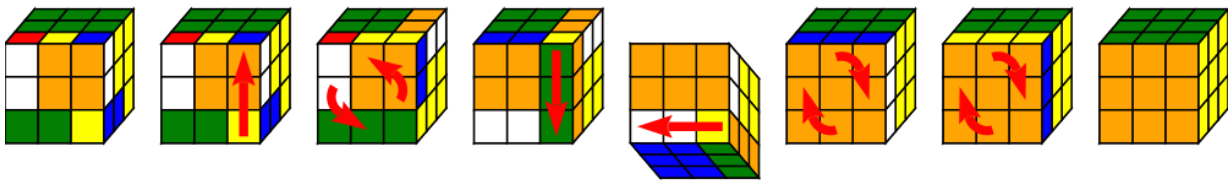
Lenses und Zauberwürfel



Tim Baumann

Curry Club Augsburg
13. August 2015

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine :: String
  }

data Person = Person
  { _firstName :: String
  , _lastName :: String
  , _address :: Address
  }
```

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine :: String
  }
```

```
data Person = Person
  { _firstName :: String
  , _lastName :: String
  , _address :: Address
  }
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }
```

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine :: String
  }

data Person = Person
  { _firstName :: String
  , _lastName :: String
  , _address :: Address
  }
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }

address :: Lens† Person Address
address = Lens†
  { getter = _address
  , setter = \a p ->
      p { _address = a }
  }
```

```
streetLine, townLine :: Lens† Address String
firstName,  lastName :: Lens† Person String
```

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine :: String
  }
```

```
data Person = Person
  { _firstName :: String
  , _lastName :: String
  , _address :: Address
  }
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }
```

Lens-Gesetze:

1. `a = getter l (setter l a s)`
2. `setter l a . setter l b = setter l a`
3. `s = setter l (getter l s) s`

```
streetLine, townLine :: Lens† Address String
firstName,  lastName :: Lens† Person String
address              :: Lens† Person Address
```

Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
```

```
personTownLine :: Lens† Person String
personTownLine = compose address townLine
```

Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
compose l m = Lens†
  { getter = getter m . getter l
  , setter = \b s -> setter l (setter m b (getter l s)) s
  }

personTownLine :: Lens† Person String
personTownLine = compose address townLine
```

Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
compose l m = Lens†
  { getter = getter m . getter l
  , setter = \b s -> setter l (setter m b (getter l s)) s
  }

personTownLine :: Lens† Person String
personTownLine = compose address townLine
```

Folgende Hilfsfunktion ist oft nützlich:

```
modify :: Lens† s a -> (a -> a) -> s -> s
modify l f s = setter l (f (getter l s)) s
```

Zum Beispiel, um die Stadt in der Adresse in Versalien zu schreiben:

```
person' = modify personTownLine (map toUpper) person
```

Alles wunderbar? Leider nein

Problem Nr. 1: Bei der Auswertung

```
  modify (compose l m) f s
= setter (compose l m) (f (getter (compose l m) s)) s
= setter l (setter m (f (getter m (getter l s))) (getter l s)) s
```

wird `getter l s` zweimal berechnet. Besser wäre

```
  modify (compose l m) f s
= let a = getter l s in setter l (setter m (f (getter m a)) a) s
```

Alles wunderbar? Leider nein

Problem Nr. 1: Bei der Auswertung

```
  modify (compose l m) f s
= setter (compose l m) (f (getter (compose l m) s)) s
= setter l (setter m (f (getter m (getter l s))) (getter l s)) s
```

wird `getter l s` zweimal berechnet. Besser wäre

```
  modify (compose l m) f s
= let a = getter l s in setter l (setter m (f (getter m a)) a) s
```

Problem Nr. 2: In `modify` wird die Datenstruktur zweimal durchlaufen: Einmal, um den gesuchten Wert zu extrahieren, dann nochmal, um den neuen Wert abzulegen.
Das kann kostspielig sein, z. B. bei der Lens rechts.

```
data NonEmpty a =
  Cons a (NonEmpty a) | Last a
last :: Lens' (NonEmpty a) a
last = Lens' getter setter
 where
  getter (Cons _ xs) = getter xs
  getter (Last x) = x
  setter a (Cons _ xs) = setter a xs
  setter a (Last _) = Last a
```

Alles wunderbar? Leider nein

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.

```
data Lens‡ s a = Lens‡
  { getter  :: s -> a
  , setter  :: a -> s -> s
  , modify  ::              (a ->   a) -> s ->   s
  }
```

Alles wunderbar? Leider nein

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.
Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens‡ s a = Lens‡
  { getter  :: s -> a
  , setter  :: a -> s -> s
  , modifyF :: Functor f => (a -> f a) -> s -> f s
  }
```

Alles wunderbar? Leider nein

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.
Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens‡ s a = Lens‡
  { getter  :: s -> a
  , setter  :: a -> s -> s
  , modifyF :: Functor f => (a -> f a) -> s -> f s
  }
```

Bahnbrechende Einsicht von Twaan van Laarhoven:

`modifyF` umfasst `getter` und `setter`!

Alles wunderbar? Leider nein

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.
Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens‡ s a = Lens‡
  { {-getter  :: s -> a
  , setter  :: a -> s -> s
  ,-} modifyF :: Functor f => (a -> f a) -> s -> f s
  }
```

Bahnbrechende Einsicht von Twaan van Laarhoven:

`modifyF` umfasst `getter` und `setter`!

`modifyF` umfasst `getter` und `setter`

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

`modifyF` umfasst 1. `getter` und `setter`

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1. `(.^) :: s -> Lens' s a -> a`

`modifyF` umfasst 1. `getter` und 2. `setter`

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1. `(.^) :: s -> Lens' s a -> a`

2. `(.~) :: Lens' s a -> a -> s -> s`

`modifyF` umfasst 1. `getter` und 2. `setter`

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1. `(.^) :: s -> Lens' s a -> a`

2. `(.~) :: Lens' s a -> a -> s -> s`

```
newtype Id a = Id { getId :: a }
instance Functor Id where
  fmap f (Id a) = Id (f a)
```

modifyF umfasst 1. getter und 2. setter

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1. `(.^) :: s -> Lens' s a -> a`

2. `(.~) :: Lens' s a -> a -> s -> s`

   ```
   (.~) l a s = getId (l (\_ -> Id s) s)

   newtype Id a = Id { getId :: a }
   instance Functor Id where
     fmap f (Id a) = Id (f a)
   ```

modifyF umfasst 1. getter und 2. setter

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1. `(.^) :: s -> Lens' s a -> a`

```
newtype Const a b = Const { getConst :: a }
instance Functor (Const a) where
  fmap _ (Const b) = Const b
```

2. `(.~) :: Lens' s a -> a -> s -> s`

```
(.~) l a s = getId (l (\_ -> Id s) s)
```

```
newtype Id a = Id { getId :: a }
instance Functor Id where
  fmap f (Id a) = Id (f a)
```

modifyF umfasst 1. getter und 2. setter

```
type Lens' s a = Functor f => (a -> f a) -> s -> f s
```

1.
```
(.^) :: s -> Lens' s a -> a
s .^ l = getConst (l Const s)

newtype Const a b = Const { getConst :: a }
instance Functor (Const a) where
  fmap _ (Const b) = Const b
```

2.
```
(.~) :: Lens' s a -> a -> s -> s
(.~) l a s = getId (l (\_ -> Id s) s)

newtype Id a = Id { getId :: a }
instance Functor Id where
  fmap f (Id a) = Id (f a)
```

Komponieren von `Lens'`es

Gegeben: `l :: Lens' s a`
und `m :: Lens' a b`

Gesucht: `? :: Lens' s b`

Komponieren von `Lens'`es

Gegeben:
```
l :: Functor f => (a -> f a) ->  s -> f s
```
und
```
m :: Functor f => (b -> f b) ->  a -> f a
```
Gesucht:
```
? :: Functor f => (b -> f b) ->  s -> f s
```

Komponieren von `Lens'`es

Gegeben:
```
l :: Functor f => (a -> f a) -> (s -> f s)
```
und
```
m :: Functor f => (b -> f b) -> (a -> f a)
```
Gesucht:
```
? :: Functor f => (b -> f b) -> (s -> f s)
```

Komponieren von `Lens'`es

Gegeben:
```
l :: Functor f => (a -> f a) -> (s -> f s)
```
und
```
m :: Functor f => (b -> f b) -> (a -> f a)
```
Gesucht:
```
l.m :: Functor f => (b -> f b) -> (s -> f s)
```

Komponieren von `Lens'`es

Gegeben:
```
l :: Functor f => (a -> f a) -> (s -> f s)
```
und
```
m :: Functor f => (b -> f b) -> (a -> f a)
```
Gesucht:
```
l.m :: Functor f => (b -> f b) -> (s -> f s)
```

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!

Komponieren von `Lens'`es

```
Gegeben:   l   :: Functor f => (a -> f a) -> (s -> f s)
     und   m   :: Functor f => (b -> f b) -> (a -> f a)
Gesucht:  l.m :: Functor f => (b -> f b) -> (s -> f s)
```

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
address f (Person first last addr) =
  fmap (Person first last) (f addr)
```

Komponieren von `Lens'`es

```
Gegeben:   l   :: Functor f => (a -> f a) -> (s -> f s)
     und   m   :: Functor f => (b -> f b) -> (a -> f a)
Gesucht:  l.m :: Functor f => (b -> f b) -> (s -> f s)
```

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
address f (Person first last addr) =
  fmap (Person first last) (f addr)

streetLine, townLine :: Lens' Address String
firstName,  lastName :: Lens' Person String
```

Komponieren von Lens'es

```
Gegeben:   l   :: Functor f => (a -> f a) -> (s -> f s)
     und   m   :: Functor f => (b -> f b) -> (a -> f a)
Gesucht:  l.m :: Functor f => (b -> f b) -> (s -> f s)
```

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
address f (Person first last addr) =
  fmap (Person first last) (f addr)

streetLine, townLine :: Lens' Address String
firstName,  lastName :: Lens' Person String
```

Dann haben wir `address.townLine :: Lens' Person String`

API

```
(.^) :: s -> Lens s a -> a
```

Welche Bibliothek?

Abbildung: Picking a Lens Library (Cartesian Closed Comic)