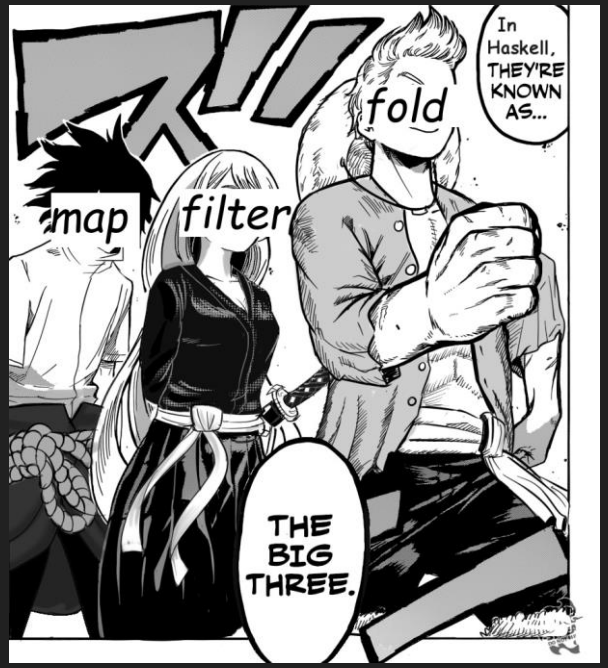
# FIT2102 Programming Paradigms Tutorial 9

## Foldable and Traversable




Faculty of Information Technology

MONASH University

# Haskell Type Holes

We can use ‘_’ as type holes! Which means haskell will tell us the type of what goes there.

```
add1 :: Num a => [a] -> [a]

add1 = map _
```

```
error:
    • Found hole: _ :: a -> a
      Where: ‘a’ is a rigid type variable bound by
               the type signature for:
                 add1 :: forall a. Num a => [a] -> [a]
               at test.hs:2:1-27
    • In the first argument of ‘map’, namely ‘_’
```

# Monoid

In Haskell, the Monoid typeclass is a class for types which have a single most *natural operation for combining values*, together with a *value which doesn't do anything* when you combine it with others

# Monoid

```
ghci>:i Monoid
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a    -- has alias (<>)
  mconcat :: [a] -> a
  {-# MINIMAL mempty, mappend #-}
       -- Defined in `GHC.Base'

-- defining mconcat is optional, since it has the following default:
mconcat = foldr mappend mempty
```

# Examples

*value which doesn't do anything when combined:* 0

*natural operation for combining values:* +

```
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)
```

# Foldable

A foldable is a structure which can be reduced to a single value. Think of it as a structure on which we can use foldr. To define an instance of foldable, we need to define the following function:

```
foldMap :: (Monoid m) => (a -> m) -> t a -> m
```

# Why is this useful?

Defining an instance of foldable allows us to derive a number of very useful functions for free. For example, a generalised fold, both left- and right-fold, a list converter, a length function, element existence, etc.

```
class Foldable t where
 fold :: Monoid m => t m -> m
 foldMap :: Monoid m => (a -> m) -> t a -> m
 foldr :: (a -> b -> b) -> b -> t a -> b
 foldr' :: (a -> b -> b) -> b -> t a -> b
 foldl :: (b -> a -> b) -> b -> t a -> b
 foldl' :: (b -> a -> b) -> b -> t a -> b
 foldr1 :: (a -> a -> a) -> t a -> a
 foldl1 :: (a -> a -> a) -> t a -> a
 toList :: t a -> [a]
 null :: t a -> Bool
 length :: t a -> Int
 elem :: Eq a => a -> t a -> Bool
 maximum :: forall a. Ord a => t a -> a
 minimum :: forall a. Ord a => t a -> a
 sum :: Num a => t a -> a
 product :: Num a => t a -> a
```

# Traversable

A Traversable type is a kind of upgraded Foldable. Where Foldable gives you the ability to go through the structure processing the elements (foldr) but throwing away the shape, Traversable allows you to do that whilst preserving the shape and, e.g., putting new values in.

A traversable has to be a foldable and a functor. They are defined using traverse.

```
traverse :: (Traversable t, Applicative f) => (a-> f b) -> t a -> f (t b)
```