



Beyond correct and fast: Inspection Testing

Joachim Breitner, University of Pennsylvania

December 4, 2017, IBM PL days, T.J. Watson Research Center

The background of the slide features several strips of paper that appear to be torn and floating in the air. These strips are rendered with a sense of motion and depth, some showing a light beige color and others a slightly darker, greyish tone. They are scattered across the white background, creating a dynamic and artistic effect.

An anecdote

The generic-lens library

```
data Employee = MkEmployee String Int
```

```
ageLens :: Lens' Employee Int
```

```
ageLens f (MkEmployee name age)
```

```
    = fmap (\newAge -> MkEmployee name newAge) (f age)
```

The generic-lens library

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

```
ageLens :: Lens' Employee Int  
ageLens = typed @Int
```

The generic-lens library

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

```
ageLens :: Lens' Employee Int
ageLens = typed @Int
```

-- In the libraries (simplified):

```
from :: Generic a => a -> Rep a
to   :: Generic a => Rep a -> a
typed :: Generic s => Lens' s a
typed = ravel (dimap from (fmap to)) . gtyped)
```

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

B. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. *Closed*

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

B. A. Eisenberg, D. Mitzel, S. Peyton Jones, and S. Weirich. *Closed*

A promise broken

Manual code:

```
ageLensManual :: Lens' Employee Int
ageLensManual =
  \ (@ (f_a6t0 :: * -> *))
    ($dFunctor_a6uA :: Functor f_a6t0)
    (f1_a6ps :: Int -> f_a6t0 Int)
    (ds_d7Kk :: Employee) ->
  case ds_d7Kk of _ -> MkEmployee name_a6pt age_a6pu ->
  fmap
    @ f_a6t0
    $dFunctor_a6uA
    @ Int
    @ Employee
    (\ (newAge_a6pv :: Int) -> MkEmployee name_a6pt newAge_a6pv)
    (f1_a6ps age_a6pu)
  }
```


A promise broken

Manual code:

```
ageLensManual :: Lens' Employee Int
ageLensManual =
  \ (@ (f_a6t0 :: * -> *))
    ($dFunctor_a6uA :: Functor f_a6t0)
    (f1_a6ps :: Int -> f_a6t0 Int)
    (ds_d7Kk :: Employee) ->
  case ds_d7Kk of _ -> { MkEmployee name_a6pt age_a6pu ->
    fmap
      @ f_a6t0
      $dFunctor_a6uA
      @ Int
      @ Employee
      (\ (newAge_a6pv :: Int) -> MkEmployee name_a6pt new
        (f1_a6ps age_a6pu)
      }
  }
```

Generic code:

```

ageLensGeneric1
  :: forall x_X7NQ (f1_X7NS :: * -> *).
    Functor f1_X7NS =>
      (Int -> f1_X7NS Int)
    -> M1
      D
        ('MetaData "Employee" "GenericLens" "main" 'False)
      (M1
        C
          ('MetaCons "MkEmployee" 'PrefixI 'False)
          (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceUnpackedness)
            x_X7NQ
              6pv) -> f1_X7NS (M1
                D
                  ('MetaData "Employee" "GenericLens" "main" 'False)
                (M1
                  C
                    ('MetaCons "MkEmployee" 'PrefixI 'False)
                    (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceUnpackedness)
                      x_X7NQ)
                )
              )
            )
          )
        )
      )
ageLensGeneric1 =
  \ (@ x_X7NQ) (@ (f1_X7NS :: * -> *)) ($dFunctor_X7NU :: Functor f1_X7NS) =>
    let {
      f2_a7MF
        :: f1_X7NS (M1
          C
            ('MetaCons "MkEmployee" 'PrefixI 'False)
            (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceUnpackedness)
              x_X7NQ)
          )
    }
    in f2_a7MF

```

A promise broken

Manual code:

Generic code:

[illegible]

A promise broken

```
proj :: Functor f => f a -> Coyoneda f a
proj fa = Coyoneda id fa

ravel :: Functor f => ((a -> Coyoneda f b) -> (s -> Coyoneda f t))
    -> (a -> f b) -> (s -> f t)
ravel coy f s = inj $ coy (\a -> proj (f a)) s
```



mpickering commented on 3 Sep

Collaborator +

Hi Joachim,

After making such a claim in the talk, I tried it out again and indeed with the most recent version of the library the situation is worse and the generated core is different. I just tried it again with an older version, and I see what is claimed with the paper.

@kcsongor What have you changed since about the 0.2 version? The precise commit is in the paper repo.



kcsongor commented on 3 Sep

Owner +

I'm investigating this regression and will try to get back later today



nomeata commented on 3 Sep

Contributor +

Heh. At least a good story for me to advocate for "optimization unit tests" that I envision :-)

The background of the slide is an abstract architectural sketch. It features various geometric shapes, lines, and shaded areas in a light, sketchy style. The sketch appears to be a conceptual drawing of a building or a site plan, with some areas highlighted in a light orange or yellow color. The overall style is artistic and hand-drawn.

A definition

The background of the slide is an abstract architectural drawing. It features various geometric shapes, lines, and shaded areas in a light, sketchy style. The drawing appears to be a plan or section of a building, with some areas highlighted in a light brown or tan color. The overall style is artistic and technical.

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.

An abstract architectural drawing in the background, featuring various geometric shapes, lines, and shaded areas in a light, sketchy style, resembling a floor plan or a technical drawing.

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.

An abstract architectural drawing in the background, featuring various geometric shapes, lines, and shaded areas in a light, sketchy style. The drawing includes rectangular blocks, curved lines, and some areas with light brown shading, suggesting a floor plan or a conceptual architectural sketch.

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.

An abstract background pattern consisting of various overlapping geometric shapes, primarily rectangles and polygons, in shades of gray, beige, and light brown. The shapes are arranged in a complex, non-repeating manner, creating a textured, architectural feel.

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.

An abstract geometric pattern composed of various overlapping shapes, including rectangles, triangles, and polygons, in shades of gray, white, and light orange. The pattern is dense and intricate, resembling a complex architectural plan or a stylized map.

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.

Haskell Implementation

The screenshot shows the GitHub repository page for 'nomeata / Inspection-testing'. The repository has 58 commits, 1 branch, 5 releases, 4 contributors, and is licensed under MIT. The file list includes 'Test', 'examples', '.gitignore', '.travis.yml', 'ChangeLog.md', 'LICENSE', 'README.md', 'Setup.hs', 'Test.hs', and 'inspection-testing.cabal'. The 'README.md' file is selected, showing the title 'Inspection Testing for Haskell' and a description: 'This GHC plugin allows you to embed assertions about the intermediate code into your Haskell code, and have them checked by GHC. This is called inspection testing (as it automates what you do when you manually inspect the intermediate code)'. The 'Synopsis' section states: 'See the Test.Inspection module for the documentation, but there really isn't much more to it than:'. A code block shows the template Haskell code:

```
{-# LANGUAGE TemplateHaskell #-}
```

nomeata / **Inspection-testing** Unwatch 2 Star 32 Fork 4

[Code](#) [Issues](#) [Pull requests](#) [Projects](#) [Wiki](#) [Insights](#) [Settings](#)

Inspection Testing for Haskell Edit

[Add topics](#)

58 commits 1 branch 5 releases 4 contributors MIT

Branch: master - [New pull request](#) [Create new file](#) [Upload files](#) [Find file](#) [Clone or download](#)

nomeata Make '[--]' a bit more liberal	Latest commit 4658d3 13 days ago
Test	Make '[--]' a bit more liberal 13 days ago
examples	In GenericLens test, also test the non-Yoneda variant 19 days ago
.gitignore	Start converting ghc-proofs into inspection-test a month ago
.travis.yml	Disable ghc-8.0.1 26 days ago
ChangeLog.md	Make '[--]' a bit more liberal 13 days ago
LICENSE	Initial check-in 10 months ago
README.md	Do not fail because of expected failures 21 days ago
Setup.hs	Initial check-in 10 months ago
Test.hs	Support '-dplugin-opt=Test.Inspection.Plugin-keep-going' 26 days ago
inspection-testing.cabal	Make '[--]' a bit more liberal 13 days ago

[README.md](#)

Inspection Testing for Haskell

This GHC plugin allows you to embed assertions about the intermediate code into your Haskell code, and have them checked by GHC. This is called *inspection testing* (as it automates what you do when you manually inspect the intermediate code).

Synopsis

See the `Test.Inspection` module for the documentation, but there really isn't much more to it than:

```
{-# LANGUAGE TemplateHaskell #-}
```

generic-lens: Happy end

```
56 -----
57 -- * Tests
58 -- The inspection-testing plugin checks that the following equalities hold, by
59 -- checking that the LHSs and the RHSs are CSEd. This also means that the
60 -- runtime characteristics of the derived lenses is the same as the manually
61 -- written ones above.
62
63 fieldALensName :: Lens' Record Int
64 fieldALensName = field @"fieldA"
65
66 fieldALensType :: Lens' Record Int
67 fieldALensType = typed @Int
68
69 fieldALensPos :: Lens' Record Int
70 fieldALensPos = position @1
71
72 subtypeLensGeneric :: Lens' Record Record2
73 subtypeLensGeneric = super
74
75 typeChangingGeneric :: Lens (Record3 a) (Record3 b) a b
76 typeChangingGeneric = field @"fieldA"
77
78 typeChangingGenericPos :: Lens (Record3 a) (Record3 b) a b
79 typeChangingGenericPos = position @1
80
81 typeChangingGenericCompose :: Lens (Record3 (Record3 a)) (Record3 (Record3 b)) a b
82 typeChangingGenericCompose = field @"fieldA" . field @"fieldA"
83
84 inspect $ 'fieldALensManual === 'fieldALensName
85 inspect $ 'fieldALensManual === 'fieldALensType
86 inspect $ 'fieldALensManual === 'fieldALensPos
87 inspect $ 'subtypeLensManual === 'subtypeLensGeneric
88 inspect $ 'typeChangingManual === 'typeChangingGeneric
89 inspect $ 'typeChangingManual === 'typeChangingGenericPos
90 inspect $ 'typeChangingManualCompose === 'typeChangingGenericCompose
```

Already used by:

- generic-lens
- generic-sop
- vec

Many applications

- Equality of generic vs. manual code.
- Equivalence of generic vs. manual code.
- Elimination of intermediate data structures (fusion)
- Strictness/laziness properties
- Absence of allocations
- Absence of slow function calls
- Absence of branches
- Vectorization and SIMD
- *insert more good ideas here*

The background of the slide is a complex, abstract geometric pattern. It consists of numerous overlapping, semi-transparent shapes in various shades of gray, beige, and light brown. These shapes include rectangles, triangles, and irregular polygons, some of which are further divided into smaller sections by thin black lines. The overall effect is a dense, layered composition that resembles a stylized architectural plan or a modernist artwork.

Thank you