COMP4075: Lecture 7

Functional Programming Patterns: Functor, Foldable, and Friends

Henrik Nilsson

University of Nottingham, UK

Type Classes and Patterns

- In Haskell, many functional programming patterns are captured through specific type classes.

Type Classes and Patterns

- In Haskell, many functional programming patterns are captured through specific type classes.
- Additionally, the type class mechanism itself and the fact that overloading is prevalent in Haskell give raise to other programming patterns.

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

- ***Semigroup***: a set (type) S with an ***associative*** binary operation $\cdot : S \times S \to S$:

$$\forall a, b, c \in S \;:\; (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

- ***Semigroup***: a set (type) S with an ***associative*** binary operation $\cdot : S \times S \to S$:

$$\forall a, b, c \in S \ : \ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

- ***Monoid***: a semigroup with an ***identity element***:

$$\exists e \in S, \forall a \in S \ : \ e \cdot a = a \cdot e = a$$

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.
- Being explicit about when such structures are used
  - makes code clearer
  - offer opportunities for reuse

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.
- Being explicit about when such structures are used
  - makes code clearer
  - offer opportunities for reuse
- The standard Haskell libraries provide type classes to capture these notions.

Class $Semigroup$

Class definition (most important methods):

$$\begin{aligned}&\mathbf{class}\ Semigroup\ a\ \mathbf{where}\\&\quad(\diamond)\qquad\quad :: a \to a \to a\\&\quad sconcat :: NonEmpty\ a \to a\end{aligned}$$

Minimum complete definition: $(\diamond)$ (ASCII: $<>$)
(There is thus a default definition for $sconcat$.)

$NonEmpty$ is the non-empty list type:

$$\mathbf{data}\ NonEmpty\ a = a :|\ [a]$$

Instances of $\mathit{Semigroup}$ (1)

A list $[a]$ is a semigroup (for any type a):

$$\mathbf{instance}\ \mathit{Semigroup}\ [a]\ \mathbf{where}$$
$$\quad (\diamond) = (+\!\!+)$$

Instances of *Semigroup* (1)

A list $[a]$ is a semigroup (for any type a):

$$\textbf{instance } \mathit{Semigroup}\ [a]\ \textbf{where}$$
$$(\diamond) = (+\!\!+)$$

$\mathit{Maybe}\ a$ is a semigroup if a is one:

$$\begin{aligned}&\textbf{instance } \mathit{Semigroup}\ a\\ &\qquad \Rightarrow \mathit{Semigroup}\ (\mathit{Maybe}\ a)\ \textbf{where}\end{aligned}$$
$$\begin{aligned}\mathit{Nothing} &\diamond y &&= y\\ x &\diamond \mathit{Nothing} &&= x\\ \mathit{Just}\ x &\diamond \mathit{Just}\ y &&= x \diamond y\end{aligned}$$

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

But which one to pick? Both are equally useful!

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

But which one to pick? Both are equally useful!

Idea:

- $Sum\ a$: the semigroup (a, $(+)$)
- $Product\ a$: the semigroup (a, $(*)$)

Instances of *Semigroup* (3)

Semigroup instances for $Sum\ a$ and $Product\ a$:

$$\begin{array}{l}\textbf{instance}\ Num\ a \Rightarrow Semigroup\ (Sum\ a)\ \textbf{where} \\ \quad (\diamond) = (+) \\ \textbf{instance}\ Num\ a \Rightarrow Semigroup\ (Product\ a)\ \textbf{where} \\ \quad (\diamond) = (*)\end{array}$$

Instances of *Semigroup* (4)

Similarly, any type with a total ordering forms a semigroup with maximum or minimum as the associative operation:

- $Max\ a$: the semigroup (a, max)
- $Min\ a$: the semigroup (a, min)

Semigroup instances:

$$\textbf{instance}\ Ord\ a \Rightarrow Semigroup\ (Max\ a)\ \textbf{where}$$
$$(\diamond) = max$$
$$\textbf{instance}\ Ord\ a \Rightarrow Semigroup\ (Min\ a)\ \textbf{where}$$
$$(\diamond) = min$$

Instances of *Semigroup* (5)

All products of semigroups are semigroups; e.g.:

$$\begin{aligned}&\textbf{instance}\ (Semigroup\ a, Semigroup\ b)\\ &\qquad\qquad \Rightarrow Semigroup\ (a, b)\ \textbf{where}\\ &\quad (x, y) \diamond (x', y') = (x \diamond x', y \diamond y')\end{aligned}$$

Instances of *Semigroup* (5)

All products of semigroups are semigroups; e.g.:

$$\begin{aligned}&\textbf{instance}\ (\mathit{Semigroup}\ a, \mathit{Semigroup}\ b)\\ &\qquad\qquad \Rightarrow \mathit{Semigroup}\ (a, b)\ \textbf{where}\\ &\quad (x, y) \diamond (x', y') = (x \diamond x', y \diamond y')\end{aligned}$$

$a \to b$ is a semigroup if the range b is a semigroup:

$$\begin{aligned}&\textbf{instance}\ \mathit{Semigroup}\ b\\ &\qquad\qquad \Rightarrow \mathit{Semigroup}\ (a \to b)\ \textbf{where}\\ &\quad f \diamond g = \lambda x \to f\ x \diamond g\ x\end{aligned}$$

Exercise: *Semigroup* Instances

What is the value of the following expressions?

$$[1, 3, 7] \diamond [2, 4]$$
$$Sum\ 3 \diamond Sum\ 1 \diamond Sum\ 5$$
$$Just\ (Max\ 42) \diamond Nothing \diamond Just\ (Max\ 3)$$
$$sconcat\ (Product\ 2 :|\ [Product\ 3, Product\ 4])$$
$$([1], Product\ 2) \diamond ([2, 3], Product\ 3)$$
$$((1:) \diamond tail)\ [4, 5, 6]$$

Class *Monoid*

Recall: A monid is a semigroup with an identity element:

$$
\begin{aligned}
&\textbf{class } \mathit{Semigroup}\ a \Rightarrow \mathit{Monoid}\ a\ \textbf{where} \\
&\quad \mathit{mempty} \;::\; a \\
&\quad \mathit{mappend} :: \; a \to a \to a \\
&\quad \mathit{mappend} = (\diamond) \\
&\quad \mathit{mconcat} \;::\; [a] \to a \\
&\quad \mathit{mconcat} \;=\; \mathit{foldr}\ \mathit{mappend}\ \mathit{mempty}
\end{aligned}
$$

Minimum complete definition: *mempty*

Instances of *Monoid* (1)

A list $[a]$ is the archetypical example of a monoid:

$$
\begin{array}{l}
\mathbf{instance}\ Monoid\ [a]\ \mathbf{where} \\
\quad mempty = [\,]
\end{array}
$$

Any semigroup can be turned into a monoid by adjoining an identity element:

$$
\begin{array}{l}
\mathbf{instance}\ Semigroup\ a \\
\qquad\qquad \Rightarrow Monoid\ (Maybe\ a)\ \mathbf{where} \\
\quad mempty = Nothing
\end{array}
$$

Instances of *Monoid* (2)

Monoid instances for $Sum\ a$ and $Product\ a$:

$$\mathbf{instance}\ Num\ a \Rightarrow Monoid\ (Sum\ a)\ \mathbf{where}$$
$$\quad mempty = Sum\ 0$$
$$\mathbf{instance}\ Num\ a \Rightarrow Monoid\ (Product\ a)\ \mathbf{where}$$
$$\quad mempty = Product\ 1$$

Instances of $Monoid$ (3)

Monoid instances for $Min\ a$ and $Max\ a$:

$$
\begin{aligned}
&\mathbf{instance}\ (Ord\ a, Bounded\ a) \Rightarrow \\
&\qquad\qquad Monoid\ (Min\ a)\ \mathbf{where} \\
&\quad mempty = maxBound \\
&\mathbf{instance}\ (Ord\ a, Bounded\ a) \Rightarrow \\
&\qquad\qquad Monoid\ (Max\ a)\ \mathbf{where} \\
&\quad mempty = minBound
\end{aligned}
$$

Instances of $Monoid$ (4)

All products of monoids are monoids; e.g.:

$$\begin{aligned}
&\textbf{instance}\ (Monoid\ a, Monoid\ b) \\
&\qquad\qquad \Rightarrow Monoid\ (a, b)\ \textbf{where} \\
&\quad mempty = (mempty, mempty)
\end{aligned}$$

Instances of *Monoid* (4)

All products of monoids are monoids; e.g.:

$$\begin{aligned}&\textbf{instance } (Monoid\ a, Monoid\ b)\\ &\qquad\qquad \Rightarrow Monoid\ (a, b) \textbf{ where}\\ &\quad mempty = (mempty, mempty)\end{aligned}$$

$a \rightarrow b$ is a monoid if the range b is a monoid:

$$\begin{aligned}&\textbf{instance } Monoid\ b \Rightarrow Monoid\ (a \rightarrow b) \textbf{ where}\\ &\quad mempty\ \_ = mempty\end{aligned}$$

Functors (1)

A Functor is a notion that originated in a branch of mathematics called Category Theory.

However, for our purposes, we can think of functors as type constructors T (of arity 1) for which a function map can be defined:

$$map :: (a \to b) \to T a \to T b$$

that satisfies the following laws:

$$\begin{array}{rcl} map\ id & = & id \\ map (f \circ g) & = & map\ f \circ map\ g \end{array}$$

Functors (2)

Common examples of functors include (but are not limited to) ***container types*** like lists:

$$
\begin{aligned}
&mapList :: (a \to b) \to [\,a\,] \to [\,b\,] \\
&mapList \; \_ \; [\,] \qquad\quad = [\,] \\
&mapList \; f \; (x : xs) = f \; x : mapList \; f \; xs
\end{aligned}
$$

Functors (3)

And trees; e.g.:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
&\mathit{mapTree} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b \\
&\mathit{mapTree}\ f\ (\mathit{Leaf}\ x) \qquad\ = \mathit{Leaf}\ (f\ x) \\
&\mathit{mapTree}\ f\ (\mathit{Node}\ l\ x\ r) = \mathit{Node}\ (\mathit{mapTree}\ f\ l) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (f\ x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{mapTree}\ f\ r)
\end{aligned}
$$

Class $\mathit{Functor}$ (1)

Of course, the notion of a functor is captured by a type class in Haskell:

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \textbf{ where}\\
&\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\\
&\quad (<\$) :: a \to f\ b \to f\ a\\
&\quad (<\$) = \mathit{fmap} \circ \mathit{const}
\end{aligned}
$$

Class *Functor* (2)

There is also an infix version that can be viewed as function application lifted over a functor:

$$(<\$>) :: (a \to b) \to f\ a \to f\ b$$
$$(<\$>) = fmap$$

Compare the standard infix function application operator:

$$(\$) :: (a \to b) \to a \to b$$

Class *Functor* (3)

However, Haskell's type system is not powerful enough to enforce the functor laws.

Class *Functor* (3)

However, Haskell's type system is not powerful enough to enforce the functor laws.

In general, the programmer is responsible for ensuring that an instance respects all laws associated with a type class.

Class *Functor* (3)

However, Haskell's type system is not powerful enough to enforce the functor laws.

In general, the programmer is responsible for ensuring that an instance respects all laws associated with a type class.

Note that the type of $fmap$ can be read:

$$(a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$$

That is, we can see $fmap$ as promoting a function to work in a different context.

Instances of $Functor$ (1)

As noted, list is a functor:

$$\textbf{instance } Functor\ [\,]\ \textbf{where}$$
$$\quad fmap = listMap$$

Instances of *Functor* (1)

As noted, list is a functor:

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where} \\ \quad \mathit{fmap} = \mathit{listMap} \end{array}$$

Maybe is also a functor:

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\ \quad \mathit{fmap}\ \_\ \mathit{Nothing} = \mathit{Nothing} \\ \quad \mathit{fmap}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x) \end{array}$$

Instances of *Functor* (2)

Container types are in general instances of functor, including $Array$:

$$\textbf{instance } Functor\ (Array\ i)\ \textbf{where} \ldots$$

E.g, given a matrix $m :: Array\ (Int, Int)\ Double$, we can double all elements:

$$fmap\ (*2)\ m$$

Instances of *Functor* (3)

As functors are so common, there is a GHC extension for deriving *Functor* instances in standard cases.

For example, the functor instance for our tree type can be derived:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Tree}\ a = {} & \mathit{Leaf}\ a \\
\mid {} & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
& \mathbf{deriving}\ \mathit{Functor}
\end{aligned}
$$

Instances of $Functor$ (4)

The type of functions from a given domain is an example of a functor that is ***not a container*** type. Map is just function composition:

$$\textbf{instance}\ Functor\ ((\rightarrow)\ a)\ \textbf{where}$$
$$\quad fmap = (\circ)$$

Instances of *Functor* (4)

The type of functions from a given domain is an example of a functor that is ***not a container*** type. Map is just function composition:

$$\textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ a)\ \textbf{where}$$
$$\quad \mathit{fmap} = (\circ)$$

Note that a ***curried*** function type, like

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

thus is a ***nesting*** or ***composition*** of functors:

$$(((\rightarrow)\ a)\ (((\rightarrow)\ b)\ c)) = (((\rightarrow)\ a) \circ ((\rightarrow)\ b))\ c$$

Nesting functors (1)

In practice, functors often appear nested inside other functors, e.g.

$$mxs :: [\mathit{Maybe}\ \mathit{Double}]$$

Such a structure can of course be processed by repeated mapping, e.g.:

$$\mathit{fmap}\ (\mathit{fmap}\ (*2))\ mxs$$

One reading of this is “use fmap to lift $(*2)$ to work on Maybe, and then map that over the list”.

Nesting functors (2)

However, in general $f\ (g\ a) = (f \circ g)\ a$, meaning

$$fmap\ (fmap\ (*2)) = (fmap \circ fmap)\ (*2)$$

suggesting the following combinator:

$$\begin{aligned}(<\$\$>) &:: (Functor\ f, Functor\ g) \Rightarrow \\ &\qquad (a \rightarrow b) \rightarrow f\ (g\ a) \rightarrow f\ (g\ b) \\ (<\$\$>) &= fmap \circ fmap\end{aligned}$$

Nesting functors (2)

However, in general $f\ (g\ a) = (f \circ g)\ a$, meaning

$$fmap\ (fmap\ (*2)) = (fmap \circ fmap)\ (*2)$$

suggesting the following combinator:

$$\begin{aligned}(<\$\$>) &:: (Functor\ f, Functor\ g) \Rightarrow \\ &\qquad (a \to b) \to f\ (g\ a) \to f\ (g\ b) \\ (<\$\$>) &= fmap \circ fmap\end{aligned}$$

This allows us to simplify $fmap\ (fmap\ (*2))\ mxs$ to

$$(*2)\ <\$\$>\ mxs$$

Nesting functors (3)

Note that the composition of $fmap$s is mirrored by composition of functors at the type level:

$$[\,Maybe\ (a)] = [\,]\ (Maybe\ a) = ([\,] \circ Maybe)\ a$$

This can be generalized to any number levels; e.g.

$$(<\$\$\$>) = fmap \circ fmap \circ fmap$$
$$(*2)\ <\$\$\$>\ [[[1,2],[3]],[[4]],[[5]]]$$
$$\Rightarrow [[[2,4],[6]],[[8]],[[10]]]$$

$Data.Functor.Syntax$ defines $<\$\$>$, $<\$\$\$>$ ...

Nesting functors (4)

Note that we also could have defined:

$$(\lt\$\$\gt) = fmap\ fmap\ fmap$$

Why?

Exploiting that curried function types are composed functors, $\lt\$\$\gt$, $\lt\$\$\$\gt$ ... can compose functions where the second function has arity 2, 3, ... :

$$f :: Bool \to Double \to Int \to Double$$
$$(\gt 0)\ \lt\$\$\$\gt\ f :: Bool \to Double \to Int \to Bool$$

This is often quite handy in practice.

Class *Foldable* (1)

Class of data structures that can be folded to a summary value.

Many methods; minimal instance $foldMap$, $foldr$:

$$
\begin{array}{l}
\textbf{class } Foldable\ t \textbf{ where} \\
\quad \begin{array}{ll}
fold & :: Monoid\ m \Rightarrow t\ m \to m \\
foldMap & :: Monoid\ m \Rightarrow (a \to m) \to t\ a \to m \\
foldr & :: (a \to b \to b) \to b \to t\ a \to b \\
foldr' & :: (a \to b \to b) \to b \to t\ a \to b \\
foldl & :: (b \to a \to b) \to b \to t\ a \to b \\
foldl' & :: (b \to a \to b) \to b \to t\ a \to b
\end{array}
\end{array}
$$

Class *Foldable* (2)

(continued)

$$
\begin{aligned}
foldr1 &:: (a \to a \to a) \to t\ a \to a \\
foldl1 &:: (a \to a \to a) \to t\ a \to a \\
toList &:: t\ a \to [a] \\
null &:: t\ a \to Bool \\
length &:: t\ a \to Int \\
elem &:: Eq\ a \Rightarrow a \to t\ a \to Bool
\end{aligned}
$$

(Note that $length$ should be understood as $size$.)

Class *Foldable* (3)

(continued)

$$
\begin{array}{ll}
maximum & :: Ord\ a \Rightarrow t\ a \rightarrow a \\
minimum & :: Ord\ a \Rightarrow t\ a \rightarrow a \\
sum & :: Num\ a \Rightarrow t\ a \rightarrow a \\
product & :: Num\ a \Rightarrow t\ a \rightarrow a
\end{array}
$$

Class *Foldable* (3)

(continued)

$$\begin{aligned}
maximum &:: Ord\ a \Rightarrow t\ a \rightarrow a \\
minimum &:: Ord\ a \Rightarrow t\ a \rightarrow a \\
sum &:: Num\ a \Rightarrow t\ a \rightarrow a \\
product &:: Num\ a \Rightarrow t\ a \rightarrow a
\end{aligned}$$

Note: $foldl$ typically incurs a large space overhead due to laziness. The version with strict applictaion of the operator,$foldl'$ is typically preferable.

Instances of *Foldable* (1)

All expected instances, e.g.:

- **instance** *Foldable* [ ] **where** . . .
- **instance** *Foldable Maybe* **where** . . .

Instances of *Foldable* (1)

All expected instances, e.g.:

- $\textbf{instance}\ \mathit{Foldable}\ [\,]\ \textbf{where} \ldots$
- $\textbf{instance}\ \mathit{Foldable}\ \mathit{Maybe}\ \textbf{where} \ldots$

And GHC extension allows deriving instances in many cases; e.g.

$$\textbf{data}\ \mathit{Tree}\ a = \ldots \textbf{deriving}\ \mathit{Foldable}$$

Instances of *Foldable* (2)

But there are also some instances that are less expected, e.g.:

- $\textbf{instance}\ \mathit{Foldable}\ (\mathit{Either}\ a)\ \textbf{where} \ldots$
- $\textbf{instance}\ \mathit{Foldable}\ ((,)\ a)\ \textbf{where} \ldots$

Instances of *Foldable* (2)

But there are also some instances that are less expected, e.g.:

- $\textbf{instance}\ \mathit{Foldable}\ (\mathit{Either}\ a)\ \textbf{where} \ldots$
- $\textbf{instance}\ \mathit{Foldable}\ ((,)\ a)\ \textbf{where} \ldots$

This has some arguably odd consequences:

$$
\begin{aligned}
&\mathit{length}\ (1, 2) && \Rightarrow 1 \\
&\mathit{sum}\ (1, 2) && \Rightarrow 2 \\
&\mathit{length}\ (\mathit{Left}\ 1) && \Rightarrow 0 \\
&\mathit{length}\ (\mathit{Right}\ 2) && \Rightarrow 1
\end{aligned}
$$

Example: Folding Over a Tree (1)

Consider:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \\
\qquad \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
\qquad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{array}
$$

Example: Folding Over a Tree (1)

Consider:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \\
\quad\ \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
\quad\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{array}
$$

Let us make it an instance of $\mathit{Foldable}$:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Foldable}\ \mathit{Tree}\ \textbf{where} \\
\quad \mathit{foldMap}\ f\ \mathit{Empty} = \mathit{mempty} \\
\quad \mathit{foldMap}\ f\ (\mathit{Node}\ l\ a\ r) = \\
\qquad \mathit{foldMap}\ f\ l \diamond f\ a \diamond \mathit{foldMap}\ f\ r
\end{array}
$$

Example: Folding Over a Tree (2)

We wish to compute the sum and max over a tree of Int. One way:

$$sumMax :: Tree\ Int \to (Int, Int)$$
$$sumMax\ t = (foldl\ (+)\ 0\ t, foldl\ max\ minBound\ t)$$

Example: Folding Over a Tree (2)

We wish to compute the sum and max over a tree of Int. One way:

$$
\begin{aligned}
&sumMax :: Tree\ Int \to (Int, Int)\\
&sumMax\ t = (foldl\ (+)\ 0\ t, foldl\ max\ minBound\ t)
\end{aligned}
$$

Another way, with a single traversal:

$$
\begin{aligned}
&sumMax :: Tree\ Int \to (Int, Int)\\
&sumMax\ t = (sm, mx)\\
&\quad \mathbf{where}\\
&\qquad (Sum\ sm, Max\ mx) =\\
&\qquad\quad foldMap\ (\lambda n \to (Sum\ n, Max\ n))\ t
\end{aligned}
$$

Example: Folding Over a Tree (3)

The latter can be generalized to e.g. computing the sum, product, min, and max in a single traversal:

$$
\begin{array}{l}
foldMap \\
\quad (\lambda n \rightarrow (Sum\ n, Product\ n, Min\ n, Max\ n)) \\
\quad t
\end{array}
$$

Aside: Foldable?

Note that the kind of “folding” captured by the class $Foldable$ in general makes it impossible to recover the structure over which the “folding” takes place.

Aside: Foldable?

Note that the kind of “folding” captured by the class $Foldable$ in general makes it impossible to recover the structure over which the “folding” takes place.

Such an operation is also known as “reduce” or “crush”, and some authors prefer to reserve the term “fold” for ***catamorphisms***, where a separate combining function is given for each constructor, making it possible to recover the structure.

Aside: Foldable?

Note that the kind of "folding" captured by the class $Foldable$ in general makes it impossible to recover the structure over which the "folding" takes place.

Such an operation is also known as "reduce" or "crush", and some authors prefer to reserve the term "fold" for ***catamorphisms***, where a separate combining function is given for each constructor, making it possible to recover the structure.

One might thus argue that $Reducible$ or $Crushable$ would have been a more precise name.

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google’s original MapReduce framework
- Apache Hadoop

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google's original MapReduce framework
- Apache Hadoop

Functional mapping and folding with ***associative*** operator (semigroup) is amenable to parallelization and distribution.

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google's original MapReduce framework
- Apache Hadoop

Functional mapping and folding with ***associative*** operator (semigroup) is amenable to parallelization and distribution.

However, achieving scalability in practice required both careful engineering of the frameworks as such, and a good understanding of how to use them on part of the user.