Chapter 14

```
import Data.Foldable (Foldable (..))
```

Exercise 1: Complete the following instance declaration from `Data.Monoid` to make a pair type into a monoid provided the two component types are monoids:

```
data Pair a b = Pair a b

instance (Semigroup a, Semigroup b) => Semigroup (Pair a b) where
    (<>) :: Pair a b -> Pair a b -> Pair a b
    Pair x1 y1 <> Pair x2 y2 = Pair (x1 <> x2) (y1 <> y2)

instance (Monoid a, Monoid b) => Monoid (Pair a b) where
    mempty :: Pair a b
    mempty = Pair mempty mempty
```

Exercise 2: In a similar manner, show how a function `type a -> b` can be made into a monoid provided that the result type `b` is a monoid.

```
newtype Hom a b = Hom (a -> b)

instance (Semigroup b) => Semigroup (Hom a b) where
    (<>) :: Hom a b -> Hom a b -> Hom a b
    Hom f <> Hom g = Hom $ \x -> f x <> g x

instance (Monoid b) => Monoid (Hom a b) where
    mempty :: Hom a b
    mempty = Hom $ const mempty
```

Exercise 3: Show how the `Maybe` type can be made foldable and traversable, by giving explicit definitions for `fold`, `foldMap`, `foldr`, `foldl` and `traverse`.
First, wrap the type since these definitions are already provided in the standard library:

```
newtype M a = M (Maybe a)
```

Also define a Functor for this new type otherwise we cannot define a Traversable instance:

```
instance Functor M where
    fmap :: (a -> b) -> M a -> M b
    fmap f (M (Just x)) = M $ Just $ f x
    fmap _ (M Nothing) = M Nothing
```

Finally define the instances for Foldable and Traversable:

```haskell
instance Foldable M where
    fold :: (Monoid m) => M m -> m
    fold (M (Just x)) = x
    fold (M Nothing) = mempty

    foldMap :: (Monoid m) => (a -> m) -> M a -> m
    foldMap f (M (Just x)) = f x
    foldMap _ (M Nothing) = mempty

    foldr :: (a -> b -> b) -> b -> M a -> b
    foldr f y (M (Just x)) = f x y
    foldr _ x (M Nothing) = x

    foldl :: (b -> a -> b) -> b -> M a -> b
    foldl f x (M (Just y)) = f x y
    foldl _ x (M Nothing) = x

instance Traversable M where
    traverse :: (Applicative f) => (a -> f b) -> M a -> f (M b)
    traverse f (M (Just x)) = M . Just <$> f x
    traverse _ (M Nothing) = pure $ M Nothing
```