JARED: We can't put it off any longer. We have to finally talk about my actual favorite programming language.

Welcome to Dead Code. I'm Jared, and today we're going to be talking about ReScript.

Before we get into that, though, somebody signed up for the hat giveaway with a Feedbin email, and if that was you, you need to respond so we can send you your hat. We sent you an email, but no response yet as of my recording this, which, admittedly, is three weeks before it comes out, and possibly even longer before you hear it, so maybe you'll see the email in the intervening time. But I've got a box that I want to send you with a hat in it, so please check your feeds.

Today we are joined by Josh Vlk, who is our first guest whose last name doesn't contain any vowels. And, like I said, we're talking about ReScript, so let's get into it.

Josh, welcome to the podcast.

JOSH: Hi. Hello.

JARED: Could you tell our audience a little bit about who you are?

JOSH: Sure. So, my name is Josh Vlk. I am a front-end developer, mostly been focusing on React for the past...since about 2016, I believe, and now I am also a contributor to ReScript.

JARED: I think probably a good first question that at least some people who haven't heard me rant about it are going to be asking, coming from that intro, is, what's ReScript?

JOSH: Yeah. So, ReScript is a programming language that is designed for web development, focusing on being able to compile and interop with JavaScript and TypeScript. So, you're able to write code in ReScript, which is...it has a nice, light, clean syntax. It doesn't have some of the rough edges that you would find in JavaScript. There's generally only a single way to do something. And it's also a very strongly typed language, so things like you're not going to have to worry about null and undefineds, and setting up a lot of the common things that we get from ESLinting rules. It's all just part of the language.

The syntax is very, very similar to JavaScript. You're going to see curly braces, and you're going to see the same kind of structures. Like, some functions, you could write it exactly the same way in ReScript as you would in JavaScript, but that doesn't mean that all JavaScript is valid ReScript. It's not the same as TypeScript.

TypeScript is probably the language that you would...the first thing that's going to pop into people's heads to compare it to. And the difference is that TypeScript is a language that...it's not even really a language. It is, but it isn't. It's a layer of types on top of JavaScript, so it is a superset of JavaScript. So, anything that's valid JavaScript is also valid TypeScript.

And TypeScript's very ambitious goal, that they have done an excellent job on, right, is to provide types for JavaScript. But the issue that you hit there is that there are things you can do in JavaScript that are insane, right [chuckles]?

JARED: Yes.

JOSH: Anyone who's worked with the language, I don't need to explain this. You've seen there's all these...some of these are people maybe doing, like, just things that we would never actually do. But there's just little clips, and videos, and snippets, and websites around that just make you scratch your head and say, "What [laughs]?"

And JavaScript is used because it's the language of the web. And we have had to make JavaScript good. JavaScript wasn't good initially. It just kind of became the language that we all used. And then we've had to create TypeScript. We've had to create ESLint. We've had to create bundlers. You know, JavaScript existed for almost 20 years before it actually got a module system as part of the language. And we've had to add all this stuff to it.

So, now you look at a common JavaScript project, and you're going to have TypeScript config. You're going to have ESLint config. You're going to have your Prettier config, or maybe you're using Biome or some of these other modern tools. You end up having to configure all of this stuff to handle JSX. Maybe you have Babel in there. And you just have to add all of these things. And the starting point for "How do I start writing JavaScript?" is a lot.

And with ReScript, instead of trying to add types or data structures to JavaScript, it's just another language that compiles to JavaScript and has a very easy interop with JavaScript. So, you can use your existing JavaScript packages in ReScript. So, like, React is first class support for React within ReScript. The core team maintains all of the...we call it bindings. It's the way that you handle interop to JavaScript.

You can think about it as a .d.ts file in TypeScript. So, if you're trying to add types to untyped JavaScript, you can write a .d.ts file to write type definitions. You can do something very similar in ReScript; we just call it a binding. So, the ReScript team manages the React bindings, and JSX is part of the language it's built in.

So, to get started, you can just add a ReScript.json file, and ReScript comes with the compiler. So, you have all of the strong types sound types. So, you don't have the options to just turn it off, like you can in TypeScript. There's no any type. There's no ts-ignore, ts-expect-error. You can't just bail out.

So, if you're working within ReScript, and you don't have any compiler issues, you know that everything is going to be correct. So, this is known as a sound type system, which TypeScript is not. It's a strong type system, and if you lean into TypeScript and you don't use the any type, and you don't add ts-ignores, you can get to a state like that. But it requires very strict usage of

the language and very strict linting rules to get there. With ReScript, you don't need that. It's just the language, so you don't need a linter, right?

I've gone through all of the common linting rules, like, the recommended when you set up ESLint or TypeScript ESLint. And my initial stab at it was to try and figure out, if we had a linter in ReScript, this is something that we probably should have at some point, because of maybe more opinionated things, what are the rules that would be nice to have? And I went through the common rules, and what I found was that none of them really applied to ReScript. The only ones that I found were around regex, which we are looking into ways to maybe make a bit better, like treating regex like an embedded language so we can do some better analysis of it, which still wouldn't require getting a linter, right? You don't need things like "no unsafe access," because if I get something from an array, I'm just going to get back an option of that type, right?

So, there's no nulls or undefined by default in ReScript. It's just, like, option is the type that handles something that may or may not exist. So, if I say array, you know, [0], like the brackets and then a 0, I'm going to get back an option of whatever that array type was. So, I don't need that linting rule. I don't need the linting rules for...I'm trying to think now of some good examples of the common linting rules. But I went through, and none of them really applied, because the compiler just handled it for you and guided it towards you, things like forgetting to handle something inside of a switch statement.

ReScript does pattern matching, which is equivalent to, like, switch statements. You might want exhaustive pattern matching. There are linting rules for that. You don't need that in ReScript. ReScript also comes with a formatter. It's just built in. So, I don't need to install a separate formatter.

JARED: I love that.

JOSH: And it's not opinionated; zero config options for it. It's even more restrictive than Prettier, right, like, it's just nothing. You can't tweak anything for the formatter, which sometimes people say, "I wish I could do this," and we're just like, no. The team has, like, no interest in allowing any kind of tweaking to it. The way that quotes work it's just double quotes. There are no semicolons. That is the language. If you add semicolons, it would break the language. And we don't need to have some people using semicolons and some people not using semicolons, right? It's just, no, this is the way the language looks.

If I go into any ReScript codebase, it's going to behave the same, which is something you don't get with TypeScript and ESLint. Because if you've seen the full tsconfig options, the way that that code can behave can be wildly different in one codebase to another. And then you add ESLint on top of that, which E-plugins are you using, the way that you write code changes.

ReScript forces everyone into an ecosystem and an environment where the code all just looks the same and behaves the same. And this goes back to the concept that we try and just offer a single way to solve a problem, and making that way to solve the problem easy, and not difficult

to find. And then once that problem is solved, everyone does it the same way. Like, it's just...instead of having lots and lots of options to solve the problem, it's very simple. So, the brief answer, I guess, to "What is ReScript?" is that it is a [laughs]...it's a lot to explain, and I want to make sure that people who aren't familiar with it kind of get a full picture.

But the short answer is that it is a type-safe language with a strong type system that compiles to JavaScript that is designed to allow developers to do good web development and to ship good things with confidence that the code will be correct the first time. And if they need to go and change it and do refactoring, that you can do so without worrying that you've missed some edge case or have forgotten something.

And it is part of the JavaScript ecosystem. There are other languages, like, there's PureScript or Elm, or you can go even further than that into ClojureScript or ScalaJS, that compile to JavaScript, but they're not really part of the JavaScript ecosystem. They have their own build tools. They have their own bundlers, their own package managers. With ReScript, it's just npm install ReScript, or pnpm, or I have projects where I'm using Deno with ReScript.

And then you just write your ReScript. It spits out a JavaScript file, which is human-readable and optimized by the compiler to be probably faster than what you can write. I definitely have seen it spit out things where it's, like, I would not have thought to maybe write my JavaScript that way, but then I maybe benchmark it, and it's actually faster than what I would write. It does a very good job of trying to optimize your code.

And then you just run it through Vite, or Parcel, Webpack, whatever your bundler or build tool is. I've used it with Next.js. I've used it with React Router. I've done some backend stuff with Express on it, just Node. I have a project that I'm working on that's using Deno's fresh framework, which is Preact. Anywhere that you can have a .js file, you can use ReScript, which is great.

JARED: Cool. So, I guess, for any listeners that don't follow me on other socials where I talk about this, I love ReScript. It is one of my favorite languages. I wish...I sadly can't really justify putting it into projects at work just because these are...It is, unfortunately, not the technology everyone is using, and we try to do normal Rails things in our Rails apps and not ship less common tooling in our apps. But if it were up to me, I would be writing everything in ReScript.

JOSH: Same [laughs].

JARED: Yeah, yeah. I, unfortunately, have to deal with all these other developers that don't love this language or even know it as much as me. But there's a lot to, I think, to dive into about ReScript.

But I think maybe one thing we should touch on is the sort of history, because I think there's maybe some names in there that people may have heard over the years. So, maybe you could

give a rundown of where ReScript came from. And what's BuckleScript? What's Reason? Why all these names come up when I started googling things.

JOSH: [chuckles] Yeah, and this probably leads to some of the confusion, or people get things a bit mixed up, and things that aren't necessarily true about the current state of ReScript. There's a blog article that I wrote on ReScript has come a long way. And in this, I try and take some of the common misconceptions that people have, or things that were true in 2018, 2020, maybe 2021, that just aren't true anymore. And they were valid concerns, right? Things that people brought up that made ReScript harder to adopt at that time. And there are things that the team has really taken to heart and have made huge leaps in addressing.

The history of ReScript, so ReScript as a name came about in the year 2020. So, you might hear that and think, oh, it's a newer language. I guess five years isn't that new, but in terms of the lifetime of languages, that seems fairly new. But the history of it goes back much farther.

The first place where you could say ReScript as a concept was born back in about 2016, where out of Meta there was Reason. This was created and pitched by Jordan Walke, who's also the creator of React. And Reason was an alternate syntax for OCaml. So, it's meant to be...which, if you don't know what OCaml is, that's fine. It is a programming language that goes back to the '90s. That's the same type system. It's in the ML family of languages, which is not machine learning. It's Meta Language. They're very extensible. Rust also has some origins in OCamls, so it's a great starting point for a lot of these modern languages.

And he was trying to create this alternate syntax to make it easier for JavaScript developers to work with OCaml and to use React with OCaml. So, the type system, all of the nice features like pattern matching, the option type, result types, typed errors, very easily being able to create variant types, which are, like, you can think of it as, like, a tagged union inside of TypeScript, but making that, like, the default way to solve, how do I structure my data, which can integrate patterns?

And then about the same time in 2016, out of Bloomberg, there came something called BuckleScript, which was an OCaml to JavaScript compiler. The two of these got paired together into a stack that was called ReasonML. And that was kind of the first thing that people heard about, right? So, 2016, Jordan Walke gave, like, a presentation at ReasonConf, kind of introduced ReasonML, which was the Reason and BuckleScript combo.

And the issues with that was that it targeted both compiling to OCaml and compiling to JavaScript. And it was in this hybrid place where you needed the JavaScript build tools and, like, package.json, and npm install, and, you know, 2016, it probably was webpack. And then you also needed the OCaml build tools and bundlers. So, you ended up having to still use OCaml and then also use JavaScript. And everything had to be abstracted in such a way where, like, your data types, like lists, arrays, objects, whatever, had to be done in such a way where they could work in OCaml and in JavaScript, which led to everything being kind of wrapped around in these abstractions.

So, the standard library at this time was called Belt, and that allowed me to do things like create an array and map over it. But it always created this, like, runtime cost because there was always this function wrapper around, like, array.map because it really wasn't just making an array and calling .map on it, right? It had these abstractions to make it work in OCaml and in JavaScript. And this led to not a great experience for JavaScript devs or OCaml developers [laughs], right? Everyone wanted to have their cake and eat it too.

And probably one of the biggest issues was because we were having to target OCaml, there was a complete lack of async, await, and promises, like, in the language, because those don't exist in OCaml. They have different ways of handling asynchronous things. So, everything in the land of BuckleScript and Reason was callbacks. And even in 2016, we were already over callbacks. We were like, please don't send me back to this [laughs], right?

So, what happened was the project essentially split because you had two very different focuses. So, on the JavaScript side, there came ReScript, which dropped the ability to compile to OCaml. So, this was in 2020. And dropping that enabled the team to begin working on better interop directly with JavaScript. So, there was, like, a JS module in there. So, you had, like, Belt.array, and then you had JS.array because we wre trying to begin to move the standard library closer to what was in JavaScript.

Through other changes in the language that allowed us to drop some more of the OCamlisms, Belt, and JS, it started to not become exactly what we wanted. There was the best interop story. So, then you had things like JS.array2, which was the newer way of doing it [chuckles]. And now that's all deprecated, right? It's all pushed out of the side as of version 11. V12 is coming up, which completely pushes all of that out and even further pushes that down.

There's a new standard library that's just called standard library. You don't even have to really know the name of it. You don't have to think about it. You can just directly say array.map, right?

JARED: Nice.

JOSH: And that just spits out zero runtime costs; no abstractions over into the code. It's just an array with .map being called on it. And it has just an API surface that's just the JavaScript API, right? We didn't try and add fancy stuff to it. We didn't try and make it different. We didn't try and think about, how would this change if you were doing it maybe in a more functional way or more of an OCaml? It's just, we just try to take JavaScript, the APIs that exist there, and just make them work in ReScript.

And this leads to a much better user experience, right? Because if I'm trying to pick up a language, and I know how the data types work in JavaScript, I don't want to have to learn some other way of doing it. And I don't want to be presented with four different functions that essentially do the same thing, which goes against one of the core principles that we try and do with ReScript, which is there's a single way to do things, right? And in the moment that there are

four ways to do things, we've gone against that [laughs], obviously. But the reason those existed was because of that transition away from the OCamlisms and kind of that history. But now we're at a point where you don't have to worry about any of that.

You don't have to understand BuckleScript, or Reason, or OCaml or Belt, Array, Array2. None of that is applicable anymore. There's a single way to just do things, and the underlying data structures are just the same as in JavaScript, right? So, date is the same. You get all the functions that you would expect to find in a date, array, map, set, int, big int, strings, numbers. They're all there, right?

We've tried to create as much of the same API that exists in JavaScript as possible in a way that feels ergonomic and works well within ReScript that produces, like, zero runtime cost. So, it's just spitting out just the JavaScript, like, methods and functions that you would call.

JARED: Yeah, and now we all have to do the hard work of writing lots and lots of ReScript code with the new API so that Copilot will stop spitting out suggestions that contain the old APIs.

JOSH: That also [chuckles] has become, you know, a huge issue, right? Is the birth of AIs, right, and the fact that they're training on this data. And this is something that we've actually talked a lot about on the team and how to address. It's something that we're trying to push for and to try and improve.

Things like if you have the VS Code extension, it would be great if we prompted, like, hey, do you want to add, like, an agents file to this? This is something that some other tooling has started doing. Like, if you use NX, it's a monorepo management thing, it'll just pop up and say, “Do you want to, you know, configure this to work with agents,” right? “Do you want to set it up for AI?” And it'll just set up the agent's file. It sets up an MCP. That would be great to be able to kind of guide towards that experience.

I'm actually taking a pass on our website right now to remove kind of all of the legacy documentation to make it...so, if you go to the website, it's just the latest version of ReScript, and that's it. That's all you see. Because what I've found is that some of the AI will still go through and find...the fact that version 8, 9, 10, 11 all still exist on there, sometimes the AI is not really smart enough to pick the right thing, right? So, if you just have one version on there, that makes the AI better at doing that.

And then also, too, like, on the forums, we've been talking about, like, what are some good rules that you can put out? Actually, this is a great place for me to call out, if you want to talk about ReScript or learn more, go to ReScript-lang.org. There's a link to the forums. That is the primary place for talking about ReScript. There's not, like, a Discord. There's not an active Subreddit or anything like that.

The forums are where the conversations are happening. So, jump in there. We are friendly. We love to have people ask us questions about ReScript. And just, like, there's several of us that

will respond within minutes and then immediately drop everything we're doing to try and help you because we love ReScript, and we love to see people trying ReScript [laughs].

JARED: Yeah, I've found a ton of useful stuff in there. Like, a lot of problems I've run into, somebody else has hit that hiccup before, and a quick search of the forums often gets me moving. And even, like, early on when using the language when I had no idea what I was doing and I'd never written OCaml before, I'd never saw any ML-style language before, I was, like, why can't I do this thing that I can do in other languages? And, you know, finding someone who had the same sort of programming intuition I had trying to do the thing, and the kind folks there being, like, oh, that's not...you can't do it like that.

JOSH: [chuckles]

JARED: You have to do this other thing this way because that's how this language works. And it's, like, okay, okay, I can deal with that. So, you...I think the first time I remember seeing your name was last year you published the article you mentioned, "ReScript has come a long way, maybe it's time to switch from TypeScript." And I'm pretty sure I shared it. What prompted you to write that article?

JOSH: Yeah, so, you know, I have been aware and involved in ReScript since the rebranding back in 2020. I've always been looking for something better in the JavaScript ecosystem. Because, like I said earlier, like, we've had to really push to make JavaScript good, and we've kind of all just accepted that JavaScript is the runtime, but I also have to write JavaScript. And that is something that doesn't exist really anywhere else, right? Like, if I'm going just write a backend, like, just I just want a backend service, I have lots of choices that I could pick from, right? I could use Java. I could use maybe C, C#. Rust is very popular and common these days. You have choices, Python, right?

JARED: Ruby even [laughs]?

JOSH: Ruby. Ruby. Yes, Ruby is very popular. PHP. And depending on the problems that you're trying to solve and the business logic and the needs of what you're trying to do, maybe you need to focus more on concurrency. Maybe you need something that's super performant. Maybe you need type safety. Maybe you need to interop with popular libraries and frameworks. Maybe you're doing, you know, machine learning. There's all these different needs, and there are different languages that have different solutions and could be a good fit for what you're trying to do.

And then for web development and the front-end world, JavaScript. And in the year 2025, if you're not completely crazy, TypeScript [laughs], right? And that's just become the only option. And I've never been happy with that state of things. And TypeScript has also come a long way over the past 5, 10 years. But it's still writing JavaScript with a layer of types over it, and I don't think it encourages you to get into writing good code.

And especially, like, if you ask an AI to write some TypeScript, it's just going to stick as any, like, if you ask it, solve this type problem, it'll just be, like, any [laughs]. It's, like, well, technically, you did what I asked, but that's not what I wanted.

JARED: It's just going to typecast something. It'll be like, oh, this isn't the right type. I'll typecast it. It's, like, okay...[laughs]

JOSH: Yes, right, and it's not what I want. And things like being able to do...I'm a huge fan of, like, domain-driven design. There's an excellent book and talk by Scott Wlaschin called “Domain Driven Design Made Functional,” which he's using F#, but you don't have to write it in F# or understand F# to do this. F# is another ML dialect. It's referred to generally in the community as OCaml for .NET [laughs].

But, basically, it's like use your types to represent your business logic, right? It's, like, one of the common patterns that I see in TypeScript is we end up creating an object, and then we have some optional keys on it. And then as the needs of the business logic expand, we stick more optional keys on that object. So, like, a good use case that I like to raise and something that I've been doing a lot of e-commerce development for the past several years is a shopping cart, right?

You might just create the base type called it's a shopping cart. What do I have in it? I have some products. I have a promotion. Maybe I have, like, a cart ID. Then there's a promo code. You just start to expand on that address. Maybe you can stick, like, a payment method, like, token, like, they used a credit card and through this provider. And then you start to build your application around that type.

Well, I just hit the page. I don't have a cart yet. So, maybe now we just, is the cart null, right? Is the cart type null or not there? Do we add a is loading type to it? What if I try and get the cart and there's an error? Do I have an error value on the cart, or do I just check for errors as I'm accessing the cart? What if the cart exists, but there's nothing in it? Now, I just check, okay, length of items is zero. Do this thing. And do they have a promo code? Let me check.

And then we end up just, as you add more use cases, you just add more of these, like, if this key exists, do this. And then you don't really represent what does that mean? So, instead of just doing that, you can just create types that represent these things, right? Like, my type of cart is empty, error, loading. Perhaps I haven't tried to get it yet, so it's not initialized. So, we just hit the page. I haven't even begun the request. It's not even loading yet.

There's a, you know, has items. There is has items with a special promo code where we want to show a different experience to the end user, or it means something to our business, like, in an impactful way, where it's not just that you have items. You are in this specific cart state where we want to show you something different. We're hitting a different, you know, endpoint for when you hit payment, right, like, special cases. And just write types like that.

And then when I have to do things within ReScript, you can just do pattern matching. So, if I just say, hey, I'll switch on my cart, and if it's empty, render the empty component for the page. If it's an error, render the error component. Send off some error tracking metrics maybe to Sentry or New Relic. If it has items, render the has item state. And if it's, you know, this special case, render the special case component.

And the pattern matching is exhaustive. So, if I go back and the business now says, hey, not only do we have with items in the special case, we've added special case number two, right? Where the business is adding, you know, we're going to spend millions of dollars on marketing, and we're going to pitch this new product, and we need to represent this across the website in a meaningful way.

You add that new case to your default type, and because of the exhaustive pattern matching, you're going to see compiler-like warnings or errors across...just hit compile, and you're just going to see spit out in your terminal just, hey, you forgot to handle this here. And you're going to have to go back through and add that handling explicitly to all of those places. And you're not going to have to remember every single place in the website where I'm dealing with a cart, which on an e-commerce site is most of the website [laughs], right?

JARED: Yes, yeah.

JOSH: That's a lot of what you do. And then the reverse is true, too. Like, let's say the business is, like, okay, special case two is taken off, and it's great, and that's what we're going to do now. We want to get rid of special case one. I go back to the type; I get rid of it. I'm now going to get compiler errors that say, hey, this type isn't valid. And now I'm going to delete all of the special stuff and all that handling instead of just having to go through and find all of those if length of items is greater than zero...or maybe there's a...maybe I have five items in the cart. We're going to do something special. If length of items is greater than five, do this.

So, it pushes you to do things in that way. It changes how you think about writing the code, and it forces you to do it. And that's what I want out of a programming language. I want a programming language that's going to help me do that.

And you can do this in TypeScript, right? You can represent your business logic through types by using tagged unions or discriminated unions, but you have to opt into it, right? You have to really enforce it. And ReScript helps solve problems by allowing me to define the state of the application, right? Like, there's the saying, make it impossible to have unrepresentable state in your application, right? Like, just don't do that. And ReScript just makes that the default way of doing things. You just start by defining types in this way. You use pattern matching as a default, and you don't get that in TypeScript.

So, why I wrote, like, this article was to try and raise the awareness of it because I want people to use ReScript, right? I want to use ReScript because it allows me to move faster. Anything that I'm writing these days and prototyping, even if we're not directly writing, like, ReScript right now

where I work, I'm going to do the prototype or the POC or the first pass at something in ReScript, right? Because I can move quicker with that. And you can go back and refactor a lot easier. Like, that is one of the huge, huge selling points of ReScript is refactoring.

Code is not written and done. It evolves. It changes. We add new stuff. We're really bad at deleting stuff [laughs], right? We like to just add things, and we often miss things. And with something like ReScript where you get this fast feedback loop of the compiler, it makes it easy to go in and do this stuff. So, everything gets accelerated, writing it the first time, understanding what something is supposed to do. Like, if I've never seen something before and I see this pattern matching, I can just read these different things, and I get an awareness of it.

But people maybe have heard of ReScript, but they just discarded it because of these issues, right? Like, oh, it doesn't have await async. I don't understand build. I'm not going to learn OCaml. And those things just aren't true. So, I'm trying to just raise the awareness that those issues aren't there. They were valid issues, right? They're not there anymore. So, give it another shot. Like, maybe you've looked at ReScript in the past, or maybe you looked at ReasonML, and maybe you downloaded it; you tried it; and it just wasn't a good fit for you five years ago, four years ago. Try it again. Give it a shot. Learn it. Go through the documentation.

And even, I can guarantee you, even if you don't end up using ReScript or dropping everything and writing it at work, it will make you a better developer to work in a language that has less surface area and forces you to solve problems through the concept of, like, domain-driven types and pattern matching. You're going to come out on the other side with a much simpler way of doing things. You're going to shed a lot of, like, oh, I could do this in TypeScript. I could do this in TypeScript. And then when you can't, you slim down kind of what your surface area is, right?

And this is how I write TypeScript. I'm using tagged unions as much as possible. I'm using, like, TS-pattern for pattern matching. I can't remember the last time I've written a switch statement or if else, like, it's just TS-pattern. It's exhaustive. It's great. I have, like, all the ESLint rules turned up all the way into hard mode [laughs]. And you can kind of get there. And then you add in something like fp-ts, or Neverthrow, or Effect to add, like, option and result types.

Just as a little sidebar, I also have written a fork of fp-ts called fp-tsm, which is, like, a rewrite and a simpler surface area of fp-ts, which is CommonJS and has been absorbed by Effect. And for my own needs and kind of the projects that we're working on, adding Effect isn't...Effect is awesome. If you haven't checked out Effect, please check out Effect. I don't work for them, but it's awesome. Now, for us, we just needed a smaller surface area, so I have fp-tsm. It's, like, a little lightweight library that does what fp-ts does, but in a much easier to learn way that gives you option and result types.

Then you have to...you bring all that into TypeScript, and you could do that. But some of those concepts, you might not think to reach for them in TypeScript. So, learning something like ReScript where you work with these, like, variant types, which become your tagged unions in

TypeScript, and pattern matching, and working with option and result types, it's something in your brain [chuckles] will change and for the better.

JARED: Yeah, I think there's something to that. So, I'm a Rubyist. I love Ruby. And Ruby has a very large surface area. You can do a lot of weird things with it. There's a lot of corner cases, and edge cases, and weirdness, and I love it all. But because it's a language that I write on the backend, I feel like I have a lot more control over the runtime and ability to debug it and handle all that weirdness, enjoy it, use it when it's appropriate.

When we move into browser land where this is code running on other users' machines, suddenly, despite loving all the weirdness and dynamism of Ruby, I get in this different mindset where I'm, like, I can't control what's happening on their machine. I want it to be as safe as possible. And moving into the ReScript world where I have this sound type system that I can trust that, you know, barring some weird thing on my user's machine, you know, some extension messing with my JavaScript or something, that I can trust everything will work as expected. And I just love that.

And I think the, like you say, the smaller surface area of ReScript has resulted in me writing better JavaScript and better TypeScript just by training me to think about it differently, which I think is really cool.

One thing I did want to...actually, there's a couple of things I do want to make sure we touch on. One of them is the adoption story. So, if I'm someone who's building a project in JavaScript or TypeScript and I want to try a little ReScript, what's that look like?

JOSH: You can go into an existing project, or create a new project, right? Maybe you just want to try it out. There's a create ReScript app command. You can find all the installation details on the website. But let's say I have a project that's existing. It's been around for a long time; we're using TypeScript for everything, and I want to try out adding some ReScript into it, right? Just add a ReScript.json file at the top level, set up some config stuff, npm install ReScript, and then just run ReScript in watch mode or build. And it's going to just spit out a JavaScript file next to that ReScript source file. And you can also add some annotations within your ReScript code to also generate .d.ts files, so you can get TypeScript types for what I'm writing in ReScript.

And then inside of the existing JavaScript or TypeScript, I can just import what JavaScript or ReScript has built. So, if I want to just try making a React component using ReScript, I go in; I can write my React component. It's going to spit out my JS file or MJS file or CJS, if that's still your thing [laughs], and a TypeScript file. And then I can just import that and use it in my project.

And so, with TypeScript, it kind of has this, just you turn it on for your whole project. Everything's untyped. You just turn off strict mode, and then you work your way through things and add strictness across your project. With ReScript, it's going to be, like, take a file. Convert it over to ReScript, or create new things in ReScript. And then you just pull that into your existing project.

So, it's done like a...instead of just a breadth, like, everything approach, it's more of a depth-based approach.

So, I've attempted migrating a few projects to it, just pulling down something from open source. I can't remember specifically which one I did, but I just went through GitHub, and I found some React apps. And I was just, like, oh, let me just see what it's like to rewrite this in ReScript. And the starting point is, like, just take your TSX file, rename it to .res, and then just walk through the compiler [laughs] errors until you get it working, you know, just comment it out and start whacking through it, right?

First things are, like, okay, there's no const or var. Everything is just let declarations. It doesn't mean that it's mutable. Mutable is opt-in in ReScript. You have to specifically say this thing is mutable. Just let binding is how it all works. There's no function definition. It's all just let something equal whatever, and then all functions are just arrows.

So, just start by search and replace for const and var, change it to let, and then just walk through until you get it working. And then you have a single file that's now in ReScript. And you can just work your way through the project. A great place, I think, to adopt and add ReScript to any existing project is going to be state management. So, like, Redux, pull in...there's existing, like, bindings you can install on NPM for working with Redux. And then once you write a reducer with, like, pattern matching, it's like the head exploding, like, oh my god, that was so easy to do. And just get some...that's a great place where often business logic is done or representing what's possible.

Get that in ReScript, and then that just immediately spreads across your entire application, right? So, it's like, pick something small where there's a lot of complexity that you want to simplify. It could be state management; it could be a specific component. Maybe there's just a function that just is ugly and keeps breaking and causing problems. And you want some extra type safety around it or to make it easier to model out the data, like what I was talking about, like, the shopping cart, like, defining those types. Do it in ReScript and just spit out in the TypeScript.

And if you're using it in TypeScript, you can get a discriminate union, basically, just a tagged union. You can then just work with that, right? Like, I can just spit that out from the ReScript side to generate those tagged unions in a very easy way. And, you know, hopefully, then you delete all the TypeScript [laughs], and your whole project is in ReScript, and everyone's happy.

JARED: Yeah, no, I took...I have, on a few projects, ported some React components to ReScript. It was easy. I have found bugs because of it. I've also taken some fun components. I had an app that was built in Hyperapp, an esoteric JavaScript framework that I'm not sure anyone uses anymore. And sort of over a weekend, it was like, I wonder what it would take to rewrite this thing, because it wasn't very big, in ReScript? And just banged it out. And the resulting code was so much better than what was there before. It was really fun to do.

JOSH: Yeah, you end up with a lot less code. Like, you think about how many hoops you have to jump through to do things like tagged unions and defining types. And, like, there's a lot of need for explicit types within TypeScript, where ReScript, everything is just inferred. Like, you can do type annotations, but 90% of the time you don't have to. So, everything just gets reduced way, way, way down. And if you're worried about, well, how do I understand what it looks like? If you're using VS Code or any modern editor with a ReScript, like, plugin, you can turn on, like, the thing that'll just add a line above your functions with the type definitions.

So, you can just, at a glance, see what it all is, but you don't have to write it all by hand. So, it makes it a lot easier to kind of flow into that pattern. And then once you start using, like, pattern matching or defining types with, like, variant types and...everything gets smaller, like, something that took a hundred lines of code to do in TypeScript ends up being 25, 50 in ReScript. And it's just easier to do it well [chuckles].

JARED: Yeah, it's hard to really, like, communicate to people who haven't used it how much, like, just how powerful variants and pattern matching are just in terms of simplifying just the vast majority of the conditional logic that you write on a day-to-day basis. It all just gets sort of compressed down into variants and pattern matching.

And it makes it really easy to scan through this code and understand what it's doing and handling special cases, like the whole, you know, story you described around, you know, the business wants something new in a special case, and you need to figure it out.

Like, you know, I've written some fun little projects in ReScript, and it's like, oh no, I thought of a new thing. And you just add a variant to some type somewhere, and it just cascades down and tells you all the places that you need to handle that. And you're like, oh yeah, I wouldn't have even thought of that. And you just...you fix them all and it takes you, like, two seconds, because you just got a list of things you got to bang through. And it'll give you nice errors if you do things wrong, and then it's handled, and you just move on. It's quite a nice experience.

JOSH: Yeah. And I think you touched on something great, which is it's hard to understand if you haven't tried it or worked with it. And that's where I really, really want to stress, just try it, right? You may not immediately convert your entire project over to it, but try and just write a to-do app, right? Like, just take one component in your existing codebase and try and rewrite it in ReScript. And there's a good chance you're going to fall in love with it.

JARED: Yeah. And worst case scenario, there's an escape hatch because the code it spits out is pretty decent, so...

JOSH: Yeah. That's another huge goal is that the JavaScript they're emitting is not, like, some compiled minified thing. It looks like what you wrote in ReScript. The names of all the functions and variables are the same. It's human-readable. You could just delete the RES files and commit the JavaScript it spits out and be like, you know what, we tried it. My team didn't love it. But you're not stuck.

JARED: Yeah. I think probably one last feature I wanted to sort of touch on, because I just think it's really cool here, is the, like, zero overhead JavaScript bindings. I did a fair bit with the language before I got into attempting to, like, bind to JavaScript libraries and external JavaScript. What are these zero-cost bindings and how does that work?

JOSH: Yeah. So, just to define it first, so, a zero-cost binding means that I write something in ReScript that will directly use whatever that source JavaScript is without any additional, like, wrapper functions. So, the array, like, .map is a great example, and it's just, in ReScript, everything is, like, function-based, right? So, like, I'm not going to have my data and call a method on it. I'm going to have a function that piping is the easiest way to kind of take things around. So, I'll have my array, and I pipe that into array.map.

And if you're using a VS Code plugin, you can hit dot, and it will auto complete out to piping to it. So, you get very nice ergonomic experience with it. Don't think that you have to immediately be like, oh, now I got to write a pipe and remember what to pass it to. You're going to get auto completes for the functions that will accept the type of what you're passing into that pipe, which is great.

But you're going to hit places where you want to use something or a library in the JavaScript ecosystem that you might be the first person to be doing this, right [laughs]? There is an ever-growing number of community written bindings you can find on NPM. There is a place on the ReScript site where you can search for packages, like, on NPM that are tagged with ReScript. So, you can kind of see what's out there. Most of the common ones are going to exist, right? Like, React is there, Preact, Jotai, Redux, Node, Deno. Deno is, like, standard libraries. Like, if it's something where it's in the top 50, like, NPM packages or the things that everyone's using, you're probably going to find existing bindings out there, which is great. It makes it super easy.

But you might have some library that it’s just you [chuckles], right? It's like, you and 20 other people really need that particular library. So, there's a way within ReScript where you can essentially define what the types are for that JavaScript library. Usually, it's easy enough as just pulling open the type definitions for that library, or if they have great documentation, you can use that.

I find if you're pulling in some NPM library that, you know, isn't very, very commonly used, you're probably going to be better off just opening up the type definitions to try and do this. And then you can basically use, like, these...they look like decorators essentially. They're, like, attributes, but it's like the little at symbol and be like at module. You give it the name of where you're, you know, the library that you're importing from. And then instead of doing, like, let add be something, you're going to just be, like, external add and then the type definition for that.

And then at the end of that, you'll just put, like, equals and the name of the function that you're importing from that module. And if it's the default export, you just put equals default. Now within

my ReScript code, I can just call that function with those types that I've defined. So, it may seem a little bit daunting at first or may seem like, well, now I want to use this library, so I've got to write 800 bindings for something. You really don't. You just write a binding for what you need, right?

So, I use, like, date-fns. It's a very popular library, and I pulled it in for something. I didn't write type definitions for the 8,000 functions [chuckles] that are in that library, right? It's like, I needed to do a diff of, is this date in the future? You find that one function. You write a binding for it, and you're done, right?

So, don't think that you have to go through and do everything. And this is probably why some of these libraries you're not going to find stuff published on NPM because probably one of the most common practices in the ReScript community is that people just one-off write a quick binding in a file, like, a bindings folder, I think, exists in every project I've worked on.

Like, I'm currently using React Router with it for a few projects, and I didn't go through and write all of the whole API surface area of React Router into ReScript, right? It's like, okay, I need useRouter, useLocation, okay, I wrote those. What does useLocation return? An object of the shape, cool, so now I just have that. It's a pathname and state or whatever that returns. That's it, right?

There's the scripts tag and the links components that you've got to put in your root file. It's like, I think I have, like, maybe 20 lines of bindings in a file called React Router, and that covers my whole application, right? Thankfully, React Router doesn't have a huge API surface. That's one of its selling points, but I didn't just start by being like, well, I'm pulling in React Router. Let me write every single thing.

It's, oh, I need this, oh, I didn't bind it yet. Let me just write it real quick. And then what it spits out is just the import from that module, so this is where that zero cost comes in. It just pulls in that import, and it calls that function or uses that value exactly as if that's how you had written it in JavaScript or TypeScript.

JARED: Yeah, yeah, it was a little daunting the first time I tried to write my own binding, but then as soon as I got one working, I was like, oh, I could do that for anything now. And I really, really appreciate that there's no binding layer. You write your bindings, and then it's all just native calls after that. It's really cool.

JOSH: Yeah. And this is something that I think really helps with the adoption and the selling points of ReScript compared to other compiled to JavaScript languages. So, like, Elm, I think, is a great example that was very popular back in 2013, 2014, and for a while seemed like it was beating out TypeScript, right? Of course, TypeScript in 2013 wasn't that widely used, and back then it was mostly just AnyScript, because nothing was written in TypeScript [laughs]. There were no bindings, right? It was in a state of just, well, write all the types yourself. And it's more

difficult to learn how to handwrite .d.ts files than it is to write ReScript bindings, especially in, like, 2013, 2014.

So, Elm was very popular, and it had this whole pitch of, if it compiles, you won't have runtime errors, which is great and is true, right? But in order to do that, it had to treat everything in JavaScript as if it would explode. So, all of the places where you would call these JavaScript functions would get wrapped in this layer of essentially expecting it to blow up and then safely handle it and recover, and would verify that everything that you did was correct. And it was a lot more overhead.

And it ended up with, you're not...Elm has its own, like, package library, and everything...you're going to pull in something written in Elm because that's the easiest way to do it. And there's a lot of great things to do in Elm, right? But at the end of the day, Elm is just a front-end framework, and it's a language. It's a package manager. It is the framework. It's all of it.

And that's not really how we build things, right [chuckles]? We want to have options, and things change, right? So, like, Solid exists; TanStack Start exists, Next, React Router. There's new frameworks popping up all the time. And we want to evolve and change. And you can use ReScript still with that.

JARED: So, ReScript has a new version coming out, possibly even by the time this goes live, you mentioned.

JOSH: Fingers crossed it might be out by the time it goes live. If it's not, there's a release candidate out for version 12, which a good chunk of this effort has been on some of the internals, on further removing the dependencies internally on some of the OCamlisms. And there's a lot of stuff that we marked as deprecated in version 11 that now is actually gone.

And so, things like that, the zero-cost bindings to the built-in JavaScript APIs it's not in v11. It was, like, you could install it as a separate library, but, like, JS, Array, and Array2, and Belt.Array were all still there. Those are all now pushed way, way, way down, and the standard library is just built into the language now. It's just there. And just a lot of cleanup on some of the things that were marked as deprecated it's just, like, [vocalization]; it's gone. It's not even there now.

And we have improved the interop with JavaScript errors and exceptions, so that's a lot more streamlined and easier to write ReScript code that can handle JavaScript errors. There's always been a differentiating between a ReScript error and then a JavaScript exception. And that story is a lot cleaner and a lot better.

We have new ReScript tools commands that you can run that'll do things like generate a JSON file based on your types and your comments, create a documentation in JSON, which, ideally, we're getting that to a point where you can pull that into Astro or Starlight or something and spit that out. But the first step was to just create those JSON files.

And this is all dogfooded by us. That whole standard library that I talked about, it's like, we're not manually writing all the documentation for it [laughs], right? We're just running a command, and what you see on the website is correct. Those are the types. Those are the comments in the code. You'll see that documentation when you hover over it in your IDE, and that is huge, just to get to that point.

Probably the biggest feature within v12 and the thing that's the largest impact that's taken up a good chunk of the time is that there's a complete rewrite of the compiler. So, the compiler was originally written in OCaml, because that's the history of everything. The new compiler is written in Rust, which has a lot of benefits, like, internally being able to do a full rewrite because we're able to shed a lot of the BuckleScriptisms. And a lot of this is just internal and won't really impact you today, but it's going to enable the team to really extend on things in the future. But what it will immediately mean to you is that, number one, it's way faster.

JARED: Nice. I mean, it was already fast [laughs].

JOSH: It was already fast [laughs], right? I did a thing where I compiled 32,000 TypeScript files. Like, you measure it in minutes, like, at that point, whereas ReScript can just take those in, and you measure it in seconds, right? The new compiler, you measure in milliseconds. So, huge improvements in the feedback loop of that. So, take something that was already fast and makes it even faster, which is huge. We've got to stay ahead of the TypeScript Go, right [laughs]? It's like, oh, you made TypeScript faster, well, we made ReScript faster.

And with that compiler rewrite, one of the other things is that there's now built-in monorepo support, which is huge, because I think monorepos have become a lot more broadly accepted as a way to do things, especially as we have more of these frameworks that are the whole stack, right? It's like your frontend, your backend, it's all in there, but then you still want to organize things into, here's my React components; here's my ORM; here's my database code; here's stuff that's specific to the server. That all gets stitched together, like, in Next.js or React Router, maybe, but you still want some separation in there.

And now with the new version of the compiler, that's great built-in monorepo support and being able to compile the whole thing, or only specific packages in the project that are namespacing to define what package that is in the monorepo. There'll be a lot of documentation and blog posts about this once it comes out. There's already some chatter about this on the forums and stuff, but more of this will be built out.

The other thing is that we're really putting a huge emphasis on the documentation and making sure that that's very up-to-date and that all of the legacy stuff is stripped out, and that if you go look at the website, you're not going to see things like carryovers from BuckleScript or older versions that are no longer applicable. And it's just going to be like, this is ReScript.

And when I said a lot of that stuff in version 12 that are marked as deprecated is removed, and now there's even more deprecations on top of it, don't get scared by that and say, now I've got

to go in and update everything. One of the things that we're able to pull off inside of the new version is that this can be automated [chuckles].

So, you're going to run your project, and you're going to get warnings, like, hey, this array.2 is deprecated. Switch to just array map from the standard library. And it's going to say, do you want to run this command and do that? So, you can a have a thousand warnings for deprecated things, run a command, and in less than a second, it's all just going to be fixed for you.

So, when I saw the dev who worked on that show it to us, I was like, that is impressive. That is huge. And then I went in and tried it on some projects. Like, I've been working on updating the ReScript website. I'm actually switching it over to React Router and deleting a lot of legacy stuff that hasn't been used.

And it's, like, I updated, you know, I'm updating it to the latest release candidates and stuff. And then I saw those warnings like, hey, this is all deprecated. Do you want to run this command? And I had, I think, like, 80 to 100 warnings that popped up for deprecated things, and it just fixed it. I didn't even have to go through and find it all and manually update it and search and replace. It just did it, which was awesome.

So, the upgrade path to v12 is really, really easy, and you get that faster compiler, monorepo support, and all of that legacy baggage stuff is just gone. And you don't even have to think about it anymore.

JARED: So, you've been involved in ReScript for five years now. Where do you see the language going in the next few years?

JOSH: Well, adoption [chuckles], number one, right? It's great that we're all working on this, that we're making it better, and we're extending the language. With v12, it really feels like a major release that finally sheds all of the baggage. Like, everything I went through in that article, it's not even, like, you have to look at it and be like, well, that's not actually true; that exists. But you don't have to do it that way. There's a newer way of doing it. That's all just gone, right?

All the documentation's updated, all of that legacy stuff, just chuck it out of the way. Hopefully, we can get all the, you know, AI to not think [chuckles] about it. And it just simplifies everything and solidifies our goals of there's a single way to do things and to do it well.

So, now it's like, once this is out there, it's like getting as many people to try it and expect to hear about us on a lot of podcasts, see a lot of blog posts, social media chatter. We want to try and get some presentations at conferences over the next year and just really get it out there and be like, hey, ReScript is still here. It's like, maybe you heard about it, and you didn't think about it ever then, or maybe it didn't do what you needed it to do five years ago, but it's here. Try it again; adopt it.

And improving the existing bindings that are out there, like, one of the things that has kind on been on my to-do list is I have...there's a lot of those packages that are out there for bindings. It's just opening up PRs and updating them to v12. Just get them all up to date. And just spread that out there and make it so there's less confusion in what's in the ecosystem, which if you have only 12 things on GitHub or 100 things on GitHub that are using version 12 and 1,000 that are using 9, 10, 11, that makes it harder. But if you get all that updated, it makes it easier for AI.

And that's probably another huge thing that I think there's a lot of potential for with ReScript is this emergence of pair programming with your assistant, right [laughs]? It's AI. And maybe you've been hesitant about it or you don't use it. Other devs are. A lot of devs are. I'm not one of the people who says that AI will solve all of our problems and take away our jobs, but it can be helpful in getting rid of a lot of the busy work that devs face.

And if you've used AI in different languages, you may have noticed that there are some languages it does a better job of than with other languages. So, like, TypeScript, there's a lot of data for it to train on. And remember that it's not training on good TypeScript. It's training on your TypeScript and my TypeScript [chuckles], right? It's training on some random person's TypeScript on the internet who just learned it and pushed it up and did not have strict mode enabled, and didn't have ESLint TypeScript installed because they just want to make their lives difficult. That's what it's training on, just thousands of ways to solve the same problem.

You can ask it, parse this CSV file, and it's going to find...I've had it be like, okay, let me install this library, or just manually do it, or install this other library. And then it's like, well, that library's in CommonJS, and that doesn't work anymore. And there's such a huge surface area that it's harder to train on. And because there's not that built-in compiler feedback loop, it might write something that runs, but it didn't cover all the edge cases.

So, now if I'm in ReScript and I have this compiler with a strong type system, and there's a lot of context with things like variant types that make it easy for humans to understand, like you mentioned, it's easier for you to get the context of what something's doing with pattern matching, that also makes it easier for AI to handle, which is, I think, it has a lot of possibility.

And, like I said, once we get v12 out, we're going to continue to try and improve that by figuring out how to make the documentation better. Maybe the plugin will have an MCP server or set up an agents file for you with some rules to try and guide it to write more modern ReScript. But just that compiler and feedback loop makes working with these agents a lot better.

Like, I've tried to do some...I've worked on writing some code mods or, like, linting rules or stuff in TypeScript, getting the AI to try and remove some imports and update some types. And it just kept bombing on me [laughs] because it would just stick an as any on there and then call some function that didn't exist. And it was just like, what are you doing? And I kept trying to hit it, and I was banging my head against it.

And then I went and just rewrote the same code mod in Rust using just the SWC, like, AST. And yeah, its first attempt to bang its head on it, but then the compiler is just like, no, you can't do that. And it just creates this feedback loop that...AIs pay attention to compiler errors and warnings. It gives them immediate feedback that they can act on. And if it compiles, it'll probably work, right? You're still going to have to update your prompts and make sure that it works like how you expect, but you're getting rid of a whole issue of problems of it doing something poorly or incorrectly because it doesn't represent your business logic, right?

It's going to see, oh, you forgot to handle a case here, and be like, oh, I forgot to handle a case here. Let me look for where else this data switched on, right? Let me see other usages in your codebase. And it's going to do a better job. And for better or for worse, we're in an era right now where writing code with AI is huge and more and more people are doing it. And I don't know how much better the actual agents are going to get themselves.

I think where we're at right now, we're in a period where the tooling needs to catch up and improve. The actual agents might not get better or smarter, but the way that we can use them in their existing state will get better, so things like MCP servers becoming more common or VS Code plugins saying, hey, do you want to set up an agents file? And all of that will improve.

And I think to the point of the libraries that we pick will be impacted by which ones do AI use or is better at? The languages that we pick will be impacted by how is AI good...Is AI good at this language? Is the output going to be good? And using something that has a sound type system, strong types, and a compiler is going to lead to less bugs, whether it's written by a human or by an AI.

JARED: Cool. Well, where can people go to follow what you're doing online?

JOSH: Yeah, so, you can find me over on Bluesky on vlkpack, V-L-K-P-A-C-K-.com. Or you can just look for Josh Vlk, and you'll find me on there. I really don't do any other socials. I have accounts on there, but I'm not active [chuckles] on them. Bluesky is the place I am. And then also go check out ReScript-lang.org. Check out the forums. I'm on there. Send me a message on there if you want. Say hi.

And yeah, I talk a lot about ReScript. And then you'll also hear me talking a lot about Pathfinder, which is a tabletop role-playing game. And I 3D print a lot of minis, and you'll see pictures of me making LED torches [chuckles] and things like that.

So, I don't just talk about ReScript. I talk about TypeScript and software development in general. And then also you'll see lots of long threads of me painting minis. I try not to post 800 pictures at once. I'll post one and then just thread that. So, if you want it, you can find it.

JARED: Cool. Well, thanks so much for coming on the podcast.

JOSH: Yeah, thank you for having me. This was great.

JARED: Well, I think my personal focus when examining languages is not necessarily around the LLM story around them. I do think it's interesting. I've found LLMs so far kind of mediocre to bad at writing ReScript. But I do see that potential upside that with a reasonable corpus to draw on, they could get very good, both due to the nature of the type system and the compiler and its errors, but also due to the fact that the consistency in the actual source in that corpus would be really high because the language ships with a no-options formatter. And the surface area of the language is relatively small and tries to force you into the single right way of doing things.

The training set would probably be very good or would result in LLMs that would have good patterns well-reinforced very consistently. There'd be a lot less noise, which I think is an interesting idea anyway. I don't know if we'll get to the point that the ReScript out in the wild, especially the newer versions and everything are...there's just enough out there that this actually has a meaningful effect. But it certainly is an interesting idea and does make a compelling argument for switching to the language.

I personally love ReScript. Like I said in the episode, it is a fantastic language. It's just really great to use. It sort of gives me the story that I'm looking for when I'm writing JavaScript. So, I want to just echo Josh in saying that you should try it out. If you don't like it, you don't like it. It's not going to be for everyone, but it's certainly for me. And I do honestly wish that I could write all the JavaScript that I write in ReScript.

This episode has been produced and edited by Mandy Moore.

Now go delete some...