Lenses and optics

Advanced functional programming - Lecture 11

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

STANDARDIZED LADDER OF FUNCTIONAL PROGRAMMING

The LambdaConf Ladder of Functional Programming (LOFP) is a standardized progression of different concepts and skills that developers must master on their journey to becoming expert-level functional programmers. LOFP can be used to rank workshops, talks, presentations, books, and courseware, so that aspiring functional programmers have a better understanding of what material is appropriate for them given their current experience.

NOVICE

CONCEPTS
- Immutable Data
- Second-Order Functions
- Constructing & Destructuring
- Function Composition
- First-Class Functions & Lambdas

SKILLS
- Use second-order functions (map, filter, fold) on immutable data structures
- Destructure values to access their components
- Use data types to represent optionality
- Read basic type signatures
- Pass lambdas to second-order functions

ADVANCED BEGINNER

CONCEPTS
- Algebraic Data Types
- Pattern Matching
- Parametric Polymorphism
- General Recursion
- Type Classes, Instances, & Laws
- Lower-Order Abstractions (Equal, Semigroup, Monoid, etc)
- Referential Transparency & Totality
- Higher-Order Functions
- Partial-Application, Currying, & Point-Free Style

SKILLS
- Solve problems without nulls, exceptions, or type casts
- Process & transform recursive data structures using recursion
- Able to use functional programming "in the small"
- Write basic monadic code for a concrete monad
- Create type class instances for custom data types
- Model a business domain with ADTs
- Write functions that take and return functions
- Reliably identify & isolate pure code from impure code
- Avoid introducing unnecessary lambdas & named parameters

COMPETENT

CONCEPTS
- Generalized Algebraic Data Types
- Higher-Kinded Types
- Rank-N Types
- Folds & Unfolds
- Higher-Order Abstractions (Category, Functor, Monad)
- Basic Optics
- Efficient Persistent Data Structures
- Existential Types
- Embedded DSLs using Combinators

SKILLS
- Able to use functional programming "in the large"
- Test code using generators and properties
- Write imperative code in a purely functional way through monads
- Use popular purely functional libraries to solve business problems
- Separate decision from effects
- Write a simple custom lawful monad
- Write production medium-sized projects
- Use lenses & prisms to manipulate data
- Simplify types by hiding irrelevant data with existentials

PROFICIENT

CONCEPTS
- Codata
- (Co)Recursion Schemes
- Advanced Optics
- Dual Abstractions (Comonad)
- Monad Transformers
- Free Monads & Extensible Effects
- Functional Architecture
- Advanced Functors (Exponential, Profunctors, Contravariant)
- Embedded DSLs using GADTs, Finally Tagless
- Advanced Monads (Continuation, Logic)
- Type Families, Functional Dependencies

SKILLS
- Design a minimally-powerful monad transformer stack
- Write concurrent and streaming programs
- Use purely functional mocking in tests
- Use type classes to modularly model different effects
- Recognize type patterns & abstract over them
- Use functional libraries in novel ways
- Use optics to manipulate state
- Write custom lawful monad transformers
- Use free monads / extensible effects to separate concerns
- Encode invariants at the type level
- Effectively use FDs / type families to create safer code

EXPERT

CONCEPTS
- High-Performance
- Kind Polymorphism
- Generic Programming
- Type-Level Programming
- Dependent-Types, Singleton Types
- Category Theory
- Graph Reduction
- Higher-Order Abstract Syntax
- Compiler Design for Functional Languages
- Profunctor Optics

SKILLS
- Design a generic, lawful library with broad appeal
- Prove properties manually using equational reasoning
- Design & implement a new functional programming language
- Create novel abstractions with laws
- Write distributed systems with certain guarantees
- Use proof systems to formally prove properties of code
- Create libraries that do not permit invalid states
- Use dependent-typing to prove more properties at compile-time
- Understand deep relationships between different concepts
- Profile, debug, & optimize purely functional code with minimal sacrifices

FANTASYLAND


how many levels of functional programming are you on
like,, maybe expert beginner or competent. my dude
this was supposed to be legs
you are like a little baby
watch this
$A \xleftarrow{f} B$
g
$g \circ f$
C
PROFUNCTOR
OPTICS

Motivation: managing nested records

Records in Haskell provide a convenient way to organize structured data.

```haskell
data Person = {name :: String, address :: Address}
data Address = {street :: String, city :: String}
```

In practice, these records can be *huge*.

Nested records

We can use the record fields to project out the desired information.

If we need to access nested fields, we can define our own projection functions:

```
personCity :: Person -> City
personCity = city . address
```

Record projections compose nicely.

What about record updates?

Nested records

Setting nested fields is pretty painful:

```
setCity :: City -> Address -> Address
setCity newCity a = a {city = newCity}

setAddress :: Address -> Person -> Person
setAddress newAddress p = p {address = newAddress}

setPersonCity :: City -> Person -> Person
setPersonCity newCity p =
  setAddress (setCity newCity (address p)) p
```

This is already quite some ‘boilerplate’ code – code that is not interesting and follows a fixed pattern.

Intro: simple lenses

To automate this, we can package the getter and setter functions in a single data type, sometimes reffered to as a *lens*:

```
data (:->) a b = Lens
  { get : a -> b
  , put : b -> a -> a
  }
```

Such lenses compose nicely:

```
compose :: (b :-> c) -> (a :-> b) -> (a :-> c)
```

Example: lenses

In our example, suppose we are given lenses for every record field:

```
city :: (Address :-> String)
address :: (Person :-> Address)
```

We can compose these lenses by hand, to assemble the pieces of data that we're interested in:

```
personCity :: Person :-> City
personCity = compose city address

updateCity :: City -> Person -> Person
updateCity newCity = put personCity newCity
```

Example: lenses

Lenses make the manipulation of nested records manageable.

But who writes the lenses?

This is not hard to do by hand:

```
city :: Address :-> City
city = Lens { get = \a -> city a
            , put = \nc a -> a {city = nc}
            }
```

But could clearly use some automation – this can be done by using Template Haskell.

Lenses: more generally

The view-update problem: given a (compound) data type `s`, define functions:

- `view : s -> a` – that extract some value of interest from the data type;
- `update : (s,a) -> s` – that overwrites the value of interest

This pattern is quite common once you have (nested) records, data types, database tables/rows, or any non-trivial data model.

```
data Lens a s =
  Lens {view :: s -> a, update :: (a,s) -> s)
```

Generalization

We can generalize this slightly:

- allowing the source `s` and target `t` to be different;
- allowing the `view` and `update` functions to manipulate values of different types.

```
data Lens a b s t =
  Lens { view :: s -> a
       , update :: (b,s) -> t
       }
```

Example

For example, we can have a `fst` lens that allows you to enrich values with an additional ‘context’ `c`:

```
fstLens :: Lens a b (a,c) (b,c)
fstLens = Lens v u
  where
  v :: (a,c) -> a
  v = fst
  update :: (b,(a,c)) -> (b,c)
  update (x, (_,y)) = (x,y)
```

Products are to lenses as Coproducts are to...

A lens lets you project out or update one particular part of a product:

```
data Lens a b s t   = Lens
  { view   :: s -> a
  , update :: (b,s) -> t}
```

But what should we do if we have more than one constructor?

Products are to lenses as Coproducts are to...

A lens lets you project out or update one particular part of a product:

```
data Lens a b s t   = Lens
  { view   :: s -> a
  , update :: (b,s) -> t}
```

But what should we do if we have more than one constructor?

By reversing all the arrows, replacing products with coproducts, we arrive at the following definition of a *prism*:

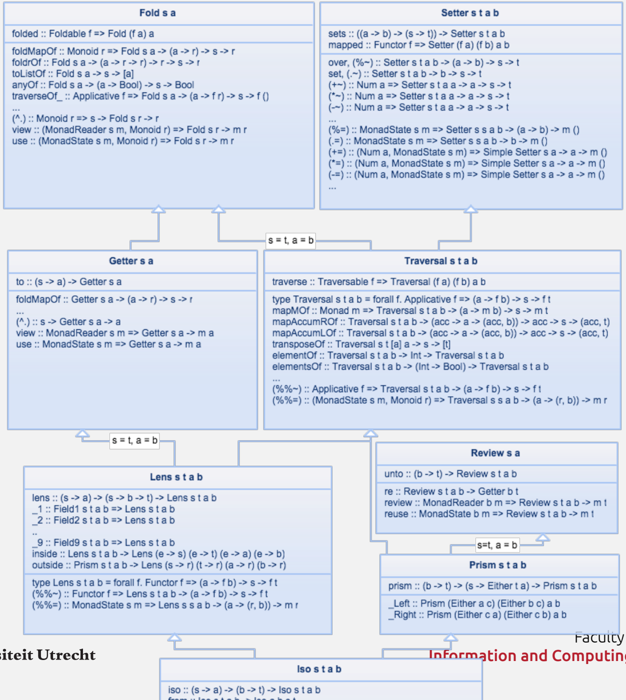
```
data Prism a b s t  = Prism
  { build :: b -> t
  , match :: s -> Either a t}
```

Why prisms?

Just as lenses let us focus on parts of a record…

… prisms let us focus on a particular constructor of a data type.

```
the :: Prism a b (Maybe a) (Maybe b)
the = Prism match build
  where
  match : Maybe a -> Either a (Maybe b)
  build :: b -> Maybe b
```


Fold s a
folded :: Foldable f => Fold (f a) a
foldMapOf :: Monoid r => Fold s a -> (a -> r) -> s -> r
foldrOf :: Fold s a -> (a -> r -> r) -> r -> s -> r
toListOf :: Fold s a -> s -> [a]
anyOf :: Fold s a -> (a -> Bool) -> s -> Bool
traverseOf_ :: Applicative f => Fold s a -> (a -> f r) -> s -> f ()
...
(^.) :: Monoid r => s -> Fold s r -> r
view :: (MonadReader s m, Monoid r) => Fold s r -> m r
use :: (MonadState s m, Monoid r) => Fold s r -> m r
Setter s t a b
sets :: ((a -> b) -> (s -> t)) -> Setter s t a b
mapped :: Functor f => Setter (f a) (f b) a b
over, (%~) :: Setter s t a b -> (a -> b) -> s -> t
set, (.~) :: Setter s t a b -> b -> s -> t
(+~) :: Num a => Setter s t a a -> a -> s -> t
(*~) :: Num a => Setter s t a a -> a -> s -> t
(-~) :: Num a => Setter s t a a -> a -> s -> t
...
(%=) :: MonadState s m => Setter s s a b -> (a -> b) -> m ()
(.=) :: MonadState s m => Setter s s a b -> b -> m ()
(+=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
(*=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
(-=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
...
s = t, a = b
Getter s a
to :: (s -> a) -> Getter s a
foldMapOf :: Getter s a -> (a -> r) -> s -> r
...
(^.) :: s -> Getter s a -> a
view :: MonadReader s m => Getter s a -> m a
use :: MonadState s m => Getter s a -> m a
Traversal s t a b
traverse :: Traversable f => Traversal (f a) (f b) a b
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
mapMOf :: Monad m => Traversal s t a b -> (a -> m b) -> s -> m t
mapAccumROf :: Traversal s t a b -> (acc -> a -> (acc, b)) -> acc -> s -> (acc, t)
mapAccumLOf :: Traversal s t a b -> (acc -> a -> (acc, b)) -> acc -> s -> (acc, t)
transposeOf :: Traversal s t [a] a -> s -> [t]
elementOf :: Traversal s t a b -> Int -> Traversal s t a b
elementsOf :: Traversal s t a b -> (Int -> Bool) -> Traversal s t a b
...
(%%~) :: Applicative f => Traversal s t a b -> (a -> f b) -> s -> f t
(%%=) :: (MonadState s m, Monoid r) => Traversal s s a b -> (a -> (r, b)) -> m r
s = t, a = b
Review s a
unto :: (b -> t) -> Review s t a b
re :: Review s t a b -> Getter b t
review :: MonadReader b m => Review s t a b -> m t
reuse :: MonadState b m => Review s t a b -> m t
Lens s t a b
lens :: (s -> a) -> (s -> b -> t) -> Lens s t a b
_1 :: Field1 s t a b => Lens s t a b
_2 :: Field2 s t a b => Lens s t a b
..
_9 :: Field9 s t a b => Lens s t a b
inside :: Lens s t a b -> Lens (e -> s) (e -> t) (e -> a) (e -> b)
outside :: Prism s t a b -> Lens (s -> r) (t -> r) (a -> r) (b -> r)
type Lens s t a b = forall f. Functor f => (a -> f b) -> s -> f t
(%%~) :: Functor f => Lens s t a b -> (a -> f b) -> s -> f t
(%%=) :: MonadState s m => Lens s s a b -> (a -> (r, b)) -> m r
s=t, a = b
Prism s t a b
prism :: (b -> t) -> (s -> Either t a) -> Prism s t a b
_Left :: Prism (Either a c) (Either b c) a b
_Right :: Prism (Either c a) (Either c b) a b
Iso s t a b
iso :: (s -> a) -> (b -> t) -> Iso s t a b

Profunctor optics

Rather than give a complete tour of the library, I want to highlight a few of the key ideas.

And give a general presentation of *profunctor optics*.

Profunctors

```haskell
class Functor f where
  fmap :: (a -> a') -> f a -> f b

class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') ->
           p a b -> p a' b'
```

A *profunctor* generalizes the familiar concept of functors.

A profunctor takes two type arguments and requires two maps: one contravariant and one covariant.

Function space profunctor

The ‘canonical’ choice for profunctor is the function space type constructor:

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') ->
           p a b -> p a' b'

instance Profunctor (->) where
  dimap f g h = g . h . f
```

We could also attach more information, such as a boolean flag to our types; or restrict ourselves to functions that consume or produce pairs.

Other profunctors

```haskell
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') ->
           p a b -> p a' b'

data Upstar f a b =
  UpStar {unstar :: a -> f b}

instance Functor f => Profunctor (Upstar f) where
  dimap f g u = ...
```

Other profunctors

```haskell
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') ->
           p a b -> p a' b'

data Upstar f a b =
  UpStar {unstar :: a -> f b}

instance Functor f => Profunctor (Upstar f) where
  dimap f g (UpStar h) = UpStar (fmap g . h . f)
```

Cartesian profunctors

There are many further specific properties of profunctors we can identify, such as the *cartesian* profunctors:

```
class Profunctor p => Cartesian p where
  first :: p a b -> p (a,c) (b,c)
  second :: p a b -> p (c,a) (c,b)
```

Why?

Why do all this work studying profunctors?

Why?

Why do all this work studying profunctors?

It turns out, that these profunctors give a uniform description of lenses, prisms, adapters, traversals, and many other lens constructs.

```
type Optic p a b s t = p a b -> p s t
```

Where p is typically a profunctor.

Example: profunctor lenses

Lenses are another example of profunctors:

```
data Lens a b s t =
  Lens {view :: s -> a, update :: (b,s) -> t)

instance Profunctor (Lens a b) where
  dimap f g (Lens v u) = ...
```

Similarly, lenses are also cartesian.

Example: profunctor lenses

Lenses are another example of profunctors:

```
data Lens a b s t =
  Lens {view :: s -> a, update :: (b,s) -> t}

instance Profunctor (Lens a b) where
  dimap f g (Lens v u) =
    Lens (v . f) (g . u . cross id f)
```

Similarly, lenses are also cartesian.

Example: profunctor lenses

In fact, we can convert freely between lenses and the following profunctor representation:

```
type LensP a b s t =
  forall p . Cartesian p => Optic p a b s t

lensC2P :  Lens a b s t  -> LensP a b s t
lensP2C :  LensP a b s t -> Lens a b s t
```

Example: profunctor prisms

```
data Prism a b s t  = Prism
  { build :: b -> t
  , match :: s -> Either a t}

instance Profunctor (Prism a b) where
  dimap f g (Prism m b) = Prism (plus g id . m . f)
```

We can show that prisms are cocartesian profunctors.

And similarly, we can give an abstract representation of prisms as cocartesian profunctors:

```
type PrismP a b s t =
  forall p . Cocartesian p => Optic p a b s t
```

Why?

Why go all through this effort?

Why?

Why go all through this effort?

The 'profunctor view' of lenses and optics give a single framework in which to study a series of related concepts.

This provides a compositional manner of describing and composing lenses, prisms, traversals, adapters, and other optics.

Van Laarhoven lenses

Many lens libraries use the Van Laarhoven representation:

```
type LensF a b =
  forall f. Functor f => (b -> f b) -> (a -> f a)
```

By instantiating the functor argument `f` differently, you can 'project out' different bits of information.

Van Laarhoven lenses

This representation lets us project out information:

```
type RefF a b =
  forall f. Functor f => (b -> f b) -> (a -> f a)

newtype Const a b = Const {getConst :: a}

get :: RefF a b -> a -> b
get r = getConst . r Const
```

The key trick is the choice of functor…

We can also choose a different functor that pairs up things, the identity functor to define modification operations, etc.

What I haven't talked about

- Isos – for converting between different type representations;
- Optics and Traversals (in the applicative sense);
- Laws and properties of lenses;
- …

Further reading

- Profunctor Optics by Matthew Pickering et al.
- Control.Lens library documentation and tutorials
- Van Laarhoven lenses: `https://www.twanvl.nl/blog/haskell/cps-functional-references`