Uh, right, some chandler crews. Um, I work at google, I work on c plus plus and google. I worked on all human google. Um, recently been take the mic down just a touch. Um, you know recently been working on lots of old him. I and I started working on kind of re designing parts of the l v m past manager are. I'm sorry if i'm a little bit quiet and hard to hear today. Turns out this time slide is, in fact, cursed. Um, I started losing my voice about eight o'clock last night.

I'm going to try and make it all the way through, though. All right. So this is actually part two of a kind of two part talk. Um, I actually intended to give this talk in april at your lv. I wanted to tell people about from the work had been doing on the past manager. And I realize that we really need to kind of go over the background for how the past manager help, as is an l v m or even designed, how we think about them, what the kind of basic components are. And so I ended up talking a lot about that instead.

But I still really want to talk to you guys about the the new design that i've been working on for the past manager. Um, I think it sells a lot of the problems that we have today. I think it's pretty promising going forward. Um, and so i'm gonna spend most of this talk actually talking about the core of the new passed manager design. IT's going to be very low level and have tons of code up on the slide decks. Hopefully that works for you guys. Um, I can't point at the code that's tried to highlight the pieces of the code that I really want you guys to look at, and we're gonna kind of step through each of the parts of the code. One of the big complaints about the previous past manager is that almost no one understood how it worked. Like, how many people here feel like they really have a good handle on how the old past manager works?

Yeah, okay, sure, chris. Um, uh, uh, yeah, there will be a quiz if you actually want to to step up to this, but it's it's really confusing. And so my goal is that by the end of this talk, at least that isn't true for the new design. And you guys can let me know how I do. Okay, so from the previous talk, you don't need to have all the background to the previous talk. Uh, but from there there's a couple of key things. Right? A pass on LLVM is kind of world is something which operates on some unit of I are uh units kind of fuzzy little find. You can have a function, you can have a module.

Um all kinds of ways we can think about how to break up the IR but you've got some chunk of it. You're going to transform it from one thing into another thing, probably equivalent for some definition of equivalence. Right? Um. An alternative use of pass million is to try and analyze that chunk of I r and drive some kind of higher order information, right? Some higher level information about exactly what that I r does. So so I started from just this kind of very basic ideas about what a pass looks like.

And I tried to see if we could actually design the past management layer around this kind of simplistic view and still have it. So all the needs and, uh, you know, like what if we actually just let the code model this very basic idea? Maybe it won't work, but it might give us a better starting point. And so I started off with this idea of a pass. Super, super simple, right? Like you have a class, it has a constructor, the constructor sets things up. It has a run method, the run method, except some unit of I r in this case, a function. Right? And it does something with that function.

Couldn't get much simpler than this. The question is how close to this could we keep everything and actually address the use cases we have for the past manager? Well, let's let's just start kind of experiment again. This is actually almost literally what a pass looked like on the first iteration of the new pass minute when I first checked. And if you go back through the commit history, you'll actually find a test case looks almost exactly like this code, right? And so we're like, okay, well, then if that's the simplest idea of a pass, what what is the simplicity of the past manager look like? How do we actually kind of, you know, take that down to its core principles?

Well, it it does aggregation. That's the core idea of a past managers. It takes a bunch of other passes, aggregate them together into some sequence, right? Runs the whole sequence over a particular unit of I r one of the key ideas of a past manager is that a past manager is itself a pass. Right? And so you can see down here, this is a class, and it has a run method which accepts a unit of I r right? So it satisfies our definition of a pass. But it also has this like really fancy clever template meta programming blah that you guys probably think it's ridiculous up at the that's actually taking care of aggravating passes.

All right? And so I I think we at least need to understand how all this works because there's no virtual. There's no like type hierarchy at all here. And and this is this is a bit of clever programming and I can take no credit for this clever programming. The entire idea came from sean parent. There are all kinds of links in the actual code if you want to read more about it, but i'm gonna give you the the like you know, super, super fast version. So what does this whole concept model thing? All right, the idea of a concept is that it is this kind of totally abstract virtual interface, right? We have virtual destructor, we have a virtual run method, right? We actually take the unit of I r as a template parameter, right? is this great kind of distillation of what a passes around this. We can build any past we want.

And then we have a model of this particular, you know, kind of high level concept, which is a class template, right? It derives from that, and it over rides that run method. And IT delegates that run method to some concrete type. Right? IT's just a rapper, but it's a wrapper that hides a particular interface inside of a virtual interface. And it doesn't for us so that our past doesn't have to understand virtual doesn't have to have a type hierarchy. Doesn't have to deal with any of that business. All of this is completely hidden. No one has to deal with that, right?

All you have to do is you have a class. If it has a run method, it is a pass. you can add it to a pass manager that makes some vague sense to people. I wanna belabor this too much. All right, so if we have a past manager and we can add random passes to it, the next thing we need is we need the ability to kind of you know move between layers of the eye are right. We need some way of adapting between different layers of the eye are the simplest idea I could come up with for this. So you have some kind of and and yes, these names are hilarious module to function pass adapter. Um, i'm sorry if anyone here gets kind of like java throw backs, but we have a model to function pass adapter and it accept it. You know wraps a function pass, and it is a module pass.

Right? And it does the kind of obvious thing. It takes the module, and it finds every function in the module, and it runs the function pass on each function in the model. There's nothing fancy here, right? Like the entire code for this is actually on the screen, right? With this, we're done. So good keynote. Ok. So it's a little bit more complicated than that, right? But this is actually the simplest baseline we could get for kind of past management. But now we need to actually work. We need to actually serve the needs that l b m has, which are much more complicated. And primarily, that has to do with analysis.

Everything would be very simple. If all you did was transfer my art. Unfortunately, this this whole idea of analyzing the I r is really, really complicated. All the complexity comes from here. So an analysis passes, you know, in theory, just a special kind of pass. All right. But it has a lot of very special properties, and we want to treat it totally differently from a lot of other passes. Okay, the kind of really, really important aspects of an analysis pass. IT's got an immutable view of the I r right? IT's never mutate ing the I r it produces a result. Ok. It doesn't just transform the I r it can't the irs immutable to an analysis. It produces some result and that result can be queried.

The result may actually be the logic of the analysis. In many cases, we don't actually compute anything. What we do is we just give you a result that will lazily compute what you need when you need it. Alright, lg and really likes to be lazy. I'm a big fan of this. So that's that's kind of the core attributes of an analysis pass. And I want to look at a concrete example, because wITh analysis passes, if I give you these like, you know theoretical examples, IT's just going to be boring. So we're gonna look at a concrete example. Turns out that this is pretty easy.

Here's the dominate a tree. Now i've deleted a bunch of comments, and I believe a bunch of other stuff. But this is actually the core of the dominate a tree. And what you might notice here is that dominate a tree doesn't have a single thing to do with the pass in it. There is nothing going on here about a pass. All right? And dormitory is art passes dom into trees are the result of running an analysis pass. Right? They're actually the result. The analysis pass looks something like this, okay?

You have a dominate a tree result, right? And we need a little bit more machinery to define analysis passing the new infrastructure. The first piece of machinery is that we need some way to identify analysis passes because we're going to want to run them automatically. We're going to want to manage dependencies between them. We do a lot of things with analysis passes that we just dont do with transformation passes, and in order to do that we need to have some good way of identifying them. And we do that with this kind of abstract idea that returns a void star that's unique to the past.

Uh. This is pretty much the same trick the current past manager uses and the piles of other parts of l t means nothing too crazy. The rest of it though, looks a lot more like the passes that i've been showing it. Right? There's a constructor, it sets things up. Turns out this one's really easy to setup, right? There's a run function, it runs over some I r i've even got a bug in my slide. That's supposed to be a con structure. Nba.you know, that's fine, right? And it computes a result that it returns it.

nothing crazy here. All the fun stuff is actually in the result. And this tends to be a pretty common pattern. In l v m the actual analysis passes are relatively uninteresting. Most of the interesting parts are the result, but it's a pretty nice way to kind of think about how you take a piece of I r and you get this result that tells you something at a very high level about that. I are vaguely happy with this so far. Totally have to have questions. Everyone's super quiet. IT's early.

All right. So the hard part of this then, like, you know, we have results already. The analysis pass itself is super, super simple. The hard part is, well, when do we run the analysis? Pass? This is actually what is the question? Ok, so historically, we had a very strange approach to this. In my mind, we actually tried to solve this as a scheduling problem. Now you may know from your computer science classes that scheduling problems are hard. In fact, they have like special classifications of how hard they can be. And it turns out that lg n scheduling problem is no different. If you actually go on you profile, um, a debug build and you run all the regression test a new profile, the regression test, you will find that about like ten percent of the entire time of the regression test suite is spent scheduling analysis passes, which blows my mind.

Okay, we're spending all of our time figuring out in which order to run the analysis passes to satisfy their dependence is IT's really wasteful. And IT's completely unnecessary because we we can use much simpler ways to do this by catching. So I want to think of the entire thing as a caching problem. Way better at solving that. IT's super easy to think about. And IT also opens the door to doing lots of more interesting things about picking women where we run analysis passes. So we pick an analysis schedule right by catching the results of lazy runs and memorizing them and just producing them later. Right? Analysis past can't you take the I r so IT doesn't really matter if some other pass runs in between. When the analysis pass, you want IT runs and when your analysis pass runs. There's no constraints to solve here. Right? IT's totally fine to just take whatever top logical order you have, right? You cache the results you produce the cache results at each step when you need to ensure that the analyses only run once. And you've picked a perfectly good schedule.

Super simple, crazy questions. Anyone think? I'm just like often left field? Ok, if you pass this, then you have repeated, oh, sorry. So if you if you pass the analysis past, and then you pass another pass that changes the I r you can't use the cache again. So you have to reap as, again, hang on to that question for like two slides. So this is what a cashing based analysis manager might look like for functions. Ok. So we first have to have another one of these concepts and pass.

We have a rule but we also have a result concept in the past concept that are distinct. This just allows us to to you know actually talk about the idea of running a pass to get a result and then holding onto a result. The result actually has no interesting interface that we really care about. The idea is just to cash it somewhere. Right? A void star would be enough, but there's a little bit of fun stuff in there. The interface for the analysis manager is very, very simple. You can get a result right? If you're just getting a result then maybe it will lazily run it. If it doesn't have one if it already has one it will simply provide it. You can also do get cash result.

This this has the kind of obvious property of being constants. Never going to do anything. IT's not going to change anything. IT's just gonna hand you whatever IT happens to have. If IT doesn't have anything that's gonna hIT. You know, right, you also have the first sign of registration. You can register and analysis pass with an analysis manager. This makes a certain amount of sense, because we want we want to kind of know the total set of analyses that this manager is kind of dealing with in its life. Right? We also want to have the particular ability to set up an analysis past with some initial state, you know, parameters tunable, whatever they may be, and plug those into the manager so that whenever it gets queried, the query path doesn't have to know how to set up the analysis. Right?

The registration took care of that for it. And the final thing, and this should kind of start to give away the the ghost here is invalidate, because naturally, the problem immediately becomes an invalidation problem of cache invalidation problem, because this is computer science, right? And so we have to develop the cache invalidation. So the question is, uh, you know, at what point, right? Like, if you if you run a pass to transform the I r you have to somehow go and invalidate all the cache results, which are no longer usable. And that turns out to be where most of the complexity is.

Ok. So how do we do cache invalidation of analyses? Well, the first thing we need to be able to talk about is kind of, you know what what are we even thinking about in in terms of analysis? what are we trying to preserve? We need some way of actually naming a set of analyses that we care about preserving or not preserving. And so the new pass manager infrastructure provides you a pretty straightforward set class. IT's got some convenient methods, right? IT's a set of preserved analyses can get say to preserve any you can say you preserve all you can you know um mark a set as preserving something right? You can query um actually deleted one method in here that's kind of important. You can also intersect two sets and that lets you figure out. Ok, so now I can actually talk about the sets of analyses I care. I'm trying to preserve or I mean preserved or any of the other queries we might have.

We end up using this in all the all the a p i's make some sense, okay, so now we have to start complicating our beautiful, beautiful, simple interfaces. Ok. The first thing to realize, uh, remember, a function past manager is a pass. So the first thing we realize is that our past interface has to get a little bit more complex. Now when we run a pass, it has to return something. I can't return void. The specific thing in its return is what set of analyses are preserved after that past has run right?

That's that's the first complication to the interface the next. And you can kind of see how this is going to play together. Right? And the next thing we do is is in the past manager, we actually have to kind of accumulate these things and and figure out how to model the preserve set of passes. When I when i'm writing on aggregation. And it's not entirely surprising. What I do is I see what each of the sub par sis preserves. I intersect the sets. And the result of that is the set I preserve.

Nothing crazy everyone sort of happy with this? Yes. But then the but the than the transforming passes have to know about the analysis passes in order to know which analyses to uh, discredit in which to leave alone. So I mean unless they query the analysis class. Yeah, so absolutely. So so the you know the question is like, doesn't this mean that all the transform passes have to know about all the analysis passes that they're gonna preserve? Absolutely. And they already do today. There's always a conservative answer of I don't preserve anything.

I don't have to know what analysis passes there are to say I preserve nothing. But if you know about the specific analyses, you can say that you preserve those specific analyses. And there are some cases where you can say like, no, no, I actually preserve every analysis. The reason I know I preserved every analysis that I didn't change, the I r got it right. And all of those are kind of model than that, um, preserved analysis set. It gives you kind of special access to none and all without knowing what they're actually comprising.

Yeah. So I I get that you don't actually have to have knowledge of all the analysis. If all you're doing is saying, well, i've I know I preserve this analysis versus like, I know i've been invalidated that one. Okay, got it. Yeah, this is why it's a preserved analysis, not an invalidated analysis. Makes it much easier to specify. Yes, I can repeat it. For example, the question is, what happens across I r units? I promise. I've got a slide for that. So does it make sense to have some passes that check the validity and verify that the analysis is verified? I don't see too many of these.

Yeah. So so the the this is a common desire. It would be nice if we actually checked the things. We say we we actually preserve actually do a little bit of this today, but very, very little were very bad about it. And it's one of the things I think we're going to have to get better at in order to adopt this new design, because it relies on it much more heavily from my experiences complicated bud as whole, because people pretend IT's preserving. Well, IT's not. Yeah, I saw another hand at the back.

I can't hear you. I'm happy to repeat it, but I can hear you. Sorry. Thanks. This is louise. Can you hear me? Okay? I just wanted to get back to the beginning a little bit. When you introduced the past concept. I believe when you mentioned that you also have result on sept. And all those things. Um, I just wanted to uh, get a clear picture of whether these are purely synthetical contacts, or whether there is some semantic properties that these also encapsulate ing. Um, there are semantics. this is uh, at a core, this is just a concept based polymorphism.

Um, if you want to look it up, sean parent describes it in a bunch of toxics concept based polymorphism. And there's a semantic contract. And and i'm sorry, I I deleted all of the comments. These classes actually have nice comments that actually kind of clarify a lot of this, but my slides are small. Ok. Alright, thank you. I just wanted to make movement. Absolutely. All right. So jumping back into this a little bit, we've got a function past manager. The first thing we notice is that the idea of a past has become more complicated. We're now recording what analyses are preserved.

We're doing so conservatively. And we're returning that whenever we're run over a unit of I r the next complication is that we have this analysis manager inside the past manager and in the interface to the pass. Ok. And this is important because if the analysis manager that your passes query ing isn't visible to the past manager that's running your pass, then we can't ensure that the in validation steps actually occur at the right times. And and we certainly can ensure they occur at the optimal times. And so we always want a pass to kind of get its analysis manager from whatever past manager is running it, okay, as we pass it down through the run method.

Now, one thing that I I couldn't find a good way to show on this slide is that this is actually optional. You don't have to accept and analysis manager in your past interface. If you just only have one parameter to your run function, the past manager knows how to do that. Deal with that, and it just won't pass the past minute. The the analysis manager to you. So you pay for what you use. If you don't need analyses, you don't pay for them, right? If you do, you add the parameter, it will automatically be populated when your passes. Right.

That if should be by the data. And the question is, uh, shouldn't you assume that if you don't know anything right, then you should evaluate everything? And that's certainly true. But there may be cases where you happen to know like, no, no, no, I don't mutate the I r I can't have invalidated things. So so there is still a good use case for preserving all things. Okay, so with analysis manager, right? All all the kind of interesting stuff that's going to happen here is that this past manager is going to first pass that analysis manager down into each of the subordinate passes. Right? And the second thing is going to is going to invalidate any analyses that each pass actually fails to preserve. And so the past managers taking care of all of the cache invalidation for you, and it's using this this preserved analyses set to communicate from the past, which runs to the analysis manager, what needs to be invalidated making some sense, folks.

All right, so as we've been going through this, we're talking about analysis managers and they all operate over units of I are just like the passes to. So the interesting thing is how do you actually model this when it crosses boundaries between I are um and unlike normal passes, this is a much more complex operation because this is bidirectional. All right. You can imagine a module to pass query ing function analyses. You can also imagine a function of pass query module analyses. So we need to support both directions of this. Um. The other thing that's a little bit tricky is that in validation has to be propagated by directional e write a function transformation can invalidate the entire module analysis, right?

And clearly a module transformation and validates every function analysis potentially. Right? IT's even trickier, right? How do we even identify the function for which the analysis was run if the transformation to the module removed that function? So this is actually a really hard kind of cache invalidation problem, where your keys may be invalidated at the same time as the values are invalidated. Ok. So is is where things started to get a little bit tricky. And you can tell, but close my slides get a bit harder to read. And I apologize. I worked really hard, but it was hard. It was very hard to make this fit.

So bear with me. Also the names just get silly. This is a function analysis manager, module, proxy. Okay. Now we can debate whether this is a good name or a useful name. All we want, I don't really care. The ultimate idea is that this is a module analysis pass, okay? And what it's doing is a module analysis path is absolutely nothing. You can see it's run. And all its run does is construct a result. All of the interesting logic here is in the result and the result of this weird proxy analysis doesn't do anything either, except provide two very important methods.

Oh, come to myself. the first method that's really super important here is that we can get the function analysis manager from this proxy. That's really, really important. Okay. This proxy needs to be the path through which we always get that subordinate level of I r analysis manager, because this proxy is responsible for ensuring and validation occurs at the appropriate points. Ok. So this is the primary interface for kind of extracting a different level of I r s analysis manager. The second important thing is in validation. Ok. The nice thing is that the model of a module analysis being invalidated pretty cleanly, maps on to, you know, fanning that in validation out across all the functions within that module. Ok, so we go and we look at what's going on. We actually kind of delegate this invalidation down. Now we do something really interesting. Here. We check whether this analysis is preserved, and we only invalidate the function analyses.

If this mod proxy analysis fails to be preserved, the whole point of this is that if you have a module pass and you actually mark explicitly that you preserve this function analysis manager module proxy thing, right, you actually mark that you preserve this. What that says is, hey, no, i've taken care that all of the function analyses, all the functionality is that we have ever cashed are correct after whatever i've done to the module and their times. And this is actually pretty straightforward. For example, if you have a module paths which only manipulates global, maybe it only delete un referent to global.

Okay. You might be able to confidently say like no no no. The function analysis passes are completely validated. You might have to do some work. You might have to go query them, but you can actually say that. Ok. And it's important that you have the ability to preserve function. Alice is when you know you didn't touch them. It also means that when the I r changed and you reserve, you return that. You know, I know I preserved everything. It actually works in this delegation layer as well. Make some sense. So some concerned looks about that, okay, yes.

The question is, is it only deletion? What about addition? Uh, well, the nice thing about addITion is that if you added a function to an analysis, uh, to a module, um, IT's pretty easy to tell that you you don't have a cashed analysis result for that function, because IT didn't exist before the function manager. Uh, if I mean, so if you are, if if your passes actually directly manipulating the function analysis manager and you say you preserved the function analysis manager, then you either have a bug or you did.

I mean that's that's within your hands as manager to include that new function. Uh, there there's a whole big interface on the function analysis manager. You can actually manually run analyses if you need to, and inject results and do other things. So it's possible if you wanted to manually update it, and that certainly use case, right? A one use case for for actually preserving analyses isn't there intrinsically preserved, but actually do work to ensure that they are correct after your transformation, and then mark them as preserved. And that's one of these cases you want to support.

Ok. So this handles kind of the invalidation from, you know, a higher order chunk of I r down to a lower order trunk of I r so how do we handle the reverse? Turns out the reverse is easier, ok, because in the reverse, you simply have fewer choices. There is no way to lazily run a module analysis pass and produce a useful cache entry from within a function transformation. Because if you are within a function transformation pass, you're not allowed to go and look at other functions, right? You're not allowed to to to do anything with them. You can't touch them. So unless you already have some analysis cashed and ready, you just you can't touch it. Okay?

And so we have this lovely thing which just provides uh access to the module analysis manager, but it's a contax s so it says that you can't create new cache entries. You can only query what's already in the cache. Um and it also doesn't do anything to handle and validation because the actual past managers themselves propagate and validation back up the stack. Right? If you remember what our function pass finisher looked like, it actually returned the intersected set of preserves passes up to whatever is managing it. And that takes care of the in validation for you. So this is this turns out to be the simpler of the two problems.

Um, whether or not it's actually invalidated, uh, it just prevents us from rerunning a trivial path analysis. Past is just going to reconstruct a result. That's all, uh, running a little short on time. So i'm going to skip ahead. You guys had good questions. Uh, so the first thing I want to briefly mention is that we still need to understand how we funnel these analysis managers back and forth between the layers of I r and this happens inside of the adapter between the two layers of I r so this is the less simplified version of the adapter.

Ok? When the adapter actually receives uh and analysis manager in its argument, it has to get the next level of analysis manager out of it. Right? And then IT has to pass IT down, and IT has to invalidate IT as the thing is running. Right? This isn't too surprising. IT's just responsible for actually doing that step of delegation. So I was going to try and go through and actually show you guys how to use them. But all of this code is checked in. And i've only got a few minutes. I want to skip ahead.

Um, I want to talk about a few loose ends that I haven't really covered yet. Uh, automatic registration. Uh, lots of people talk about automatic registration. IT's really, IT's a really fundamental part of our existing stuff. And my answer is simply no. Uh this causes endless problems for us, right? Like if you want to understand how frustrating automatic registration of passes talked either owen or christina men like they will tell you about the pain and suffering they have gone through trying to support this use case. Um, I don't actually think the use case is really that important. I would like passes to stop being special. I would like them to just be like any other piece of code, right? We know how to you know or like orchestrate your a p i's so that you can you know construct an object with the various pieces of input you need and then pass it around to another object we know how to like you know you know collect things together into containers and these types of things. These are simpler concepts. We don't need an automatic magical registry hiding behind the scenes.

Um. When it comes to things like a command line stuff we we need to throw out the entire command line passing anyways because the existing command line passing magically infer structure from this like flat sequence of arguments that doesn't actually have structure. And so there are these like even more magical arguments which kind of slightly change how that structures inferred because their special barrier passes like it's a terrible pile of hacks. Instead we could define a super simple textual syntax for specify a pass puppy pline including structure and then we could pass that and we could actually put that code in a library and that library could be registered with the names of all the passes and then all the front ends that ever want to you know use textual ways of registering passes could use that library and there's no more need for registering command line flags for passes just to get there you know pass name in there.

So that kind of obvious all these things we still need to solve something for plug ins, but it seems much more straightforward to give plug into dedicated access to registering their particular name of pass rather than trying to tie it to global variables with weird initialize as really scary to me the other losing uh one of the other loosens I I kind of feel like I need to talk about is reuse ing pa sizz. So one thing you might notice is that you actually pass these things around by value. Okay? And so so the expectations are actually going to move them most of the time into the past managers between the past managers are going to take ownership of the things and a lot of people are worried about like well but what if that's going to increase the compilation cost? The idea here is just factor that out of the pass if you need to have a past context that maintains long live data structures have that have it outside of your pass pass it into the constructor of the past right and and like reference it lately there are lots of ways to actually manage data structures externally to the past infrastructure. And this keeps writing passes very simple which is what the majority of passes want.

So super quickly, quite a things stand. Um, most of the instructors entry. So almost all of the code i've shown you today has been crib and like massage to fit into a slide out of like checked in code. All right. I'm not showing you like any future things, so if you want to play with it, go play with it, check it out. I encourage to try it out. Um, there's also a bunch of uh, support for other things they didn't talk about, like sc cs. Uh, there's not support for loops yet. Um, i'm not planning on doing basic blocks are instructions, because they don't really seem relevant.

Um, I have a whole section on like what's next, but we're out of time. And so i'm gonna go ahead and stop it. I'm sorry. I I made a terrible mistake of like adding things because I was worried I would didn't have enough material. Um, spc past to have a together function analysis information. So the question is, how will these work with s e c passes to query function? Alice's passes, uh, the same way, like so I use the module because the modules a lot easier to talk about. But there's there's a there's kind of a proxy that lets the module query the function analysis manager and get results for function analysis function analyses.

You can set a a totally analogous proxy for the function analysis manager to a s e c past manager, right? One of the things is that it's it's really decouple how many layers of I r we have. So one particularly obnoxious thing today about the way things are is we actually have multiple past managers, not just for like different levels of our past, but also for the machine passes and all that. Do you have thoughts on how to clean that up and maybe unify it in some same way and get rid of all the static initialize as and all that random stuff that you know I care about?

I have thoughts, but I don't actually trust them. Um, I haven't spent enough time looking at it to really understand this problem and to have any confidence in the answers. As for I hate to punt, but about his question earlier about the verification to see if it actually invalidate or not. Uh, it could be as complex as running the whole pass again. So this could be something like a search that you just turned on when running the tests or validation, whatever that you turn off in production.

That's that's uh, why would like to see. So so you can actually look at some of the existing pass and allah sis passes and they have a way to kind of compare them and to to to validate the results. And we just need to actually, you know, do that more consistently as an engineering thing. And then once this kind of consistently available add support to the analysis layers to kind of speculative liriodendron allah sees even though they're cashed in, debug builds or in special builds and validate like no, we actually we got the same thing again. Um, IT's gonna be a lITtle bIT tricky because IT will actually become exponential cost if you're not careful. But like we know how to solve these problems. So this isn't scary.

I would like to go back to the uh, flow of the past manager floor. So basically we are supporting like iterative compilation and IT's really hard to see. So IT's really hard to know what's the flow of the compiler. So, well, how the ordering is. So can we have a way or to expose the loop so that people know and they look the file, they know that something's going on here. so this stuff is happening. the best hope there is actually kind of logging. Um and if you actually look at the test cases that are checked in for this infrastructure, there's there's a past debugging mode and what it does is it actually prints every step of what it does as it does it. And that's actually how a bunch of testers structured. They're they're not they're not looking at the I r there making sure that we actually like you know we do this and then we do this and we do this we do this and the sequence is a reasonable sequence.

Um, I don't know of a better solution to that, especially because uh, one of the things I want to do next is to paralyse the optimizer, which makes it really hard to know what's happening next. And silence reigns. Talk right here. Uh, so so what you've presented as a new mechanism for uh, managing the past, what i'm confused about is how you handling the transition from the old. Do do you support both in the code base or how does that work? So the first next thing is that we have to port the existing pipelines and all the passes. And while the ports going on, we absolutely are going to support both. There's just there's just no other way.

And IT turns I having started on this a lITtle bIT, IT's not that uh you end up just being able to just factor the code out so that the code shared between both pieces of like you know glue and eventually will fold that away. But it it actually ends up cleaning up most of the analysis passes when you do it. So it's it's the the loss isn't very is very high there. Ok. and then to promote the new style, have you updated tutorial so that people who are adding new code use it a new way? I I don't want people to add you to to use the new way when they're adding new code because it's not actually enabled yet. Um. Right now it's on me to keep up with new code coming in uh during the port. Once we start talking about enabling it. That's when that's when we want to make sure people uh take over. But this is kind of the first step of documenting how things were probably going to look run up.

I have you right now, uh, the mic can get the other mic, okay? The past manager, um, is in that is updating the state of the analysis manager during the passage that is running, but it's returning the intersection of all the passes. All the analysis is invalid, isn't overly conservative, because an earlier pass that it run may invalidate, say the deaf use change in the later pass may recreate them, but not destroy them. Yeah, we should fix that. IT's conservatively correct, but I agree.

So I ran enough time. The question is, does this help out the static pipeline? Is that what you said? Structures make that. Ok. so the question is, right now, you know, if you have a fixed number of passes and nothing is varying, uh, IT's IT's IT's hard because you have to have global constructors, all the same, and all kinds of dependency stuff. Um, IThink IT gets simpler, right? Because now like you just have like have global variables of those types. If you want, IT's not gonna matter because they're not like, you lose ownership, right?

You're you're gonna have a past manager to do the aggregation. But it's it's past managers are now as heavy weight as a vector of pointers with a virtual call to actually run something on that pointer. On the on the object. That pointer is reference ing, right? Like can't I can't imagine how to make something lower overhead than that guys are enough time. Thank you very much.