Not-So-Boring Haskell

Andrew Lelechenko
andrew.lelechenko@gmail.com

Haskell Ecosystem Workshop, 05.06.2025

Michael Snoyman, Boring Haskell Manifesto, Nov 2019

Haskell has [two overlapping] subcultures:

- Explore interesting and revolutionary concepts in computer science, software engineering, and mathematics.
- Make better software.

...

Exploring these [interesting and revolutionary] concepts can be fun, rewarding, and—long term—a huge benefit for productivity. Short and medium term, however, this exploration can lead to slower and less reliable results.

...

...a simple, well-defined subset of Haskell's language and ecosystem will deliver large value for a project, while introducing little to no risk compared to alternative options. We call this subset, somewhat tongue-in-cheek, "boring Haskell".

The Simple Haskell Initiative, Dec 2019

Accessibility Commercial software is a team endeavor. Fancy Haskell is costly to teams because it usually takes more time to understand and limits the pool of people who can effectively contribute.

Maturity Things that have been around longer will be more well-tested and understood by a larger group of people. Prefer tried and true techniques over the latest shiny library or language feature. The more foundational something is in your tech stack, the more conservative you should be about adopting new versions or approaches to that thing.

Leaking Complexity If you adopt a new thing, how much of its complexity will spread throughout the rest of your codebase? You should be more hesitant to adopt something if its complexity is going to spread through a larger portion of your codebase.

Haskell has [two overlapping] subcultures:

- Explore interesting and revolutionary concepts in computer science, software engineering, and mathematics.
- Make better software.

. . .
Exploring these [interesting and revolutionary] concepts can be fun, rewarding, and—long term—a huge benefit for productivity. Short and medium term, however, this exploration can lead to slower and less reliable results.

. . .
. . . a simple, well-defined subset of Haskell's language and ecosystem will deliver large value for a project, while introducing little to no risk compared to alternative options. We call this subset, somewhat tongue-in-cheek, "boring Haskell".

Year 2025

- Haskell is no longer an exploration vehicle *per se.*
- It is a productionalisation lab, dedicated to converting designs to practical tools and *making better software.*
- Limiting feature set is not going to make your code simpler, quite the opposite of it.
- Our unique selling proposition:
  explore without sacrificing reliability or time-to-market.
- Our core enabler:
  fearless refactoring.

The Simple Haskell Initiative, Dec 2019

Accessibility Commercial software is a team endeavor. Fancy Haskell is costly to teams because it usually takes more time to understand and limits the pool of people who can effectively contribute.

Maturity Things that have been around longer will be more well-tested and understood by a larger group of people. Prefer tried and true techniques over the latest shiny library or language feature. The more foundational something is in your tech stack, the more conservative you should be about adopting new versions or approaches to that thing.

Leaking Complexity
If you adopt a new thing, how much of its complexity will spread throughout the rest of your codebase? You should be more hesitant to adopt something if its complexity is going to spread through a larger portion of your codebase.

Year 2025

Accessibility Curiosity is what has driven us to use Haskell in the first place. We should exploit this instinct and lean on the fact that people like puzzles.

Maturity Let people indulge themselves and experiment with the very latest shiny libraries. Learning new things is exciting and not tedious. It's easy to replace libraries.

Leaking Complexity It's cheap to demote and dismantle complexity, if it does not work out. Refactoring is Haskell superpower.

In 2025 anything boring is better left to AI. Jump on the new train:

Puzzling Haskell

Rules of the game

- It is not a pub quiz! You are not supposed to know the answer beforehand.
- Puzzle should offer a systematic way to analyze and solve it without prior knowledge.
- Puzzle solving should not be tedious; write as little code as possible.
- Each puzzle on its own should be small enough to take in at a glance. Long-distance interaction is bad.
- Keep asking yourself: can this puzzle be solved in a finite amount of time?
- Solving process offers a tight feedback loop. Both compile time and runtime should be fast.
- It's not fun to solve the same puzzle twice. Solutions must be robust and survive through ecosystem and tooling upgrades.

Upgrade policy — 1

Fearless refactoring is both a blessing and a curse:

- It's very easy to accommodate almost any change.
- *Because of this,* library authors tend to make breaking changes, they would never have attempted in Python or JavaScript.

Rock solid build environments are similar:

- It's easy to set up and protect a uniform reproducible environment everywhere.
- But if / when you get out of this ivory tower, the real world can overwhelm you.

Upgrade policy — 2

Vicious cycle:

- Wait until a last minor GHC release.
- Wait until Stackage LTS is available.
- Try to upgrade.
- Hit a bug in GHC.
- Find that a package is missing from Stackage.
- Rinse and repeat.

Solution:

- Set in stone production environment.
- As fluid as possible development environment.
- You have to fix breakage anyway, so why not do it early?
- Constant pain is the best trainer to avoid unmaintainable code.

Template Haskell is a pub quiz

In GHC 9.8 `TyVarBndr ()` changed to `TyVarBndr BndrVis` in multiple places.

```
-- | The @flag@ type parameter is instantiated to
-- * 'Specificity' (examples: 'ForallC', 'ForallT')
-- * 'BndrVis' (examples: 'DataD', 'ClassD', etc.)
-- * '()', a catch-all type for other forms of binders...
data TyVarBndr flag
  = PlainTV  Name flag      -- ^ @a@
  | KindedTV Name flag Kind -- ^ @(a :: k)@

data BndrVis
  = BndrReq    -- ^ @a@
  | BndrInvis -- ^ @\@a@
```

GHC plugins are not even puzzles

- Plugins have to deal with raw AST, so fragile for the same reason TH is.
- But even worse: plugins are black boxes.
- Plugin errors are custom, hardly relatable to code you wrote.
- Reading finite amount of manuals is good.
- Reading infinite amount of code is bad.

Tools to solve puzzles

Enable your code for hole-driven development: employ as many fine-grained types as possible

```
> :set -fno-show-provenance-of-hole-fits
> import Data.Monoid
> _ [1,2,3] :: Sum Integer
* Found hole: _ :: [Integer] → Sum Integer
  Valid hole fits include
    mempty :: forall a. Monoid a ⇒ a
      with mempty @([Integer] → Sum Integer)
```

Hole-driven development

```
> :set -frefinement-level-hole-fits=1
> _ [1,2,3] :: Sum Integer
* Found hole: _ :: [Integer] → Sum Integer
  Valid hole fits include
    mempty :: forall a. Monoid a ⇒ a
      with mempty @([Integer] → Sum Integer)
  Valid refinement hole fits include
    const (_ :: Sum Integer)
      where const :: forall a b. a → b → a
      with const @(Sum Integer) @[Integer]
    foldMap (_ :: Integer → Sum Integer)
      where foldMap :: forall (t :: * → *) m a.
                       (Foldable t, Monoid m) ⇒
                       (a → m) → t a → m
      with foldMap @[] @(Sum Integer) @Integer
```

Core rule: **avoid features which weaken type inference.**

So many extensions, so little time

- AllowAmbiguousTypes
- AlternativeLayoutRule
- ApplicativeDo
- Arrows
- AutoDeriveTypeable
- BangPatterns
- BinaryLiterals
- BlockArguments
- CApiFFI
- ConstrainedClassMethods
- ConstraintKinds
- CPP
- CUSKs
- DataKinds
- DatatypeContexts
- DeepSubsumption
- DefaultSignatures
- DeriveAnyClass
- DeriveDataTypeable
- DeriveFoldable
- DeriveFunctor
- DeriveGeneric
- DeriveLift
- DeriveTraversable
- DerivingStrategies
- DerivingVia
- DisambiguateRecordFields
- DoAndIfThenElse
- DoRec
- DuplicateRecordFields
- EmptyCase
- EmptyDataDecls
- EmptyDataDeriving
- ExistentialQuantification
- ExplicitForAll
- ExplicitNamespaces
- ExtendedDefaultRules
- ExtendedLiterals
- FieldSelectors
- FlexibleContexts
- FlexibleInstances
- ForeignFunctionInterface
- FunctionalDependencies
- GADTs
- GADTSyntax
- GeneralisedNewtypeDeriving
- GHCForeignImportPrim
- HexFloatLiterals
- ImplicitParams
- ImplicitPrelude
- ImportQualifiedPost
- ImpredicativeTypes
- IncoherentInstances
- InstanceSigs
- InterruptibleFFI
- JavaScriptFFI
- KindSignatures
- LambdaCase
- LexicalNegation
- LiberalTypeSynonyms
- LinearTypes
- ListTuplePuns
- MagicHash
- MonadComprehensions
- MonoLocalBinds
- MonomorphismRestriction
- MultilineStrings
- MultiParamTypeClasses
- MultiWayIf
- NamedDefaults
- NamedFieldPuns
- NamedWildCards
- NegativeLiterals
- NPlusKPatterns
- NullaryTypeClasses
- NumDecimals
- NumericUnderscores
- OrPatterns
- OverlappingInstances
- OverloadedLabels
- OverloadedLists
- OverloadedRecordDot
- OverloadedRecordUpdate
- OverloadedStrings
- PackageImports
- ParallelArrays
- ParallelListComp
- PartialTypeSignatures
- PatternGuards
- PatternSignatures
- PatternSynonyms
- PolyKinds
- PostfixOperators
- QualifiedDo
- QuantifiedConstraints
- QuasiQuotes
- RankNTypes
- RebindableSyntax
- RecordPuns
- RecordWildCards
- RecursiveDo
- RelaxedLayout
- RelaxedPolyRec
- RequiredTypeArguments
- RoleAnnotations
- Safe
- ScopedTypeVariables
- StandaloneDeriving
- StandaloneKindSignatures
- StarIsType
- StaticPointers
- Strict
- StrictData
- TemplateHaskell
- TemplateHaskellQuotes
- TransformListComp
- Trustworthy
- TupleSections
- TypeAbstractions
- TypeApplications
- TypeData
- TypeFamilies
- TypeFamilyDependencies
- TypeInType
- TypeOperators
- TypeSynonymInstances
- UnboxedSums
- UnboxedTuples
- UndecidableInstances
- UndecidableSuperClasses
- UnicodeSyntax
- UnliftedDatatypes
- UnliftedFFITypes
- UnliftedNewtypes
- Unsafe
- ViewPatterns

Language extensions

- Syntactic extensions are fine, even exotic ones, because they allow us to write less code.
- Record extensions are fine. `RecordWildCards` are tempting, but essentially a form of and as bad as name shadowing.
- Deriving extensions are great, saving boilerplate.
- Safe Haskell is too fragile, avoid at all costs.
- Overloading harms type inference, so undesirable.

OverloadedLists harms hole-driven development

```
> :set -XOverloadedLists
> _ [1,2,3] :: Sum Integer
<interactive>:1:1: error: [GHC-88464]
* Found hole: _ :: t0 -> Sum Integer
  Where: 't0' is an ambiguous type variable
  Valid hole fits include
    mconcat :: Monoid a => [a] -> a
    sconcat :: Semigroup a => GHC.Base.NonEmpty a -> a
    head :: HasCallStack => [a] -> a
    last :: HasCallStack => [a] -> a
    maximum :: (Foldable t, Ord a) => t a -> a
    minimum :: (Foldable t, Ord a) => t a -> a
<interactive>:1:3: error: [GHC-39999]
* Ambiguous type variable 't0' arising from an overloaded
  list prevents the constraint IsList t0 from being solved.
* Ambiguous type variable 't0' arising from the literal '1'
  prevents the constraint Num (Item t0) from being solved.
```

Type system extensions

- Core rule: **make the type system more powerful without weakening inference.**
- GADTs are double-edged: they can greatly refine typed hole suggestions, but also complicate type inference.
- Type-level programming and Dependent Haskell are fine!.. as long as you can avoid `TemplateHaskell` and GHC plugins.
- Linear Haskell is also fine.
- Advanced types are easy to demote to `Haskell2010`.

Ad-hoc polymorphism

- Resist the urge to introduce a type class.
- Resist the urge to introduce a type class with a type family at all costs.
- Ripping off badly designed type class can get very costly.

```
class IsList l where
  type Item l
  fromList :: [Item l] → l
  toList :: l → [Item l]

> :set -XOverloadedLists
> _ [1,2,3] :: Sum Integer
<interactive>:1:3: error: [GHC-39999]
* Ambiguous type variable ’t0’ arising from an overloaded
  list prevents the constraint IsList t0 from being solved.
* Ambiguous type variable ’t0’ arising from the literal ’1’
  prevents the constraint Num (Item t0) from being solved.
```

Type classes and open type families

```
{-# LANGUAGE TypeFamilies #-}

class Consume what how where
  type Output what how
  consume :: what → how → output

instance Consume Soup Spoon
instance Consume Steak Fork
instance Consume Petrol Car
instance Consume Stream Fold

foo :: Consume what how ⇒ [what] → Maybe how → Maybe (Outp
```

Quiz time

```
log :: String → IO ()
log msg = putStrLn $ "[LOG] " ++ msg

main :: IO ()
main = log "Hello World"

Log.hs:5:8: error: [GHC-87543]
    Ambiguous occurrence ’log’.
    It could refer to
       either ’Prelude.log’,
              imported from ’Prelude’ at Log.hs:1:1
              (and originally defined in ’GHC.Float’),
           or ’Main.log’, defined at Log.hs:2:1.
  |
5 | main = log "Hello World"
  |        ^^^
```

Until `ImportShadowing` language extension arrives...

- Always use explicit export lists.
- Prefer to use qualified imports and explicit import lists.
- Avoid `hiding` clause, even if it requires `NoImplicitPrelude`.

ImportQualifiedPost is available since GHC 8.10

```haskell
import Data.Map (Map)
import qualified Data.Map as M
```

$$\Rightarrow$$

```haskell
{-# LANGUAGE ImportQualifiedPost #-}
import Data.Map (Map)
import Data.Map qualified as M
```

Quiz time

```
data Infinite a = a :< Infinite a

foo :: Int
foo =
  let xs = 1 :< xs in
    case xs of
      x :< _ → x
```

- What's the result of `foo`?
- Your coworker moved it to another module and now it freezes. Why?

Strictness

- Good extensions should not change semantics silently. Code should either compile and work in the same way, or do not compile at all.
- Thus `Strict` and `StrictData` are discouraged.
- Using `BangPatterns` or `deepseq` is a sign that your data structures lack bangs.
- "Make invalid laziness unrespresentable": sow the bangs of strictness generously and prematurely across all datatypes.

Generic programming and boilerplate generation

- `TemplateHaskell` offers good compile time and runtime, but it is a pub quiz, not a puzzle.
- Scrap-your-boilerplate is extremely slow in runtime.
- `DeriveGeneric` is least bad, but you get
  - either bad compile time,
  - or bad runtime.
- Old but gold: it's often the best to use `CPP` for boilerplate generation.

Thanks all!

Haskell: where stability meets evolution.
A language that grows with you.

andrew.lelechenko@gmail.com

github.com/Bodigrim

Bodigrim