Next steps for Template Haskell stability

See also: Plan-to-Stabilise-Template-Haskell
See also: https://gitlab.haskell.org/adamgundry/template-haskell-next/-/issues
See also: https://informal.codes/posts/stabilising-th/

The aim of this document is to raise awareness and gather feedback about **a few short-term initiatives** to improve the stability of `template-haskell`.

The main cause for `template-haskell`'s instability is the set of AST types it exposes: `Expr`, `Type`, `Pat`, etc. These types are fundamental to the core functions of the library such as:

- Splicing : `$(foo :: Q Expr)`
- Quoting: `[|...|] :: Quote m => m Expr`

We would like these types to be able to express the breadth of the Haskell language, but as the language changes and grows, these AST types need to be updated to keep track. This leads to breaking changes to the interface of `template-haskell`.

Example: A few releases ago, the `ConP` constructor, which represents constructor patterns, was updated to add a new field. This broke many downstream users who were using it when generating code.

When I wrote Plan-to-Stabilise-Template-Haskell, I was optimistic that many of the direct usages of TH AST constructors could be replaced by uses of TH quotes. TH quotes are much more stable. Yet, TH Quotes also have a variety of usability issues. I successfully migrated some of the code in esqueleto to use quotes rather than hand crafted syntax trees (PR). I encountered some issues I had to work around:

- non-top-level declaration splices are not allows (#25436), which makes it difficult to split up instance definitions.
- GHC does not allow splicing in a name into a method definition with arguments (#25435)
- We can't easily easily splice in do notation statements (#24953)

All of these things can be worked around, but some of the workarounds are non-trivial and require refactoring the code a bit. It makes me less enthusiastic that we can rewrite large parts of TH usages to use quotes, especially since a lot of this code hasn't been touched for years (other than to keep it compiling). In the long-term, I think it would definitely be worthwhile to improve them and have them used more widely, but for now I think we should focus our energy elsewhere.

While improving the situation around the AST types is the most impactful change (this impacts type (D) users, who are most exposed to breaking changes, and corresponds to the first step in the roadmap from Plan-to-Stabilise-Template-Haskell), it is also a difficult problem with some open design questions. I have recently been focussing on smaller issues since I have had

limited capacity in the last year (although I'm finally able to find more time to work on these things).

Step 1: Decoupling `template-haskell` from the TH AST

The TH AST types that live in `ghc-internal` are fixed by our version of GHC. If we want to be compatible with multiple versions of GHC, then `template-haskell` needs to have some sort of compatibility shim on top of them.

Going into ZuriHac 2024, I thought the best way to implement this was through exposing pattern synonyms. During ZuriHac, Adam Gundry investigated this and found that pattern synonyms didn't work well. If I remember correctly, there were issues with Haddocks along with others.

The new line of investigation since then has been to expose views over the internal AST types and to keep these types themselves abstract. One of these views could be an older version of the AST or versions of the AST linked to language editions. This is tracked here: template-haskell-next#2. Users would use smart constructors to generate ASTs and these views to consume them.

A downside of the views approach is that end-users will probably have to change their code to accommodate it, whereas the pattern synonyms idea could be invisible to users.

We should investigate introducing a stable smart constructor layer. Most users could depend on this package and it should cover most type (B) usages. See: #20828

Step 2: Split out the `Lift` and `QuasiQuoter` interface from `template-haskell` (#25262) (implemented at !13569) (also tracked as template-haskell-next#6)

(Does not depend on Step 1)

Even with a stage-1 compiler you should be able to:

- Quotes (with no splices)
- Write an instance for Lift (using quotes)
- Hence, derive Lift
- Build code with smart constructors, or even the underlying constructors.

Moreover, some libraries on which GHC depends do these things (e.g. `filepath`). This forces us to vendor these libraries, because...

- `filepath` depends on `template-haskell` because it does derive(Lift) and exposes nice quasiquoters
- but `template-haskell` depends on `filepath`.

Solution:

- `template-haskell-quasiquoter` depends on ghc-internal (but not `filepath`!)
- `filepath` depends on `template-haskell-quasiquoter`
- `template-haskell` depends on `filepath`.
- Moreover, PVP version of `template-haskell-quasiquoter` is super-stable (unlike `template-haskell` which is very unstable. Hence version bounds on `filepath` don't need to be updated. Hooray.
- Moreover: th-quasiquoter can also contain stable shims to unstable th functions. For example
    - th-lift: `liftAddrCompat`

Main goal: avoid version bumps!

Criteria:

- ~~Does not need to run splices~~
- Largest stable TH-related API. But why two libraries? Not one, or three, or seven?
- Avoid the need to import the (unstable) `template-haskell`

And release shim'd versions of A and B, for the previous N versions of GHC.

What is `template-haskell`?

- The `Lift` class
- QuasiQuoters
- Syntax tree
- Q monad and operations
    - Code generation operations (e.g. mkName, addTopDecls, reportError)
    - Impure operations (e.g. addForeignFile, addModFinalizer)
- The reification interface (a long way off)
    - Operations (e.g. reifyType, lookupTypeName)
    - Reified information about Ids, TyCons, etc. (currently the TH AST but this should change in the future)

A possible alternative proposal:

- Identify a stable API for Template Haskell,
- ... in one package; call it `th-stable`
- Release shims for `th-stable` (best efforts, anyway)
- Grow the API of `th-stable` over time to support more use-cases.
- Downside: non-shimmable changes are more likely in a big library than in multiple small ones

Both the `Lift` typeclass and the interface for `QuasiQuoter`s are relatively stable, widely used, and don’t depend on the implementation of the TH AST types. This makes them great

candidates for being split out into their own libraries: `template-haskell-lift` and `template-haskell-quasiquoter` . I have a draft implementation at !13569 .

Both of these packages expose the minimum interfaces necessary for users along with all relevant types. `Template-haskell-lift` also includes some backwards compatibility utilities, which are helpful for the custom instances required by `bytestring` and `text`.

Most boot libraries depend on `template-haskell` but they only do so in order to generate instances of `Lift` or to expose a `Quasiquoter`. They could easily just depend on these new libraries instead, and that would mean that `template-haskell` could be eliminated from the dependency footprint of `ghc` the library (`ghc` only depends on `template-haskell` transitively through depending on boot libraries such as `containers`, which exposes a `Lift` instance). This would mean that users who want to use the `ghc` library bundled with the compiler, wouldn't be forced to also use the bundled version of `template-haskell`.

I'm hoping that this will greatly reduce the amount of bounds bumping needed in the ecosystem since many libraries depend on `template-haskell` just for the `Lift` identifier, so they can derive an instance.

Once these packages no longer depend on `template-haskell`, we can then add a dependency in the opposite direction. For instance `template-haskell` currently has to vendor parts of `containers` and `filepath` since otherwise we would run into a dependency loop. This also expands the types of things we can put into `template-haskell`. Something simple like an efficient variable substitution utility (that uses `Map`) becomes possible. This would reduce the amount of internals that consumers of the library need to interact with.

I still need to tidy up the MR and add some documentation, but feedback on the design would be really helpful.

Abstract `Quasi` typeclass

As it stands the `Quasi` typeclass plays two roles. It is both the internal interface between GHC and `template-haskell` and it is an external interface users can use. These need to be disentangled. If we add a new `reifyFoo` method to `Quasi` in `ghc-internal` it shouldn't immediately be visible in `template-haskell`. It is not enough to hide the method because then it becomes impossible for users to derive their own instances of `Quasi`, which is something users do.

I'm planning to write up a GHC proposal to improve this situation. My plan is to have another interface distinct from `Quasi` that acts as the internal interface, and `Quasi` can then exist purely in `template-haskell` rather than `ghc-internal` and become the external interface.

Once we have something like this, it becomes much easier to add new methods or otherwise have breaking changes in the `Quasi` class since we are much more decoupled from GHC.

Reinstalling `template-haskell` with cabal-install

As of `GHC-9.12` it is possible to reinstall `template-haskell` with cabal-install. All you need to do is:

- Pass `--allow-boot-library-installs`
- Ensure that the installed version of `template-haskell` is excluded by a constraint
- Ensure that we don't depend on `ghc` (which has `template-haskell` as a transitive dependency).

All of these could be improved and I have a tracking ticket on the cabal tracker that links to issues: cabal#10440

cabal#10087 concerns the list of non-reinstallable packages in cabal-install. Initially I thought we could just make this a list exposed by GHC but Matthew Pickering convinced me that this is bad idea here: !13297. See also Matt's reply on the GHC devs mailing list.

So, as Matt says, we can unit-id for `ghc` the library, and cabal-install can use that in the solver to force users who use plugins to use that unit. Similarly we can expose the unit-id of `ghc-internal` and cabal-install can add a constraint to force usage of that unit. Matt has made a draft commit for the cabal side of this. And I've updated my GHC PR to use this approach instead.

cabal#9669 on the other hand should be easy to make progress with. There is broad support for adding a `--prefer-newest` flag that doesn't force the solver to pick packages from the global database (changing the default is understandably more controversial). The PR for adding `--prefer-oldest` is a helpful starting point: cabal#8261