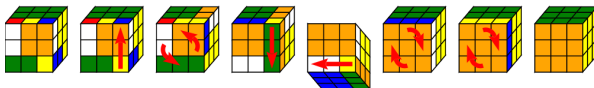


Lenses und Zauberwürfel



Tim Baumann

Curry Club Augsburg
13. August 2015



Wo kann ich mehr über `lens` erfahren?

- **Das Lens-Wiki:** <https://github.com/ekmett/lens/wiki>
- Blogserie “Lens over Tea” <http://artyom.me/lens-over-tea-1>
- Vortrag von Simon Peyton Jones bei Skills Matter
- Vortrag von Edward Kmett:
“The Unreasonable Effectiveness of Lenses for Business Applications”
- Blogpost: “Program imperatively using Haskell lenses”
- School of Haskell: “A Little Lens Starter Tutorial”
- Cheat Sheet für `Control.Lens`:
<https://github.com/anchor/haskell-cheat-sheets>





<http://timbaumann.info/lens>

<https://github.com/timjb/presentations/tree/gh-pages/lens>

Plated



Plated

```
data Inline
  = Str String
  | Emph [Inline]
  | Math MathType String
  | Link [Inline] Target
  | Image [Inline] Target
  ...
deriving (... , Typeable, Data, Generic)
```

```
data Block
  = Para [Inline]
  | BlockQuote [Block]
  | BulletList [[Block]]
  | Header Int Attr [Inline]
  ...
deriving (... , Typeable, Data, Generic)
```

```
data Pandoc = Pandoc Meta [Block]
deriving (... , Typeable, Data, Generic)
```



Anwendung: Ausnahmebehandlung

```
class (Typeable e, Show e) => Exception e where ...  
data SomeException = forall e. Exception e => SomeException e  
handle :: Exception e => (e -> IO a) -> IO a -> IO a
```

```
handle (\(exc :: AssertionFailed) -> return "caught")  
  (assert (2+2 == 3) (return "uncaught"))  
~> "caught"
```

```
handle (\(exc :: AssertionFailed) -> return "caught")  
  (assert (2+2 == 4) (return "works"))  
~> "works"
```

Es ist doof, dass man das Argument im Exception-Handler mit einem Typ annotieren muss. Doch zum Glück gibt es `Control.Exception.Lens!`



Anwendung: Ausnahmebehandlung

```
import Control.Exception.Lens
catching :: MonadCatch m => Prism' SomeException a
         -> m r -> (a -> m r) -> m r
```

```
catching _AssertionFailed (assert (2+2 == 3) (return "uncaught"))
                          (const (return "caught"))
```

≈> "caught"

```
catching _AssertionFailed (assert (2+2 == 4) (return "works"))
                          (const (return "caught"))
```

≈> "works"

In `Control.Exception.Lens` sind ganz viele `Prisms` vordefiniert:

```
_IndexOutOfBounds :: Prism' SomeException String
_StackOverflow     :: Prism' SomeException ()
_UserInterrupt     :: Prism' SomeException ()
_DivideByZero      :: Prism' SomeException ArithException
_AssertionFailed   :: Prism' SomeException String
-- (usw)
```

Anwendung: Defaultparameter

Angenommen, wir schreiben eine HTTP-Library (Beispiel geklaut von Oliver Charles)

```
data HTTPSettings = HTTPSettings
  { _httpKeepAlive :: Bool
  , _httpCookieJar :: CookieJar
  } deriving (Show, ...)
makeLenses ''HTTPSettings

defaultHTTPSettings :: HTTPSettings
defaultHTTPSettings = HTTPSettings True emptyCookieJar

httpRequest :: HTTPSettings -> HTTPRequest -> IO Response
instance Default HTTPSettings where
  def = defaultHTTPSettings

httpRequest
  (def & httpKeepAlive .~ True
   & httpCookieJar .~ myCookieJar)
  aRequest
```

Kritik: Default ist eine gesetzlose Typklasse!



Anwendung: Defaultparameter

Angenommen, wir schreiben eine HTTP-Library (Beispiel geklaut von Oliver Charles)

```
data HTTPSettings = HTTPSettings
  { _httpKeepAlive :: Bool
  , _httpCookieJar  :: CookieJar
  } deriving (Show, ...)
makeLenses ''HTTPSettings

defaultHTTPSettings :: HTTPSettings
defaultHTTPSettings = HTTPSettings True emptyCookieJar

httpRequest :: State HTTPSettings a -> HTTPRequest -> IO Response
httpRequest mkState req =
  let config = execState mkConfig defaultHttpSettings in ...

httpRequest
  (do httpKeepAlive .= True
      httpCookieJar  .= myCookieJar)
  aRequest
```

Besser!



Traversal1

Ein `Traversal1 s a` ist ein `Traversal s a`, dass immer mindestens über ein `a` iteriert. Dies lässt sich mit der Typklasse `Apply` umsetzen:

```
type Traversal1 s t a b =  $\forall$  f. Apply f => (a -> f b) -> s -> f t  
type Traversal1' s a = Traversal1 s s a a
```

```
class Functor f => Apply f where  
  (<.>) :: f (a -> b) -> f a -> f b
```

Zur Erinnerung:

```
type Traversal s t a b =  $\forall$  f. Applicative f => (a -> f b) -> s -> f t  
type Traversal' s a = Traversal s s a a
```

```
class Functor f => Applicative f where  
  (<*>) :: f (a -> b) -> f a -> f b  
  pure  :: a -> f a
```

