Beyond correct and fast: Inspection Testing

Joachim Breitner, University of Pennsylvania

December 4, 2017, IBM PL days, T.J. Watson Research Center
An anecdote
The generic-lens library

```haskell
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

```haskell
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 \textit{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 \texttt{GHC.Generics} as a solid basis on which to build their libraries.

References

A promise made

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

A promise broken

Manual code:

```haskell
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:

```haskell
ageLensManual :: Lens' Employee Int
ageLensManual =
\ (@ (f_a6t0 :: * -> *))
  (\Functor_a6uA :: Functor f_a6t0)
  (f1_a6ps :: Int -> f_a6t0 Int)
  (ds_d7Kk :: Employee) ->
case ds_d7Kk of _ { MkEmployee name_a6pt age_a6pu ->
  fmap
 fmap
$fFunctor_a6uA @ f_a6t0

let
(f1_a6ps age_a6pu)

ageLensGeneric = \ (@ (f_a6sQ :: *->*)) ($dFunctor_a6t1 :: Functor f_a6sQ) ->
casedTyped @ Employee @ Int $fGenericEmployee (ageLensGeneric1 'cast' ...) @ f_a6sQ

ageLensGeneric :: Lens' Employee Int

−−
ageLensGeneric1

Generic code:

```haskell

```
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)
    }
```

Generic code:

```
ageLensGeneric1 ::
forall x_X7NQ (f1_X7NS ::*->*).Functor f1_X7NS =>
  (Int -> f1_X7NS Int) ->
  M1D('MetaData
    "Employee" "GenericLens" "main"
    'False)(M1C('MetaCons
      "MkEmployee"
      'PrefixI
      'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))x_X7NQ->
    f1_X7NS (M1D('MetaData
      "Employee" "GenericLens" "main"
      'False)(M1C('MetaCons
        "MkEmployee"
        'PrefixI
        'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))x_X7NQ)
ageLensGeneric1 =
  (@ x_X7NQ) (@ (f1_X7NS ::*->*)) ($dFunctor_X7NU :: Functor f1_X7NS) ->
  let {f2_a7MF:: f1_X7NS (M1C('MetaCons
      "MkEmployee"
      'PrefixI
      'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))x_X7NQ)->
        f1_X7NS (M1D('MetaData
          "Employee" "GenericLens" "main"
          'False)(M1C('MetaCons
            "MkEmployee"
            'PrefixI
            'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))x_X7NQ)
f2_a7MF =
    fmap f1_X7NS$dFunctor_X7NU (M1C('MetaCons
      "MkEmployee"
      'PrefixI
      'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))x_X7NQ)@ (M1D('MetaData
      "Employee" "GenericLens" "main"
      'False)(M1C('MetaCons
        "MkEmployee"
        'PrefixI
        'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))x_X7NQ)@ (mIso1 @ (M1C('MetaCons
          "MkEmployee"
          'PrefixI
          'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))@ x_X7NQ)'cast' ...) }
  in
  let {f4_X7Ne:: f1_X7NS (:*:)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String)) (S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)) x_X7NQ)->
        f1_X7NS (M1C('MetaCons
          "MkEmployee"
          'PrefixI
          'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))@ (M1C('MetaCons
            "MkEmployee"
            'PrefixI
            'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int))@ x_X7NQ)'cast' ...) }
    in
    let {f5_X7Ni :: f1_X7NS (K1 R Int x_X7NQ) ->
          f1_X7NS (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness ... 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (K1 R Int) x_X7NQ) ((mIso1 @ (K1 R Int) @ x_X7NQ) 'cast' ...) }
    in
      (x1_X7Nt :: Int -> f1_X7NS Int)(x2_a7MM:: M1D('MetaData
        "Employee" "GenericLens" "main"
        'False)(M1C('MetaCons
          "MkEmployee"
          'PrefixI
          'False)(S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 String) :*: S1 ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy) (Rec0 Int)))x_X7NQ) ->
    f2_a7MF(f4_X7Ne( case x2_a7MM 'cast' ...
      of _ { :*: a1_a7N7 b1_a7N8 ->
          fmap f1_X7NS$dFunctor_X7NU (M1 S ('MetaSel 'Nothing 'NoSourceUnpackedness 'NoSourceStrictness ... f1_X7NS $dFunctor_X7NU @ Int @ (K1 R Int x_X7NQ) (($fGHasTypeK1a1 @ Int) 'cast' ...) (x1_X7Nt (b1_a7N8 'cast' ...)))))))}
```

```
ageLensGeneric :: Lens' Employee Int
ageLensGeneric =
  (@ (f_a6sQ ::*->*)) ($dFunctor_a6t1 :: Functor f_a6sQ) ->
  $w$ctyped @ Employee @ Int $fGenericEmployee (ageLensGeneric1 'cast' ...) @ f_a6sQ $dFunctor_a6t1
```
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
```
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.
### Haskell Implementation

#### 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:

```haskell
{-# LANGUAGE TemplateHaskell #-)
```
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*
Thank you