Deriving Lenses using Generics

Csongor Kiss
Imperial

Matthew Pickering
Bristol

Toby Shaw
Imperial

Which functions can we derive just by knowing the structure of a data type?

Overall Structure

Types

```
data T = T { a :: Int
           , b :: String
           , c :: Bool
           }
```

Named Fields

Which lenses can we derive just by knowing the structure of a datatype?

Lenses

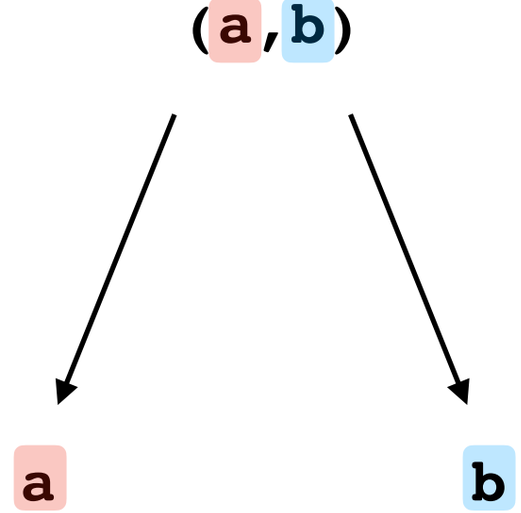
- An abstraction for product-like data structures

```
data Lens s a =
Lens { view :: s -> a
     , set :: a -> s -> s }
```

```
data Lens s a =
  Lens { view :: s -> a
       , set :: a -> s -> s }
```

_1 is the lens which focuses on the first element in a tuple

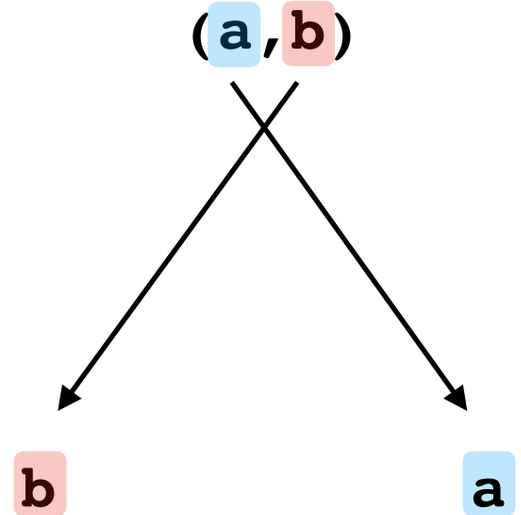
```
_1 :: Lens (a, b) a
```



```
data Lens s a =
  Lens { view :: s -> a
       , set :: a -> s -> s }
```

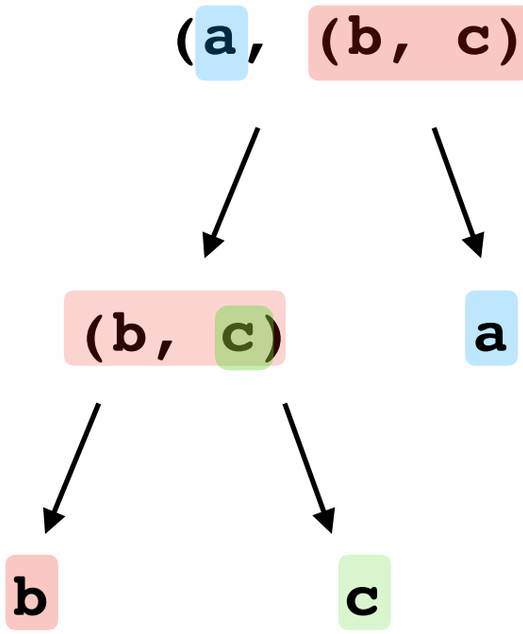
_2 is the lens which focuses on the second element in a tuple

```
_2 :: Lens (a, b) b
```



Lenses are compositional

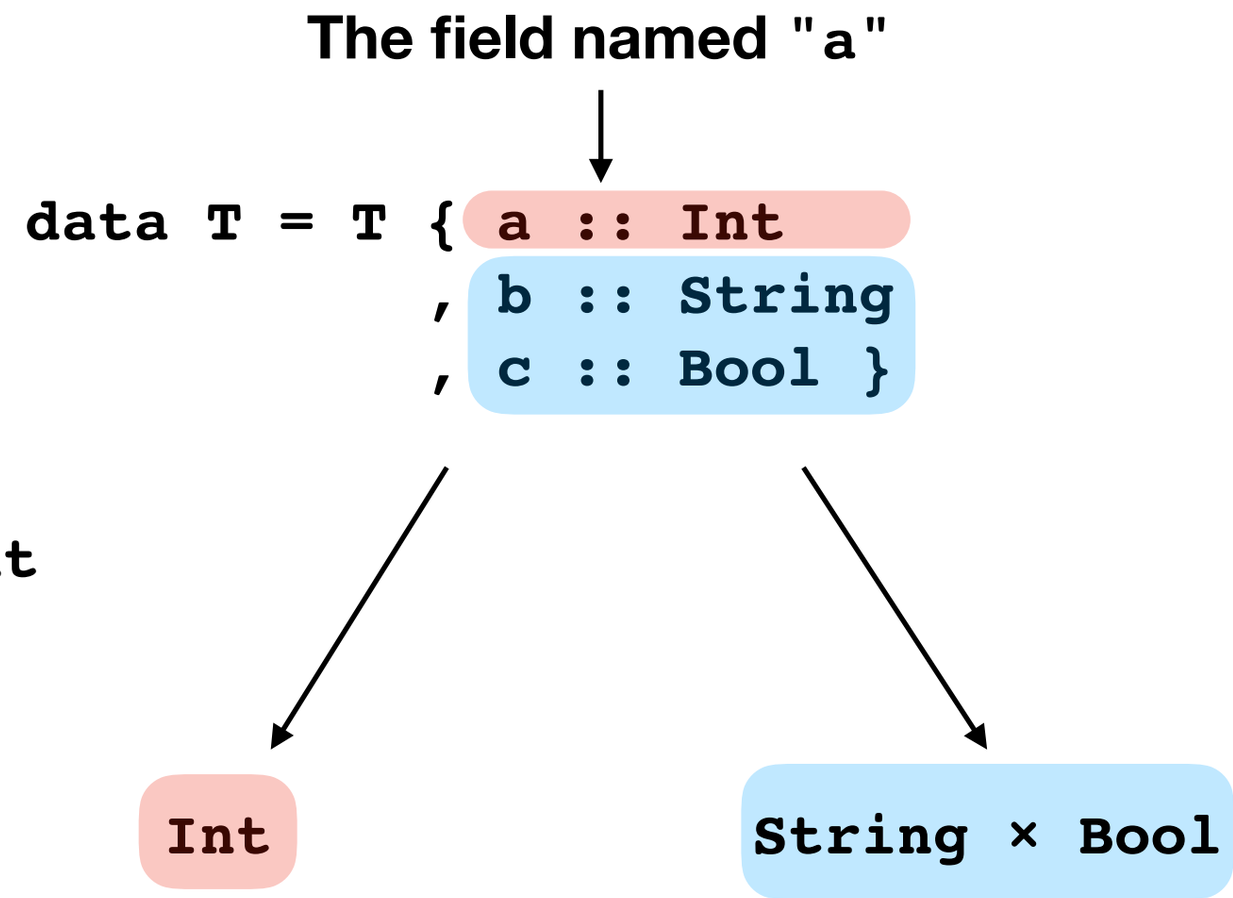
```
<o> :: Lens s t -> Lens t u -> Lens s u
```




```
_2 :: Lens (a, (b, c)) (b, c)

 <o>

_1 :: Lens (b, c) b
```

Name-directed

```
field @“a” :: Lens T Int
```

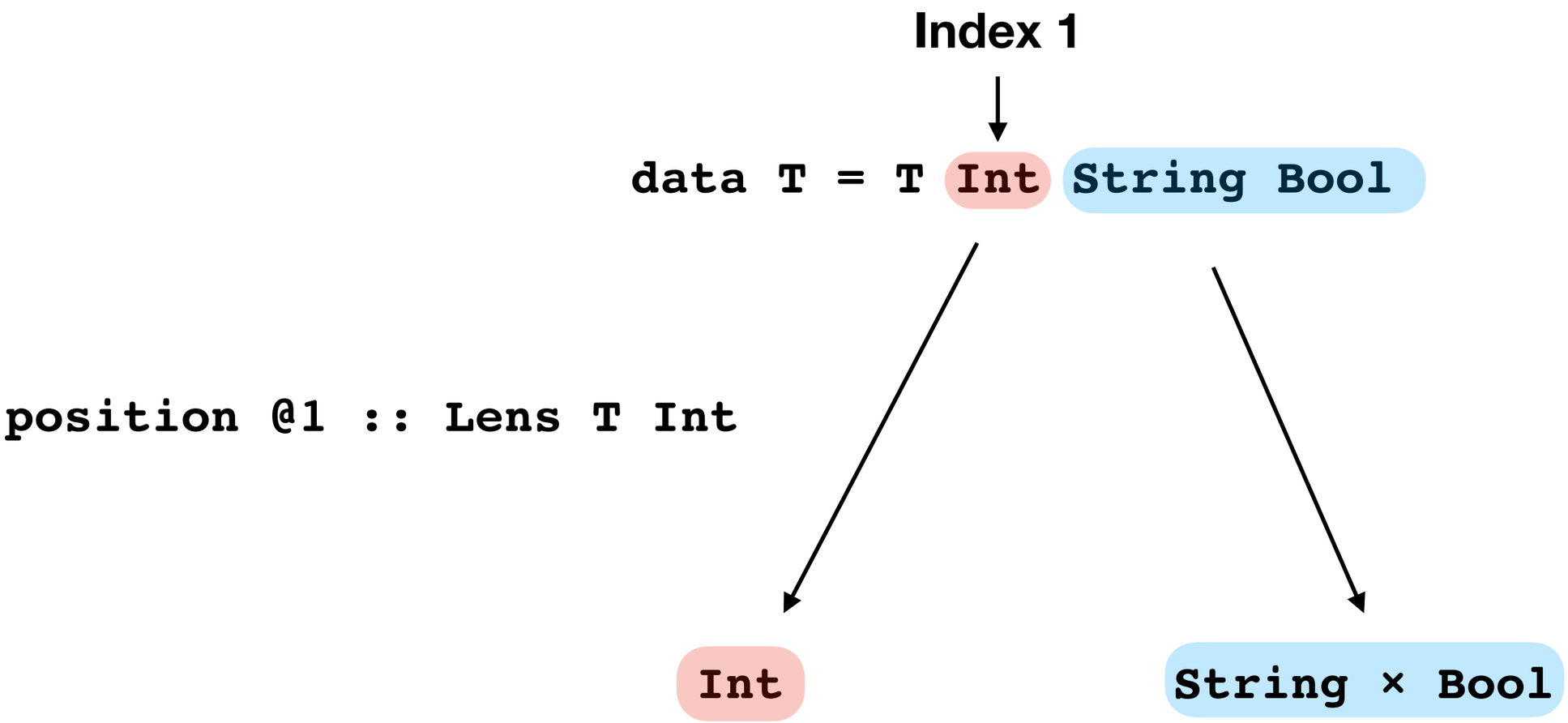


Name-directed

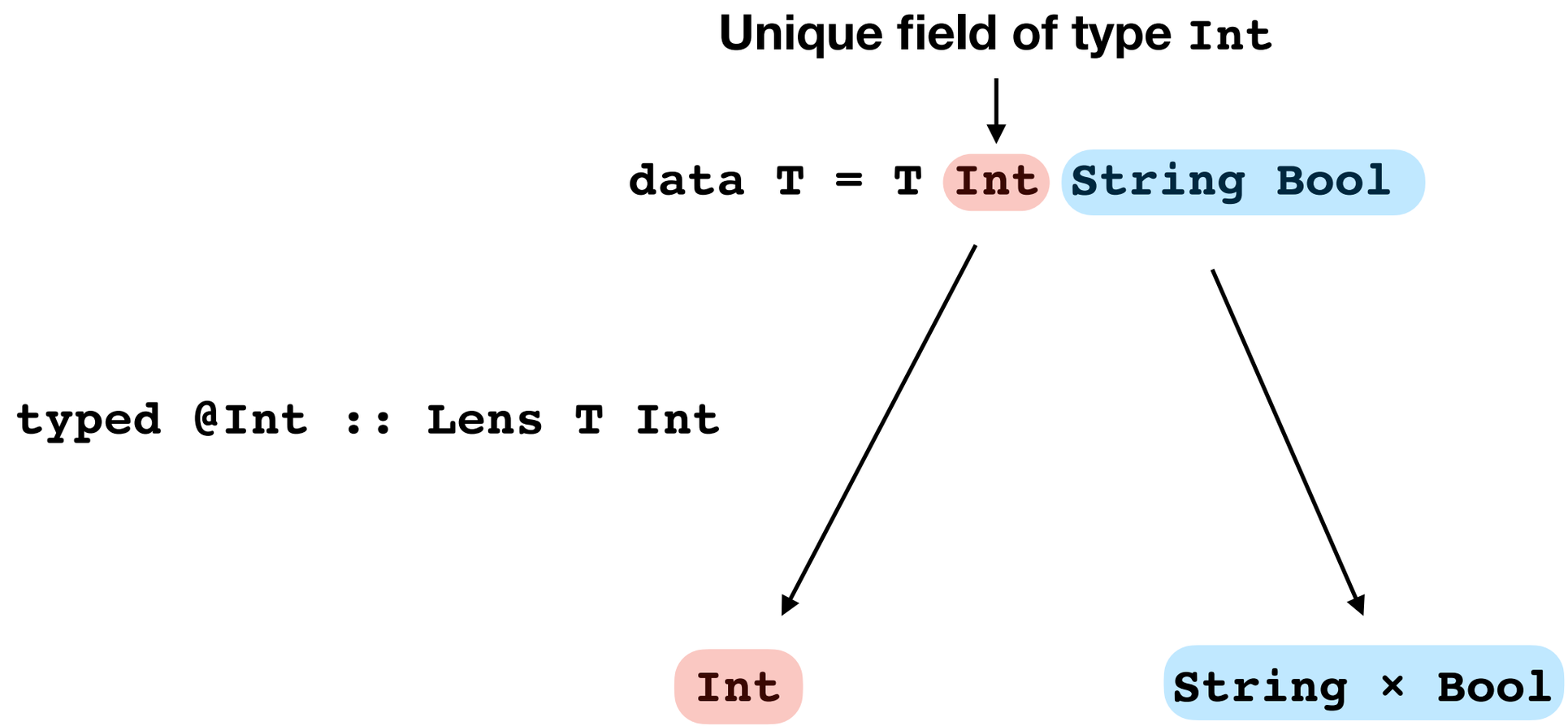
```
data T = T { a :: Int
           , b :: String
           , c :: Bool }

field @“a” :: Lens T Int
```

```
> v = T 5 “foo” True

> view (field @“a”) v
5
> view (field @“c”) v
True

> set (field @“b”) “bar” v
T 5 “bar” True
```

Index-directed



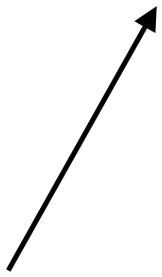
Type-directed



Structure-directed

```
data S = S { a :: Int
           , b :: String
           }
```

```
data T = T { a :: Int
           , b :: String
           , c :: Bool
           }
```



`T` is like `S` but has an additional field

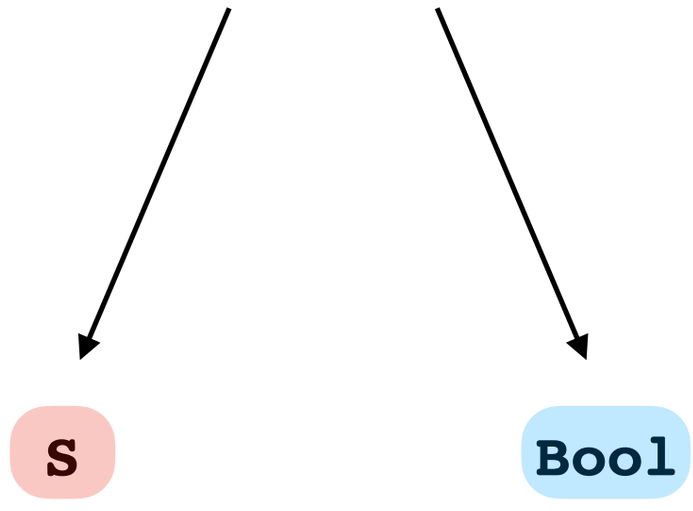
Structure-directed

```
data S = S { a :: Int
           , b :: String
           }
```

Specify the super type

```
super @S :: Lens T S
```

```
data T = T { a :: Int
           , b :: String
           , c :: Bool
           }
```



Structure-directed

```
data S = S { a :: Int
           , b :: String
           }
```

```
data T = T { a :: Int
           , b :: String
           , c :: Bool
           }
```

```
> :t (super @S)
Lens T S
> view (super @S) (T 5 “Csongor” True)
S 5 “Csongor”
> set (super @S) (S 6 “Matt”) (T 5 “abc” True)
T 6 “Matt” True
```

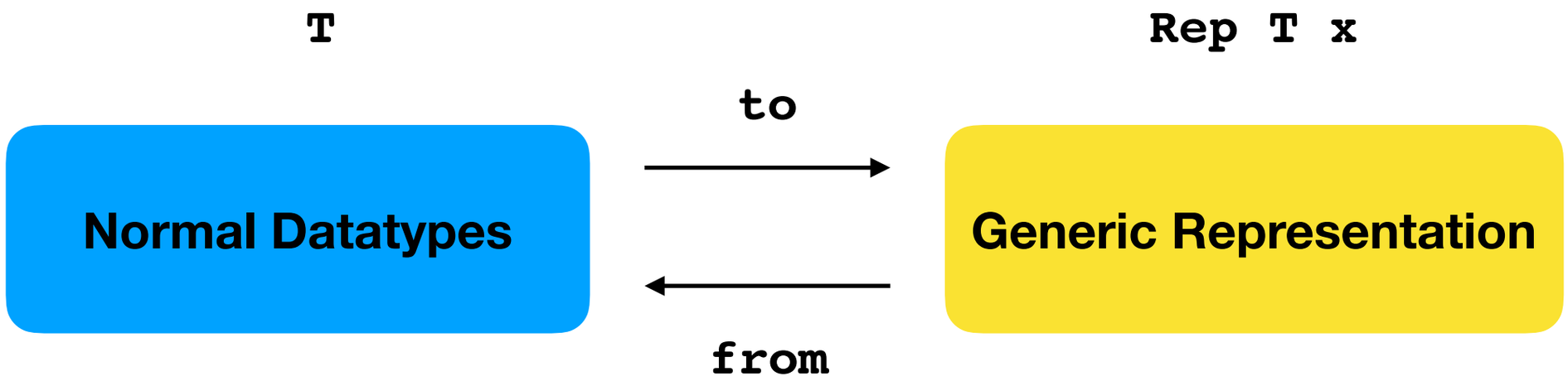
```
data FormResponse = FormResponse { name :: String
                                 , age :: Int }

data Person = Person { name :: String
                     , age :: Int
                     , uniqueID :: UID
                     , dateModified :: Date
                     , userSettings :: Settings }
```

```
updatePerson :: Person -> IO Person
updatePerson p = do
formResponse <-
 getFormResponse (view (typed @Settings
                          <o> super @FormSettings))
                       p)
date <- getCurrentDate
return $ p & set (super @FormResponse)
                 formResponse
           & set (field @“dateModified”)
                 date
```

Which lenses can we derive just by knowing the structure of a datatype?

Implementation

GHC.Generics




```
class Generic a where
   type family Rep a :: * → *
   from :: a → Rep a x
   to :: Rep a x → a
```

GHC.Generics

Empty Types	`data V1 p`
Unary Types	`data U1 p = U1`
Sum Types	`data (:+:) f g p = L1 (f p)` `  | R1 (g p)`
Product Types	`data (:*:) f g p = (f p) :*: (g p)`
Constant Types	`newtype K1 i c p = K1 { unK1 :: c }`

Structure becomes evident by inspecting the type

GHC.Generics

From GHC.Generics documentation

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving Generic
```

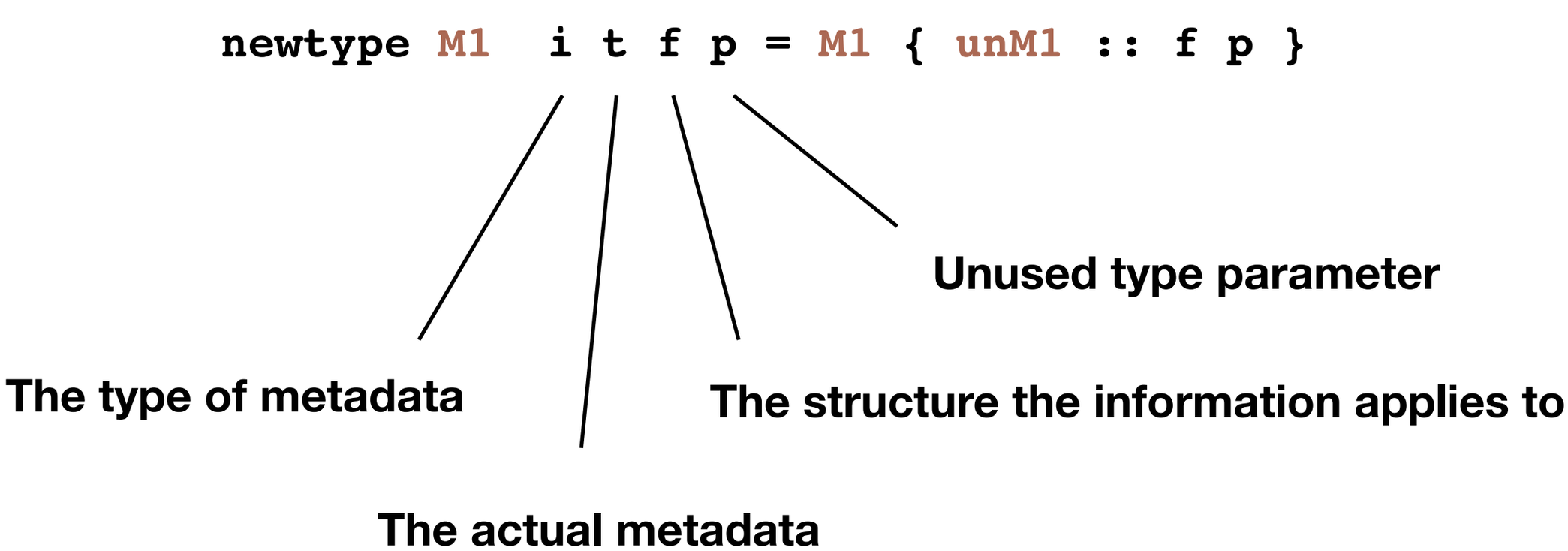
GHC.Generics

From GHC.Generics documentation

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving Generic
```

```
instance Generic (Tree a) where
  type Rep (Tree a) =
    Rec0 a
    :+:
    (Rec0 (Tree a) :*: Rec0 (Tree a))
```

Metadata Fields



```
data T = T { a :: Int } deriving Generic

type Rep T =
  M1 D ('MetaData "T" "T" "main" False)
    (M1 C (‘MetaCons "T" 'PrefixI ‘GHC.Types.True)
      (M1 S
        ('MetaSel
          (Just "a")
          'NoSourceUnpackedness
          'NoSourceStrictness
          ‘DecidedLazy)
        (Rec0 GHC.Types.Int)))
```

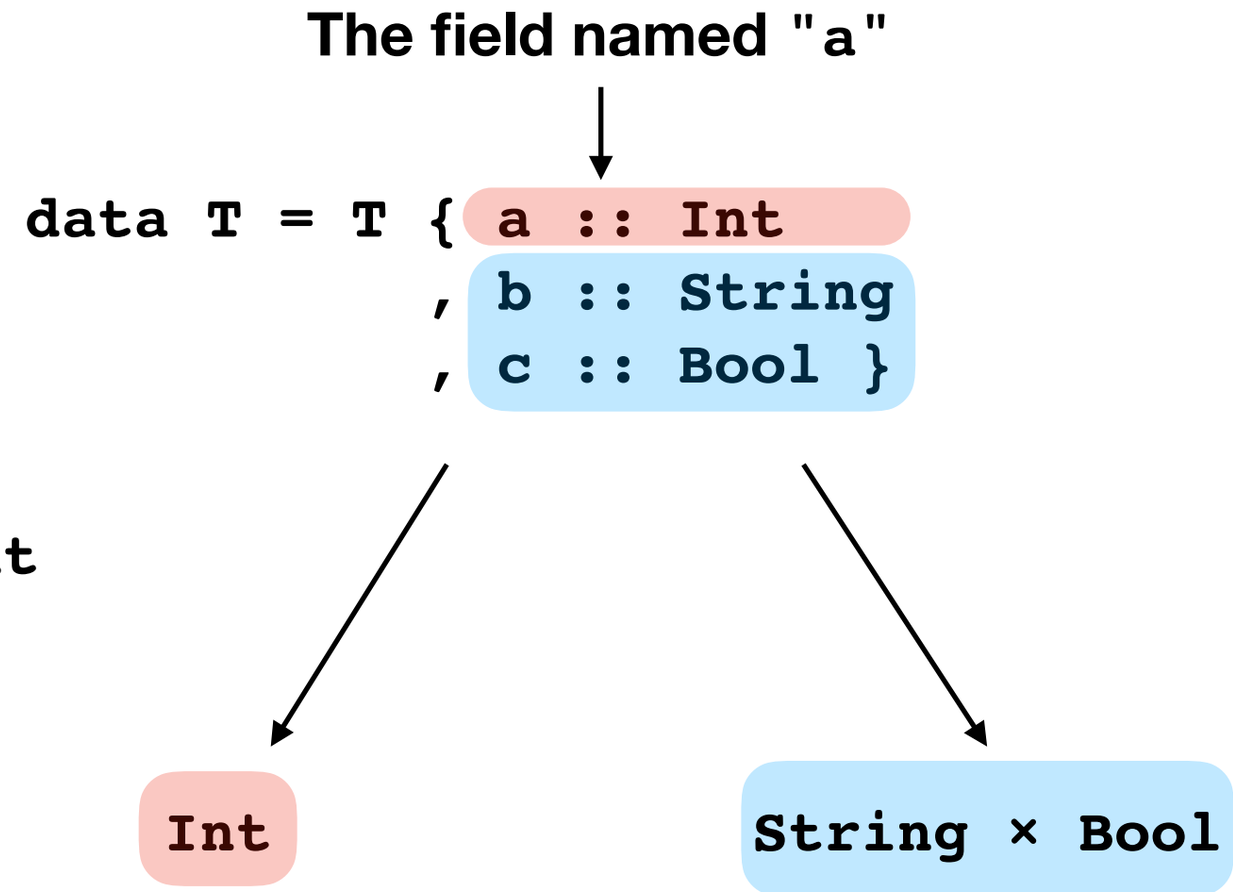
Meta information about the datatype

Meta information about the data constructor

Meta information about the record field

Implementation: `field`

Name-directed




```
field @“a” :: Lens T Int
```

Implementation: `field`

```
class HasField (field :: Symbol) a s | s field → a where
  field :: Lens s a
```

```
instance
  (Generic s
  ,ErrorUnless field s (HasTotalFieldP field (Rep s))
  ,GHasField field (Rep s) a
  ) ⇒ HasField field a s where

  field = … gfield …
```

Idea

- Type class instances for each of the generic constructors

- Specify each simple case in turn

Now of kind `* -> *` as this is the type of the generic constructors

```
class GHasField (field :: Symbol) (s :: * → *) a
        | s field → a where
  gfield :: Lens (s x) a
```

Boring Cases

Ignore the metadata for data types and constructors

```
instance GHasField field s a
         ⇒ GHasField field (M1 D meta s) a where
  gfield = mLens <∘> gfield @field


instance GHasField field s a
         ⇒ GHasField field (M1 C meta s) a where
  gfield = mLens <∘> gfield @field


mLens :: Lens (M1 i c f p) (f p)
```

Selector Metainfo

Select the field which has the correct field name

Requested field name

```
instance GHasField field
          (M1 S ('MetaSel ('Just field) p f b) (Rec0 a))
          a where
 gfield = mLens <.> kLens
```

Actual field name of the field

Tricky Case: Products

Problem: Pick the correct side of the product to recurse into

Tricky Case: Products

First Attempt

```
instance GHasField field f a
         ⇒ GHasField field (f :*: g) a where
instance GHasField field g a
         ⇒ GHasField field (f :*: g) a where
```

REJECTED: Due to duplicate instances

Tricky Case: Products

Solution: Say which branch of the product the field is in.

```
instance (GProductHasField field f g a (Contains field f ))
         => GHasField field (f :*: g) a where
  gfield = gproductField @field @_ @_ @_ @(Contains field f )
```

Tricky Case: Products

```
instance (GProductHasField field f g a (Contains field f )) ⇒
         GHasField field (f :*: g) a where
  gfield = gproductField @field @_ @_ @_ @(Contains field f )
```

```
instance GHasField field l a
         ⇒ GProductHasField field l r a 'True where
  gproductField = first <$> gfield @field

instance GHasField field r a
         ⇒ GProductHasField field l r a 'False where
  gproductField = second <$> gfield @field
```

More!

- generic-lens - a library which implements the paper
- Prisms
- Better Error Messages using TypeErrors
- Performance Evaluation
- Reflections on building an API around type applications