Dave Abrahams, Oct 19, 1:51 PM
Non-monomorphizability of Swift and C++ Templates

Today in a Swift/C++ interop sync up we had a discussion about how to bridge Swift and C++ generics

The old idea that we can maybe just force Swift to instantiate stuff or simply change the model to have "compile-time-only" generics came up and I mentioned that I don't think those are feasible approaches.

I thought we could talk about the reasons here.

@Brennan Saeta rightly pointed out that it should get written down.

Dave Abrahams, Oct 19, 2:01 PM
Background on monomorphizability:
https://forums.swift.org/t/swift-type-checking-is-undecidable/39024/40

Swift type checking is undecidable - Discussion - Swift Forums
forums.swift.org

Dave Abrahams, Oct 19, 2:06 PM
In the the simple example there, the set of types constructed/used is unknowable at compile-time. If we were to write analogous code in C++ and tried to instantiate X<int>, compilation would fail because the template instantiation depth limit would be hit.

But that is legal Swift code.

Does anyone want to argue that we can make that code illegal in Swift?

Brennan Saeta, Oct 19, 2:19 PM, Edited
My understanding is that some folks are interested in adding features to Swift to guarantee static monomorphization. It would not be for all Swift code, but programmer requested in certain contexts. I imagine we would be able to re-use these features for C++-interop...

Dave Abrahams, Oct 19, 2:21 PM
In that case we need to talk about how these contexts might be created and what restrictions they might impose on users.

Dave Abrahams, Oct 19, 2:42 PM
For example, can you create/use separately compiled modules?

Dave Abrahams, Oct 19, 2:56 PM
Ideal interop with C++ templates, if you can't monomorphize, requires template instantiation at runtime, which is a **lot** to swallow. So maybe we *should* think in terms of monomorphization. I would be very interested in knowing what “some folks” are thinking.

Dmitri Gribenko, Oct 19, 11:16 PM
I think we don't necessarily need to design a set of rules to prohibit that while type-checking the definition; we could attempt monomorphization and fail compilation at the use site if it is not possible. How common are such cases?

Dave Abrahams, Oct 20, 9:41 AM
Depends how you approach it. C++ gets a lot of mileage out of treating each member of a class template as separately instantiable and only instantiating them “if used:”

```
template<class T> struct X {
  X<X> g() { return X<X>(); }
};

int main() {
  X<char>().g(); // instantiates X<X<char>> but not X<X<X<char>>>
}
```

If we did something similar for monomorphization in Swift, most cases, such as the C.Indices.Indices.Indices… case I cited in the linked post, would go away.

Dave Abrahams, Oct 20, 9:44 AM
That said, I still don't know how this works with separate compilation of modules.

Dave Abrahams, Oct 20, 9:53 AM
Say you have a Swift module that uses a C++ template:

func f<T>(x: T) { someCxxFunctionTemplate(x) }

Dave Abrahams, Oct 20, 9:55 AM
You have to compile this into something that includes the ability to instantiate someCxxFunctionTemplate<T> for arbitrary T, which only becomes known at the point where you have the whole program (essentially, link time).

Dave Abrahams, Oct 20, 9:59 AM
Worse yet,

Dave Abrahams, Oct 20, 10:01 AM
func f<T>(x: T) -> some_cxx_class_template<T>::type { ... }

Dave Abrahams, Oct 20, 10:03 AM
This one is a real problem; we can't even typecheck uses of f from other generics.

Parker Schuh, Oct 20, 10:37 AM
What if some_cxx_class_template<T>::type was instead some protocol modeling the c++ template and you had to explicitly list the instantiations you wanted to use in order to use c++ templates in swift generics?

Dave Abrahams, Oct 20, 10:48 AM
Example please. I don't know what “protocol modeling a C++ template” could mean or how this would help.

Dmitri Gribenko, Oct 20, 11:52 AM
> You have to compile this into something that includes the ability to instantiate someCxxFunctionTemplate<T> for arbitrary T, which only becomes known at the point where you have the whole program (essentially, link time).

we would need to serialize this generic function into the .swiftmodule. that module would record a dependency on the clang C++ module, so when someone calls it, they would have access to both the implementation of the function and the C++ module dependency

this model is exactly as costly as headers in C++ today

Dave Abrahams, Oct 20, 11:55 AM
Yes. This still does not solve the typechecking problem.

Dmitri Gribenko, Oct 20, 11:55 AM
oh yes for sure. i don't think that is solvable though

c++ templates are a "substitute and see what you get" model

even with concepts i believe

(iirc there is no guarantee that a requires clause perfectly describes the requirements of a template)

Dave Abrahams, Oct 20, 11:56 AM
Yes, even with concepts. Those concepts lite are… not scottish.

“ i don't think that is solvable though” doesn't get us off the hook. We need **some** answer.

Dmitri Gribenko, Oct 20, 11:58 AM
we would have to produce this error from SIL's generic specialization pass

Dave Abrahams, Oct 20, 12:02 PM
Got a meeting

Parker Schuh, Oct 20, 1:01 PM

template <typename T> struct ToyBox { T something(); ... };

protocol ToyProtocol { associatedtype T func something() -> T ... }
// Instantiates for ToyBox<int> all the functions needed to conform to ToyProtocol (just like as if they were referenced individually).

extension @cplusplus(ToyBox<int>) : ToyProtocol {}
func apply<S: ToyProtocol>(var t: S) -> S.T { return t.something(); }

Here the user has to be explicit that they're forcing instantiation to conform to the protocol.

Dave Abrahams, Oct 21, 4:10 PM
waitaminit, this is nonsense:
func f<T>(x: T) -> some_cxx_class_template<T>::type { ... }
Of course that can't typecheck unless Swift knows that some_cxx_class_template<T>::type exists for all T.

Dave Abrahams, Oct 21, 4:17 PM, Edited
And if Swift knew nothing more than that, it would be as-if f returned some associatedtype type of T's conformance to a synthesized protocol __some_cxx_class_template, with only the constraints on T.type declared in that protocol. @Parker Schuh that sounds a bit like what you were suggesting (though I still have to grok your code)?

And of course you'd have to spell it:
func f<T>(x: T) -> @cplusplus(some_cxx_class_template<T>::type) { ... }

or if it was somehow mapped to a Swift generic type,

func f<T>(x: T) -> some_cxx_class_template<T>.type

Dave Abrahams, Oct 21, 5:41 PM
OK, I'm starting to see how this can work…

Parker Schuh, Oct 21, 6:30 PM
A synthesized protocol is a little sketchy because of sfinae which is why I was thinking that you would have to talk about the template in a generic context via a manually specified protocol. Of course, when you extend a c++ class to conform to a protocol, that would explicitly request those properties of the template needed for the conformance. I was still thinking that the swift generics code wouldn't be able to reference a template explicitly, but would be used like so:

func f<T, R: SomeManualModelOfCxxClassTemplate>(x: T) -> R where R.T == T { ... }

I think you could make a synthesized protocol for

func f<T>(x: T) -> some_cxx_class_template<T>.type

but it would have to be formed from just those functions of the c++ type used in f<T>(x: T) or just conservatively include all the template functions.

Anyways, my SomeManualModelOfCxxClassTemplate stands in place of your synthesized protocol.

Dave Abrahams, Oct 21, 9:37 PM
I can't imagine that SFINAE creates any new issues for interop that aren't already present due to C++ overloading and template specialization.

I wasn't talking about the template conforming to the protocol, but T.

Dave Abrahams, Oct 21, 9:43 PM
So
protocol __some_cxx_class_template_type { associatedtype __type }

extension Z: __some_cxx_class_template_type { typealias __type = ... }
func f<T: __some_cxx_class_template_type>(x: T) -> T.__type

obviously to generalize this you need something like

extension Tuple2<Y,Z>: __some_cxx_class_template_type

to deal with class templates having more than one parameter

Dave Abrahams, Oct 21, 9:48 PM
I do get what you're saying, but I don't think your signature is quite right:

func f<T, R: SomeManualModelOfCxxClassTemplate>(x: T) -> R.type
where R.T == T { ... }

but either one is problematic because there's no way to deduce R.

Dave Abrahams, Oct 21, 10:02 PM
I actually don't believe in the synthesized protocol idea, and do believe that something more like manual conformance declaration is going to be essential. In simple cases,

protocol ThingWithType { associatedtype type }
extension some_cxx_class_template: ThingWithType {}
func f<T>(x: T) -> some_cxx_class_template<T>.type { ... }

And then you check the conformance in SIL's generic specialization pass.

Dave Abrahams, Oct 21, 10:05 PM
It may be possible to do a preliminary check in most cases, but you'll always need the check during specialization, essentially equivalent to C++ phase 2 type checking. And some conformances would have to be labeled @no_preliminary_check.

Dave Abrahams, Oct 21, 10:08 PM
More complicated cases have to do with parts of a class template's interface that are conditionally available, and where the conditions can't be expressed in Swift's type system

Dave Abrahams, Oct 21, 10:11 PM

protocol ThingWithF { f() -> Self }
extension some_cxx_class_template: ThingWithF where @cplusplus(sizeof(T) > 42) {}

Maybe those conditions can just be assumed to be satisfied in phase 1 and again, checked in phase 2.

Dave Abrahams, Oct 22, 8:07 AM
in which case it may not be important to express them at all. Checks that would fail in phase 2 produce instantiation backtraces, as usual.

Parker Schuh, Oct 22, 10:32 AM
I suppose I don't really mean sfinae, but the rule where the members of a class template are not fully type-checked until they are used. (Like in std::vector where copying is only enabled if the

element type supports copying). R would be deduced at the call site when you do something like this:

var x: some_cxx_class_template<T>.type = f(some_t)

I feel like your syntax where there is no reference to the protocol assumes a feature tying them together because for normal swift generic structs is there some hidden implicit protocol that models the generic type?

Dave Abrahams, Oct 22, 11:06 AM
> where the members of a class template are not fully type-checked until they are used yes, I mentioned that above (Tue 9:41) AM.

I think I'd rather not discuss the protocol synthesis idea anymore if you don't mind; as you can see from last night's posting I've moved on 🙂

Parker Schuh, Oct 22, 11:08 AM
Ah, so if there is a manually specified protocol, how does it fit into the

func f<T>(x: T) -> some_cxx_class_template<T>.type { ... }

signature?

Dave Abrahams, Oct 22, 11:11 AM
It depends.

Dave Abrahams, Oct 22, 11:12 AM, Edited
We can (preliminarily) typecheck this in phase 1 if:

a) some_cxx_class_template has no specializations.

Parker Schuh, Oct 22, 11:13 AM
What do you mean by phase 1?

Dave Abrahams, Oct 22, 11:14 AM
I feel like you're missing some context here. Did you read everything I posted above from 10:02PM last night?

Dave Abrahams, Oct 22, 11:16 AM
Phase 1 is regular Swift type checking

Phase 2 happens in SIL's generic specialization pass.

LMK when you're caught up and I'll continue

Parker Schuh, Oct 22, 11:17 AM
Yup, mostly caught up. I missed the T is what conforms to the protocol section.

Dave Abrahams, Oct 22, 11:18 AM
Uhm, but that was part of the synthesized protocol idea, which I'm no longer pursuing. The valuable stuff is after that.

Parker Schuh, Oct 22, 11:22 AM
why do you need the extra associated type here:

protocol ThingWithType { associatedtype type } ?

Dave Abrahams, Oct 22, 11:24 AM
We don't need it at all unless we need to model the availability of some_cxx_class_template<T>::type for some Ts in Swift.

We probably don't need to model that.

Parker Schuh, Oct 22, 11:32 AM
You were just getting somewhere, but I don't think this is correct: "a) some_cxx_class_template has no specializations." I can always write a template that just forwards to another template that has specializations.

Dave Abrahams, Oct 22, 11:39 AM
It doesn't matter; it's only a preliminary check.

If it has no specializations and its definition has no type, we can diagnose it in phase 1. If we want.

Dave Abrahams, Oct 22, 11:43 AM
But the diagnostic in that case is no better than what we'd get by leaving it to phase 2, so there's probably no point in trying to typecheck anything about C++ templates in phase 1.

Parker Schuh, Oct 22, 11:44 AM
Yes, I think at some point you build the witness tables and that is where you type check the templates.

Dave Abrahams, Oct 22, 11:45 AM
You can only do that when you generate specialized witness tables.

Parker Schuh, Oct 22, 11:46 AM
Just curious, couldn't that happen earlier if you had an explicit extension some_cxx_class_template<T> : SomeProtocol {} ?

Dave Abrahams, Oct 22, 11:46 AM
That's what SIL's generic specialization pass is about.

Well, yes and no.

Parker Schuh, Oct 22, 11:47 AM
I'm saying T, but I really mean explicitly listing out all the types (like int, float, etc...)

Dave Abrahams, Oct 22, 11:47 AM
In the sense that you can now typecheck anything that comes from the conformance to SomeProtocol (like the existence of a nested ::type) in phase 1, but…

since we can't check that conformance declaration until phase 2 when we know what T is

you may get a diagnostic for the conformance declaration in phase 2.

Dave Abrahams, Oct 22, 11:52 AM
That still would be a better diagnostic than what you get without the conformance declaration, because it could be reported in one place (at the conformance declaration) rather than every place you name some_cxx_class_template<T>.type (for any T).

To make it good we would want to avoid reporting the different Ts for which some_cxx_class_template<T> fails to conform with distinct diagnostics.

Oh, I just saw: ”I'm saying T, but I really mean explicitly listing out all the types (like int, float, etc...)”

yes, that might change the picture so you could do the diagnostic in phase 1. However, I suspect that model is not ergonomic enough. Having to do stuff like that is one reason Ada's generics system was not really successful.

Parker Schuh, Oct 22, 12:06 PM
I guess there are points where you go, "lookup protocol P for some_cxx_class_template<T>. If that never happens in a generic context, then you don't need to explicitly specify which types conform.

Parker Schuh, Oct 22, 12:11 PM
I think anything else would require a feature that attaches some information like, "These are the template protocol conformances that are required for this generic function." to each generic function and class.

Dave Abrahams, Oct 22, 12:22 PM
Those points occur during monomorphization.

Every Swift generic that uses a C++ template must be monomorphized, as must every swift generic that uses such a generic, transitively.

Parker Schuh, Oct 22, 12:24 PM
I must be really behind, if you're requiring momomorphization, why even bother with a protocol?

oh, I guess to type-check in phase 1?

Dave Abrahams, Oct 22, 12:26 PM, Edited
Yes, you can type-check more things in phase 1, the more *each one is* captured *at* a common point where a statement is made about them in the Swift type system.

Of course you still have to check those statements in phase 2 for all the used types, but as I said you get better diagnostics this way.

(At least potentially)

Make sense?

Parker Schuh, Oct 22, 12:29 PM, Edited
Makes sense. I was just thinking that for something like: func f<T>(arg: vector<T>) { ... } technically, a witness table could be passed into the generic function and you could avoid the momomorphization.

Dave Abrahams, Oct 22, 12:30 PM
I don't think so.

Aren't I allowed to declare a specialization of std::vector<X>?

if X is my type.

Sure I am; that's how iterator_traits works.

Parker Schuh, Oct 22, 12:31 PM
Hmm, I guess it would be just the case where you conform vector<T> to Sequence.

Dave Abrahams, Oct 22, 12:32 PM
details?

what would be the case?

Parker Schuh, Oct 22, 12:33 PM
var v: vector<Int> = fetchSomeVector();

fn_taking_sequence(v)

In that case, you would explicitly have a statement that gets the witness table for Sequence from vector<Int> and then use it in a generic context.

In that case, fn_taking_sequence would be able to operate over vector<T>, but not actually require momomorphization.

Dave Abrahams, Oct 22, 12:43 PM
I don't know what statement you have in mind, but I imagined:
extension std::vector: Sequence { ... }

is written explicitly, and at the call site of fn_taking_sequence(v) that conformance would be checked for vector<Int> (in phase 1).

Parker Schuh, Oct 22, 12:44 PM
Yup, All I'm trying to say is that in this particular case fn_taking_sequence would not require momomorphization.

Dave Abrahams, Oct 22, 12:45 PM
If the call to fn_taking_sequence were in a generic context where the T parameter to vector was dependent, that context (function) would have to be monomorphized and the conformance of vector<T> to Sequence would be checked in phase 2.

Yes, that is correct!

No forced monomorphization of fn_taking_sequence, because it does not use a C++ template in a way that depends on a generic parameter of fn_taking_sequence.

Parker Schuh, Oct 22, 12:48 PM
I think the fn_taking_sequence use case is what a lot of the low hanging fruit is (calling map on std::vector, etc), and the monomorphization version can come later.

Dave Abrahams, Oct 22, 12:50 PM
You can't get away from the basic problem of monomorphization. The caller of fn_taking_sequence, if generic, must be monomorphized

If you want to say, in the first development step, Swift generics can't use C++ templates that depend on the generic's parameters, then I buy that as a strategy though.

I mean, as a strategy for bring-up.

Is that what you meant?

Parker Schuh, Oct 22, 12:53 PM
Well, that was what I was trying to say, but now I'm slightly concerned. When you're modeling the templates manually as protocols, you probably want them to return other templates. Consider the case of modeling std::vector and begin(),end()

Dave Abrahams, Oct 22, 12:57 PM
Who's modeling templates manually as protocols?

Parker Schuh, Oct 22, 12:58 PM
Oh, I thought that was required for phase 1 typechecking.

Dave Abrahams, Oct 22, 1:00 PM, Edited
phase 1 typechecking works OOTB on everything that isn't:
- a use of a C++ template whose parameter is dependent on a Swift generic parameter
- a declaration of conformance of a (not-fully-specialized) C++ template to a Swift protocol

Parker Schuh, Oct 22, 1:02 PM
Yes. I agree.

Dave Abrahams, Oct 22, 1:02 PM
I'm not sure what you mean about modeling templates as protocols. If you think that's still relevant in light of what you just agreed to, maybe you should write out the swift code for the begin/end thing.

so I can see what you mean.

GTG, meeting the boss.

Parker Schuh, Oct 22, 1:03 PM
mmk

Parker Schuh, Oct 22, 1:34 PM
Ok, maybe begin and end don't quite illustrate my point, but
:

```
protocol ConstIteratorManualModel {
  associatedtype ElementType
  static func ==(_ a: Self, _ b: Self) -> Bool
  static func !=(_ a: Self, _ b: Self) -> Bool
  func dereference() -> ElementType
  func next() -> Self
}

protocol VectorManualModel {
  associatedtype ElementType
```

```
  associatedtype ConstIterator: ConstIteratorManualModel
    where ConstIterator.ElementType == ElementType
  subscript(_ i: Int) -> ElementType { get }
  func begin() -> ConstIterator
  func end() -> ConstIterator
}

func convertToSwiftArray<Vect: VectorManualModel>(_ vect: Vect) -> [Vect.ElementType] {
  var out = [Vect.ElementType]()
  var iter = vect.begin()
  let endIter = vect.end()
  while iter != endIter { out.append(iter.dereference()) iter = iter.next() }
  return out
}
```

A better example might be:

```
protocol MyCppMatrixType {
  associatedtype ElementType
  func slice_row(_ row_id: Int) -> cpp_span<ElementType>
}
```

Dave Abrahams, Oct 22, 2:03 PM
Ah, I think I understand what you're going for: write protocols corresponding to C++ generics so that you can get phase-1 typechecking of their (dependently-typed) uses from Swift generics.

Yes, that seems like a technique one might use in this system in order to bound the bad effects of the template instantiation model.

However, of course, it is an anti-pattern to write convertToSwiftArray when you could make std::vector conform to Sequence.

I don't know if I understand what you're up to with MyCppMatrixType or why it's a better example.

care to elaborate?

Parker Schuh, Oct 22, 2:10 PM
Ah, it is a better example because it returns a c++ type from the function. The first example has protocols that can be implemented entirely with swift types.

Dave Abrahams, Oct 22, 2:11 PM
Not sure I see how that makes a difference.

Parker Schuh, Oct 22, 2:12 PM
You can't really use that modeling protocol in a generic context anymore because it references a specialized template type.

std::vector doesn't naturally conform to Sequence and that will require generic code to bridge between the two models (much like convertToSwiftArray)

Dave Abrahams, Oct 22, 2:13 PM
Seems to me you *can* use it in a generic context. Since there are no constraints on cpp_span, that protocol is almost equivalent to:

```
protocol MyCppMatrixType {
  associatedtype Span
  func slice_row(_ row_id: Int) -> Span
}
```

i.e. you just don't know anything about that result type in a generic context.

I don't know what you mean by "naturally." You can make it conform with an extension. That's a natural thing to do in Swift and I imagine we'd have prepackaged conformance declarations for std::vector to RandomAccessCollection, MutableCollection, etc.

Parker Schuh, Oct 22, 2:16 PM
Do you think those prepackaged conformace declarations would be in terms of std::vector<T> or in terms of something like VectorManualModel?

Dave Abrahams, Oct 22, 2:17 PM
The former, if nothing else because it will put less strain on the compiler not to have to go through an additional layer.

Parker Schuh, Oct 22, 2:18 PM
Makes sense, but that means that you'd have to momomorphize those generic functions.

Dave Abrahams, Oct 22, 2:18 PM
It's not clear to me whether VectorManualModel is needed as a prepackaged thing.

sorry, which generic functions?

Parker Schuh, Oct 22, 2:20 PM
like func next() -> ElementType? when conforming to IteratorProtocol.

Dave Abrahams, Oct 22, 2:21 PM
Yes, the conformance needs to be monomorphized.

Also, I guess, anything that conformance passes through needs to be monomorphized. Yuck 😞

No wait, that's not right.

Everything whose implementation might demand a new instance of such a conformance needs to be monomorphized.

Parker Schuh, Oct 22, 2:25 PM
I think those are the same rules we went over before.

Dave Abrahams, Oct 22, 2:25 PM
I hope you're right.

[Note: it's not full monomorphization that's needed; it's monomorphization in the generic parameters on which C++ template instantiations may depend].

Parker Schuh, Oct 22, 2:29 PM
I'm not sure you can avoid VectorManualModel. How will you typecheck the generic code that conforms std::vector<T> to Sequence without having some list of functions that are available in std::vector<T> ?

Dave Abrahams, Oct 22, 2:36 PM
Phase 2. It gets checked for every specific T.

Dave Abrahams, Oct 22, 2:38 PM, Edited
There's no advantage to using VectorManualModel for this beacuse these phase 2 typechecks for vector<X>: Sequence all succeed, assuming we write the conformances correctly.

Dave Abrahams, Oct 22, 2:46 PM
Ah—VectorManualModel does offer one advantage in development effort.

Dave Abrahams, Oct 22, 2:48 PM, Edited
If we don't have it, we need to make it possible to disable phase 1 typechecking of Swift generics that use dependently-typed C++ templates, for example that conformance you were talking about.

Dave Abrahams, Oct 22, 2:59 PM
But without that capability, you can't use dependently-typed C++ templates directly from Swift generics at all—you have to go through VectorManualModel or something like it.

Parker Schuh, Oct 22, 3:05 PM
phase 1 typechecking is pretty important. I'm not sure you can lower to sil without it.

Dave Abrahams, Oct 22, 3:07 PM

I can believe that.

Dave Abrahams, Oct 22, 3:10 PM
Well, OK. Let's say you can't use dependently-typed C++ templates from Swift generics. How bad would that be?

If it's problematic maybe we can create a ManualModel synthesis tool to help people create these protocols. If that becomes good enough we might turn it into a language feature for default synthesis.

Parker Schuh, Oct 22, 3:12 PM
I'm pretty convinced with your previous argument that they're basically like opaque associated types.

Dave Abrahams, Oct 22, 3:13 PM
So sorry—is that a response to my question? Can you tell me what "they" is?

Parker Schuh, Oct 22, 3:14 PM
oops, your

```
protocol MyCppMatrixType {
  associatedtype Span
  func slice_row(_ row_id: Int) -> Span // Span instead of cpp_span<ElementType>
}
```

they being "dependently-typed C++ templates"

Dave Abrahams, Oct 22, 3:16 PM
Ah… true, but while I'm sure that works for the signature, I don't think it works for bodies of generic functions/methods that have to be lowered to SIL in phase 1.

Parker Schuh, Oct 22, 3:17 PM
Well, since we don't know anything about Span (or cpp_span), we can only just return it and otherwise treat it as an opaque type.

Dave Abrahams, Oct 22, 3:19 PM
Agreed

How about my question:

> Let's say you can't use dependently-typed C++ templates from Swift generics. How bad would that be? If it's problematic maybe we can create a ManualModel synthesis tool to help people create these protocols. If that becomes good enough we might turn it into a language feature for default synthesis.

Parker Schuh, Oct 22, 3:21 PM
Are dependently-typed templates something like: llvm::InlinedVector<int, 4> ? Can you provide an example of what you think would be disallowed?

Dave Abrahams, Oct 22, 3:26 PM
A dependently typed template is one like A<X> where X is a generic parameter of a Swift generic that uses A<X>.

Dave Abrahams, Oct 22, 3:27 PM, Edited
llvm::InlinedVector<int, 4> is not dependently typed in any context because the template parameters are all concrete types, not dependent on any Swift generic parameter.

Parker Schuh, Oct 22, 3:28 PM
ah ok. I was more referencing the 4 which is a value instead of a type.

But I guess you're saying the same thing except the value is a generic metatype.

Dave Abrahams, Oct 22, 3:29 PM
?? Ya lost me here.

Let me give you a fuller example:

Dave Abrahams, Oct 22, 3:32 PM

```
// this is swift code
struct B<X> {
  // illegal: C++ template parameter depends on Swift generic parameter
  typealias C = some_cxx_template<X>
}

struct D<X> {
  typealias C = some_cxx_template<int> // legal
}
```

No metatypes here.

Parker Schuh, Oct 22, 3:38 PM
I think that that has implications for conforming vector<T> to Sequence. Especially since you'll want to do something very similar for making the iterator type but I think you can work around those limitations with VectorManualModel style helper protocols.

Dave Abrahams, Oct 22, 3:39 PM
Yes, I am assuming we always use VectorManualModel style helpers to do that.

It does mean you can't write:

struct VectorUser<X> { var impl: std::vector<X> }

instead you'd have to write:

struct VectorUser<X, V: VectorManualModel>
  where V.Element == X { // or something var impl: V }

Parker Schuh, Oct 22, 3:41 PM
In google3, that won't be a huge problem. The style guide wants most leaf code to not be generic.

Dave Abrahams, Oct 22, 3:42 PM
Hmm, I really don't buy that this is an acceptable level of ugliness in the long run.

Speaking of which, I'm going out for a 🏃

Parker Schuh, Oct 22, 3:45 PM
Ok, I don't think that this particular problem is tied to the phase 1 typechecking problem we were discussing.

Dave Abrahams, Oct 22, 3:45 PM
Interested to hear why when I get back

Parker Schuh, Oct 22, 4:25 PM, Edited
This is what I was thinking:

```
// Context:
protocol VectorManualModel : Sequence { /* contains a model of c++ methods like
init(), push_back(), etc */ }
// Because VectorManualModel doesn't reference any c++ types, the conformance to
Sequence just uses the associated types of VectorManualModel.

// Level 1: Should work by default and just build up the witness table via the
standard lookup into concrete specialized vectors.
extension std::vector<Int> : VectorManualModel {}

// Level 2: Build witness table when std::vector<int>() is passed into f(t) (or at the
first point where vector<T> is bound to VectorManualModel)
extension std::vector<T> : VectorManualModel {}
func f<V: VectorManualModel>(_ v: V) {
  print(v.map() { $0 })
}

// Level 3:
```

```
//   Every time you see std::vector<T>, think about it as an existential
//   which conforms to all the conditional conformances like VectorManualModel
//   when doing phase 1 type checking.
extension std::vector<T> : VectorManualModel {
  associatedtype ElementType = T
  // ... other associated types need to be explicit ...
}
protocol SomeOtherCppGenericModel {
   associatedtype ElementType
   func getVector() -> std::vector<ElementType> // This is known to be concrete at
some point which drive the witness table construction.
}
func f<SO: SomeOtherCppGenericModel>(_ so: SO) -> std::vector<SO.ElementType> {
  return so.getVector()
}

// Level 4: Requires monomorphization: (Here, std::vector<T> is typed checked as
SomeVector : VectorManualModel where ElementType == T in phase 1, but this function
has to be specialized in phase 2)
func f<T>(_ t: T) -> std::vector<T> {
  var v = std::vector<T>()
  v.push_back(t)
  return v
}

// Level 5: VectorManualModel is not needed. We can also write Sequence conformance
// directly in terms of std::vector features. Requires the feature of skipping phase 1
type-checking.
func f<T>(_ t: T) -> std::vector<T> {
  var v = std::vector<T>()
  v.push_back(t)
  return v
}
```

Dave Abrahams, Oct 22, 6:10 PM
s/tensor/vector/ surely. I'm analyzing this

Dave Abrahams, Oct 22, 6:14 PM
protocol VectorManualModel : Sequence { /* contains a model of c++ methods like init(), push_back(), etc */ }
and it also contains also the parts needed for conformance to Sequence, which needs to be typechecked somehow. Not sure how that's supposed to work.

extension std::vector<Int> : VectorManualModel {}
sure, no challenge there.

// Level 2: Build witness table when std::vector<int>() is passed into f(t) extension std::vector<T> : VectorManualModel {} func f<V: VectorManualModel>(_ v: V) { print(v.map() { $0 }) }

Dave Abrahams, Oct 22, 6:18 PM, Edited
nit: technically I think you need to build this witness table when std::vector<int> is first bound to VectorManualModel, which might arbitrarily many call contexts away if f is being called by another generic function.

Dave Abrahams, Oct 22, 6:22 PM
And I think this might need https://forums.swift.org/t/an-implementation-model-for-rational-protocol-conformance-behavior to be addressed 😢

An Implementation Model for Rational Protocol Conformance Behavior - Pitches - Swift Forums
forums.swift.org

Parker Schuh, Oct 22, 6:24 PM
Because VectorManualModel doesn't refer to any associated types, the conformance to Sequence can just happen in extensions to VectorManualModel.

Parker Schuh, Oct 22, 6:29 PM
What do you think runs afoul of your forums link?

Dave Abrahams, Oct 24, 1:10 PM
a) Yes, you are right: the conformance to Sequence has to just happen in extensions to VectorManualModel because we don't have the ability to create default implementations outside of extensions. Normal Swift rules.

Dave Abrahams, Oct 24, 1:14 PM
b) The inability to declare retroactive conformances of VectorManualModel is unfortunate, but I'm not sure how important that is; maybe we don't need extension VectorManualModel: Sequence because we can have extension vector: VectorManualModel, Sequence.

c) I don't know what you mean about referring to associated types. FWIW, VectorManualModel certainly needs to *declare* some associated types.

Dave Abrahams, Oct 24, 1:21 PM
d) Nothing runs afoul of my forums link. I may have misphrased what I wrote there. Probably I should have said, that the ability required to cause distinct witness tables to be generated for every specialization of std::vector is closely related to what's needed for solving that problem. Sadly, solving the forums link demands generating those witness tables at runtime, without monomorphization, in the general case. It also has ABI implications, so maybe not that closely related 😉

Parker Schuh, Oct 26, 5:14 PM
wrt c) I was saying that if you're able to write some_cpp_type<ElementType> when defining protocols in generic contexts, you don't need extra associated types to write the modeling protocol. The type that is inside std::vector<T> is still necessary.

Dave Abrahams, Oct 26, 5:15 PM
“defining protocols in generic contexts?” Example please?

Parker Schuh, Oct 26, 5:30 PM
Umm, bad phrasing. Just pointing out that while you are defining a protocol with associated types, you're in a bit of a generic context.

Consider:

```
protocol ConstIteratorModelingProtocol { associatedtype ElementType }
protocol VectorModelingProtocol {
  associatedtype ElementType
  associatedtype ConstIterator : ConstIteratorModelingProtocol where
ConstIterator.ElementType == ElementType
  func begin() -> ConstIterator
}
extension std::vector<T>::const_iterator :
ConstIteratorModelingProtocol { associatedtype ElementType = T }
extension std::vector<T> : VectorModelingProtocol {
  associatedtype ElementType = T
  func begin() ->  std::vector<T>::const_iterator
```

```
}
```

vs being able to use std::vector<T>::const_iterator like an abstract associated type symbol (in place of `associatedtype ConstIterator : ConstIteratorModelingProtocol where ConstIterator.ElementType == ElementType`) when defining `VectorModelingProtocol`:

```
protocol ConstIteratorModelingProtocol { associatedtype ElementType }
protocol VectorModelingProtocol {
  associatedtype ElementType
  func begin() -> std::vector<ElementType>::const_iterator // This
just tells us that it is a template and we can also deduce from the
protocol extension below that it conforms to
`ConstIteratorModelingProtocol` and also that ElementType ==
std::vector<ElementType>::const_iterator.ElementType
}
extension std::vector<T>::const_iterator :
ConstIteratorModelingProtocol { associatedtype ElementType = T }
extension std::vector<T> : VectorModelingProtocol {
  associatedtype ElementType = T
  func begin() ->  std::vector<T>::const_iterator
}
```

This extra slight level of conciseness (and the ability to constrain it to a particular conforming type of ConstIteratorModelingProtocol) falls out naturally from adding the level 3 feature described above.

Dave Abrahams, Oct 27, 8:36 AM
I understand what you're saying now. I'm not convinced that a) it will “fall out”, since these kinds of circularities have often caused fits for the type checker in the past even when they “should work” or b) that it's desirable. IMO at this stage we need to focus on defining the formalisms that will allow the system to work rather than concision trix; ergonomic improvements that are needed/useful will become apparent with usage.

Parker Schuh, Oct 27, 10:06 AM
Makes sense. I think it isn't just a matter of making things simpler. It does give the slightly extra information to the callee of these functions that these associated types are universal. This is just like the difference between Array<T> and a hypothetical ArrayProtocol.

Dave Abrahams, Oct 27, 10:10 AM
universal?

Parker Schuh, Oct 27, 10:11 AM

Well, that all Array<T> refer to the same thing, but two generic types conforming to ArrayProtocol might not be the same implementation.

Parker Schuh, Oct 27, 10:17 AM
Consider two modeling protocols that return std::vector<T> but written above as associatedtype VectT: VectorModelingProtocol. The compiler would not be able to know that these vectors are the same type without having to propagate some potentially awkward generic constraints.