedural programming, normally is that the io system or the runtime environment expects to be called back. It expects the program to [inaudible 0:10:10.5] in certain ways, to call interrupts or other things in that.

With functional programming it's simply like an inversion of control. Instead of calling the runtime environment back, we simply return a value to the runtime environment and lets it know what it needs to do.

[0:10:28.0] AB: How does mutation work, or actually why is immutability important to functional programming?

[0:10:34.6] RB: Well, immutability is important, because mutation is a side effect. It's not referentially transparent. It breaks this substitution model, because the sort of value of a particular variable will change and you mutate it, and so, yeah, it's going to be important to not mutate memory directly, or if you do so, then to do it in a very controlled way, and there are methods for using immutable memory really functionally, which we talked about in the book towards the end.

[0:11:04.0] AB: So if I think of the first program I ever wrote, it was a C program and it had like a for loop, like for I = I + 1, so I = I + 1. That's verboten.

[0:11:17.1] RB: It's not totally forbidden. It depends on the context. It depends on whether anyone can observe that happening. If someone has a reference to the value I, and then I go and — Let's say I'm using the variable I in two places in the program and then in one place I increment I by 1, then it's going to come as a surprise to the other part of the program or it's going to lose the sort of referential transparency that evaluating or substituting in like I ++, substituting the value of I ++ is not going to give you the same program as I ++. Say, I is 4, then the value of I ++ is going to be 5, and then taking I ++ and substituting 5 is not going to have the same meaning, right?

[0:12:05.3] AB: Yeah, it's breaking the substation, and in that case it's because the I is shared, because there's a shared state between these two areas of the program.

[0:12:15.1] RB: Yeah. It's usually safe to mutate local state. If you are in a function, say, when you create this local variable, I, like in a loop, like you said, you create this local variable, I, and you proceed to mutate it and you don't actually ever look at it except in this one place. Then it doesn't really matter. You can consider the for loop as just as a single referentially transparent expression. If you don't look inside of it, then you can consider it to be referentially transparent, because it will always evaluate to the same result as long as the expression that's inside of the loop doesn't have any side effects.

[0:12:52.5] AB: Yeah, because if I understand, because my for loop is inside of a function, and that function from the outside has no side effects. It doesn't so much the inside. I'm actually mutating this variable because nobody from the outside can see that.

[0:13:07.7] RB: Exactly.

[0:13:07.8] AB: Is that right?

[0:13:08.5] RB: Yeah, and in a chapter, I think it's chapter 14 of the book, we talk about a data structure that allows you to reason about this sort of formally and how much it actually share around values that compute with immutable state locally, and you get this sort of guarantee that if you are looking at a mutable variable, then it's always safe to mutate it and that you're the only one who can see it.

[0:13:30.2] AB: What data structure is that?

[0:13:32.2] RB: It's called the ST Monad.

[0:13:33.2] AB: So state transformer or —

[0:13:35.5] RB: Yeah. It's not totally clear what ST stands for. It's like state thread or state transformer. It's a mutable state Monad.

[SPONSOR MESSAGE]

[0:13:52.1] JM: Your company needs to build a new app, but you don't have the spare engineering resources. There are some technical people in your company who have time to build apps, but they're not engineers. They don't know JavaScript or iOS or android, that's where OutSystems comes in. OutSystems is a platform for building low code apps. As an enterprise grows, it needs more and more apps to support different types of customers and internal employee use cases.

Do you need to build an app for inventory management? Does your bank need a simple mobile app for mobile banking transactions? Do you need an app for visualizing your customer data? OutSystems has everything that you need to build, release and update your apps without needing an expert engineer. If you are an engineer, you will be massively productive with OutSystems.

Find out how to get started with low code apps today at outsystems.com/sedaily. There are videos showing how to use the OutSystems development platform and testimonials from enterprises like FICO, Mercedes Benz and Safeway.

I love to see new people exposed to software engineering. That's exactly what OutSystems does. OutSystems enables you to quickly build web and mobile applications whether you are an engineer or not.

Check out how to build low code apps by going to outsystems.com/sedaily. Thank you to OutSystems for being a new sponsor of Software Engineering Daily, and you're building something that's really cool and very much needed in the world.

Thank you, OutSystems.

[INTERVIEW CONTINUED]

[0:15:44.7] AB: Back to my for loop example, should I be writing for loops to loop over a list of whatever integers? Is that the functional programming style or —

[0:15:55.4] RB: It's not. A for loop like that, it creates an incoherency and that the structure of the loop, it doesn't really reflect the structure of the list. It's possible to have off by one error and other things like that. So you don't really have much of a guarantee that you're going to correctly loop over the list.

The functional way of going over a list is by using a fold. So you start with the NT list and you say, "What is going to be the value at this NT?" and then you say, "How am I going to add or how am I going to evaluate one of the elements from this list and add that to my result?" Then you recurs over the list with those two things.

[0:16:35.6] AB: You recurs over the list, but not explicitly set, right?

[0:16:39.6] RB: You can recourse explicitly, but that recursion is always going to have the same structure. So you can abstract it out into a function, and that function is usually called fold left, fold right for the lists.

[0:16:52.5] AB: I could write a recursive function wherein to operate over my list I handle the base case and then for the next element, I call the function on itself, but what you're recommending is to use fold, and fold abstracts over this concept? Is that right?

[0:17:10.2] RB: Yeah. If you write the explicit recursion over a list, you're always going to write that the same way. So you're going to repeating yourself a lot. You can actually just write recursion over a list once as a concept and call that fold, and then the only thing you have to specify are the base case and the function with which to fold.

[0:17:29.8] AB: I think fold is a great example of kind of the power of abstraction of a functional programming language. In your book you state that a fold is a polymorphic higher order function. Could you define those terms or explain that?

[0:17:46.1] RB: It's polymorphic. That just means that it will accept a list of any type, a list of integers, a list of strings or whatever, and you choose which type that is at the call site. That's what's called parametric polymorphism. And a higher order function, which is the function that accepts another function as its target. So a fold will take the base case and it will take a function that accepts the elements of the list.

[0:18:15.7] AB: Pair metric polymorphism, is that similar to polymorphism that I learned in my intro to object oriented, like a circle is a square?

[0:18:26.0] RB: It's not. It's been so long since I've thought about object oriented programming, but I don't' really know anymore what polymorphism means there, but —

[0:18:35.6] AB: So polymorphism — I think that parametric polymorphism is maybe more like generics, and that polymorphism in the traditional OO sense is like inheritance. Does that make sense?

[0:18:47.0] RB: Yeah, that makes sense. Although I think the purpose of both is sort of similar. It's going to be allowing to call a function with multiple different values and allowing — Sorry. Calling a function with multiple different types and choosing which type that is at the call stack. I guess that's sort of the purpose. With parametric polymorphism, it's usually — We can choose any type, not just like the sub-type of another class.

[0:19:14.2] AB: Map is another higher order polymorphic function. Could you describe map?

[0:19:20.1] RB: Yeah, map over a list will take the list and it will take a function that accepts an element of that list and it will turn it into a value of a different type, and that will simply apply that function to every element of the list and they'll return the result in list.

[0:19:38.1] AB: Exceptions in the Java sense are a violation of this encapsulation, a pure function throwing a null pointer exception seems to be violates the substitution rule that you were discussing. How does functional programming deal with exceptional circumstances?

[0:19:57.7] RB: Right. Exceptions absolutely do violate the [inaudible 0:20:01.4] transparency. If you throw an exception inside of a function, it's no longer a function, because it's not returning a value. The way we deal with exceptional conditions in functional programming is simply to return a different value. For instance, in Scala we use option. In Haskell it's called maybe. So you're going to have a function that takes an int and a maybe string or maybe the other way around. It makes more sense. It's going to take a string and a maybe int. In the case that the string actually can be parsed to an integer and it's going to return that integer, and if it doesn't parse to an integer, it's going to return a special value called nothing or none, which is the not the same thing as null, which we can talk about later. But it's going to be — This special case, nothing or none, is going to be encoded in the type. So that value is going to be a value of the return type of the function. It's just going to be options int, or maybe int depending on your language.

[0:21:02.3] AB: What makes it different than null?

[0:21:04.0] RB: Null is sort of a special — Like in Java, null is a special value of every type, and it's not like a part of the type. Null is not an integer, technically, but if you say null and you claim that that is of type integer, than a compiler is just going to accept that. Then when somebody receives that value, it's sort of like a landmine in their code. If they go and they de-reference that and they say, "Okay. Well, now show me — Add for to this integer. The whole program is going to blow up. But with the value or a type like maybe or option, it's really like a list that can have either no elements or one element. And so you have to just fold that list in order to get at the elements that might be inside. So it's a fundamentally different way of talking about apps and values.

[0:21:59.5] AB: So it's much more explicit than it seems, rather than any specific value can be null. You're specifying the return type here either can be an integer or it can be a none, and then the caller has to deal with that explicitly.

[0:22:16.5] RB: Yeah, and then you can do other things. Like there's another data type called either. So instead of just having like a none or nothing, like an empty value, you can say this function returns either an error or a string or an integer. You can say, "If I'm parsing my string to an integer, either it's going to return an integer or it's going to return a string telling you what went wrong or something," something like that.

Then you just have to deconstruct that either value and look at whether you have an error or an actual value, and that's just going to be a normal value. It's not going to be like a caching exception .

[0:22:53.9] AB: The either is valuable if you want to return more information about what went wrong?

[0:22:59.3] RB: Right, exactly. It's just a normal value. You can make that type sort of arbitrarily complex.

[0:23:06.3] AB: So it doesn't — We started off talking about exceptions, but it doesn't have to be exceptional. An either could be used if I wanted a function that returned a — Say it takes a number from 1 to 10 and if the number is below 5, it returns a string, and if it's above 5, it returns an integer. I don't know why that would exist, but an either could be used.

[0:23:28.8] RB: Yeah, you can totally do that.

[0:23:29.9] AB: So if we're making the exceptions less exceptional, we're making them explicit values. Does that make it more difficult for the caller of the function? They have to explicitly deal with the possible failure cases as supposed to an exception which just travels up the call stack?

[0:23:45.3] RB: It doesn't really, because of things like map. So if you don't actually care about the exception. Let's say you have an either like this and if you don't care about the error condition, then you can map over it. So you take a function that just computes with your integer that you have inside of your either and you map that over the either and that will ignore the error side, and you can just keep doing this and you never have to explicitly talk about your errors unless you actually care about them.

[0:24:13.4] AB: The higher order abstractions, like map and fold that we have let us kind of work — Let us deal with these exceptions without a ton of boiler plate, I think. Is that what you're saying?

[0:24:23.5] RB: Yeah, that's what I'm saying. It's the case that if you have an either E, A, either E or A, then you have a function from A to B. We can always compose those two things together. The function from A to B just snaps where that A is going to occur and it will turn that into an either E or B.

[0:24:43.3] AB: The exception can travel up the call site just by mapping inside of this either structure.

[0:24:49.4] RB: Yeah. It sort of travels in a different way, but it has the same effect.

[0:24:54.0] AB: You mentioned earlier functional data structures now. I don't know if we'll get into the ST Monad, but just basic data structures that are used in an imperative way, like an array, seems to not be appropriate for functional programming, like an array involves mutation. How do we deal with list in a functional manner?

[0:25:17.4] RB: Right. The traditional way dealing with list and functional programming is they used a length list. So then you'll have a base case, which is the empty list, and then you'll have a constructor which deconstructs a list out of a single element and another list, and so it's going to be a recursive data structure. Yeah, you're very rarely going to be working with arrays, but you might be working with abstractions that hide away the fact that there are arrays under the hood. That's a very common pattern.

[0:25:49.2] AB: So there's no mutation involved, but it looks like an array.

[0:25:52.9] RB: Other way around. So there might be data structures that you have that are actually mutable arrays under the hood, but since you can't see that, you can only operate on these arrays using referentially transparent functions. You're never going to see the fact that there are arrays under the hood, but that's abstracted away from you. It's just an implementation detail performance mechanism for the implementer of the data structure.

[0:26:18.0] AB: It looks like a length list and it looks referentially transparent, but under the hood, there's actually a bunch of memory and that it's being mutated. Why?

[0:26:29.5] RB: Mainly for performance reasons, like Vector and Scala. It's like this — It's a purely functional data structure that you can't mutate in place, but under the hood it's doing all kinds of tricks using regular arrays to make it fast.

[0:26:45.3] AB: Makes sense, because the performance — Let's dig in on the performance. I guess implied with that is that there's some performance implications to this functional programming style that this lack of mutation has a cost, I guess.

[0:27:01.7] RB: Well, in the case of a length list, the performance implications is just inherent to the nature of that data structure. If you want to ask for the last element of the length list, you have to have a pointer to the last element or you have to traverse from that head of the list. You have to go all the way to the end, and that will take time, which is proportional to the length of the list. So if you can have a structure that you have random access to, like an array, that's going to be much faster.

Your question is sort of in general whether there are — The question is whether there are performance implications with functional programming in general.

[0:27:38.1] AB: Yeah.

[0:27:39.0] RB: I don't think that's inherent, but I think with a lot of languages, there are implications depending on how you do things. For instance, in Java and Scala, a function is an object on the heap, and so if you use a lot of functions, you're going to generate a lot of garbage, and like garbage has to be collected.

Another thing is that a functional call is going to occupy space on Stack, and so if you have a long chain of function calls, you might run out of stack and you're going to get a stack overflow exception.

[SPONSOR MESSAGE]

[0:28:17.5] JM: If you are building a product for software engineers or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an email, jeff@softwareengineeringdaily.com if you're interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers because I talked to them all the time. I hear from CTOs, CEOs, directors of engineering who listen to the show regularly. I also hear about many newer hungry software engineers who are looking to level up quickly and prove themselves, and to find out more about sponsoring the show, you can send me an email or tell your marketing director to send me an email, jeff@softwareengineering.com.

If you're listening to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company. So send me an email at jeff@softwarengineeringdaily.com.

Thank you.

[INTERVIEW CONTINUED]

[0:29:45.1] AB: So how does that work with recursion? If we’re talking about our fold previously and it’s using some sort of recursive construct internally to map over a list, how do we deal with stacks-base there?

[0:29:57.5] RB: Well, so what Scala does, it will do a tail call elimination. So it will turn your recursion into a while loop. The compiler will basically unroll your fold into a while loop. It can’t always do this. If you are making a call to a function. So if your function is calling something other than itself, Scala can’t figure out that that’s going to be a tail call.

A tail call for those who don’t know, it’s a call out of the function, which is the last step of that function. There’s no further work to be done after that call. There’s no need to return to the body of that function. There’s no need for a call stack to — The stack frame to exist. So what a lot of languages do is that they will eliminate stack frames if they’re not necessary, but languages like Java don’t do this, and Scala does this only in very specific cases. So what you got to do is use clever tricks, like in Scala, you have to use a trampoline, which is a data structure that allows you to make recursive calls and then sort of bounce back on a trampoline. Basically, all the recursion happens in a single loop that is outside of your program.

[0:31:14.7] AB: Functional programming as we discussed here, it's a constraint on how we write software. A pure function can necessarily do less than an impure function. So why would we constraint ourselves to subset of the possible ways of writing software?

[0:31:32.3] RB: Well, because constraints liberate them.

[0:31:36.3] AB: Could you expand on that?

[0:31:37.2] RB: Yeah. So the fact that they can do less, that fact means that you can reason more about what they're going to do. The smaller the set of possibilities that the functions might potentially do, the more you're going to be able to reason about what they will actually do, and I think that's tremendously important if you're going to be building a large system. At every level, you want to be able to reason to sort of get your head around and to write software about tests and things like that, what the system is going to be doing. If you constraint that space, you're going to have a lot less work to do.

[0:32:19.4] AB: It's easier to reason about a function because what I know it does is constraint to this specific use case. Is that the idea or [inaudible 0:32:29.4]?

[0:32:29.7] RB: Yeah. If you think about a procedure that could have a side effect, versus a function. Understanding what a function does is very simple. It takes an argument and it returns a value, and you can — It's guaranteed it will always just only do this. Whereas if you have a procedure that might have a side effect, it could do literally anything. Like it could mutate some memory similar. It might throw an exception, and these are the small cases, or it could like take down your whole system and erase all your hard drives and destroy a small country. You just don't know what it's going to do, unless you read the source code and fully understood it.

[0:33:13.8] AB: You're arguing that in general, constraints are good. Is that right?

[0:33:18.9] RB: Yeah. I mean I think it depends. We should not forget that constraints constrain, but in general you want to impose constraints that sort of serve you, that constrain you to some space where you actually want it to be.

[0:33:33.4] AB: So you had mentioned in a talk relating to constraints, I think about how abstraction and precision are related. Could you expand on that?

[0:33:44.4] RB: Yeah. I say that abstraction is what makes precision possible. It's a common mistake that people make that abstraction is about being vague, that it's about sort of omitting information or that is about sort of sweeping things under the rug if you will, but it's not — Abstraction is really about going to a higher semantic level where we can talk in a language where we can be absolutely precise about the things that we want to be talking about.

A sort of trivial example of where abstraction makes precision possible is if you consider the function that just takes its argument and returns it. This is the identity function. So think of this as a function from integers to integers. If you look at the type signature of this, it's not very constraint. So it could be doing anything. It could multiplying your integer by four. It could be adding to or whatever. It could always just return to zero. You don't really know what it's going to do. It's very unconstrained.

Now, if you introduce an abstraction, if you way, "Well, this is going to be polymorphic in the type." The type is going to be A to A for any type A. Now I have abstracted alpha type and I say, "The caller is going to supply the type A and then they're going to supply a value of that type, and then I'm going to return a value of that same type. Now, there's only implementation of this function. This function has to return exactly its [inaudible 0:35:12.4], because it doesn't know about any other values of that type and it doesn't know what the type it's going to be until it sees that value.

[0:35:22.1] AB: That makes sense.

[0:35:22.9] RB: Yes, because abstraction creates this very precise specification of this function.

[0:35:29.3] AB: Because it's generic, because it's abstract over the type, we know precisely what the definition of it could be, because there's only one possible result that could be that function the identity.

[0:35:40.9] RB: Right.

[0:35:41.3] AB: That makes sense.

[0:35:42.4] RB: Think about how the fact that you're abstracting creates precision. The fact that it could be absolutely any type means that the space of implementations shrinks down to just one, because the implementation actually is now unable to look at what the value is.

[0:36:03.6] AB: That's a great example. Switching gears, your book, Functional Programming in Scala. I have the book here. I've been working on it in a while. I really love the book. The thing that has be not finishing it so far as actually the thing I love about it, which is that it's full of exercises that really kind of drill the concept into you. What brought you to that idea when writing this book?

[0:36:26.8] RB: When Paul and I were training our coworkers to do purely functional programming, we found that there were sort of a handful of people that [inaudible 0:36:36.6] and that were kind of like doing the work on their own and trying to figure things out and really making functional programming a part of their own daily work. Then there were people that would just sort of show up and watch the lecture, that they would show up every week just like everyone else, but they learned significantly less to use functional programming in their daily work.

So what we sort of concluded was that — And this was consistent with our prior experience, was that in order to really learn something, you have to make it a part of your work. You have to play around a little bit. You have to experience it for yourself, and so that led us to the idea of having a book that you don't just read. It's a book that you do, the book that you work through and gain practice with functional programming.

[0:37:25.3] AB: I think it works very well. Like myself, I found the exercises can actually be tackled with pencil and paper, but are greatly effective at drilling this concept. Is the pencil and the paper aspect of it, is that intentional or is it just the nature of learning about functional programming and it's very mechanical?

[0:37:43.9] RB: I think — Yeah, just in the nature of things. It's not really intentional, although when I was learning Haskell, I went through a book by Paul Hudak, which was amazing, and I could do most of the exercises just in a notebook with a pencil and paper, but I think it's just the nature of things. The functional programming doesn't actually require a computer, because it's pure and you can execute functions just in your head, because they're so simple. You don't need like an io subsystem. You don't need an operating system. You don't need any of these complicated stuff in order to evaluate a function. You're just doing mathematics essentially.

[0:38:20.3] AB: Mm-hmm! Because you can substitute, you can do the model of evaluation in your head.

[0:38:25.0] RB: Yeah. You can do sort of — You only have to hold one step in your head at a time. So we can just do the evaluation. You write out the next step on a piece of paper just like you're writing out a proof in mathematics.

[0:38:36.7] AB: If all new software were written using the principles of functional programming, if everybody bought your book, worked through it or learned about the style in some other way, what would the software industry look like 5, 10 years from now?

[0:38:51.4] RB: I don't know. That's hard to say. I think we would all have software — We have an easier time understanding. We'd have libraries that would be easier to pick up and maintain. We probably also have better software compatibility, because we'd be able to compose a software that wasn't necessarily to be used together, because often the problem with a particular system is that it's doing some kind of side effect that isn't appropriate in a different context, but with pure functions, you can always kind of snap it together. I think we'd have a bunch [inaudible 0:39:26.1] that are mix and match autonomy.

[0:39:28.4] AB: Mm-hmm. Better composition, for sure, because everything would be declaring its inputs and outputs.

[0:39:34.6] RB: Yeah, exactly.

[0:39:35.4] AB: I need to be conscious of your time. So thank you for coming on the show Rúnar. It's been great, and I love your book.

[0:39:43.2] RB: Thanks, man.

[END OF INTERVIEW]

[0:39:46.6] JM: GoCD is an open source continuous delivery server built by ThoughtWorks. GoCD provides continuous delivery out of the box with its built-in pipelines, advanced traceability and value stream visualization. With GoCD you can easily model, orchestrate and visualize complex workflows from end-to-end. GoCD supports modern infrastructure with elastic, on-demand agents and cloud deployments. The plugin ecosystem ensures that GoCD will work well within your own unique environment.

To learn more about GoCD, visit gocd.org/sedaily. That's gocd.org/sedaily. It's free to use and there's professional support and enterprise add-ons that are available from ThoughtWorks. You can find it at gocd.org/sedaily.

If you want to hear more about GoCD and the other projects that ThoughtWorks is working on, listen back to our old episodes with the ThoughtWorks team who have built the product. You can search for ThoughtWorks on Software Engineering Daily.

Thanks to ThoughtWorks for continuing to sponsor Software Engineering Daily and for building GoCD.

[END]