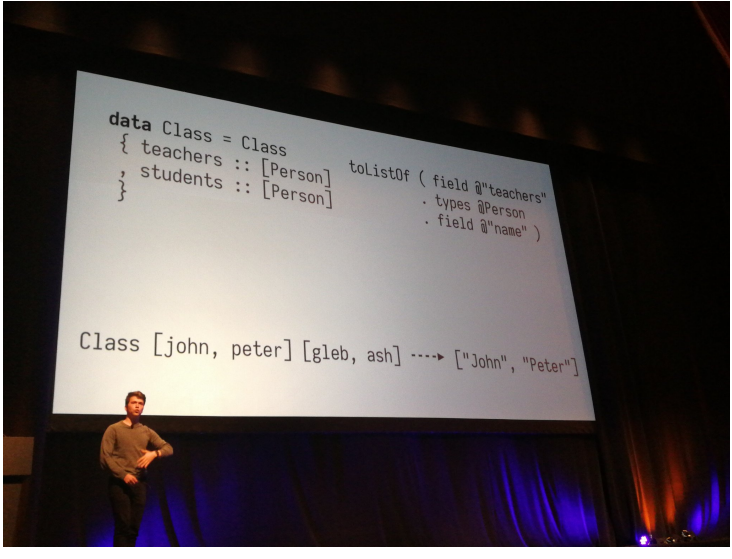
A promise checked is a promise kept: Inspection Testing

Joachim Breitner, DFINITY Foundation

March 22, 2019
BobKonf
Berlin

An anecdote

Matthew Pickering talking about generic-lens at IFL'17



Detour: Lenses

```
data Employee = MkEmployee String Int

getAge :: Employee -> Int
getAge (MkEmployee name age) = age

setAge :: Int -> Employee -> Employee
setAge newAge (MkEmployee name age)
  = MkEmployee name newAge
```

Detour: Lenses

```
data Employee = MkEmployee String Int

getAge :: Employee -> Int
getAge e = e ^. ageLens

setAge :: Int -> Employee -> Employee
setAge a = ageLens .~ a

ageLens :: Lens' Employee Int
ageLens = ...
```

Detour: Lenses

```
data Employee = MkEmployee String Int

getAge :: Employee -> Int
getAge e = e ^. ageLens

setAge :: Int -> Employee -> Employee
setAge a = ageLens .~ a

ageLens :: Lens' Employee Int
ageLens f (MkEmployee name age)
  = fmap (\newAge -> MkEmployee name newAge) (f age)
```

Detour: Lenses

```haskell
data Employee = MkEmployee String Int deriving Generic

getAge :: Employee -> Int
getAge e = e ^. typed @Int

setAge :: Int -> Employee -> Employee
setAge a = typed @Int .~ a
```

Detour: Lenses

```
data Employee = MkEmployee { name :: String, age :: Int }
    deriving Generic
getAge :: Employee -> Int
getAge e = e ^. field @"age"

setAge :: Int -> Employee -> Employee
setAge a = field @"age" .~ a
```

Using *super* in this manner might be more convenient for programmers who are not used to programming with lenses.

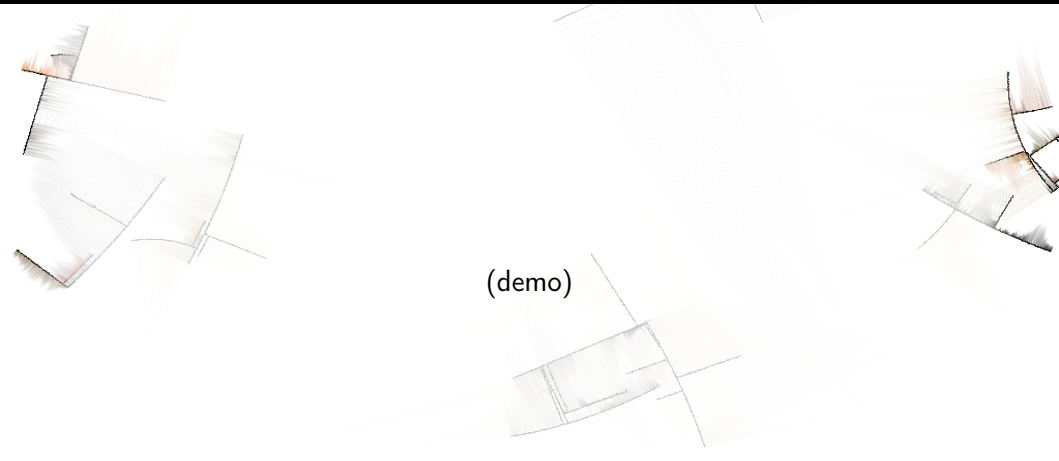
7. Conclusion

Deriving lenses generically gives the programmer the best of all possible worlds. The frugality to only define whichever lenses they need to use, the confidence that their abstraction will be without cost and the flexibility to four different types of lenses. This expressive and lightweight solution will hopefully inspire other library writers to embrace `GHC.Generics` as a solid basis on which to build their libraries.

References

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *ACM SIGPLAN Notices*

A promise broken



(demo)

Fusion

Most of the functions in this module are subject to *fusion*, meaning that a pipeline of such functions will usually allocate at most one `Text` value.

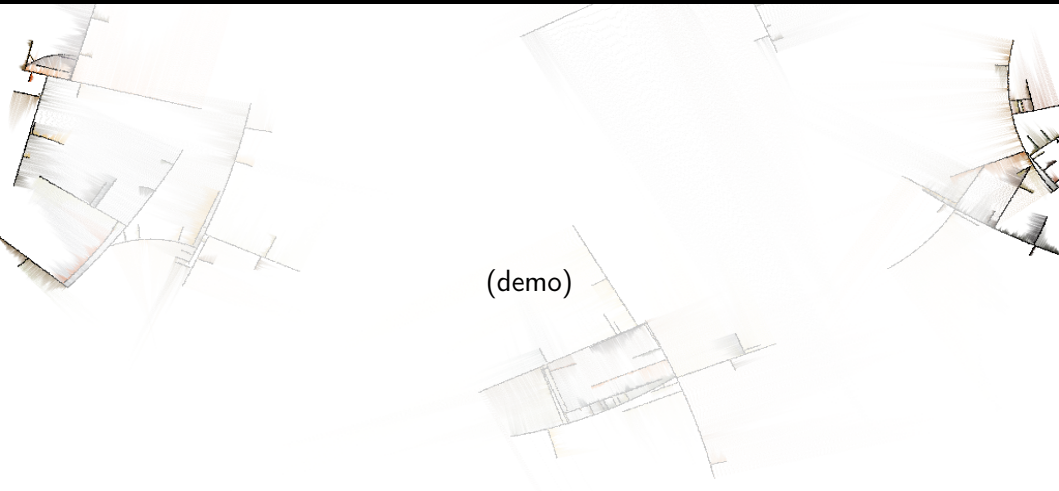
As an example, consider the following pipeline:

```
import Data.Text as T
import Data.Text.Encoding as E
import Data.ByteString (ByteString)

countChars :: ByteString -> Int
countChars = T.length . T.toUpper . E.decodeUtf8
```

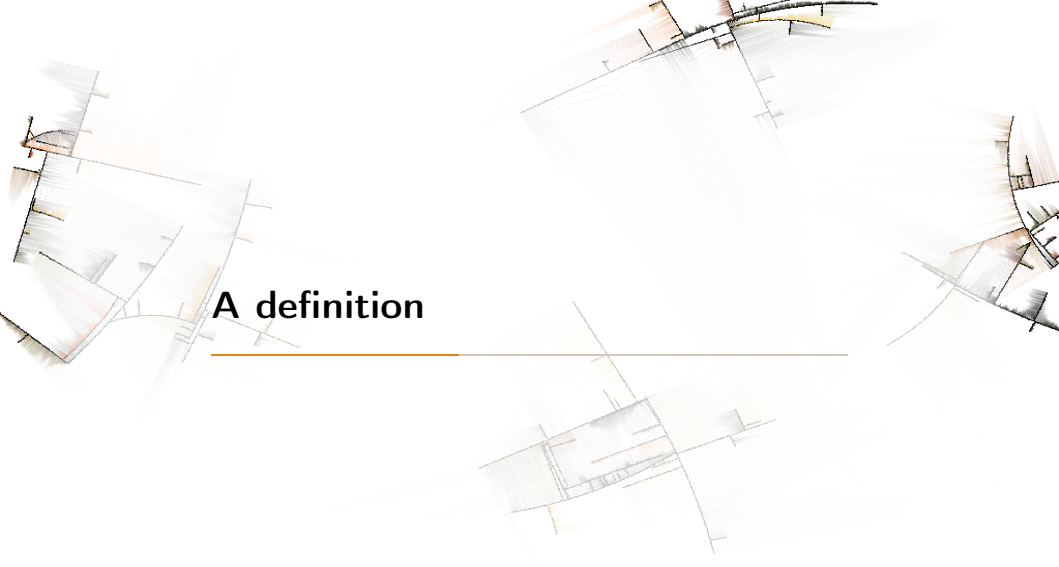
From the type signatures involved, this looks like it should allocate one `ByteString` value, and two `Text` values. However, when a module is compiled with optimisation enabled under GHC, the two intermediate `Text` values will be optimised away, and the function will be compiled down to a single loop over the source `ByteString`.

Functions that can be fused by the compiler are documented with the phrase "Subject to fusion".

A promise broken (again)



(demo)



A definition

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Inspection Testing

is when a non-functional property of a compilation artifact of a specific piece of code is specified declaratively by the programmer and checked, during compilation, by the compiler.

Brainstorming. . .

- Equality of generic vs. manual code.
- Equivalence of generic vs. manual code.
- Elimination of intermediate data structures (fusion)
- Elimination of dictionary passing
- Absence of allocations
- Absence of slow function calls
- Absence of branches
- *insert more good ideas here*

A tool

Wishful thinking

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE InspectionTesting #-}
module Example (Employee(..)) where

import GHC.Generics (Generic)
import Data.Generics.Product (field)


data Employee = MkEmployee String Int deriving Generic

                                  ...

inspect ageLensManual === ageLensGeneric
```

Wishful thinking

- New syntactic constructs
- Compiler support for inspection-testing

Next-best solution

- ~~New syntactic constructs~~
  Template Haskell top-level splice
- ~~Compiler support for inspection-testing~~
  Compiler plugin for inspection-testing

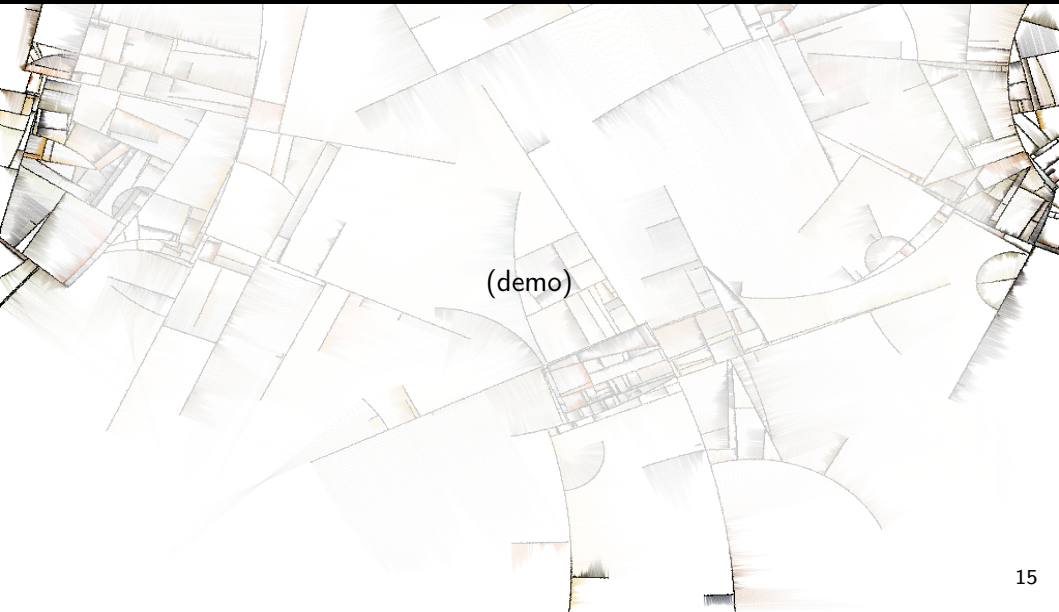
Reality

```
{-# LANGUAGE DeriveGeneric, TemplateHaskell #-}
{-# OPTIONS_GHC -fplugin=Test.Inspection.Plugin #-}
module Example (Employee(..)) where

import GHC.Generics (Generic)
import Data.Generics.Product (field)
import Test.Inspection

data Employee = MkEmployee String Int deriving Generic

                                  ...

inspect (’ageLensManual === ’ageLensGeneric)
```

Reality

```
$ ghc -O Example.hs
[1 of 1] Compiling GenericLens ( Example.hs, Example.o )
Example.hs:21:1: ageLensManual === ageLensGeneric passed.
inspection testing successful
      expected successes: 1
```

Scaling it up



Results

Data.Text

	docs	fuses
unpackCString#	✓	✗
reverse	✓	✗
scanl1	✓	✗
scanr1	✓	✗
takeWhileEnd	✓	✗
dropWhileEnd	✓	✗
dropAround	✓	✗
strip	✓	✗
stripStart	✓	✗
stripEnd	✓	✗
length	✓	✗
find	✗	✓
index	✗	✓

Data.Text.Lazy

	docs	fuses
decodeUtf8	✓	✗
toCaseFold	✓	✗
scanl1	✓	✗
dropAround	✓	✗
strip	✓	✗
stripStart	✓	✗
unfoldr	✗	✓
unfoldrN	✗	✓
find	✗	✓
index	✗	✓

This tool has been invaluable in ensuring the performance guarantees, without having to manually inspect the generated core after every single commit.

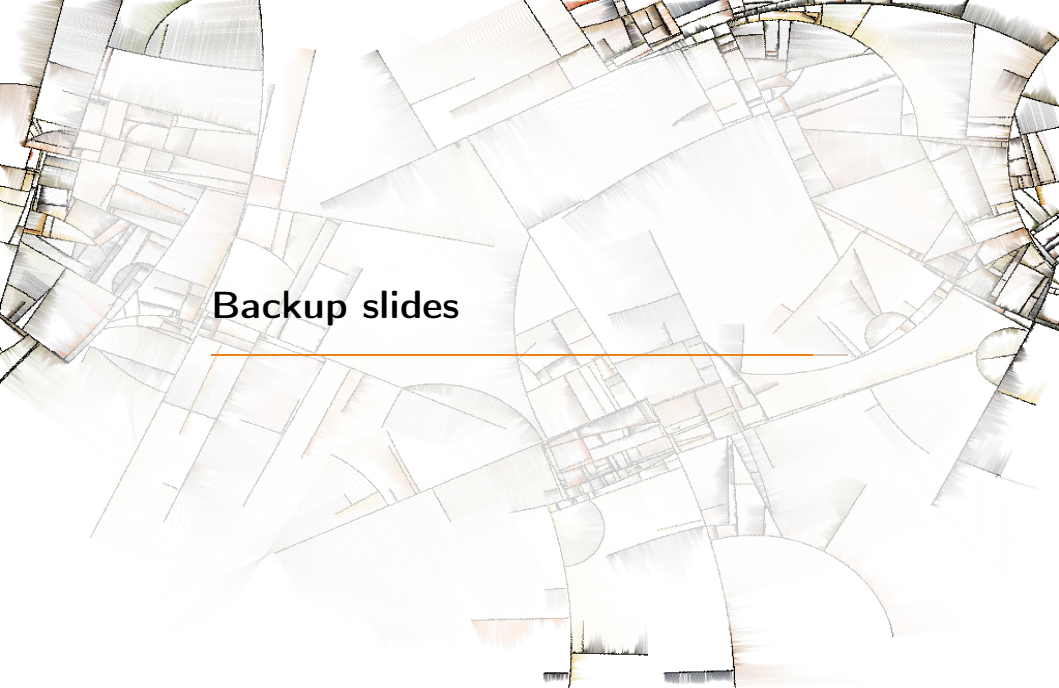
—Csongor Kiss, `generic-lens` author

[Without inspection-testing,] even writing that module in the first place would feel like a bad idea, given how much one would need to stare into the Core.

—Oleg Grenrus, `fin` and `vec` author

Further users (so far):

`generics-sop`, `hkt`, `prospect`, `ghc-justdoit`, `postgresql-simple`, `algebraic-graphs`, `prospect`,



Backup slides

How does it work?

- `Test.Inspection.inspect :: Obligation -> Q [Dec]`
  - stores the `Obligation` as a *module annotation*
  - annotates each referenced `Name`, to keep it alive
- `Test.Inspection.Plugin.plugin`
  - runs at the end of the pipeline
  - collects all `Obligation` annotations
  - uses `thNameToGhcName` to convert Template Haskell names
  - slices the relevant parts of the module
  - checks the properties
  - aborts compilation upon failure
    (or stores it in a static value in the program)