Setup

There is a bug in the Dotty compiler:

https://github.com/lampepfl/dotty/issues/5980#issuecomment-466722242

The bug's nature is as follows. If you have two types and take an intersection over them, the types of their members that share names will also get intersected in the resulting type. This way, if the member types that are intersected are F[A] and F[B], and F[+_], the resulting type will be F[A & B], and not F[A] & F[B]. This, however, violates the Liskov substitution principle. The principle states:

if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substitutedwith any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.).

That is, if S <: T then you can do with S what you can do with T.

From the issue tracker above:

```
def fc1: C     = new C {}
def fc2: A & B = fc1


// def fy1: F[A & B] = fc1.children  // Type error
def fy2: F[A & B] = fc2.children
```

Above, C <: A & B. However, we can assign A & B's `children` to F[A & B], but not C's children. Hence the bug.

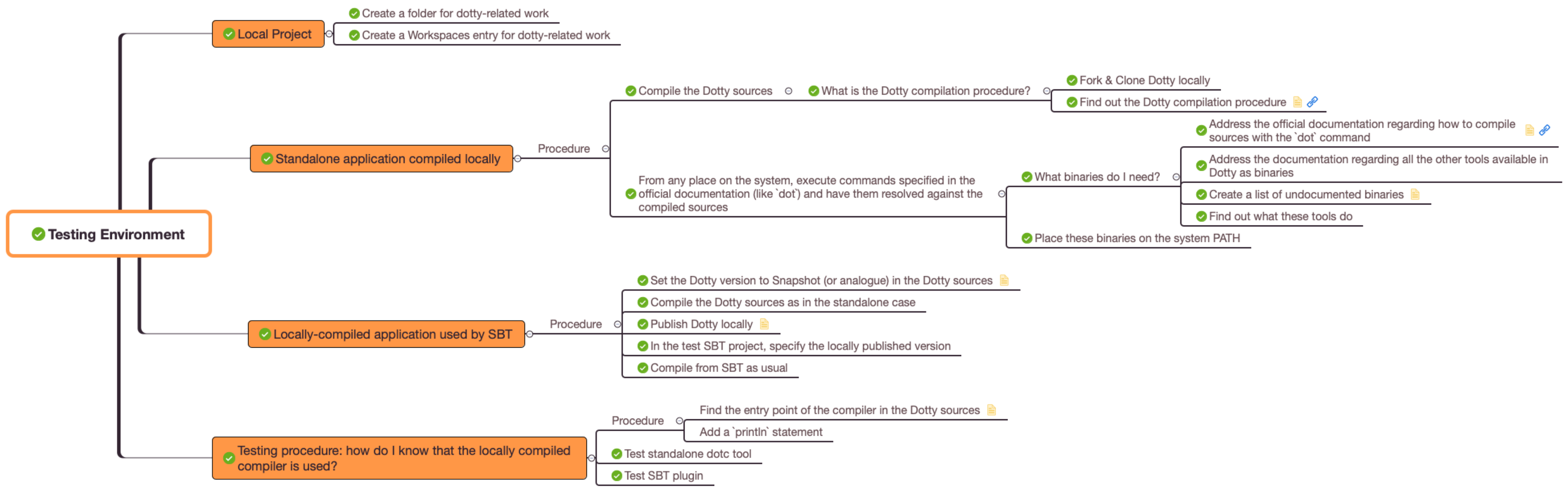
Constraint

I would like to fix the above bug in the Dotty compiler. Here are some UDEs that prevent me from reaching the goal:

- I lack testing environment. I should be able to modify & test the compiler
  - As a standalone application (`dot foo.scala`)
  - From SBT against larger projects

Addressing the Constraint

Testing Environment.xmind


✓ Testing Environment
✓ Local Project
✓ Create a folder for dotty-related work
✓ Create a Workspaces entry for dotty-related work
✓ Standalone application compiled locally
Procedure
✓ Compile the Dotty sources
✓ What is the Dotty compilation procedure?
✓ Fork & Clone Dotty locally
✓ Find out the Dotty compilation procedure
✓ From any place on the system, execute commands specified in the official documentation (like `dot`) and have them resolved against the compiled sources
✓ What binaries do I need?
✓ Address the official documentation regarding how to compile sources with the `dot` command
✓ Address the documentation regarding all the other tools available in Dotty as binaries
✓ Create a list of undocumented binaries
✓ Find out what these tools do
✓ Place these binaries on the system PATH
✓ Locally-compiled application used by SBT
Procedure
✓ Set the Dotty version to Snapshot (or analogue) in the Dotty sources
✓ Compile the Dotty sources as in the standalone case
✓ Publish Dotty locally
✓ In the test SBT project, specify the locally published version
✓ Compile from SBT as usual
✓ Testing procedure: how do I know that the locally compiled compiler is used?
Procedure
Find the entry point of the compiler in the Dotty sources
Add a `println` statement
✓ Test standalone dotc tool
✓ Test SBT plugin

Navigating the Project

Problem

I have troubles keeping track of the files I am working on in that Dotty huge code base. Precisely, I have troubles switching between Build.scala and the scripts under the bin/folder when analysing how these work. I need to have all of these files in a separate place in Sublime.

The solution is to search for a plugin by search terms like “sublime bookmark files”, “sublime selected files tab” etc.

Solution

FavoriteFiles plugin seems to do the job. I have also tried the Favorites plugin, however, it did not allow for grouping of the files. Favorite files allows for grouping, adding and removing files.

By looking at the Main.sublime-menu file, I was able to see what commands the plugin exposes. By the means of this answer, I was able to find out where hotkeys are set (precisely, Key Bindings - User menu). I set the following hotkeys to address the need of quickly adding and removing stuff from the favorite files:

```
{ "keys": ["super+g", "super+q"], "command": "favorite_files_add" },
{ "keys": ["super+g", "super+w"], "command": "favorite_files_open" },
{ "keys": ["super+g", "super+r"], "command": "favorite_files_remove" },
```

SBT Integration

Problem

After publishing the Dotty binaries via `sbt publishLocal`, I can't use them in an SBT project. The error message at first was about the inability of SBT to find dotty-doc. After doing `dotty-doc-boostrapped/publishLocal`, the error message changed to this:

```
[error] coursier.ResolutionException: Encountered 1 error(s) in dependency
resolution:
[error]     ch.epfl.lamp:dotty-library_0.14:0.14.0-bin-SNAPSHOT:
[error]         not found:
[error]             /Users/anatolii/.ivy2/local/ch.epfl.lamp/dotty-
library_0.14/0.14.0-bin-SNAPSHOT/ivys/ivy.xml
[error]             https://repo1.maven.org/maven2/ch/epfl/lamp/dotty-
library_0.14/0.14.0-bin-SNAPSHOT/dotty-library_0.14-0.14.0-bin-SNAPSHOT.pom
[error] Couldn't retrieve `ch.epfl.lamp:dotty-doc:0.14.0-bin-SNAPSHOT`.
[error] (coursierResolutions) coursier.ResolutionException: Encountered 1
error(s) in dependency resolution:
[error]     ch.epfl.lamp:dotty-library_0.14:0.14.0-bin-SNAPSHOT:
[error]         not found:
[error]             /Users/anatolii/.ivy2/local/ch.epfl.lamp/dotty-
library_0.14/0.14.0-bin-SNAPSHOT/ivys/ivy.xml
[error]             https://repo1.maven.org/maven2/ch/epfl/lamp/dotty-
library_0.14/0.14.0-bin-SNAPSHOT/dotty-library_0.14-0.14.0-bin-SNAPSHOT.pom
[error] (scalaInstance) Couldn't retrieve `ch.epfl.lamp:dotty-doc:0.14.0-
bin-SNAPSHOT`.
[error] Total time: 2 s, completed Feb 26, 2019 4:44:43 PM
```

The identical one concerning dotty-library.

Upon looking at `/Users/anatolii/.ivy2/local/ch.epfl.lamp/`, I find the following artifacts:

```
dotty-compiler_2.12
dotty-doc_0.14
dotty-doc_2.12
dotty-interfaces
dotty-library_2.12
dotty-sbt-bridge
dotty_2.12
```

There are clearly more projects under `sbt projects` in dotty, so it seems not all of them get published, yet SBT requires them all.

Solution

If there are special procedures needed to publish all the artifacts for the SBT plugin for Dotty, they must be scripted in the build files. So I searched it for the word "publish" and stumbled upon the "lazy val `sbt-dotty`" project. The project has the `scripted` task, so that if you run `sbt-dotty/scripted`, it seems to publish everything you need for integrating with SBT, along with a SNAPSHOT of the SBT plugin itself.

sbt-dotty/scripted takes too long to execute

Problem

The command takes >30 minutes to execute. This poses a problem because I need my modifications to Dotty to be usable in test SBT projects fast.

Proposed Solution

- [x] Analyze the task's output. Identify what exactly tasks are executed
- [x] Locate the long-running tasks (essentially non-publish tasks) in the build scripts
- [x] See if they are controlled from any flags I can unset
- [x] Either unset the flags or decide further from here

Solution

`scripted` task turns out to be an SBT standard task which is meant specifically for testing purposes. Hence using it for publishing goes against the spirit of this task. Hence I looked through the build script for the mentions of the projects I want to publish. The idea is to find a task that publishes all the projects in batch, and that this task will need to refer to the projects by name to publish them.

This way I found the dotty-bootstrapped/publishLocal command. It publishes all the libs without testing them. It seems you also need to run sbt-dotty/publishLocal to publish SBT plugin separately. The `boostrapped` part seems to be needed by the SBT plugin: it resolves the artifacts which are "bootstrapped".

Entry Point

Problem

I want to find the entry point of the compiler. Precisely, the `main` method that is actually executed when the `dotc` command is run. The problem is, the code base is quite big, the search for the `main` method returns a lot of results and it is not obvious how exactly the `dotc` script is generated, to what exactly it refers.

Proposed Solution

Ultimately, `dotc` must execute a Java class. Hence my task should be to obtain the name of that Java class and search the sources for that name. The name should be possible to obtain from the code that actually generates the dotc file. This must happen during the build, hence I should look through the build files for any mentions of the dotc file. Alternatively, dotc is already included in the repo. Hence I should also take into account the info from the file included in the repo.

So:

- [ ] Search all of the build files for any mentions of `dotc`; look for a Java class name mentioned together with it
- [ ] Search the standard dotc files included in the repo for the Java class name

Test files

When searching for the entry point of the compiler via naive regex search for the `main` method, I stumbled upon a bunch of files with `main` method scattered across the code base. They contained very ad-hoc code evidently used for in-place testing. I should have a look at them – maybe I can use them for extra “eyes” on what’s going on in the compiler internals.

It is hard to look for entry points

UDE
I want to find the entry point to the compiler, but the code base is so large and I don't know where to search.

Constraints
Why do I not know where to search? Because the code base looks uniform to me.
Why does it look uniform to me? Because I haven't defined the notion of the entry point and how it relates to other parts of the codebase. I haven't named what I am looking for and its relationships to the rest of the codebase, hence uniformity.

What am I looking for? What is the entry point?
An entry point from one program to another program is the call site where the first program calls the second program.

Hence to find the desired entry point, I need to search in the first program's code base for the mentions of the second program's code base.

What exactly mentions?
Program logic flow-based mentions, dependent on the task the program is performing. In case of dotty, the task is the compilation of the provided sources. The flow of the program is, in case of dotc, the execution of the `dotc <sources>` command followed by the compilation action. In case of SBT, it is the execution of the `compile` task followed at some point by the call to the dotty compiler – either mediated or not.

Hence, in order to find an entry point, I need to start from the actual start of the flow – myself calling a bash script or an SBT command. Then I need to trace these actions to the Scala sources. Then I follow the sources until the logic escapes from one program to another.

Scripts and frameworks
Problem: how do I link the dotc script to the sources, or the sbt compile call to the sources? The former obviously makes call to some binary. The latter most probably utilizes the SBT framework to determine what to do on the call.

Solution Plan
The difficulty with the dotc script is to see what exactly binary it calls. The solution is to look through the dotc script sources to see which binary it calls. If the binary is executable, find how the executable is generated.

The difficulty with the SBT framework is to see what exactly overrides change the behavior of the compile command. The solution is to look through the SBT plugin codebase (which is known to perform the override) for any mentions of the "compile" command. Otherwise, list all of the overrides and see which of them are reasonable to explore next.

I can't find dotc in Build.scala

Why did I expect to find it in first place?
I was working under assumption that one needs to specify the file name in order to generate it from a build script. Essentially, SBT is a function that takes some input information and outputs the binaries. The input information defines what the function outputs. My assumption was that the file name as well as the main class of the compiled jars should be the part of this information. Most probably they are the information required for the SBT to output dotc – however, another assumption was that this information should have been specified by the programmer. There is another alternative – this information may be generated by SBT based on the other information the programmer provided.

What is the goal?
The goal is to find the class name which is used to generate dotc binary.

Given the above information, where might this information be?
Either specified in Build.scala by a person or derived from what a person specified by SBT. There's no trace of this info in Build.scala. Hence, it must be derived by SBT.

How do I obtain this information?
If the information is derived by SBT, I need to look at the derivation rules. I need to see what information exactly is specified manually and what rules exactly are used to ultimately turn it into the binary file. The specified information is found in the build scripts. The rules are the standard SBT rules modified by any SBT plugins in use. Therefore, I need to study how the used plugins work and what exactly info SBT has.

Another problem is that I don't know exactly what part of the build file is responsible for dotc file. The build file is modular, consists of projects. I will greatly simplify my life if I only focus on the project responsible for dotc generation.

How do I find out what project to zero in?
Clean everything. Use best guess based on the names of the projects & their commands available. Run these commands. See which one leads to the generation of the dotc file.

Procedure

- [ ] Clean everything
- [ ] Find out which exactly most specific (non-aggregating) command generates the dotc binary
- [ ] Study what information does the command have
- [ ] Study which rules apply to the command; what the command does; where it comes from; which plugins modify it

---

Solution

My assumption was that the dotc file is the binary. The one under the "/dist-bootstrapped/target/pack/bin/dotc" folder. I assumed that because everything under the "target" folder is supposed to be synthetic. Hence if it is an executable file, I assumed it to be binary. Well, not every synthetic executable file is a binary one.

Dotc is a bash script, and it has the main compiler class there as plain text.

SBT Compiler Bridge

I'm now trying to make a modification to the compiler as seen from SBT. So that the "Hello from Dotty" message is seen when compiling from SBT. Dotty is injected into SBT via an SBT plugin, so I headed to the sources of that plugin. The plugin itself turned out to be quite small. It doesn't really have much logic to it – instead, it contains the info which looks like an input to some larger function. The function being the SBT machinery itself which is configured by that plugin file.

The compiler JAR is specified there, as dotty-compiler blah blah. "dotty-compiler/publishLocal" publishes a JAR with that name, so probably the result of packaging `dotty-compiler` project is used as a value there.

The problem is, I do not know what exactly logic uses this config entry. The path of the entry in question in the hierarchy is:

compilerJar <- ScalaInstance <- scalaInstance task <- projectSettings <- AutoPlugin

The doc on AutoPlugin's projectSettings says:

> The Settings to add in the scope of each project that activates this AutoPlugin.

So basically, whichever project activates the plugin gets scalaInstance overriden. The problem is, I do not know how exactly the Scala instance is utilized by SBT.

Well if that's a configuration, probably I need to find the function and read how it is used 🙂 that is, look at the sources of the compile task of the SBT.

Solution

The above reasoning (i.e. find the function that uses the target configuration) turned out to be correct.

I cloned SBT. I searched it for `ScalaInstance`, which is the object containing the `compilerJar`. Immediately I got a snippet that uses ScalaInstance to construct & run an object with a tell-tale name `RawCompiler`. So the question became, how does `RawCompiler` use ScalaInstance?

Maven search by full classname `sbt.internal.inc.RawCompiler` revealed that this object is present at the SBT's Zinc compiler. Cloned it, searched for RawCompiler. Turned out to be a tiny 100-liner, which invoked the `process(Array[String])` method on `dotty.tools.dotc.Main` via reflection. Upon inspection, this method, defined in the Dotty's Driver class (parent of Main) had the following comment:

This overload is provided for compatibility reasons: the
  ** `RawCompiler` of sbt expects this method to exist and calls*
  ** it using reflection*

Take-aways

The piece of knowledge I gained from this task is that the entry point to the compiler is `process(args: Array[String], rootCtx: Context): Reporter` method. It invokes the `doCompile` method which supposedly does the compilation.

Another thing is the skill of studying complex code bases. Turns out that there are two types of code you can encounter. One is data, other is derivation. Data generally contains no logic, just some values being set. Derivation transforms data, derives one data from another.

What causes the bug?

The current constraint is that I do not know what exactly causes the bug. There is some chunk of the compiler’s logic which makes that decision to make the F[A & B] type in place of F[A] & F[B] type. If I know that piece of logic, I can modify it to make the decision I want.

How do I find out this piece of logic?
I need to devise a simple example which reproduces the bug. This example is the input to the compiler function. I then need to trace it through the compiler internals to see at which point that decision is made.

First steps: Navigation

Working through process-setup-doCompile methods. I'm struggling with the following:

Problem	Solution
Circling though these methods is hard. I need to manually scroll from one method to another when I want to switch and waste my energy trying to visually spot it.	Line bookmarking. I need a plugin for sublime that allows me for: • A hotkey to add a bookmark • A hotkey to list all bookmarks • Possibly grouping the bookmarks
I have troubles **tracking the data that goes in and out in the methods (dataflow)**. In the setup method, I spent a few minutes tracking down a variable that was actually fed into the method as an argument.	I'd love a visual solution that helps me to keep in mind the big picture. Ideally a line diagram like in category theory.

- [x] Find a bookmarking plugin for sublime & set the hotkeys as specified above
- [ ] Find a visual solution to keep track of the dataflow through logical nodes

---

I settled for the Sublime Bookmark solution. Same keymap as with Favorite files, since the functionality is the same but with different precision:

```
  // Sublime Bookmark
  { "keys": ["super+d", "super+q"], "command": "sublime_bookmark", "args" :
{ "type" : "add" } },
  { "keys": ["super+d", "super+w"], "command": "sublime_bookmark", "args" :
{ "type" : "goto" } },
  { "keys": ["super+d", "super+r"], "command": "sublime_bookmark", "args" :
{ "type" : "remove" } },
```

Regarding the second problem, the dataflow, I uncovered something. All this time I was going for constraint elevation while not seeing how exploitation can be done.

The second problem is visualisation of the dataflow. Keeping track of it. An automated solution tailored under my needs is not available, and its manual creation requires too much work. For a considerable amount of time, I was focusing on how to create such a solution with a minimal effort.

Why create such a solution if I can instead figure out how to make the dataflow tracking work with my current system? In other words, exploitation of the constraint instead of its elevation.

Precisely, I can shift my priorities. Instead of trying to make it to the bug site as fast as possible, I can focus on feeling myself at home as I go. That is, having an understanding of each link involved in the process as well as the context (the source, the package) it operates in.

Focus

So, following my post on Navigation, I tried "feeling myself at home wherever I go in the codebase". That is, understanding the purpose of the files and methods I encounter. That indeed worked fine – after I applied a bit of effort in that direction. That is, subordinated my analytical work to the decision of exploitation of the dataflow analysis constraint.

The strategy worked fine indeed at the Driver scope. However, naturally, very soon it led me outside the Driver. And into the Run and Context. Now, Context is a quite formidable behemoth. Run seems to also be packed with some heavy logic. If I continue focusing on "feeling myself at home" with these entities, it will take too much time.

So, I'm having an UDE here: the entities I am working in now are too large to analyze as previously. Because they are scattered across too many files and occupy too many code. And, because I can only analyze so much code in my time. **Speed of analysis is the constraint now**.

The big goal is of course to fix the bug. In order to fix the bug, I need to know how the buggy example is transformed by the compiler and where exactly the erroneous sequence of logic starts.

Hence, I can exploit my constraint of limited speed of analysis by focusing only on the parts of the system that matter to the bug. That will yield the information about the sequence of logic that yields the bug.

Which parts of the system will yield this information? This is the phase of the compiler on which the bug happens. The internals of the phase.

Which is that phase? To find that out, I first need to understand the bigger picture of how the phases are run in general. Then, I need to understand the error mechanics of the compiler: what happens internally when the error in question happens. Then, I can exploit this error logic to add a debug trace there to see on which phase the error originates. The assumption here is that the error originates during the runs of the phase on which the bug occurs – which is reasonable to assume, because I also assume the error is reported as soon as it happens, and it happens when the compiler checks the types during the assignment as in the buggy example.

So, I am going to exploit my analysis speed constraint by directing my analysis on the following questions:

- [x] How are phases run together?
- [x] How does the output of one phase gets as an input to another phase?
- [x] How (and whether) does the compilation error in question short circuit the phases sequence?
- [x] How can I trace the short circuit via println?

How are phases run together?
The phases are run together in the Run class's compileUnits method. It has a loop inside

its runPhases submethod.

How does the output of one phase gets as an input to another phase?
The following line runs the phase and assigns the result to a Run-global var:

```
units = phase.runOn(units)
```

The units is a collection of CompilationUnit instances.

How (and whether) does the compilation error in question short circuit the phases sequence?
It seems that the compile errors do short circuit. This happens from individual phases. Precisely, what happens is that the error message gets printed immediately and all the subsequent phases get their isRunnable flag set to false. Only phases which have their isRunnable flag set to true will be executed.

How can I trace the short circuit via println?
Traced it trivially using the above info. The error happens on the phase with the name `frontend`. The entry point is `runOn(List[CompilationUnit])` method. Hence, I should keep tracing the logic through that method.

Evernote Observation

While looking through the Typer, I narrowed down my scope to the Typer source. At some point, I got it that typedExpr calls typed the job of which is to call typedUnadapted which is a large `match` fork that invokes the appropriate typing method for a given tree. However, I completely lost track of where typedExpr was called from in first place. I then returned to the previous entry, Focus, in this log. There, it mentioned the Forntend phase and the entry point to this phase which is invoked from the Run method. I then navigated to that entry point and recalled without a problem how exactly the call to typedExpr happens.

So, having such a log where I record the happenings on my journey is actually a helpful thing. It helps me recall the bigger picture on demand, since its attention span is ***unconstrained*** (unlike that of my memory).

Also, it seems that having a log helps to make sure the chaos that accumulates when traversing into an unknown territory gets converted into order, gets made sense of.

Reflection on today's work

After Focus, I arrived at the Typer class, typedExpr method. I did so following the speed-oriented analysis strategy. That is, I started from the environment of the compiler running the phases. I considered it to be a behemoth, hence I decided to make my exploration laser-sharp and zero in on the part of the codebase that has the bug.

After arriving at the Typer, my constraint became the same as in the Navigation. There, I just arrived into the Driver class, the entry point of the compiler. My constraint there was lack of the big picture detailed enough to continue zeroing-in. The process of laser-sharp focus while following how the data transforms yields a good speed zeroing in on relevant pieces of code. However, this speed depends on the knowledge of the big picture, since you need to make decisions while moving with focused speed. You need to make these decisions based on some information. Unfortunately, the zeroing-in mode doesn't yield much of the big picture knowledge.

Hence, I started solving the big picture constraint via the same technique as in the Navigation. Don't follow the dataflow, try to capture the big picture. The Typer turned out to be a catalogue of methods that type various parts of the AST, plus a few methods that don't follow that pattern.

Once I understood that, the speed of analysis again became the constraint. At that pace, it will take me too much time to detail the picture enough to come to the bug site. Hence, I switched to zeroing-in again as in Focus. I did so by following the dataflow from typedExpr method and trying to trace at which point the error while compiling the buggy example appears.

Btw the error there is the correct behavior:

```
def fy1: F[A & B] = fc1.children  // Type error
def fy2: F[A & B] = fc2.children
```

fc1.children is `F[A] & F[B]`. `fc2 == fc1`, hence the second line should also show an error. The idea of the current analysis is to track down the reason of why the (correct) error happens in the first line. The idea is then to see how `fc1` and `fc2` differ at the error site so that the second line does not produce an error.

Further plans

So far I've learnt that the error happens after the `adapt` call:

adapt on: this.fc1.children against: F[CovariantExample.this.A & CovariantExample.this.B]

The method is a behemoth of ~500 LoC, so tomorrow the task will be to further zero-in on the bug at that method. Probably I'll need to balance the big picture and the laser-sharp modes again.

Pretty Printing

While doing progress described in the Reflection of today, in the part where I switched from big picture to lazer-sharp analysis modes during the Typer phase, the constraint appeared which slowed down my analysis. Namely, the trees I println'ed were hard to read.

I elevated this speed constraint by looking up some standard tools Dotty has for trees exploration. On the official site, I found a small section on pretty printing which elevated the constraint.

Tracing from the error site

As per Reflection, I am tracing down the reason why the error happens. The first step was to find the error site, the second step is to backtrack on the parameters that make the error happen (or not happen).

`adapt1` method turns out to be a small `match` on the type of the submitted tree along with a ton of helper methods that are invoked from that match. The `match` is done on the type of the tree, which already contains the erroneous type that yield the bug. The assumption here is of course that nowhere does the `adapt` method recalculate the type.

So I found the parameter that causes the error – now the objective is to find out where it is set for the Liskov violation example.

typedSelect

So I am tracing the erroneously inferred type to the site where it is set. I went from `adapt` back to `typed` and from there – through `typedUnadapted` to `typedSelect`'s child method, `typeSelectOnTerm`. I was guided by my sufficient knowledge of the big picture and the `println` traces.

typedSelect chaos onset

The first thing typeSelectOnterm does is it splits the tree into the qualifier and the name. Then, it typechecks the qualifier to have the type `?{def name: ?}` (it must be def, but I don’t remember exactly). It gives us the typed qualifier – e.g. C or A&B.

Then, the chaos starts to onset. At this point, we have the tree and the typed qualifier. Next, we’re calling a 3-arg typedSelect on the tree, the prototype and the qualifier. This means that for `def fy1: F[A & B] = fc1.children`, the tree is `fc1.children`, the prototype is `F[A & B] and the qualifier is `this.fc1` of type `C`.

After the call to `typedSelect`, we are having the entire typed select tree with the type that causes the bug. So, (`fc1.children`, `F[A & B]`, `this.fc1: C`) => `fc1.children: F[A] & F[B]`. So, `typedSelect` somehow transforms the untyped entire select tree, the expected type and the typed qualifier tree into the typed entire select tree.

Information wise, how does it do so? Well, we have the type of the qualifier and the name accessed in that qualifier. Plus we have the expected type which can be used for inference. So, we can have a look at the qualifier’s type, look at the name being accessed and figure its type. Hence, it is the qualifier type which should have the information about the types of its members.

Then, if we have a look at the actual `typedSelect` 3-arg call, it is a composition of two other calls: `cpy.Select(tree)(qual, tree.name)` and `assignType(copied, qual)` (there’s a 3rd call that doesn’t do anything but check a certain Java-related condition, hence assumed to be irrelevant.

Judging from the names and the information passed around, cpy must be a copying method. `cpy.Select` takes the untyped tree, its typed qualifier and the tree name. Hence, assuming no information loss, the result will be a tree with the select tree with the typed qualifier and untyped tree name. Hence untyped tree.

`assignType` then takes this partially typed tree and a qualifier. Why would the same info, `qual`, be passed again in the `assignType` method? This is strange, this is an unknown. The output of the `assignType` is the fully typed tree, one way or another.

So, the job of accessing the member of the given type and inferring its type must be performed in the `assignType`. Hence I need to see how exactly it is done in the `assignType`, taking into account the unknown of the redundant information passed around. First, of course, I need to double-check the data passed into the method.

There is no redundancy in the info passed around:

```
Type checking fc1.children; copied: this.fc1.children; copied qualifier
this.fc1; original qualifier this.fc1 =>
CovariantExample.this.C(CovariantExample.this.fc1)assigned:
this.fc1.children; => (F[CovariantExample.this.A] &
F[CovariantExample.this.B])(
  CovariantExample.this
.C#children)
```

It turns out that the copying function returns not a partially typed but a fully untyped tree. It just makes the qualifier into a full path rather than a relative path (this assumption is made based on the `fc => this.fc` transformation). Hence, the assumption of no information loss in the previous section is false, the type information is lost during copying. Hence, the qualifier with its type is necessary when calling the `assignType` method and is needed for the type information of the qualifier.

Chaos fades away after writing the above down. Probably the stuff like the false assumption leading to unexpected behavior (same info passed twice into the method, or so it appears) happens in the "background", in subconscious and leads to the feeling of inconsistency. When brought into the denotational, in the conscious, it allows to track down the false assumption and restore the picture of the world.

I guess the **constraint here was chaos accumulation speed** that spiked up.

Denotation.&

So `assignType(tree, qual)` assigns the type to the Select tree while knowing the type of the qualifier of this tree. It does so by accessing the member of the type by name (contained in the tree).

`assignType` lives in `typer/TypeAssigner.scala` and is a few wrappers around the `selectionType` method. The wrappers in question provide special treatment to array-related methods and check whether the target name is selectable.

The `selectionType` method calls the `member` method on the qualifier type at some point. The method has a tell-tale name and is supposed to get the member of the type with the given name. Now, the member has the type under the `info` method.

The `member` lives in `core/Types.scala` and is a few wrappers around `findMember` method. This method implements the member lookup for various types recursively. In that method, there is a recursion branch `goAnd` which resolves the
members of intersection types.

To summarise, I started from the `assignType` which computes the type of the Select tree from the type of the qualifier of that tree and the name we are selecting. I found out that it does so essentially by calling the `findMember` method on the qualifier and passing the given name to it. The `findMember` method finds the member with the given name, and that member is typed. The `findMember` method has different branches depending on the type you are calling it on. There is a branch for the intersection type. I need to analyse that branch to determine why the compiler takes too much freedom when computing an intersection of covariant types and makes that intersection F[A & B] instead of F[A] & F[B].

The branch is implemented as `go(l) & (go(r), pre, safeIntersection = true)` – so, the member's type of the intersection is computed from the member types of the types that participate in the intersection. Hence, next step would be to analyse that `&` method which lives in the `Denotation` trait in `core/Denotations.scala`.

So, in the `&` method, the method that does the job is the `mergeSingleDenot` submethod. I need to analyse it next.

Arrival at distributeAnd

`mergeSingleDenot` first tries to select one of the two types before trying to compute their intersection. At some point, it does so by calling `infoMeet` method on the two types (also resides in Denotations). This method delegates the merger (at least at its part concerning our bug) to the `&` method defined on the `Type`. This method is very simple: `ctx.typeComparer.glb(this, that)`. So, it uses the context's TypeComparer to compute the greatest lowest bound of this and that types.

`TypeComparer` lives in `core/TypeComparer.scala`. Its `glb` method does a deal of checks to see if it can just use one of the operand types as GLB or if it can drop one of the types because it is a supertype of another type (hence their GLB is supposed to be the subtype). Otherwise, this method calls `andType` method which computes the intersection of two types.

Now, first thing `andType` does is a call to `distributeAnd`. Which precisely tries to push the type arguments of two types under the same brackets (`F[A & B]` from `F[A]` and `F[B]` type arguments). It then determines whether the result of `distributeAnd` exists and, if so, uses it.

`distributeAnd` method looks at whether the two types supplied to it are `AppliedTypes` (that is, `F[A, B...]` form?), and whether the name parts of the applied types are equal. If so, it computes the GLB of the pairs of the arguments via `glbArgs` call which takes two lists of types and returns one `List[Type]`. Then, if all the types in the resulting list exist, it applies the intersection of the two applied types' names to the intersection of their arguments. Otherwise, it is a `NoType` which will be a clue for the `andType` method above that it cannot use this intersection.

To summarise, merger of two denotations involves computing the GLB of their types. This GLB essentially is computed via taking an intersection of the two types involved, however, this intersection also attempts to distribute the type arguments under the same bracket.

In the Liskov example, when dealing with `F[+_]` and the denotations of types `F[A]` and `F[B]`, `distributeAnd` is able to succeed. This is because `glbArgs` computes `A & B` type which exists. However, when dealing with `F[_]` and the same denotations, `glbArgs` computes a `NoType` that does not exist.

Hence, my objective would be to determine what makes the above difference in the `glbArgs` method. The variance must add some extra info to the type arguments of `F` that are passed to `glbArgs` – I need to determine this difference in information and how it influences the `glbArgs` logic.

Bug Nature

Here we go. `glbArgs`, the following snippet:

```
        val v = tparam.paramVariance
        val glbArg =
          if (common.exists) common
          // else if (v > 0) glb(arg1.hiBound, arg2.hiBound)
          // else if (v < 0) lub(arg1.loBound, arg2.loBound)
```

In comments, it says:

```
   *    - if corresponding parameter variance is co/contra-variant, the
glb/lub.
```

Is it true?

We are trying to compute GLB of F[A] and F[B].

Is it true that its GLB would be F[glb(A, B)] if F is covariant?

```
glb(F[A], F[B]) <: F[A] and glb(F[A], F[B]) <: F[B] and
if X <: F[A] and X <: F[B] then X <: glb(F[A], F[B])


assume glb(F[A], F[B]) = F[glb(A, B)]


glb(A, B) <: A and glb(A, B) <: B
hence glb(A, B) <: A & B


covariance: F[glb(A, B)] <: F[A & B]
F[A & B] <: F[A] & F[B]


hence there exists X for which X <: F[A] and X <: F[B] and F[glb(A, B)] <:
X, hence glb(F[A], F[B]) != F[glb(A, B)]
```

Contravariance:

glb(FA, FB) = F[lub(A, B)]

FA < Flub
FB < Flub

FA & FB < Flub

Now I need to go through tests and fix them...

Failed tests

```
[info] Test dotty.tools.dotc.FromTastyTests.posTestFromTasty started
-- [E007] Type Mismatch Error:
/Users/anatolii/Projects/dotty/dotty/tests/pos/intersection.scala:22:20 -----------------
22 |  def g: C[A | B] = f
   |                    ^
   |                    Found:    intersection.C[intersection.A] & intersection.C[intersection.B]
   |                    Required: intersection.C[intersection.A | intersection.B]
```

class C[-T]
C[A | B] <: C[A] & C[B] by here.

```
-- Error: /Users/anatolii/Projects/dotty/dotty/tests/pos/templateParents.scala:17:10 -------
----------------------------
17 |  val x = new D with E
   |          ^
   |          conflicting type arguments in inferred superclass templateParents1.C[String] &
templateParents1.C[Int]
-- [E007] Type Mismatch Error:
/Users/anatolii/Projects/dotty/dotty/tests/pos/templateParents.scala:19:27 --------------
19 |  val y: C[Int & String] = x
   |                           ^
   |                           Found:    (templateParents1.D & templateParents1.E)
(templateParents1.x)
   |                           Required: templateParents1.C[Int & String]
```

Illegal: x: D & E <: C[String] & C[Int]

```
-- [E007] Type Mismatch Error:
/Users/anatolii/Projects/dotty/dotty/tests/pos/reference/intersection-types.scala:29:26 -
29 |  val ys: List[A & B] = x.children
   |                        ^^^^^^^^^^
   |                        Found:    List[intersectionTypes.t2.A] & List[intersectionTypes.t2.B]
   |                        Required: List[intersectionTypes.t2.A & intersectionTypes.t2.B]
[====================================] completed (1724/1724, 4 failed,
173s)
```

Illegal, as per my issue

Theory Constraint

I was able to solve the “bug” alright. However, my assumption that `glb(F[A], F[B]) != F[glb(A, B)]` was based on the assumption here, that `F[A & B] <: F[A] & F[B]` is always true for the covariant case, however, `F[A] & F[B] <: F[A & B]` is not always true.

The following sketch may prove this assumption to be false:

```
F[X] <: F[A] & F[B]
F[X] <: F[A] ^ F[X] <: F[B]  – because (F[A] & F[B] <: F[A]) ^ (F[A] & F[B]
<: F[B])
(X <: A) ^ (X <: B)          – because of covariance (if X >: A then F[X]
>: F[A])
X <: A & B                   – because of subtyping rules of intersection
types
F[X] <: F[A & B]             – because of covariance

Hence

F[X] <: F[A] & F[B] => F[X] <: F[A & B]

Since it is proven here that F[A & B] <: F[A] & F[B]:

F[X] <: F[A & B] => F[X] <: F[A] & F[B]

Hence

F[X] <: F[A] & F[B] <=> F[X] <: F[A & B]

Hence

F[A] & F[B] =:= F[A & B]
```

The assumptions I am making above:

- If `Y <: F[A] & F[B]` then there is a type X for which `Y <: F[X]`
- For all types `X`, if `X <: A <=> X <: B`, then `A =:= B`.

The above assumptions may be untrue.

Plus, the tests & the documentation & the comments explicitly say that the distribution of the type parameters must happen. The variance case is explicitly documented & included in tests & comments. This adds some weight to the fact that the behavior may be desired and that in fact `F[A & B] =:= F[A] & F[B]`.

So the constraint here is my knowledge. I don’t know much about the type theory. The Liskov principle violation is most certainly a bug, however, it may be caused by different factors. My assumption was that variance was to blame. However, it may be not.

The situation here is as follows. If indeed for all covariant `F[+_]`, `F[A & B] =:= F[A] & F[B]`, then my solution of removing the variance from GLB inference is wrong because they have the theoretical foundation to distribute the parameters. In that case, however, the Liskov bug holds, and is caused by the fact that the type parameters are treated differently than the concrete types.

If however the above equality is not true, then the variance is indeed the bug.

Hence, my goal right now should be to prove or disprove the above equality. I can prove it via taking into account all of the existing axioms and inference logical rules that are in play in Dotty & Scala. I can disprove it by providing a counter-example, for this I'll also need to take into account the above rules & axioms.

My theoretical knowledge is my constraint here. I don't know much of these rules & axioms. I can exploit this constraint by explicitly writing down all the laws I do know and trying to accomplish the goal based on these. If this fails, I can elevate the constraint by reading some papers on DOT calculus and seeing what they say on variance and intersection/union types.

If all of the above fails, I can comment on the issue with what I have, and hopefully obtain some information.

Theoretical Soltuion

So I wasn't able to find a solid proof that F[A] & F[B] <: F[A & B] for covariant types. However, I think I have a good heuristics so that I can solve & submit the challenge.

	type check (assignment)	**type computation (intersection)**
traits	+	+
type parameters	-	+

Above, there's an analysis on when F[A] & F[B] <: F[A & B] is true. So, during type computations for intersection (when determining the types of the members of A & B), the compiler will always go for F[A] & F[B] <: F[A & B]. For typecheck (when determining whether an expression typechecks to a given type), the compiler will only go for it in case of traits.

Two solutions are possible for the above. First one is to make the above table red. The motivation is, there is no theoretical foundation for the subtyping rule in question. However, there are tests for that rule and it is featured in the docs and the examples. Also, in code, it is consistently present everywhere but in case of the type parameters. Hence the second one is to make the type check pass for the type parameters.

What I need to do is to backtrack a bit from where I was during Bug Nature, and to work on the example at the very beginning of the GitHub issue. The question is, what's different when type checking on traits vs type parameters in the covariant example.

Subtyping

Working through the following example:

```
trait Covariant[F[+_]] {
  trait G[+X]


  def fx: F[A  &   B] = fy  // Type Mismatch Error
  def fy: F[A] & F[B] = fx


  def gx: G[A  &   B] = gy
  def gy: G[A] & G[B] = gx
}
```

The error appears to originate in the Typer, in the `adapt1` method's `adaptNoArgsOther` submethod. At some point there, it performs the `if (tree.tpe <:< pt)` subtyping check. In the example above, all assignments but the first one pass this check. So I study the `<:<` method.

That method is implemented by `core/TypeComparer.scala`. It seems like the bulk of the subtyping method is the `recur` method of that file. It is a ~1000 LoC behemoth that defines when the first argument type is a subtype of the second argument type. I'm currently studying this method to see where the behaviors of the first assignment and the 3rd assignment above start to diverge.

In thirdTry->compareAppliedType2 there's a (NESTED) match that distinguishes between TypeBounds and ClassInfo. Traits are ClassInfo and type params are TypeBounds.
The TypeBounds branch then leads to compareLower which in turn does `fallback`:

```
      def fallback(tyconLo: Type) =
        either(fourthTry, isSubApproxHi(tp1,
tyconLo.applyIfParameterized(args2)))
```

So fallback seems to try the `forthTry` or else compare `tp1` against the lower bound of the type bounds in question. In case of `F[+_]` in our example, the lower bound is nothing, so unlikely to give us the positive result (not tested this yet). So I guess `fourthTry` is our only hope here…?

In case of ClassInfo, tryBaseType method is used. There's a notion of base type in Scala, which I don't get yet. However, `tp1.baseType(cls2)` will be computed: that is, some base type of `tp1` will be computed given class of `tp2`. This will eventually be delegated to baseTypeOf of `/core/SymDenotations.scala`. There, `AndOrType` will be selected when working on And or Or types. For these two types, the base types of their constituents will be computed and then merged under the corresponding operator (& or |).

For our example, the types of constituents will remain visibly the same (`G[A]` and `G[B]`), however, `G[A] & G[B]` is `G[A & B]`. This base is obviously a subtype of the target `G[A & B]`.

So, when dealing with classes, base types will be used (whatever they are), and, under the & and | types, the corresponding operator will be used to merge the two base types.

Hence I need to see why this merging logic is absent in case of type bounds, under `forthTry`.

Do you assert it?

So, fourthTry, AndType branch, the return value is `either(recur(tp11, tp2), recur(tp12, tp2))`. Basically, `F[A] & F[B] <: C` is true iff `F[A] <: C` or `F[B] <: C`. However, we may want to distribute the type arguments for covariant `F`:

```
either(either(recur(tp11, tp2), recur(tp12, tp2), recur(tp11 & tp12, tp2))
```

Or

```
recur(tp11 & tp12, tp2)
```

Problem is, when I do it this way, the tests fail. Errors like:

```
assertion failure for Number <:< Character, frozen = false5, 0 failed, 6s)
assertion failure for Number & Comparable[_ >: Short & Byte <: Short |
Byte] <:< Character, frozen = false
java.lang.AssertionError: assertion failed while compiling
/Users/anatolii/Projects/dotty/dotty/tests/run/numbereq.scala

assertion failure for Object <:< Serializable, frozen = true 1 failed, 8s)
assertion failure for AnyRef <:< Serializable, frozen = true
assertion failure for AnyRef & Product <:< Serializable, frozen = true
assertion failure for T <:< Serializable, frozen = false
assertion failure for T <:< Serializable, frozen = false
assertion failure for T <:< Product & Serializable, frozen = false
assertion failure for Array[Product & Serializable] <:< Array[T], frozen =
false
java.lang.AssertionError: assertion failed while compiling
/Users/anatolii/Projects/dotty/dotty/tests/run/i1284.scala
```

These seem like coming from the compiler… Quite weird…

Trying to reproduce that test

So far I tweaked the test method to run the files from a custom directory where I placed only one demo file. It says:

```
object Test {
  def main(args: Array[String]): Unit = {
    val res = 0 match {
      case 0 => new java.lang.Short(0.toShort)
      case 0 => new java.lang.Byte(0.toByte)
      case 0 => new java.lang.Integer(0)
    }
  }
}
```

This is a minimization of one of the test cases that yield an error. After the following command:

```
sbt:dotty> dotty-compiler/testOnly dotty.tools.dotc.FromTastyTests --
*runTestFromTasty
```

The code above yields the following errors:

```
[warn] Multiple main classes detected.  Run 'show discoveredMainClasses' to
see the list
[info] Test run started
[info] Test dotty.tools.dotc.FromTastyTests.runTestFromTasty started
assertion failure for Short <:< Integer, frozen = false1, 0 failed, 3s)
assertion failure for Short & Byte <:< Integer, frozen = false
…
assertion failure for Short & Byte <:< Integer, frozen = falseiled, 4s)
…
assertion failure for Comparable[Integer] <:< Comparable[_ >: Short & Byte
<: Short | Byte], frozen = false
assertion failure for Integer <:< Comparable[_ >: Short & Byte <: Short |
Byte], frozen = false
assertion failure for Integer <:< Number & Comparable[_ >: Short & Byte <:
Short | Byte], frozen = false
java.lang.AssertionError: assertion failed while compiling
/Users/anatolii/Projects/dotty/dotty/tests/run-i/demo.scala
[=======================================] completed (1/1, 1 failed, 4s)
[error] Test dotty.tools.dotc.FromTastyTests.runTestFromTasty failed:
java.lang.AssertionError: Expected no errors when compiling, failed for the
following reason(s):
[error]
[error]   - encountered 1 test failures(s), took 4.522 sec
[error]     at
dotty.tools.vulpix.ParallelTesting$CompilationTest.checkCompile(ParallelTes
ting.scala:1040)
[error]     at
dotty.tools.vulpix.ParallelTesting$TastyCompilationTest.checkRuns(ParallelT
esting.scala:1455)
[error]     at
dotty.tools.dotc.FromTastyTests.runTestFromTasty(FromTastyTests.scala:49)
[error]     ...


===========================================================================
=====
Test Report
===========================================================================
=====

0 suites passed, 1 failed, 1 total
    tests/run-i/demo.scala failed




-----------------------------------------------------------------------------
-----
Note - reproduction instructions have been dumped to log file:
    /Users/anatolii/Projects/dotty/dotty/testlogs/tests-2019-03-03/tests-
2019-03-03-T23-58-50.log
-----------------------------------------------------------------------------
-----
```

I can’t reproduce the failing case without that Java class hierarchy…

Found the issue. It seems the issue is performance:

```
assert(!ctx.settings.YnoDeepSubtypes.value)
```

This line in TypeComparer restricts from going too deeply in our type resolution recursion. If commented out, the tests seem to be working… So far.

If this is indeed the culprit, I need to find out how far we can go with our recursion and what exactly causes the overhead.

More Loops

Infinite loop compiling the Scala 2 stdlib, just as I thought I nailed it:

```
dotty-compiler/testOnly dotty.tools.dotc.CompilationTests -- *onlyScala
```

(custom command)

Loop Infinity

So, the following snippet:

```
        def maybeAndType = {
          val tpe = tp11 & tp12
          tpe != tp1 && recur(tpe, tp2)
        }
```

Located in fourthTry, it leads to an infinite typecheck loop under some very special conditions that I poorly understand so far. Precisely, `tp11 & tp12` depends on the subtyping relationship check under the hood. Hence the infinite loop.

The problems start when executing the following test:

```
dotty-compiler/testOnly dotty.tools.dotc.CompilationTests --
*compileStdLibOnly
```

(uncomment that method first)

The error looks like:

```
assertion failure for collection.type(scala.collection) <:<
collection.type, frozen = true
assertion failure for
scala.collection.IterableLike[scala.collection.parallel.ParIterableLike[_,
_, _]#T,
  Iterable[scala.collection.parallel.ParIterableLike[_, _, _]#T]
] <:< collection.IterableLike[scala.collection.parallel.ParIterableLike[_,
_, _]#T,
  LazyRef(scala.collection.parallel.ParIterableLike[_, _, _]#Sequential)
], frozen = true
assertion failure for Iterable[scala.collection.parallel.ParIterableLike[_,
_, _]#T] <:<
collection.IterableLike[scala.collection.parallel.ParIterableLike[_, _,
_]#T,
  LazyRef(scala.collection.parallel.ParIterableLike[_, _, _]#Sequential)
], frozen = true
assertion failure for Iterable[scala.collection.parallel.ParIterableLike[_,
_, _]#T] <:< Iterable[scala.collection.parallel.ParIterableLike[_, _, _]#T]
&
  collection.IterableLike[scala.collection.parallel.ParIterableLike[_, _,
_]#T,
    LazyRef(scala.collection.parallel.ParIterableLike[_, _, _]#Sequential)
  ], frozen = true
assertion failure for Iterable[scala.collection.parallel.ParIterableLike[_,
_, _]#T] <:< scala.collection.parallel.ParIterableLike[_, _, _]#Sequential,
frozen = true
```

```
assertion failure for Iterable[scala.collection.parallel.ParIterableLike[_,
_, _]#T] <:< LazyRef(scala.collection.parallel.ParIterableLike[_, _,
_]#Sequential), frozen = true
assertion failure for
scala.collection.IterableLike[scala.collection.parallel.ParIterableLike[_,
_, _]#T,
  Iterable[scala.collection.parallel.ParIterableLike[_, _, _]#T]
] <:< collection.IterableLike[scala.collection.parallel.ParIterableLike[_,
_, _]#T,
  LazyRef(scala.collection.parallel.ParIterableLike[_, _, _]#Sequential)
], frozen = true
```

If you trace the types being checked in the subtyping relationship method, you'll see it is periodic. Also, it involves the type IterableLike:

```
traitIterableLike[+A, +Repr]
```

Somehow `Repr` has to do with the infinite loop there.

To solve the test, I must first understand the causes of why exactly it happens. I should collect the following info:

- Trace the trees for which subtypes are checked, for context
- Trace the types being checked in the subtyping method. Identify the infinite loop there (the period) and explain each step of that period
- Create an isolated example that reproduces the problem

Then I can think of a solution. So first step here would be to create a good trace system and collect the fine-grained trace output in a file. Second step would be to make sense of that file.

---

loop-analysis.xmind

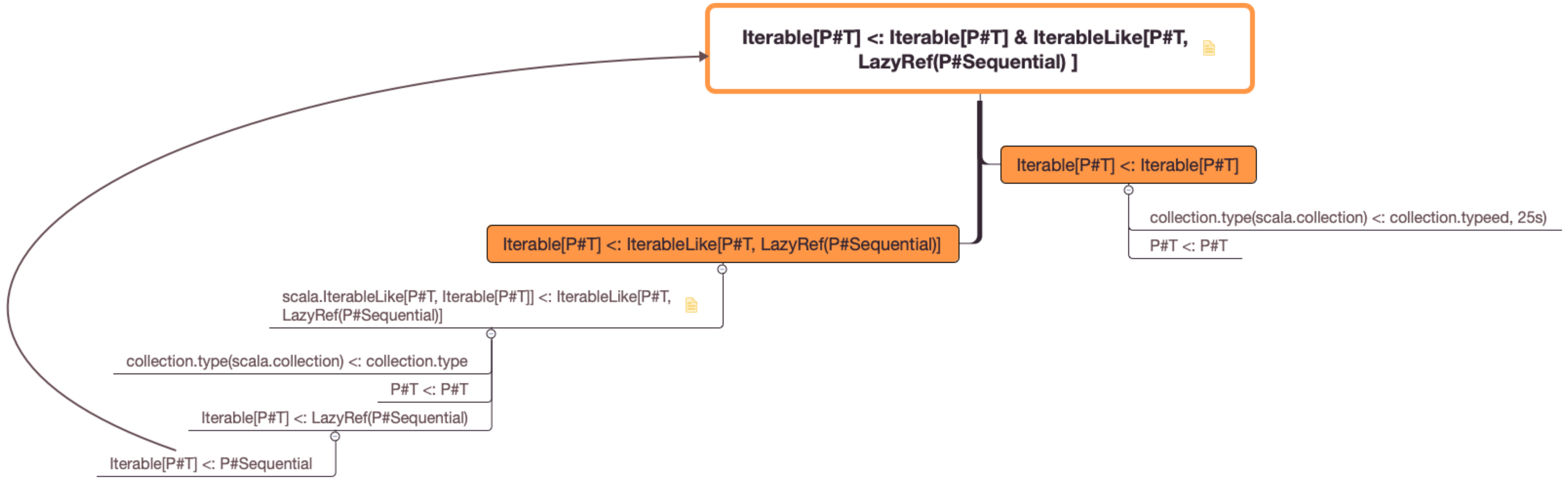
Here's what I have so far on the issue.

```
Iterable[P#T] <: IterableLike[P#T, LazyRef(P#Sequential)]
```

This is the code that appears to cause the loop. To check this, the compiler is going to lift Iterable to IterableLike, to compare according to the variance probably:

```
scala.IterableLike[P#T, Iterable[P#T]] <: IterableLike[P#T,
LazyRef(P#Sequential)]
```

Then it will compare in turn each of the arguments. When comparing



Iterable[P#T] <: Iterable[P#T] & IterableLike[P#T, LazyRef(P#Sequential)]
Iterable[P#T] <: Iterable[P#T]
collection.type(scala.collection) <: collection.typeed, 25s)
P#T <: P#T
Iterable[P#T] <: IterableLike[P#T, LazyRef(P#Sequential)]
scala.IterableLike[P#T, Iterable[P#T]] <: IterableLike[P#T, LazyRef(P#Sequential)]
collection.type(scala.collection) <: collection.type
P#T <: P#T
Iterable[P#T] <: LazyRef(P#Sequential)
Iterable[P#T] <: P#Sequential

```
Iterable[P#T] <: LazyRef(P#Sequential)
```

If Sequential is Iterable[P#T] & IterableLike[P#T, LazyRef(P#Sequential)], it will enter the infinite loop.

Loop Minimised

Basically I diverged from the paradigm I myself adopted. The constraint was minimisation. I was not able to minimise my examples (which is an entire Scala code base) and hence was not able to understand what's wrong. Turns out you can minimise the code base trivially by simply pointing individual files to compile – it works. I isolated the following example:

```
object foo {
  trait A[+X]
  trait B extends A[B]
  trait Min[+S <: B with A[S]]


  val c: Any = ???
  c match {
    case pc: Min[_] =>
  }
}
```

It yields an infinite loop under my solution.

In attempt to exploit the constraint of limited info on each compiler run, I discovered that it is a nice idea to make a debug function, then search for a common pattern in code (e.g. `case\s+.*=>`), then insert the `debug(<id>)` after this pattern, where id is automatically populated to sequential numbers via Sublime plugin. This way, you get 100+ tracing points and save a lot of time tracing code.

Also, wrong assumption here. I assumed Scala 2 codebase is monolithic and interdependent. Turns out it is not so in test environment.

Loop Analysed

Updated example (A invariant, also works):

```
object foo {
  trait A[X]
  trait B extends A[B]
  trait Min[+S <: B with A[S]]

  val c: Any = ???
  c match {
    case pc: Min[_] =>
  }
}
```

We have a trait A with a type parameter X. Also we have a trait B that extends A and sets A’s type parameter to itself. Then, we have a covariant trait Min with the type parameter which extends “B with A[S]”, where S is the type parameter. S <: B with A[S].

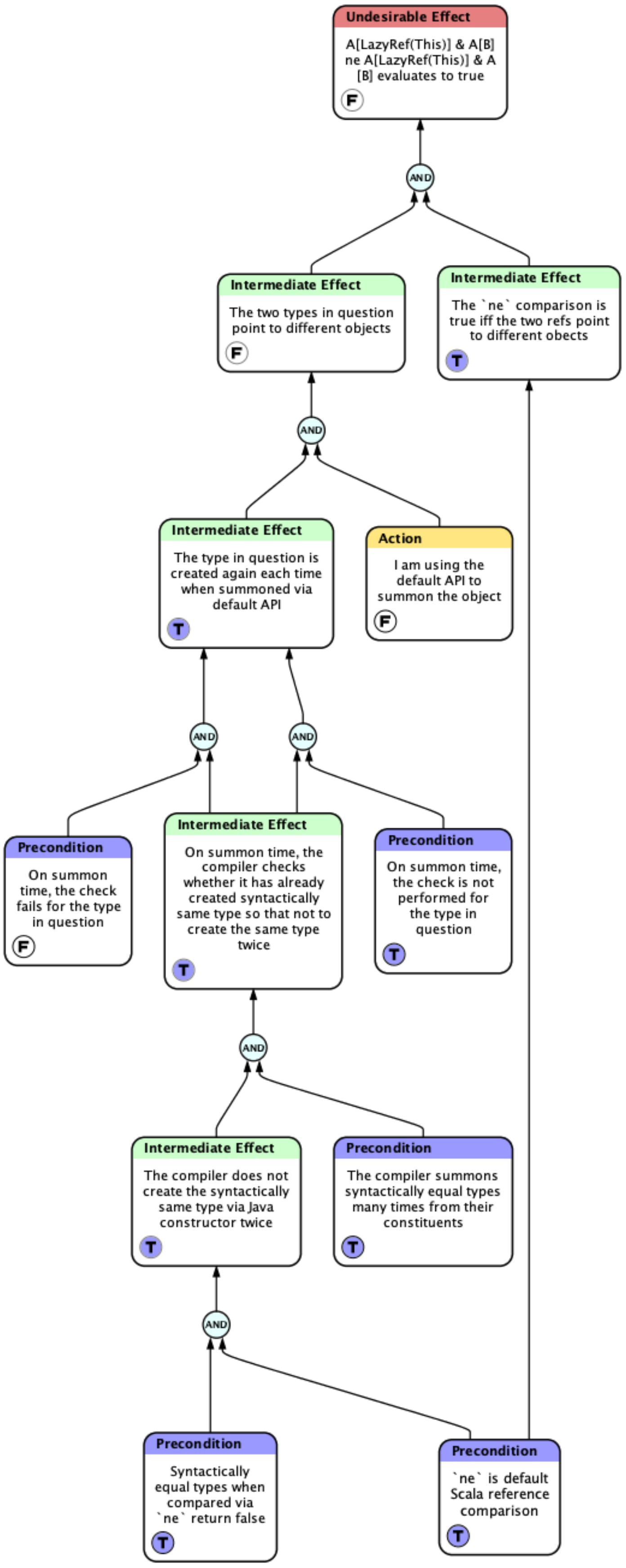
I don’t understand what’s going on inside the match statement. What exactly that Min is checked against and why. And also why exactly at some point the B & A[S] <: Nothing check is performed.

Also, it seems like the `&` operation on type bounds can be a problem. It is possible that they need to be handled somehow before “&”-ing them...

I ended up fallbacking to `andType` on this one.

PR submitted here: https://github.com/lampepfl/dotty/pull/6023

Performance


Undesirable Effect
A[LazyRef(This)] & A[B] ne A[LazyRef(This)] & A[B] evaluates to true
F
AND
Intermediate Effect
The two types in question point to different objects
F
Intermediate Effect
The `ne` comparison is true iff the two refs point to different obects
T
AND
Intermediate Effect
The type in question is created again each time when summoned via default API
T
Action
I am using the default API to summon the object
F
AND
AND
Precondition
On summon time, the check fails for the type in question
F
Intermediate Effect
On summon time, the compiler checks whether it has already created syntactically same type so that not to create the same type twice
T
Precondition
On summon time, the check is not performed for the type in question
T
AND
Intermediate Effect
The compiler does not create the syntactically same type via Java constructor twice
T
Precondition
The compiler summons syntactically equal types many times from their constituents
T
AND
Precondition
Syntactically equal types when compared via `ne` return false
T
Precondition
`ne` is default Scala reference comparison
T

I came to the conclusion that what ToC does for me is it allows me to disentangle my thoughts. Thoughts are composed of many subthoughts but since they move so fast in your head, you have no time to track them down. The problem is these thoughts are based on some assumptions about the world which may not be true. If they are not true, chaos emerges.

So I disentangled my thoughts on the reasons of the problem I encountered. It actually simplified my work compared to bare tracing since I was able to leverage my existing knowledge of the `ne` operator. I was able to pick up that assumption and examine whether it is true explicitly as opposed to just leaving it there in the background.

In fact, that was the first step I did – test out all of my assumptions in the root. This is the sufficiency based diagram, so, if all of the preconditions are true, this is sufficient to lead to the bug.

And, once I establish that they are true, I can inject a solution.

First two checks, the `ne` as the syntactic equality for the types and that `ne` is the default Scala `ne` were easy. First one I knew, the second one I tested by looking at the docs and making sure the compiler that compiles Dotty is a standard one 2.12.8.

The fact that the compiler summons the types multiple times is safe to assume based on the public construction methods of the classes that are invoked outside of the `Typer.scala`.

Troubles

Then, there is the assumption that the caching check fails for the type in question hence it returns a new copy of itself. I had to test this assumption. So I saw that types leverage `Uniques.scala` under the hood, which effectively has two methods that these types use.

Actually, even back when testing the multiple summoning assumption, I implicitly chose code analysis & tracing as my way to go. Here, I followed this path by inertia. To determine if multiple summonings happen, find the code that does multiple summonings. To determine whether there is a failure of the check, find the failure.

The problem is of course that it takes time to trace a bug in a large code base.

The additional allure was the deceptive small size of `Unique.scala`. So I traced these two methods and did not find anything fishy. I was making another implicit assumption that if the check fails, then the cache must with time grow with many instances of the same type. This did not happen, in fact, the cache remained the same size. Since I'd expect it to grow if the check is performed but fails (since failing check implies cache population because the type is not present), it falsified my assumption that the check fails.

Then the assumption was that the check is not performed at all for certain types. Here, I followed my inertia for a few hours, devising ways to trace down the location where the check is *not* performed. Which was tricky because the type in question was quite complicated and I didn't know the sequence of steps the compiler takes to create it.

I followed the inertia even further when I set myself an objective at this point to

> *Determine the sequence of steps the compiler takes to create an uncached type*

Trace the Tree

I saw two paths here. One path was to get hold of the tree for which the typecheck fails and trace the steps the compiler takes to typecheck it, until there is a place where an instance is created without caching. The problem here was that the tree was very non-trivial, there could be many ways to create an instance of the same type, and all I had is an observation of the *already created* type tree. Upon following this train of thought, I understood that it would be too costly to do trace the bug this way.

Trace `andType`

Hence I thought of another path. Instead of trying to determine the tree and reconstructing it from scratch, trace the `andType` method and try to reconstruct what it does. Problem is, its arguments are prebaked and I wouldn't be able to stub or mock them probably because I don't have that much type system knowledge.

Troubles Solution

The above solutions didn't feel feasible. Too much effort and time involved. I took a rest then, and realised my objective was set wrong. The objective is not to trace the steps the compiler takes to create the tree – the objective is to verify the assumption that the compiler doesn't perform the caching checks when creating that tree. This can be done without tracing – just by observation that each time `andType` is called with certain arguments, a new object is created.

So I did just that. I modified the call site to log all objects it created into a list and verify the object is not already there before adding it. This way I was able to verify that each time `andType` is called a new object is created and, by exclusion, hence, the compiler doesn’t do the caching check. Actually at this point I already knew `LazyRef`s are not cached and everything that contains a no-cacher is also a no-cacher.

Implementation

After I had this last assumption verified, I could focus on a solution. Which came to me as simply avoidance of creation of the object in that special scenario.

Debrief

My thought process was my constraint here. The thought process when dealing with code-based projects being to dive into the code and zero-in on the offending line of code.

The problem is that in the Dotty scenario, such a zeroing-in takes a lot of time and effort. And does not necessarily reveal the problem. E.g. I'm pretty sure the caching state of `LazyRef` is there for a reason.

All of my assumptions I tried to verify by zeroing-in on parts of code that would prove or disprove them. This strategy provides a 100% confidence as an output, but the time and energy required may prove it imprudent.

At some point, I tried to verify the assumption that in case of the types I was dealing with, no caching is performed. I followed the analytical thinking process there, trying to produce and minimise an example that would prove the point. The benefit of this approach is also that you can isolate it and submit an issue so that others can help.

However, it took too much energy and time and did not yield much as a result. To the contrary, an empirical thinking process leveraged the fact that I have control over the offending part of the code. Following this thinking process, I asked myself, "what experiment can I devise to prove or disprove my assumption?" as opposed to, "can you find the logic in the compiler which proves or disproves the assumption?".

I was able to verify the assumption in a matter of dozens of minutes.

What caused me to follow the suboptimal thinking process?

The problem here was inertia and indirect effects (causality existence category of logical reservation).

Inertia

Before going to the caching checks assumptions, I proceeded to verify the intermediate effect of the assumptions I had verified so far:

> *On summon time, the compiler checks whether it has already created syntactically same type so that not to create the same type twice*

It was quite easy to do via zeroing-in. All I had to do is to have a look at the companions of the type classes in question and see how they construct the classes.

By inertia, I followed the same metodology for the assumption about my particular case. If the general assumptions involve no more than API inspection, the assumptions of the specific case involve inspecting the state of the compiler at that particular point of execution. Tracing down under statefullness is harder, since you need to take into account that state in determining which path the program will take.

What I should have done here is stop immediately once UDEs start to appear, and try to disentangle and name these UDEs. And, disentangle them with the goal in mind. Ask questions – not only "what is necessary for it to happen" (or "why?"), but also, what am I observing because of it, what am I expecting to observe? Is this UDE sufficient for the above to happen? Or maybe I can avoid that UDE by an injection that will make the UDE in question insufficient?

Basically that's the rule of thumb: **reify UDEs (e.g. feelings) whenever they emerge, don't rely on the inertia to tackle them**. It is easier to just follow the inertia and not think, in the short run, because you're feeling like you're doing something, and you are not admitting that there is a new unknown in the equation. Unknown is scary, hence we may be wired to deny it till the last moment. In the long run, though, the chaos accumulates as we go forward without getting closer to the goal.

Categories of logical reservation applied bottom-up

Especially entity existence. It turned out that the problem lay in my own actions, and they were quite shallow in the tree. I could have inferred that my assumptions are correct from expected effects.

Cause Sufficiency & Clarity reservation failed

> *"The type in question is created again each time when summoned via default API" is sufficient for "The two types in question point to different objects"*

This is incorrect, since we also need the

> *I am using the default API to summon the type*

At the same time with the first one. Maybe also the entities & causal relationships were not clear enough here: which types in question? Well, the one supplied by the compiler for the typecheck and the one *I create* in an attempt to simplify the one fed by the compiler. Well, I do perform the creation action, hence it must be captured.

Unclear goal

When trying to verify

> *On summon time, the check is not performed for the type in question*

I set the goal

> *Determine the sequence of steps the compiler takes to create an uncached type*

And I worked with the 3rd goal in mind:

> *The type in question is created again each time when summoned via default API*

Because if I establish that 3rd goal, I can conclude the validity of the 1st assumption via expected effects, having all the info I have.

Unclear goal led to lots of time waste on investigating futile routes of action.

Failure to apply CLR properly

Overall, I feel like I haven't been formal enough with my approach. This formalism feels petty but it actually matters.

- You don't apply cause sufficiency – and you miss the real root of the problem (my own action)
- You don't articulate your goal properly and jump into action trusting your intuition only – and you're subject to your intuition and possibly flawed assumptions.

ToC is used precisely because at some point we encounter chaos where our intuition and known assumptions about the world fail us. It takes time to ask all these questions, and it may feel petty to get too detailed. However, you miss important pieces this way.

What can I do next time to prevent the above mistakes from happening?

- ToC is a tool. It is used when the existing picture of the world doesn't work anymore. Hence, when deciding to go ToC:
  - Don't trust intuition blindly when thinking
  - Don't allow inertia to accumulate
  - Get petty and detailed
    - Articulate everything I do in some way
    - Apply CLR to my thinking process – take it seriously! If I don't remember all the steps, do it with a checklist.
    - Be very precise about the goal I want to achieve