**Interviewer**
So my first question, what have been your motivations for learning Rust and choosing to use it?

**Participant**
So basically, I'm currently in an apprenticeship as a programmer in my third year. And in my first year, we mostly learned Java. And I thought it quite interesting. I liked it a lot. But after like a year, I was kind of a little bored of Java, wanted to try out something new. And I somehow stumbled upon Rust, not entirely sure where I got it from. But I heard that Rust was a cool language, so I wanted to try it out. And then my free time started learning it and spent a lot of time with Rust in my last two years.

**Interviewer**
Gotcha. Gotcha. Okay. So is there any particular set of applications, I guess, that you've focused on with Rust?

**Participant**
Yeah, most recently, like the last year, I have mostly been focused on the Rust compiler itself.

10:... Rust Project Contribut

**Interviewer**
Oh, gotcha. Okay.

**Participant**
I've contributed to the Rust compiler quite a bit, also done some reviews. And yeah, that's like my main focus. Sometimes I also write other random applications, but nothing really too big.

**Interviewer**
Gotcha. Okay.

**Participant**
Most of my Rust time is spent around the Rust project in the Rust compiler.

**Interviewer**
Gotcha. Okay. And then is there like a specific area within the Rust compiler that you tend to focus on?

**Participant**
No, not really. I'm like, I haven't really found anything specifically that interests me specifically, it's just like the general thing. I have made PRs to all kinds of regions, maybe somewhere in the parser or coach and everything in between. I haven't really found a specific area, I'm just hanging around.

**Interviewer**
Gotcha. Okay. Gotcha. Nice. So, just sort of any particular... Is there a... I guess, how do you decide a particular area to contribute to? Is it focusing on just an arbitrary issue that you see reported?

**Participant**
I just sometimes have random ideas and try them out. Sometimes I see issues and fix them, or just have an idea that comes to my mind or an idea that I see some and I get inspired from somewhere, and then I just work on it, and then something may come out of it sometimes.

**Interviewer**
Gotcha.

**Participant**
There's no real concept. I'm just like, I do whatever the fuck I want.

**Interviewer**
Nice.

**Participant**
This is just something.

**Interviewer**
So then what do you use unsafe Rust for?

**Participant**
I use unsafe Rust for mostly to create new data structures or types that are not really possible to express with safe Rust, or I've also used unsafe Rust for just for fun, for making something unsafe, learning new things about the language. It's definitely also something I've used it for, but generally when I reach for unsafe code, I do it because safe Rust can't express something I would want to express.

10:... No Other Choice

10:... No Other Choice

**Interviewer**
Gotcha. Gotcha. Okay. Is there a particular example of something that you ran into as a limitation with safe Rust that you needed to use unsafe to get around?

**Participant**
Yeah. One example, I haven't written that much unsafe code for a good reason. I've mostly just written it because I had an idea for fun. For example, one thing that I haven't quite written it myself, but I have reviewed it was rewriting the text pointer implementation inside rustc. Inside rustc there are some text pointers, an abstraction to make it more

safe. And yeah, that's basically it's unsafe because it works with pointers and it stuffs some extra data and alignment bits and then masks them out on the read because you can't do that with safe Rust. You can't just change the pointer address in weird ways because that could go very wrong if you do it wrongly. So it needs unsafe. And it's great because it allows for more efficient code because it's actually only used in one place right now, but it's basically a pointer and then it has two flags and instead of being several bytes because of the point of the pulleys, it's just a single pointer and inside the alignment bits there are the flags and yeah, that makes it more efficient and smaller.

10:4 And it's gr... Increase Performance

**Interviewer**
Gotcha.

**Participant**
So you're just adding extra metadata to track.

**Interviewer**
Is it just alignment in this case or are there other properties that you're injecting?

**Participant**
It's not the alignment that's being checked, it's the alignment that's being used. So basically the pointers in Rust, you have work losses and all other kinds of bounds and the data structure that represents these bounds, it's basically a pointer to a list of these bounds, but it also has flags. For example, whether it's currently in a const context or something like that, I think that was it. Basically you need the extra flag to have the list of bounds and also it's in a const context and you have to store the flag somewhere. So what happens is the bounds are aligned to like four or eight bytes as most types are, which means that the lower three bits of the pointer will always be zero. So you can just use one of these zeros to store the flag. So when you read the pointer, you will mask them all to zero and read it out. And if you want to get the flag, whether it's in a const context, you just read out the Bit.

10:5 Basically you nee... Contract or Invariant

**Interviewer**
Gotcha. Gotcha.

**Participant**
You pack the information together.

**Interviewer**
Okay. Gotcha.

**Participant**
I think I can find the PR.

**Interviewer**
Yeah. That would be great. Actually, if you could send that my way. I'd appreciate that.

**Participant**
I could get some more information.

**Interviewer**
Awesome. Yeah. No, that's excellent. In general, if we do get to something in this interview and there's like a specific link or a specific example that you could send, that would be amazing if you could send that in the chat.

**Participant**
Yeah.

**Interviewer**
That stuff is super helpful for this. So yeah. The next question that I have is I guess to dig just a bit deeper. In that particular case, you're using unsafe just because you're representing something at a level where you require more low-level access to the bits, in this case, to sort of that metadata. I'm not quite familiar with... I've done some work with tools that build off of the Rust compiler, like Miri, where I've been working with some of the APIs that you usually encounter, I would guess, if you're hacking the compiler. As someone who is much more familiar though with workings of the compiler than I am, how do you feel about the structure of unsafe within rustc? When you find... Is there any pattern, I guess, to how unsafe is organized or used that you've seen in your explorations with it?

**Participant**
Yeah. So generally, you have like 2,000 lines of code file with random type system structures, and then in between you have a tiny bit of sketchy unsafe. Unsafe is not very well organized. That's also part of what this PR and the follow-up made better. It improves the safety. So for example, for this tagging, the tag needs to be implemented traits to be converted to such a use size to be put into the bits, and the old version make this trait unsafe and somewhere in the big file, the tag had to be implemented for this constant thing, which required unsafe. And now this PR editor macro, it was a follow-up PR editor macro that can do this automatically, basically improving the unsafety. And yeah, generally rustc is quite an old project. It's the oldest Rust project probably. Some code dates back to like 12 years. That's quite the code base. And yeah, the unsafety is not very well organized, not as much as that like, I think I've also tried improving some of it, but generally it's not exactly awesome. It does work though. And at least some of the unsafe data structures, they pass Miri, that's something they didn't use to at some point, but I fixed some of them and then one of the structures was changed and yeah, but it's also not tested in CI. So if someone's if structure did regress and also

10:6 Y... Local, Minimal Unsafe

10:13... Macros

10:... Running tests through

some like the self-contained structure data structures, they pass Miri, some of the more involved unsafe code, those cannot be tested inside Miri because Miri cannot reasonably run the entire Rust compiler because it's very slow.

**Interviewer**
Ah, gotcha. So it's more just the performance limitation there in that case, there isn't like a, or there are additional features where it's not just performance, it's just that you run into like Miri's limitations surrounding like a syscall or something.

**Participant**
No, generally Rust doesn't use weird syscalls, just like file open and stuff like Miri supports. Although Rust C does use extra types in one place, well, there's also LLVM, the backend, which can be very tested, but you can just not run codegen, that should generally work in Miri. Rustc does use extra types, Rustc uses a bunch of unstable features, that's very cool. And extra types is one of them where basically there's a data structure that's very commonly used in the type system, it also contains unsafe and I don't think it has unit tests whether it passes Miri, although it should pass Miri because it's really simple. Basically it's a length prefix list. So it's a list like a normal slice, except instead of storing the length inside the fat pointer, it stores the length in front of the allocation. So it's basically a normal pointer and then the length and then all the elements, which is great because then the slice itself is just eight bytes instead of 16, which is nice, but also it's something that's not very well supported by the Rust language right now because the pointer is just a normal thin pointer and it needs some kind of type and you kind of have to lie about what's behind, it's not a point, it's a normal reference. And you kind of have to lie about what's behind it because you don't know how many elements statically, since it's a dynamic list. And it uses an extra type, which is basically, I guess, the proper way to do it, where it's basically saying at first there's a use size and then there's something I don't know about. But I don't think Miri supports that right now because it's a little complicated with Miri. Although maybe tree borrows allows it, the new Miri memory model, but the current tree borrows and stacked borrows, stacked borrows is a little older and it's quite strict and there's tree borrows, which is very new and more permissive, probably too permissive. I think the final model will be somewhere in the middle, but stacked borrows does not support the extra types and that would just not work, not because it's undefined behavior or wrong, just because Miri cannot support it with stacked borrows.

**Interviewer**
Gotcha. Okay. Yeah, that makes sense.

**Participant**
It's just as lacking that additional information that MIRI requires for provenance just inherently and how the data is laid out.

10:8 som... Miri is slow

10:9 And ext... Local, Minimal Unsafe

10:... Tacit Knowledge

10:11 And... Miri doesn't s...ort this

10:12 Alt... Shifting Ground

Stacked Borrows is too

77 **Interviewer**
Gotcha. I guess to go back to what you were talking about, the way you engage with unsafe within this project, you mentioned your use of a macro in improving the unsafe in that particular pull request. What do you define as improving unsafe? Let's say I have a particular chunk of unsafe that you're looking at and you're auditing it and you're thinking about, how can I make this better? How do you think about ways to take something that you feel is... I guess first, what's your definition of unsafe that is bad or somehow in need of improvement? And then what do you do to improve that unsafe in general?

78

79 **Participant**
So of course, that's the important thing that unsafe shouldn't contain undefined behavior. So if it contains undefined behavior, it's bad, but it's kind of obvious that I think the more important, not more important, but the also important thing about unsafe is it should be well encapsulated. So again, if you have a 2,000 line module that contains that with the type system and type system definitions, it should not contain unsafe because that's kind of unrelated. And if you're worried about type system, you don't want to worry about unsafe. Like the Rust language is here to protect you from that kind of thing. So I would classify bad unsafe as unsafe that's not well encapsulated. So using Rust's very cool module system as much as possible to contain the unsafe and reduce the scope of the unsafe. And that's not all there is to unsafe quality, I guess. There are also more subjective stylistic things which are hard to describe right now. Although when you see them and do kind of realize them, maybe it's also stuff like, for example, if you have some weird aliasing around, that's some weird non-standard aliasing requirements, for example, several things that point to things. And if you use references for that, it's kind of bad. Sometimes it's undefined behavior with the references, because, for example, you have aliasing with references. But also sometimes it's fine and Miri doesn't complain, but it's still a little sketchy because generally, I think unsafe should also be a little resistant to change, also comes back to the self-containers. So if I change some unsafe in one part, the other part ideally shouldn't break because it should be robust, a little self-contained. I think that's generally the most important thing, just being self-contained, being well isolated, being well encapsulated. And then if you have a small unsafe module, you can clearly document it and document why it's okay and figure out all the reasoning and everything.

10:... Preference for Safety

10:40 if you... Code Smells

10:15 I think... Documented...ract or I

Local, Minimal Unsafe

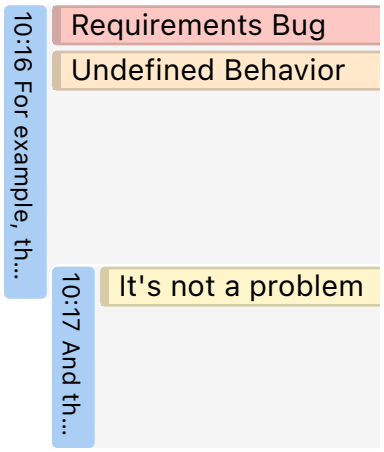
80

81 **Interviewer**
That's interesting because it's like, I could imagine having a very, very well encapsulated unsafe module, but one that is not resistant to change, at least do you feel that that's the case, that this concept of encapsulation and resistance to change, or do you feel that those are separate concepts in this case? Are there situations where you've seen unsafe that's well encapsulated, but that is somehow still too tied up or not as resistant to change as you might want it to be?

82

83 **Participant**

I use resistance of change quite loosely. It's hard to be entirely resistant because it's unsafe and it kind of relies on each other, but I think encapsulation is the more important part. And with encapsulation, you do automatically get some level of resistance to change. For example, if I have a back and I have a hashmap and the unsafe is fully encapsulated, changing the safety inside the Vec cannot break the hashmap. For example, there was a semi-recent example where Vec has this splice. I think it was a splice method and splice, you know, it's some kind of iterator and splice and Vec are kind of entangled in some way where I'm not entirely familiar with the code, but I skimmed over it and splice kind of accesses vec's internals in kind of sketchy looking ways. And at some point, the implementation of something deep inside vec's iterator was changed to use something else and it broke splice. And there's a cron job every day where Miri checks the standard library and that cron job went off and said, hey, there's something wrong. In this case, it wasn't too bad because LLVM didn't actually exploit the undefined behavior and it was quickly fixed. But basically, yeah, it's like an example where some weird unsafe that was not very well encapsulated, the entire Vec thing is complicated. I'm not sure whether it can be reasonably encapsulated, but it wasn't. And that caused issues. I'm not sure, but I can find the issue somewhere. Wasn't too long ago.




**Interviewer**
And you mentioned that in this particular case, the error was identified, but it was also identified that what was going on with this interaction wouldn't have necessarily caused an error in production because the LLVM optimizer wasn't using this undefined behavior to do something potentially bad. Is that audited just by looking at the bytecode that's generated manually? Like when you see an error like that, what's the process for determining the severity in terms of what LLVM is doing?

**Participant**
So one important thing about it is it's not exploited today. So in the future, I could already exploit it. But the nice thing with the standard library is it's directly coupled to the compiler. So we know that the compiler doesn't exploit it. So the standard library relying on it is it's still bad because you forget at some point that something relies on it. So standard library could rely on all of that stuff. It doesn't generally because it's hard to maintain. But in this case, we basically knew because just the method has a fairly strict safety requirement, but I think it just doesn't exist in LLVM. It was like, I think it was the sub pointer method, some pointer arithmetic things, which don't exist. I can't find it.

**Interviewer**
No worries. I think I've heard the... It might have been in a previous interview, someone else brought up that particular issue with Splice and Vec. So no worries if you can't find it, because I might have received a link from someone else at some point.



**Participant**

I found the link with fix.

**Interviewer**
Oh, perfect. Great.

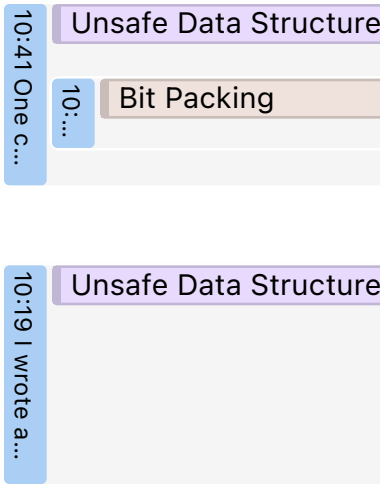
**Participant**
Going to pointer arithmetic and the allocated memory.

**Interviewer**
Gotcha.

**Participant**
And then... Let me double check. Yeah, it was Vec Splice and Vec Drain. Yeah, Vec Splice and Vec Drain have some interesting entangled interactions. And Vec Drain loses a slice iterator and splice accesses that slice iterator in weird ways, and the slice iterator was changed. And yeah.

**Interviewer**
Gotcha. And I forget, there is a different... I remember you mentioned a different application of Rust and potentially unsafe that was outside contributions to the compiler. Is that correct? I didn't write that one down. I can't quite remember which...

**Participant**
Sometimes just write weird unsafe code for fun, mostly, because I'm interested in it. One cool example, for example, is I wrote the Vec with a T, a normal Vec, basically, except for Vec, the T has to be sized. The value has to have a fixed size. But I wrote one where it doesn't have to have a fixed size. It uses the unstable pointer metadata API, and it's quite funny. And yeah, I had a lot of fun writing it. And I also wrote a pointer, basically a library, if you have a pointer or some other data. So basically, a enum of something that's a pointer and something that's normal data. And the normal data is smaller than the use size. I wrote a small library to abstract over storing those inside single users and basically manually doing the enum and tagging and checking. Except the library kind of helps you making sure that you use the strict provenance APIs, which are just generally nicer, better, and not trivial to use. That's another another crate I made. It's not widely used. And I just made it for fun.




**Interviewer**
Gotcha, gotcha. So the pattern that I'm seeing here is both with this crate or the with the two crates as well as just how you're describing your contributions to rustc. It seems like you're reasoning about some of these lower level abstractions and at a byte level in terms of how provenance information and other metadata is passed around. So you're working in the unsafe world that is supporting these safe abstractions that rustc uses. Would that be a correct way of framing your development, like your specialization, I guess?

106

107 **Participant**
Well, I wouldn't necessarily call it specialization because one of the things I do is usually I write more normal Rust code. But yeah, when I write unsafe, it's usually that kind of thing. It's just things that are fundamentally impossible with safe Rust. Like with junk, it kind of splits the metadata at the pointer in weird ways, which just cannot be expressed to see safe Rust because the borrow check has no idea what you're doing.

10:42 But ye...
No Other Choice

108

109 **Interviewer**
And you mentioned with the... I'm curious about your perspective on Miri. You mentioned that stacked borrows is too strict and you gave the example of extern types as being one thing that it can't support. And then with tree borrows, you were suggesting that that model actually might be too lax for some particular reason. I can't remember if you were specific about it. Could you describe where you see that middle ground being between the two models, if you have a particular opinion on that?

110

111 **Participant**
I don't have too many opinions on it. I was mostly relaying the opinion of other people who have thought about it more than I. Yeah, it's too strict because it can't support extern types, which is just something that is important. More generally, that's the problem. If you have a reference to something, then stacked borrows only gives you permission to read as much as the type of the reference covers. So say I had a struct with two fields, like repr(C), laid after each other. And then I took a reference just to the first field. And then using unsafe code, also access the second field that would not be allowed on stack borrows, but it's generally regarded as something that should be allowed. There's an issue on the unsafe code guidelines. It's also called the ref header problem because basically you have a reference to a header. And yeah, there's a lot of discussion here.

10:43 Yeah,...
Stacked Borrows is too
10:44 So say I had...
Shifting Ground

112

113 **Interviewer**
Gotcha. Awesome.

114

115 **Participant**
Yeah, which is something that stack borrows just can't support. So generally agreed upon that stack borrows is too strict. With tree borrows, it's a lot less strict and it does sort of the header problem. But some people that have thought a little more about possible optimizations because it's opposite the memory model, ideally, for unsafe code writers, which just allow everything. Like have aliasing mutable references, transmute everything. It's just bytes. But in the end, the reason we have undefined behavior at all is to support compiler optimizations and Rust is blazingly fast and we want it to stay that way. So yeah, it's always a balance between allowing unsafe code to be written and allowing more optimizations. And in some way it's also optimizations versus optimizations because you allow optimizations, allowing more optimizations that unsafe code writers can do allows less optimizations

the compiler can do. And all the way around, if you allow less compiler optimizations, they may be able to do more optimizations and it's kind of a trade off. And yeah, I've heard some people say that, yeah, tree borrows allow some things that should probably not be allowed. I think [name] was one of the leads of the new operational semantics team, which is also a great step towards better unsafe, the team that will finally answer these questions fully about what kind of things should be allowed and what things shouldn't. I don't have any details on which optimizations. I'm not too familiar with all of it, but yeah, basically there are some optimizations that are not allowed in tree borrows that should maybe be allowed. So yeah, it's going to be a middle ground at some point, for example, allowing the header thing, but not the other language optimizations.

**Interviewer**
Okay. Gotcha. Gotcha. It's an interesting, I like the way you describe the balance between performance and undefined behavior in terms of like, where does the burden of having performance increases go? Is it on the optimizer? Is it on the programmers themselves? If you have very defined behavior and then you rely on use of unsafe, that's explicit to enable them. So I guess transitioning away from the more general stuff, I was curious about some of the types that you mentioned in the survey response you had, specifically you selected Box, unsafe cell, MaybeUninit and ManuallyDrop. Those last two in particular, could you describe when you're using MaybeUninit and ManuallyDrop?

**Participant**
So basically ManuallyDrop the main use case of it. My opinion is basically if you have some kind of data structure and you have a method to insert something into the data structure, generally you can just use like point of right or something to properly move it over. But sometimes you have cases where you cannot just semantically move it over using rust language construct. So basically you have to copy it over, but if you copy it over into the data structure, you now have an element left over. So if that's like a string that's going to cause double free, which is rather bad. So then you first wrap it in a manually drop, then copy it over. And in the end, nothing bad will happen. That's like the way I choose ManuallyDrop.

10:21 ManuallyDrop the mai...
ManuallyDrop<T>

115
121 **Interviewer**
Okay, gotcha. And MaybeUninit?

122

123 **Participant**
MaybeUninit is just, for example, if you have an array that may be uninitialized for performance, if you have like a one kilobyte array on the stack, like in practice, it probably won't be too bad if I just initialize it and you should initialize it. But if you realize that it's too slow to initialize it, you can just use an array of MaybeUninit bytes, initialize them with uninitialized. So not initialize them and then fill it up as needed, which make it better performance. Also, if you have something that shouldn't actually like ignore the validity invariants, for example, but if you do some very sketchy things and maybe just want, or more generally, I think

10:38 MaybeUninit...
Increase Performance
MaybeUninit<T>
...
MaybeUninit<T>

another good use case of MaybeUninit is just if you want to, if you have a type, and you want to convert that type to bytes, you can't convert it to a U8 array. And you can't transmute it because it may have padding bytes. It is padding bytes, maybe uninitialized. So if you have any type whatsoever, you can transmute it to an array of the same size of MaybeUninit U8s. MaybeUninit U8 is basically anything, any byte, everything's allowed.

10:22 I think anoth...
MaybeUninit<T>
Transmute

**Interviewer**
Gotcha. And then with Box, are you typically using Box to store something that's like a type that like a raw pointer you could use in an unsafe context or are you using it as an allocator and accessing like the underlying raw pointer? Like when is Box used in conjunction with unsafe in your code?

**Participant**
Yeah, Boxes, I mostly just use Box as an allocate, like allocate and deallocate. So Box new and then immediately into raw, because there's the thing that Box has some rather strange and strict aliasing semantics. So if you store a Box and store a pointer into the Box and then move the Box, the pointer will be invalidated, which is not very useful. It's also funny because it's like, it's also exploited by LLVM to some degree. And this was just added like seven, eight more years ago at some point, because basically why not was like the comment roughly. And then people hit this sometimes. And I don't think there was any documentation whatsoever about this. It was just something you had to know. And John Gengset made a great video about the whole topic, how he was hit himself with with these aliasing requirements. At some point, I did add at least a few docs. So if you go to the docs today, the docs from, I'm going to go to the current stable docs, I think 1.69.

10:...
Box<T>
10:37 Bo...
Box<T>
10:36 And th...
Shifting Ground
10:24...
Documented...ract or I

**Interviewer**
And then who is the name of that? The person that you mentioned who made the video on the problem?

**Participant**
John Gengset. He's a great YouTuber. Great, makes some very long Rust videos. And yeah, I've learned quite some things from watching a lot of his videos.

123 **Interviewer**
How do you spell the last name?

134

135 **Participant**
It's a Norwegian name. It's just the link.

136

137 **Interviewer**
Oh, gotcha. Yeah, that'd be great.

**Participant**
Yes. John who handle?

**Interviewer**
Oh, perfect. Gotcha. Yeah. Thanks for sending all the links. I really appreciate it.

**Participant**
Yeah. Here are some docs right now that are currently inside that I added a few, maybe around a year ago that tell you that the unsafe code, that the unsafe requirements are a little strict. There's also the unsafe code guidelines issue about it. How it's kind of a foot gun because you would expect, especially for example, if you're from C++, you would just, you would just expect it to, for Box to just be a pointer that deallocates on drop. That's like how it multiplies itself. But then suddenly you write some reasonable code. You, you allocate it on the book, on the heap through Box, you create a pointer to it, and then you move the Box around. And if you move the Box, the heap stays constant. That's kind of the point of the whole thing. So we'd assume it to work, but it doesn't, at least in the current implementation of Stacked Borrows and also rustc.

10:45 Ho... Feature Disparity

10:46 But then... Box<T>

**Interviewer**
Gotcha.

**Participant**
So yeah, that's kind of interesting. Actually, I even wrote a blog post about the whole topic.

**Interviewer**
Oh perfect. Even better.

**Participant**
One of the blog posts is kind of a joke and this one is a little more serious. I guess I was a little annoyed by the situation because again, it wasn't really documented for a long time. It was just kind of done. And at least today there are no performance benefits really coming from it. I did a few benchmarks and it's basically just undefined behavior because it may be convenient, but not for real reason. My hope is that at some point we can get rid of it. But yeah, right now you have to be careful with Box and using Box as an allocator and deallocator is a great way to use Box. It's very simple. It's simpler than the allocator APIs. But yeah, if you have unsafe code, having a Box around is a little sketch, even if it's not undefined behavior, it's still a little sketch. It's kind of what I said about the bad unsafe code. Basically, again, there are things that are undefined behavior. If you do the wrong things with references or Box and that's obviously wrong, but if you use them in correct ways, it's still a little weird to have raw pointers and Boxes around because it may give, it may urge

10:25 But ye... Box<T>

10:... Code Smells

someone to change it in a way that will break. So I generally prefer to just avoid Box inside unsafe code. If there are other raw pointers around similarly, that code is just like get unchecked or something. But if there are raw pointers to the thing, it's better to avoid Box. At this point now, my hope is that in the future, it doesn't have to be avoided because it's quite convenient to just have it around and not have to worry about the allocation. Gotcha.

10:48 So... Box<T>

**Interviewer**
So I mean, I'm going to be reading your posts. So my apologies if it's already answered in that post. So you'd feel that, or would it be correct to say that you'd support having this behavior changed in just a future version of the Rust compiler so that you don't have the invalidation of pointers after you move a Box?

**Participant**
Yeah, I would support that for Box to just be a Box thing on the heap, the obvious thing. Some other people also agree with that, but other people also rightfully have few concerns that maybe this could be useful for performance in the future. But I think that the potential compiler optimization impact is going to be too small to justify having this foot gone around.

**Interviewer**
Gotcha.

**Participant**
I'm not going to be the person that will make the call or will be around to make the call. That's probably going to be on the operational semantics team. As I already mentioned, I think I don't know if you have a link already, but I can throw the link. That's the current that was like established a few months ago with the goal of establishing the exact semantics and writing the unzipped code spec, basically.

151

161 **Interviewer**
Gotcha. That's good. So then transitioning from unsafe to some of the development tools that you mentioned using, you selected most of the list. So I think I'll just be pretty broad in my question, which is which, which like, I guess just describe your bug finding workflow with using, using different static or dynamic analysis tools.

162

163 **Participant**
Basically when writing unsafe code, I try to run it all through Miri. Try to write a test suit that covers most of the code. You cannot reasonably have a test suit that covers every single branch combination, even though that would be necessary to fully make sure that it doesn't have undefined behavior because Miri just checks what you run and that you still try to have enough tests and then run those through Miri. If it works again, I cannot run the Rust compiler through Miri. Though I think funny that one

10:26 Basically... Running tests through

of the authors of Miri only at some point ran Miri inside Miri. I think it takes like six hours to run, but it sounds fun. And I guess you can kind of run Rust compiler in Miri if you have a lot of time and a lot of RAM and a lot of CPU, but it's not going to be something that's worth it. And yeah, what for again, if, if that's also not a big advantage, if the unsafe is encapsulated, it's way easier to test it in Miri. So if it doesn't depend on all the global states being set up, you could just mock the global state in some way and then run it through Miri because yeah, Miri is very slow. But Miri is like the main tool because it just finds so much things and it's better than everything else in terms of finding undefined behavior, especially since it's Rust specific. But yeah, it also doesn't hurt, especially if you, if you do FFI, some kind of other, for example, calling a C library, which I haven't really done much. So I can't talk a lot about it, but basically the LLVM sanitizers are great. Like address sanitizer and memory sanitizer and these kinds, they are not as good as Miri, but they're a lot, a whole lot faster. They have only like 30% overhead or something, which is very fast. And they also don't hurt. And if you can't use Miri, using them is definitely better than nothing. Yeah, Clippy also has some lints about undefined behavior, although I don't think I've ever really needed those, but they're especially useful if you're learning maybe and have some patterns you think or want to try to do, but then Clippy can tell you, no, don't. It's also funny, the Rust compiler has a lint. If it can statically detect that you're obviously referencing a null pointer, it will emit a lint. So if your code is that lint, please stop writing code. But yeah, some synthetic analysis isn't exactly great and it's not going to catch most issues, but for some issues it's there and that's cool. For example, one cool Clippy lint is if you have a function that takes in a raw pointer and dereferences the raw pointer, but isn't marked unsafe, that means the lint because realistically, if you're taking a raw pointer and dereference it, the function is to be unsafe because otherwise someone could just pass in a null pointer and then it's not good. Clippy also has an interesting one for signatures. If you have, if your function takes in a shared reference, but returns a mutable reference, it will lint because generally that's wrong because you can't just return a shared reference mutable reference. There are cases where it's correct. For example, an allocator, if it returns to a mutual reference to a new one. So I've hit those false positives a few times, but I think they're probably useful to include because if someone does hit them, it's great that they're told about it.

10:49... Miri is slow

10:28 But Miri is like the mai...

10:27 Cli... Clippy for Learning Ru

10:29 Clippy also has... Clippy False Positive

**Interviewer**
Are there any other cases of false positives that you found in your use of Clippy with unsafe?

163 **Participant**
Yeah, I can, at lint about turning a shared reference into a mutable reference in the signature, basically if you have an allocator, an allocator can handle mutable references because it's new memory, newly allocated memory. And it does also, it also takes in a shared reference. So in these cases, I think that's even the recent PR rusty doesn't use a lot of, doesn't really use Clippy. It's also kind of broken with the rust sees custom bootstrapping build system stuff fixed at some point, but it doesn't really

use it. So it wasn't affected. But recently I've sometimes run Clippy on rusty and then I've, yeah, I've seen in the rust sees arena.

**Interviewer**
Okay.

**Participant**
Yeah. Basically here, that's a false positive because inside the arena allocator, you can turn mut shared references into mutable reference because not really turning them. It's just creating new ones. That's like my classical case where this is false positive, but it's probably a good, still a good thing that it hits because it forces you to think about whether you're really doing it right. And I think for leads like this, it's acceptable to have false positives just because if you have false positive, you're going to put the allow and that's easy. And it's also kind of documentation telling people, Hey, this is something sketchy, but it's okay. You can either write a comment next to it. So the false positives aren't really bad. And if it is a true positive, then it's very good because it finds out. Behavior.

10:30 . B...

Clippy False Positive

**Interviewer**
Gotcha. So in terms of the dynamic tools though, have like, so have, have you been using the sanitizers a lot with your code bases and have there been particular cases where like, are there particular categories of bugs that you feel the sanitizers are better at catching than Miri can?

**Participant**
Um, generally Miri is the best thing that exists for exists for catching undefined behavior, as long as you can run it, if you can't run Miri, then the sanitizers are infinitely much better because they work at all. Basically, that's the only real reason to use them. Miri is just going to be better because first, yeah, it's very slow, which is the advantage because it means it can do complex stuff. Sanitizers cannot do anything that close to some of the things that Miri can do. For example, sanitizers, they may check, okay, you're doing something out of bounds, but they can't detect the more subtle things of invalidating pointers, which invalidating pointers at the abstract level, which is still important because compile optimizations can and probably will break your code if you do have a mistakes here, but the sanitizers just cannot know about that, but they can detect some of the worst issues like buffer overflows, use-after-freeze, the classic C undefined behavior, vilifying it all, even in Rust code. But yeah, in terms of false positives, Miri doesn't really have false positives, except the things it doesn't support. Like, yeah, if you use
167 excellent types, it will complain about undefined behavior, which is a false positive, but it's a false positive in the model, not the tool. The tool Miri implements, it doesn't, that's an advantage of dynamic analysis. Dynamic analysis generally doesn't have false positives because it does everything about the program. It doesn't have to be conservative. You can just say, this is only you be in these few cases. And while Clippy may complain, yes, I'd rather complain than not complain, the dynamic analyzer can exactly identify the cases and only complain if it's needed. But you have

to run it. And Clippy will lint on codes that you don't test and Miri will not.

**Interviewer**
Gotcha. And then the wiki frame that was interesting, you mentioned that the performance, like the fact that it was slow is a good thing because you're able to do more complex things. In saying that, do you mean that like, because there's like, because you're running in an interpreter, you already know that you're not going to be much faster than a certain baseline. So it's like more acceptable to do more expensive things because you aren't like, there isn't this expectation of performance. Would you say that that's correct? Or is there a different reason why you feel that?

**Participant**
I think I said it in the wrong direction, but it goes in both directions, I think. So first, Miri's interpreter, also Miri uses the constant evaluation interpreter, which is just fundamentally slow. I think most of the slowness in Miri today comes from the constant evaluation interpreter, not necessarily from the stack borrows checking. It used to be that the stack borrows checking was extremely slow, but someone optimized that a lot with a little caching and now it's pretty fast. Yeah, most of the slowness comes from the interpreter, although part of the interpreter is also detecting UB. So I think it does like on each move, it checks whether the type is valid and that's just going to be expensive because you have to check that the entire type value is valid. On each move. So yeah, it's slow because it knows everything. But you're also kind of right, adding some new undefined behavior detection that's slow isn't really controversial because yeah, maybe the detection is generally slow, but it decreases runtime by 1% so no one will care. Yes, that's also true. It's kind of an advantage in some ways because you could just, no one will complain a lot if you make address sanitizer two times as slow. Some people will probably be very angry because they rely on it being so fast. But if you make Miri two times slower, some people may also be angry because Miri is slow already, but probably less angry because Miri is slow already. Okay, you can just, one thing you generally do is like you normally in the normal tests you may loop 10,000 times to insert 10,000 elements to test it well. But in Miri you'll do better if you CFG Miri and if it's in Miri just insert 100 elements. It's going to be enough. So yeah, maybe you have to drop the number to 250 at some point, but that's still okay. And yeah, the code business that struggle with Miri struggle with it a lot. So if you got like two times faster, that would be cool. I would find it awesome, but it's not like a game changer. A game changer would be if you got 100 times faster. One thing that's kind of a game changer and got 100 times faster is Valgrind. Valgrind is quite interesting and cool because you can interpret arbitrary x86 machine code, but it has some false positives sometimes because it describes C semantics to the machine code, which LVM sometimes knows that the machine, for example, if you read uninitialized memory, Valgrind will complain generally correctly because it's uninitialized memory. Don't read it. But if you put in x86 machine code, uninitialized memory doesn't exist. It just dodges bytes. So sometimes LLVM exploits the knowledge of that to read uninitialized memory, not do

175

10:50 But you're also kin...
Miri is slow

anything with it, but just some LVLM transforms to weird things. And I once ran rustc inside Valgrind and it complained about uninitialized memory read somewhere in safe code. And I was pretty sure that it was probably false positives. Also, I did post it on Zulip, the Rust project communication platform, and one of the authors of Valgrind, [name], you may know them. They also came in and basically said they do work on the Rust compiler. It was probably false positive.

10:51 And I once ra...

Valgrind

**Interviewer**
Do you have a link to that thread? Actually, that would be very useful.

**Participant**
Yes, I can search for it.

179

185 **Interviewer**
Sounds good.

186

187 **Participant**
But also very recently, as of like one or two weeks ago, Felix from the Rust compiler team and AWS announced a new tool, CrabCake. It's very experimental and not really released and ready. I cannot find anything about it. I think it was written CrabCake. I can quickly check whether that's true. Which is a Valgrind implementation of stacked borrows. Implementing stacked borrows instead of inside of the compile time function evaluation interpreter of Rust C, like Miri, which is very slow. It's implemented inside Valgrind, which is slow, but a lot faster. So I can also have a link about CrabCake.

10:31 But...

Krabcake

10:32 Wh...

Valgrind

188

189 **Interviewer**
Yeah.

190

191 **Participant**
Where there was a funny comment by [name] where they were like, this is the first time someone talked about Valgrind being fast. Generally, people, I think, complain about Valgrind being slow because it is slow. If like C is 10 times faster on machine code, but compared to Miri, it's a lot faster. Just a funny comment. But more importantly, the other thing when searching for Valgrind. Yeah, I asked for a little bit, I think. I know I found it.

192

193 **Interviewer**
Oh, perfect.

194

195 **Participant**
Yeah. LLVM sometimes generates machine code. That's entirely correct. I see machine code semantics, but Valgrind describing C semantics to the machine code leads to the Valgrind thinking the machine code is invalid

because it assumes that compilers generate reasonable code that follows C semantics. But compilers sometimes don't because they don't have to.

**Interviewer**
Gotcha. Yeah, that makes sense. Gotcha. Yeah. So my final question is, are there any problems that you see in unsafe code that current development tools cannot solve that you think there should be a better solution for?

**Participant**
I think the biggest issue with Rust unsafe isn't necessarily the tools, but just the spec or rather the absence of the spec. If you write unsafe code today, you write it against the void. Isn't it how we have like stack borrows and three borrows and you can write it against three borrows or you can write it against stack borrows or neither or both or just address sanitizer or whatever. You don't really know what's allowed and what isn't. Like again, some things are currently disallowed, like the Box aliasing, but maybe it's allowed in the future. Other things are currently disallowed, but won't cause bad things in LLVM. Maybe they'll be allowed at some point. Like there are some things which are allowed in some and others. So basically to validate your unsafe code, you first need to know whether it's correct in the first place. And today you can't really tell whether the unsafe code is correct because there's no correct. There's like code that complies with the stack borrows, which is a lot of code. It's a lot of code. A stack borrows supports most code, but not all. And we know that this code isn't necessarily wrong. Like the list implementation, et cetera. See, I'm pretty confident that this implementation isn't wrong, even though it's not checked because it's very simple. But I think that's the biggest problem, not necessarily the tooling question, but just the spec question. And then with a better stack tooling can build on it. Although I think the second biggest problem is just merely being very not supporting everything. So Rust does have these very strict aliasing requirements. Like that's one of the things that's definitely not allowed is having two, for example, two mutable references that you use interchangeably to the same thing. That's just not allowed. And LLVM exploits it today. But if you do write such code, I think Miri is the only tool that will save you. Well, if you're lucky, LLVM will jumble up your code and it will, your tests will break. But if you're unlucky, it will work today. And I think Miri is the only thing that can save you. But if you, for example, call FFI, you can't use Miri. So nothing will save you. And that's one of the things that CrabCake may fix, which is very cool. And also Miri is getting support for loading shared libraries and doing FFI. So that's, at least for those people that can't use Miri because of FFI, they may be able to use Miri in the future. And those people that can't use Miri because it's just too slow, these people may be able to use CrabCake. Yeah, I think that's the two biggest problems I see in the future. The absence of any spec, the concrete spec will probably take a few years for that to be finished, but now that we have the operational semantics team, it should slowly improve and get more clear. And the tooling, just the big hole of there's lots of undefined behavior that's general, like in CEN thrust, like reading out of bounds and use after free. That will be, if you write such code, you can address

10:33 I think th... Shifting Ground

10:... It's not a problem

10:53 So... Shifting Ground

10:... Wants Solution

10:54 Th... Shifting Ground

sanitizer and it will probably fine. You don't have to be too scared of that, but if you write some more subtle rust specific undefined behavior, it may not be found unless you use Miri, which you may not be able to use.