

# Template Haskell, a Case Study in (In)stability

Teo Camarasu

CircuitHub

5<sup>th</sup> of June 2025

Haskell Ecosystem Workshop 2025

Introduction

Overview of Template Haskell

Pause

Interfaces of Template Haskell

(A) Quote-and-splice clients

(B) Syntax-construction clients

Break

(C) Reification clients

(D) Syntax-analysis clients

Build tools and Template Haskell

Conclusion

# Introduction

# Introduction

- ▶ Stable interfaces are important when working in the large.
- ▶ Template Haskell is an interesting case study.
- ▶ Essential coupling with GHC is a challenge.
- ▶ What it is, what it could be, inspiration.

## How I got involved in Template Haskell stability

- ▶ A discussion on the GHC GitLab about Template Haskell stability.
- ▶ Just finished writing a TH library, `th-deepstrict`.
- ▶ Also just finished upgrading to a new version of GHC.
- ▶ Things should be better!

# Template Haskell stability work

- ▶ Just over a year of working on and off on Template Haskell.
- ▶ Benefited from GHC culture of valuing documentation.
- ▶ Notes, tickets, design documents, and conversations are excellent.
- ▶ Great discussions at ZuriHac and in HF Stability WG.
- ▶ Many ideas and designs ready to go, but blocked on capacity.

## Get involved with GHC

- ▶ Lots of areas of GHC are similarly capacity bound.
- ▶ GHC is a large and old project, but often well documented.
- ▶ You just need to understand one small area.
- ▶ And it's fun!

# Overview of Template Haskell

# Template Haskell by Example

- ▶ Boilerplate: `makeLenses`, `makeOptics`
- ▶ Typeclass deriving: `deriveAeson`, etc
- ▶ Quasiquoters: Yesod, Shakespearean templates, raw strings, etc
- ▶ Embedding static data, eg, `file-embed` package.

# Template Haskell

- ▶ Usages of Template Haskell likely familiar.
- ▶ Allows writing *metaprograms*:
  - ▶ “Normal” programs are fully executed at run-time.
  - ▶ Parts of metaprograms may be executed at compile-time.
  - ▶ Allows examining compiler state, eg, shape of datatypes.
  - ▶ Allows generating program fragments, eg, typeclass definitions.

# Alternatives to Template Haskell

- ▶ Preprocessors
  - ▶ No understanding of syntax
  - ▶ Can modify imports.
- ▶ Generics
  - ▶ Only supports sums of products.
  - ▶ No user code run at compile-time.
- ▶ GHC plugins
  - ▶ Complete access to GHC internals.
  - ▶ Requires running user code at compile-time.

# Template Haskell is coupled to GHC

- ▶ Template Haskell understands Haskell syntax.
- ▶ Template Haskell can query *some* things about compiler state, eg,
  - ▶ definition of data structure.
  - ▶ the type/kind of a variable.
- ▶ Much more limited than GHC plugins.
- ▶ More powerful than `Generics`.

# Template Haskell and Dependency Inversion

- ▶ A copy of the Haskell syntax tree. `Exp`, `Type`, `Pat`, etc.
- ▶ A set of effects defined in the `Quasi` typeclass, eg:

```
class (MonadIO m, MonadFail m) => Quasi m where
  qNewName :: String -> m Name
  qReify   :: Name  -> m Info
  ...
```

- ▶ The base monad in TH, `Q`, is an instance of `Quasi`.
- ▶ GHC then *implements* these interfaces.
- ▶ GHC converts between the TH and its own AST.
- ▶ GHC includes an implementation of `Quasi`.

## How does it all work (example)

```
main =  
  $(varE 'print `appE` (lift "hello world") :: Q Exp)
```

- ▶ Run the splice with GHCi.
  - ▶ `varE 'print`, the print function
  - ▶ `lift "hello world"` turn a value into a syntax tree representation of the value.
- ▶ This gives us a syntax tree.
- ▶ Insert it into the program.

# Template Haskell in brief

- ▶ We can write metaprograms that have compile-time parts.
- ▶ We can manipulate syntax trees and look at some compiler state.
- ▶ Don't depend directly on GHC.
- ▶ GHC depends on these interfaces.
- ▶ Some level of coupling but there is hope.

Pause

# Interfaces of Template Haskell

# Interfaces of Template Haskell

Interfaces between Template Haskell and:

- ▶ Users (main interface):
  - ▶ Exposed via the `template-haskell` package.
  - ▶ Exposed via language extensions, eg, `TemplateHaskell`, `DeriveLift`, etc.
- ▶ GHC:
  - ▶ Defined in `ghc-internal`
  - ▶ Used to be part of `template-haskell` before GHC-9.12. See [ghc!12479](#).
- ▶ Build systems, eg, Cabal, stack, Hadrian (GHC's build system)
  - ▶ Special case for the `template-haskell` package.

# Stability Goal

- ▶ Easy to upgrade Template Haskell independently of GHC.
- ▶ Ideally a new version of GHC doesn't force an upgrade.
- ▶ As few special cases as possible, just a normal package.

# User-facing Interfaces

- ▶ Syntax trees: Datatypes and library functions represent Haskell code.
- ▶ (Typed) Splices: run some code and insert a program fragment.
- ▶ (Typed) Quotes: Syntax for representing a program fragment as a syntax tree.
- ▶ Quasiquotes: Generalisation of quotes, parse arbitrary data.
- ▶ `Lift` typeclass: Turn a value into a syntax tree representation.
- ▶ `Quasi` typeclass. Methods:
  - ▶ `qFreshName`, generating names
  - ▶ `qReify`, ..., looking up information
  - ▶ impure operations: reading files, run IO (controversial).
  - ▶ error handling, reporting errors, etc.

## Interfaces vary in stability

The main driver of instability is the GHC syntax tree.

## Classifying usages

(A) is most stable. (D) is least.

- (A) Quote-and-splice clients: Use only splices, quotes, `DeriveLift` or `Quasiquotes`. Might not even need to import the `template-haskell` library.
- (B) Syntax-construction clients: construct Template Haskell syntax trees either directly constructors, or indirectly through the smart-constructors exported by `Language.Haskell.TH.Lib`.
- (C) Reification clients: use reification to examine the program. Reification currently may return Template Haskell syntax trees.
- (D) Syntax-analysis clients: Clients who pattern match on Template Haskell syntax trees.

(A) Quote-and-splice clients

## (A) Quote-and-splice clients

- ▶ Splices, we have seen:
  - ▶ untyped: `$(... :: Q Exp)`
  - ▶ typed: `$$(... :: Code Q a)`
- ▶ Quotes:
  - ▶ untyped: `[|...|] :: Q Exp`
  - ▶ typed: `[||...||] :: Code Q a`
- ▶ Quasiquotes: `[foo|...|]` arbitrary parsing.
- ▶ `DeriveLift`, eg, `data Foo = ... deriving (Lift)`
- ▶ Each of these is stable.

## Quotes are stable

- ▶ Quotes are a way to create syntax trees.
- ▶ Language has strong guarantees that future versions will parse the same code.
- ▶ So if `1 + 1` is valid code then `[|1 + 1|]` is a valid code.
- ▶ We don't care if the underlying syntax tree representation changes.

## Quasiquotes are stable

```
module Language.Haskell.TH.Quote where
data QuasiQuoter = QuasiQuoter
  { quoteExp    :: String -> Q Exp
  , quotePat    :: String -> Q Pat
  , quoteType   :: String -> Q Type
  , quoteDec    :: String -> Q [Dec]
  }
}
```

- ▶ `[foo|...|]` desugars to `$(quoteExp foo "...")` depending on location.
- ▶ Definition of `QuasiQuoter` is stable.
- ▶ Instability for library authors as we refer to syntax tree types.
- ▶ In practice, (most) library authors also use derived lift instances and only implement `quoteExp`.
- ▶ Users of the libraries are not exposed to instability.

## Deriving and using Lift is stable

- ▶ Lift typeclass allows turning a value into a syntax tree representing that value.
- ▶ AKA cross stage persistence.
- ▶ Most instances are derived using the DeriveLift extension.
- ▶ Little exposure to changes in syntax trees.
- ▶ Lift interface has changed in the past but rarely.

## Stable in theory but still has churn

- ▶ Quotes and splices don't need any imports to use.
- ▶ All others are exposed through `template-haskell`.
- ▶ The code of type (A) users is very unlikely to break.
- ▶ But they still need to bump upper bounds with a new release of `template-haskell`.

# Splitting out Lift and Quasiquoter

- ▶ A solution is to create new package(s) for these interfaces.
- ▶ GHC proposal 696:
  - ▶ `template-haskell-lift` just exposes `Lift`.
  - ▶ `template-haskell-quasiquoter` just exposes `QuasiQuoter`.
- ▶ Much less likely to need version bumps.
- ▶ Can evolve independently.
- ▶ Boot libraries only use these interfaces and so don't need to depend on `template-haskell`.
- ▶ Then `template-haskell` can depend on containers, etc.

## (A) Quote-and-splice clients: Conclusion

- ▶ Already pretty stable
- ▶ Still requires upper bound updates often.
- ▶ By splitting out stable packages we can reduce this.
- ▶ More clients should become type (A) clients.

(B) Syntax-construction clients

## (B) Syntax-construction clients

- ▶ We want to create syntax trees to splice them into the program.
- ▶ `template-haskell` has several interfaces:
  - ▶ Syntax tree constructors of `Exp`, `Type`, `Pat`, `Dec`, etc.  
`AppE :: Exp -> Exp -> Exp`
  - ▶ Smart constructors from `Language.Haskell.TH.Lib`  
`appE :: Quote m => m Exp -> m Exp -> m Exp`
- ▶ Both are somewhat unstable.

## Case study of breakage for (B) clients

- ▶ `template-haskell-2.18` changes the definition of `ConP` and `conP`.

```
| ConP Name [Type] [Pat]
-- ^ @C1 \@ty1 p1 p2@
```

- ▶ `[Type]` is new.
- ▶ `ConP 'foo [...]` becomes `ConP 'foo [] [...]` and `CPP`.
- ▶ Sometimes smart constructors are more stable.
  - ▶ Keep old name with old type and introduce new one.

# Solutions

1. Expose stable field selectors and smart constructors  
GHC#20828.
  - ▶ eg, Haskell2010
2. Expose another syntax tree with explicit conversion functions.
3. `template-haskell-X` has a fixed syntax tree but can be built with multiple GHCs.
  - ▶ Might be implemented with pattern synonyms and CPP.
  - ▶ How do we deal with unsupported syntax?
  - ▶ Maybe we change syntax trees to be open-ended.

Adam Gundry worked on this during ZuriHac 2024.

We can approaches outside of `template-haskell`.

## Another approach: Conversion to type (A)

- ▶ Quotes are a stable means for syntax construction.
- ▶ We can try to turn (B) clients into (A) clients.
- ▶ `esqueleto#394`
- ▶ I found some bugs in TH Quotes, which we can now fix.
- ▶ We should do the same with other libraries! Happy to help.

# Conclusion

- ▶ Syntax tree construction can be quite unstable.
- ▶ Every release of GHC might force a release of TH and break code.
- ▶ Encourage use of quotes.
- ▶ Create some sort of shim on top to avoid breakage.

Break

(C) Reification clients

## (C) Reification clients

- ▶ Reification is about looking at compiler state, eg:
  - ▶ What is the definition of this datatype?
  - ▶ Look up strictness of constructor.
  - ▶ Finding typeclass/typefamily instances.
- ▶ Reifying a datatype returns a syntax tree.
- ▶ Same problems as before. Syntax trees are unstable.
- ▶ But also verbose. The compiler has a much better representation.
- ▶ `th-abstraction` package gives a better representation.

# Solution

- ▶ Return analyzed data like `th-abstraction`.
- ▶ But this is tricky to do without breaking existing users.
- ▶ Users can give definitions of `Quasi` and adding a method breaks their code.

## Quasi typeclass is problematic

- ▶ All of these methods are exposed via `Quasi` typeclass.
- ▶ It is both internal interface (GHC) and external (users).
- ▶ Split it up. GHC Proposal 700.
- ▶ `Quasi` stays as purely user-facing. And we add a new internal interface.
- ▶ Internal interface can have both new and old version of `reify`, and `Quasi` can pick which to expose.

## (C) Reification clients: Conclusion

- ▶ Reification clients can be decoupled from syntax trees, and return analyzed data instead.
- ▶ First step is to decouple `Quasi` from the internal interface.
- ▶ Then we can modify reification to make it more stable.

(D) Syntax-analysis clients

## (D) Syntax-analysis clients

- ▶ Syntax analysis clients pattern match on the entire syntax tree.
- ▶ Inherently unstable. Might break with each version of GHC.
- ▶ Maybe we can reduce the need for them by putting analyses into `template-haskell`, eg,
  - ▶ Variable substitution
- ▶ This is much more doable if we can depend on containers.

## Build tools and Template Haskell

# Build tools

- ▶ Historically `template-haskell` has needed special treatment in `cabal/stack/hadrian`.
- ▶ As of GHC-9.12, it is just a normal package.
- ▶ We moved everything wired-in to `ghc-internal`. See `ghc!12479`.
- ▶ But build tools haven't caught up.
- ▶ `cabal-install` and `stack` refuse to build it unless shipped with GHC.
- ▶ `cabal-install` can be overridden with a flag:  
`--allow-boot-library-installs`
- ▶ Ticket to teach `cabal-install`: `cabal#10087` collab with Matthew Pickering.

# Global packages

- ▶ Both `cabal-install` and `stack` have a notion of global packages.
- ▶ Global packages are bundled with GHC.
- ▶ `stack` doesn't allow reinstalling them.
- ▶ `cabal-install` only does it as a last resort – no other build plan.
- ▶ `cabal-install` issue to allow disabling this: `cabal#9669`
  - ▶ Good newcomer project.

## Build tools: Conclusion

- ▶ Build tools are lagging behind GHC.
- ▶ With a little help they can be taught to reinstall `template-haskell`.

Conclusion

# Conclusion

- ▶ Template Haskell has a large surface area.
- ▶ Some parts are quite stable.
- ▶ Some parts need improvement.
- ▶ In general, we want to decouple from GHC and especially the GHC syntax trees.
- ▶ Potential for a stable Template Haskell in the near future.