

Getting with Getters

Any function $(s \rightarrow a)$ can be flipped into continuation passing style, $(a \rightarrow r) \rightarrow s \rightarrow r$ and decorated with **Const** to obtain:

```
type Getting r s a =  
  (a -> Const r a) -> s -> Const r s
```

A **Getter** describes how to retrieve a single value in a way that can be composed with other **LensLike** constructions.

When you see this in a type signature it indicates that you can pass the function a **Lens**, **Getter**, **Traversal**, **Fold**, **Prism**, **Iso**, or one of the indexed variants, and it will just “do the right thing”.

Safe head

Perform a safe head of a **Fold** or **Traversal** or retrieve **Just** the result from a **Getter** or **Lens**.

$(^?) \equiv \text{flip preview}$

```
(^?) :: s -> Getting (First a) s a -> Maybe a
```

```
>>> Right 4 ^? _Left  
Nothing  
>>> "world" ^? ix 3  
Just 'l'
```

Viewing lenses

View the value pointed to by a **Getter** or **Lens** or the result of folding over all the results of a **Fold** or **Traversal** that points at a monoidal values.

This is the same operation as **view** with the arguments flipped.

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

```
>>> (0, -5) ^. _2.to abs  
5  
>>> ["a", "b", "c"] ^. traversed  
"abc"
```

Using MonadState

Use the target of a **Lens**, **Iso**, or **Getter** in the current state, or use a summary of a **Fold** or **Traversal** that points to a monoidal value.

```
use :: MonadState s m => Getting a s a -> m a  
  
>>> evalState (use _1) (1,2)  
1  
>>> evalState (uses _1 length) ("hello", "")  
5
```

Folding Foldables

```
type Fold s a =  
  forall m. Monoid m => Getting m s a
```

A **Fold s a** is a generalization of something **Foldable**. It allows you to extract multiple results from a container. Every **Getter** is a valid **Fold** that simply doesn't use the **Monoid** it is passed.

If there exists a **foo** method that expects a **Foldable** (**f a**), then there should be a **fooOf** method that takes a **Fold s a** and a value of type **s**.

Extracting lists from Folds

Extract a list of the targets of a **Fold**, an infix version of **toListOf**.

$\text{toList } xs \equiv xs \hat{..} \text{folded}$

```
(^..) :: s -> Getting (Endo [a]) s a -> [a]
```

```
>>> [[1,2],[3]] ^..traverse.traverse  
[1,2,3]  
>>> (1,2) ^..both  
[1,2]
```

Checking for matches

Check to see if this **Fold** or **Traversal** matches 1 or more entries. For the opposite, use **hasn't**.

```
has :: Getting Any s a -> s -> Bool
```

```
>>> has (element 0) []  
False  
>>> has _Right (Left 12)  
False  
>>> hasn't _Right (Left 12)  
True
```

Indexed Getters

For most operations, there is an indexed variant which will work as expected if the underlying target supports a notion of **Indexing**.

```
>>> ["ab", "c"] ^@..itraversed<.>itraversed  
[[((0,0), 'a'), ((0,1), 'b'), ((1,0), 'c')]  
>>> "hello" ^@..itraversed.indices even  
[(0, 'h'), (2, 'l'), (4, 'o')]  
  
>>> ifind (\i k -> i > k) [1,2,2,2]  
Just (3,2)
```

Modifying records with Setters

A **Setter** `s t a b` is a generalization of `fmap` from **Functor**. It allows you to map into a structure and change out the contents, but it isn't strong enough to allow you to enumerate those contents. Starting with `fmap :: Functor f => (a -> b) -> fa -> fb` we monomorphize the type to obtain `(a -> b) -> s -> t` and then decorate it with Identity to obtain:

```
type Setter s t a b =
  (a -> Identity b) -> s -> Identity t
```

Every Traversal is a valid Setter, since Identity is Applicative.

Modifying with a function

```
(%~) :: Profunctor p
      => Setting p s t a b -> p a b -> s -> t
```

```
>>> traverse %~ even $ [1,2,3]
[False,True,False]
```

Modifies the target of a **Lens** or all of the targets of a **Setter** or **Traversal** with a user supplied function.

This is an infix version of over.

Modifying with a constant value

```
(.~) :: ASetter s t a b -> b -> s -> t
```

```
>>> [1,2,3] & element 0 .~ 3
[3,2,3]
>>> 0 & bitAt 8 .~ True
256
>>> [1,2,3] & traversed . filtered odd .~ 0
[0,2,0]
```

Replace the target of a **Lens** or all of the targets of a **Setter** or **Traversal** with a constant value.

Prisms and Isos

Prisms can be thought of as partial isomorphisms. You can turn a **Prism** around with `re`, and a **Iso** around with `from` (which is involutive).

```
prism :: (b -> t)
      -> (s -> Either t a)
      -> Prism s t a b
prism' :: (a -> s)
       -> (s -> Maybe a)
       -> Prism' s a
```

```
iso :: (s -> a) -> (b -> t) -> Iso s t a b
from :: AnIso s t a b -> Iso b a t s
```

```
>>> 5^.re _Left ^?! _Left
5
>>> _Left # 1
Left 1
>>> has _Left (Right 1)
False
```

```
>>> 'a' ^. from enum
97
>>> 97 ^. enum :: Char
'a'
```

```
>>> Map.fromList [("hello",1)] ^. at "hello"
>>>                                     . non 0
1
>>> Map.empty & at "hi"
>>>                                     . non Map.empty
>>>                                     . at "world" ?~ "!"
fromList [("hi",fromList [("world","!")])]
```

Some setting operators

Operator	W/result	W/state	W/result	Action
+~	<+~	+=	<+=	Add to target(s)
-~	<-~	-=	<-=	Subtract from target(s)
~	<~	*=	<*=	Multiply target(s)
//~	<//~	//=	<//=	Divide target(s)
^^	<^^	^=	<^=	Raise target(s) to a non-negative Integral power
^^^	<^^^	^^=	<^^=	Raise target(s) to an Integral power
~	<~	**=	<**=	Raise target(s) to an arbitrary power
~	< ~	=	< =	Logically or target(s)
&&~	<&&~	&&=	<&&=	Logically and target(s)
<>~	<<>~	<>=	<<>=	mappend to the target monoidal value(s)