N-Queens Combinatorial Problem

Polyglot FP for Fun and Profit – Haskell and Scala

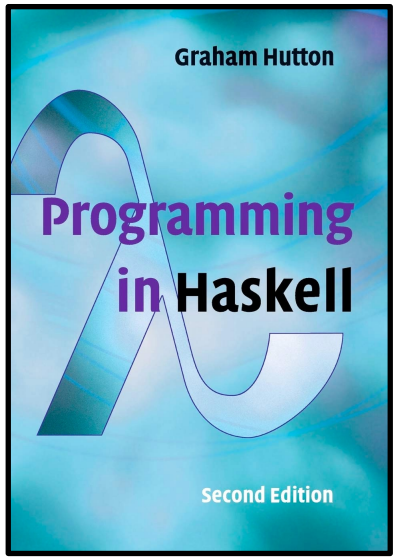
See how feeding FP workhorses map and filter with monadic steroids turns them into the intriguing mapM and filterM

Graduate to foldM by learning how it behaves with the help of three simple yet instructive examples of its usage

Use the powers of foldM to generate all permutations of a collection with a simple one-liner

Exploit what you learned about foldM to solve the N-Queens Combinatorial Problem with an iterative approach rather than a recursive one

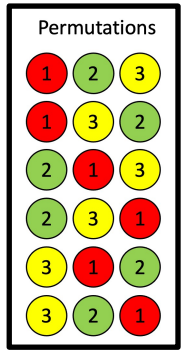
Part 4

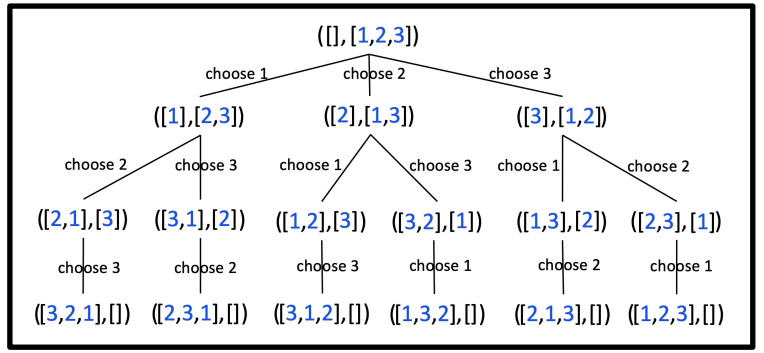


Graham Hutton

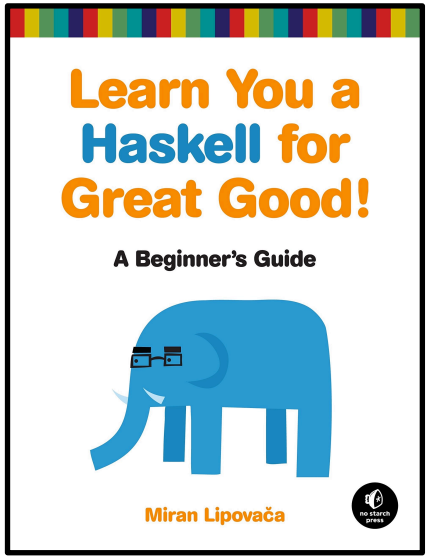
@haskellhutt

monadic mapping, filtering, folding $\begin{cases} mapM \\ filterM \\ foldM \end{cases}$





Haskell Scala



Miran Lipovača

slides by

@philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

The plan for **Part 4** of this series is to first take a quick look at the **Haskell** equivalent of the **Scala intersperse** and **intercalate** functions we used in **Part 3**, and then to come up with an alternative way of solving the **N-Queens problem** using the **foldM** function.

If on a first reading you want to get straight to the meat of this slide deck then consider skipping the first seven slides.

Here are the Scala intersperse and intercalate functions that we used in **Part 3**.

Scala

scala.collection.decorators

IteratorDecorator

```
final class IteratorDecorator[A] extends AnyVal
```

Enriches Iterator with additional methods.

```
def intersperse[B >: A](sep: B): Iterator[B]
```

Inserts a separator value between each element.

```
Iterator(1, 2, 3).intersperse(0) === Iterator(1, 0, 2, 0, 3)
Iterator('a', 'b', 'c').intersperse(',') === Iterator('a', ',', 'b', ',', 'c')
Iterator('a').intersperse(',') === Iterator('a')
Iterator().intersperse(',') === Iterator()
```

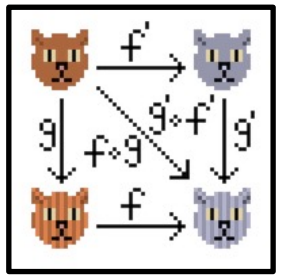
The === operator in this pseudo code stands for 'is equivalent to'; both sides of the === give the same result.

sep the separator value.

returns The resulting iterator contains all elements from the source iterator, separated by the sep value.

Note Reuse: After calling this method, one should discard the iterator it was called on, and use only the iterator that was returned. Using the old iterator is undefined, subject to change, and may result in changes to the new iterator as well.

Cats




```
def intercalate[A](fa: F[A], a: A)(implicit A: Monoid[A]): A
```

Intercalate/insert an element between the existing elements while folding.

```
scala> import cats.implicits._
scala> Foldable[List].intercalate(List("a","b","c"), "-")
res0: String = a-b-c
scala> Foldable[List].intercalate(List("a"), "-")
res1: String = a
scala> Foldable[List].intercalate(List.empty[String], "-")
res2: String = ""
scala> Foldable[Vector].intercalate(Vector(1,2,3), 1)
res3: Int = 8
```

typelevel.org/cats/api/cats/Foldable.html

cats

Foldable

@philip_schwarz

In the next slide we see how **Miran Lipovača** introduces **intersperse** and **intercalate** functions that operate on **lists**.

```
> import Data.List

> :type intersperse
intersperse :: a -> [a] -> [a]

> :type intercalate
intercalate :: [a] -> [[a]] -> [a]
```

Data.List

The Data.List module is all about lists, obviously. It provides some very useful functions for dealing with them.

We've already met some of its functions (like map and filter) because the Prelude module exports some functions from Data.List for convenience.

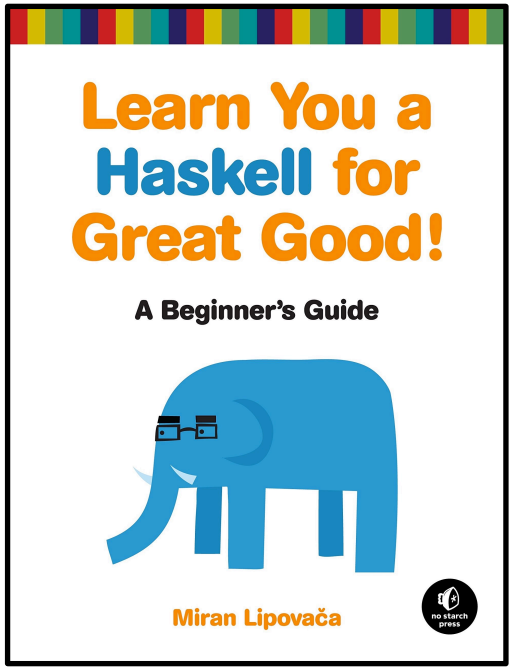
You don't have to import Data.List via a qualified import because it doesn't clash with any Prelude names except for those that Prelude already steals from Data.List. Let's take a look at some of the functions that we haven't met before.

intersperse takes an element and a list and then puts that element in between each pair of elements in the list. Here's a demonstration:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"

ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

intercalate takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"

ghci> intercalate  [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```



Miran Lipovača

When we used the Scala intersperse function in **Part 3**, we used it with lists.

```
type Grid[A] = List[List[A]]
```

```
import scala.collection.decorators._
def insertPadding(images: Grid[Image]): Grid[Image] =
  images map (_ intersperse paddingImage) intersperse List(paddingImage)
```

So the intersperse function that we saw on the previous slide will do as the Haskell analogue.

As for the Scala Cats intercalate function that we used in **Part 3**, although we used it on lists, it was much more generic in that it operated on a Foldable and so rather than simply concatenating the lists contained in a list, it folded a Foldable using a Monoid.

```
import cats.Monoid
val beside = Monoid.instance[Image](Image.empty, _ beside _)
val above  = Monoid.instance[Image](Image.empty, _ above _)
```

```
def combineWithPadding(images: Grid[Image], paddingImage: Image): Image =
  import cats.implicits._
  images.map(row => row.intercalate(paddingImage)(beside))
        .intercalate(paddingImage)(above)
```

Where can we find the Haskell equivalent of this more generic version of intercalate?

In **Programming in Haskell**, **Foldable** is said to be located in **Data.Foldable**.

If I search **Hoogle** for **intercalate**, **Data.Foldable** does not show up. In **Hoogle**, the versions of **intercalate** that do show up, either don't involve both **Foldable** and **Monoid**, or are in what appear to me to be 'not-so-mainstream' modules.

Hoogle | intercalate | set:stackage | Search

intercalate :: (MonoFoldable mono, Monoid (Element mono)) => Element mono -> mono -> Element mono

mono-traversable Data.MonoTraversable.Unprefixed

Synonym for ointercalate

intercalate :: Monoid w => w -> [w] -> w

basic-prelude BasicPrelude

```
intercalate = mconcat .: intersperse
```

intercalate :: (Sequential c, Monoid (Item c)) => Element c -> c -> Element c

foundation Foundation Foundation.Collection

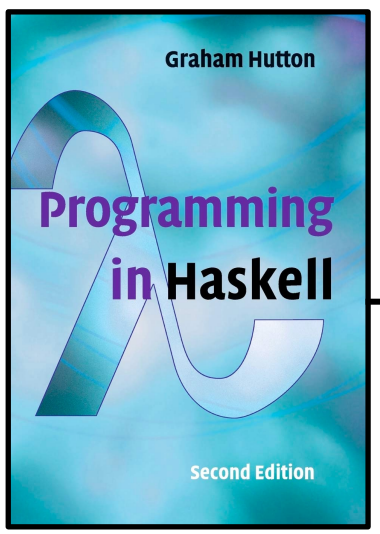
intercalate `xs xss` is equivalent to `(mconcat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

intercalate :: Monoid a => a -> [a] -> a

clay Clay.Property

intercalate :: (Foldable f, Monoid m) => m -> f m -> m

bytestring-tree-builder ByteString.TreeBuilder



In **Haskell Programming**, I see an **intercalate** in use, but it is the one that operates on lists.

B.17 Foldables

The declarations below are provided in the library `Data.Foldable`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Data.Foldable
```

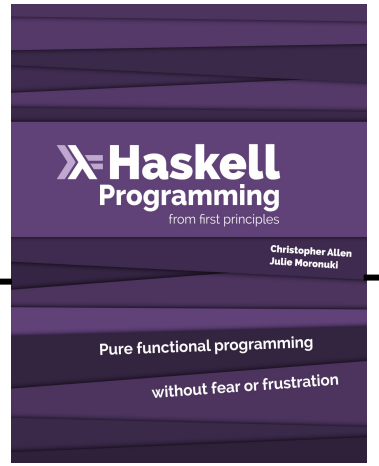
Class declaration:

```
class Foldable t where
   foldMap :: Monoid b => (a -> b) -> t a -> b
   foldr   :: (a -> b -> b) -> b -> t a -> b

   fold    :: Monoid a => t a -> a
   foldl   :: (a -> b -> a) -> a -> t b -> a
   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 :: Ord a => t a -> a
   minimum :: Ord a => t a -> a
   sum     :: Num a => t a -> a
   product :: Num a => t a -> a
```

base-4.15.0.0: Basic libraries

Data.List

```
intercalate :: [a] -> [[a]] -> [a]
```



CHAPTER 14. TESTING

```
where
  convertLine line = do
    let morse = stringToMorse line
    case morse of
     (Just str)
       -> putStrLn
          (intercalate " " str)
     Nothing
       -> do
         putStrLn $ "ERROR: " ++ line
         exitFailure
```

We can find an **intercalate** function based on **Monoid** and **Foldable** in **monoid**'s **Data.Monoids**.

Is **monoid**'s **Data.Monoids** a sensible library (the most sensible, even?) to depend on for **intercalate**?

Are there more sensible places where to find **intercalate**?

Why does **monoid**'s **Data.Monoids** not show up in **Hoogle**?

What is the status of **monoid**'s **Data.Monoids**? Is it not mainstream/recognized/official in some way?

```
intercalate :: (Monoid a, Foldable f) => a -> f a -> a
```

hackage.haskell.org/package/monoid-0.1.9

Hackage :: [Package] Search · Browse · What's new · Upload ·

monoid: Monoid type classes, designed in modular way, distinguish Monoid from Mempty and Semigroup. This design allows mempty operation don't bring Semigroups related constraints until (<>) is used.

[apache, data, library] [Propose Tags]

Build InstallOk | Documentation Available

Modules

[Index] [Quick Jump]

Data

Data.Monoids

Versions [RSS] [faq]

0.1.8, **0.1.9**

Dependencies

base (>=4.9 && <4.13), containers, lens, mtl [details]

License

Apache-2.0

Copyright

Data.Monoids

Documentation

```
type family Monoids lst :: Constraint where ...
```

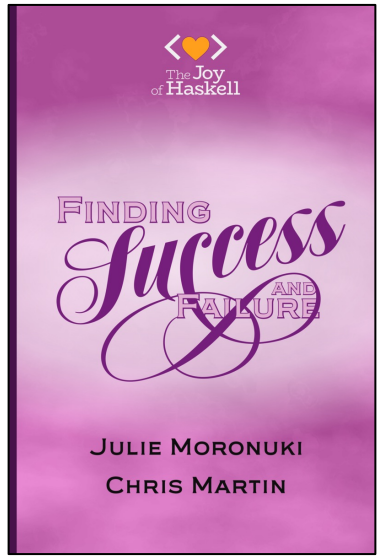
When **Chris Martin** (coauthor of **Finding Success and Failure** - **The Joy of Haskell Series**) replied to my questions, I realised that **intersperse** can be used to implement **intercalate**

So here is an example of doing just that (using the **Tree** data structure provided by https://hackage.haskell.org/package/containers-0.6.5.1)

Chris Martin
@chris_martin

Replying to @philip_schwarz and @argumatronic

It would be really cool if the Data.Foldable had an intercalate function. I don't know of any library I would recommend. Instead of (intercalate x xs), I think I would write the slightly lengthier expression (fold (intersperse x (toList xs)))




```
> import Data.Foldable
> import Data.List
> import Data.Tree

> :type toList
toList :: Foldable t => t a -> [a]

> :type fold
fold :: (Foldable t, Monoid m) => t m -> m

> :type intersperse
intersperse :: a -> [a] -> [a]

> fold (intersperse "-" (toList (Node "a" [Node "b" [], Node "c" [], Node "d" []])))
"a-b-c-d"
```

```
data Tree a
```

Non-empty, possibly infinite, multi-way trees; also known as *rose trees*.

Constructors

Node

`rootLabel :: a`	label value
`subForest :: [Tree a]`	zero or more child trees

And here on the right we do the same thing, but using the **intercalate** function provided by **monoid**'s **Data.Monoids**.

```
> import Data.Monoids
> import Data.Tree
> :type intercalate
intercalate :: (Mempty a, Semigroup a, Foldable f) => a -> f a -> a

> intercalate "-" (Node "a" [Node "b" [], Node "c" [], Node "d" []])
"a-b-c-d"
```

@philip_schwarz

After that quick look at the **Haskell** equivalent of the **intersperse** and **intercalate** functions, let's now turn to the task of finding out how the **N-Queens combinatorial problem** can be solved using the **foldM** function.

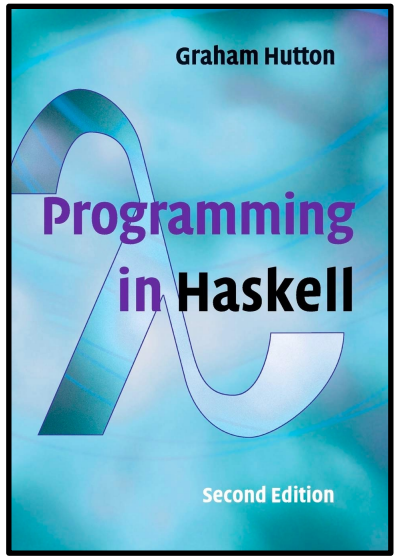
Before we start looking at the solution, we need to make sure that we fully understand how the **foldM** function works.

To prepare for that, we are first going to get an understanding (or remind ourselves) of how the **mapM** and **filterM** functions work.

On the next slide we look at how **Graham Hutton** explains the **mapM** function in his **Haskell** book.

Graham Hutton

@haskellhutt



Generic functions

An important benefit of abstracting out the concept of monads is the ability to define generic functions that can be used with any monad. A number of such functions are provided in the library Control.Monad. For example, a monadic version of the map function on list can be defined as follows:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

```
map :: (a -> b) -> [a] -> [b]
```

Note that mapM has the same type as map, except that the argument function and the function itself now have monadic return types. To illustrate how it might be used, consider a function that converts a digit character to its numeric value, provided that the character is indeed a digit:

```
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing
```

(The functions isDigit and digitToInt are provided in Data.Char.) Then applying mapM to the conv function gives a means of converting a string of digits into the corresponding list of numeric values, which succeeds if every character in the string is a digit, and fails otherwise:

```
> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a"
Nothing
```

If you are not familiar with the **traverse** function, then just skip the next slide.

Let's do the same in **Scala**. Except that I can't find **mapM** in **Cats** or **Scalaz**! It turns out, however, that the signature of the **mapM** function that we have just seen is just a **specialization** of the **traverse** function.

Here is the **mapM** function again

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

specialised for **lists**

And here is how a **more generic** version is defined in terms of **traverse**:

```
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  …
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
mapM = traverse
```

Every **monad** is also an **applicative**

generalized for any **traversable**

So here is the **Haskell mapM** example again, and next to it the **Scala** equivalent (using **Cats**)

```
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing

> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a”
Nothing
```

```
def conv(c: Char): Option[Int] = c match
  case _ if c.isDigit => Some(c.asDigit)
  case _ => None

assert( "1234".toList.traverse(conv)
        == Some(List(1,2,3,4)))

assert( "1234a".toList.traverse(conv)
        == None)
```

```
def conv(c: Char): Option[Int] =
  Option.when(c.isDigit)(c.asDigit)
```

alternatively

That was **mapM**.

Now let's move on to **filterM**.

In the next slide we look at how **Miran Lipovača** explains the **filterM** function in his **Haskell** book.

filterM

The filter function is pretty much the bread of Haskell programming (map being the butter). It takes a predicate and a list to filter out and then returns a new list where only the elements that satisfy the predicate are kept. Its type is this:

```
filter :: (a -> Bool) -> [a] -> [a]
```

The predicate takes an element of the list and returns a Bool value. Now, what if the Bool value that it returned was actually a monadic value? Whoa! That is, what if it came with a context? Could that work?

For instance, what if every True or a False value that the predicate produced also had an accompanying monoid value, like ["Accepted the number 5"] or ["3 is too small"]? That sounds like it could work. If that were the case, we'd expect the resulting list to also come with a log of all the log values that were produced along the way. So if the Bool that the predicate returned came with a context, we'd expect the final resulting list to have some context attached as well, otherwise the context that each Bool came with would be lost.

The filterM function from Control.Monad does just what we want! Its type is this:

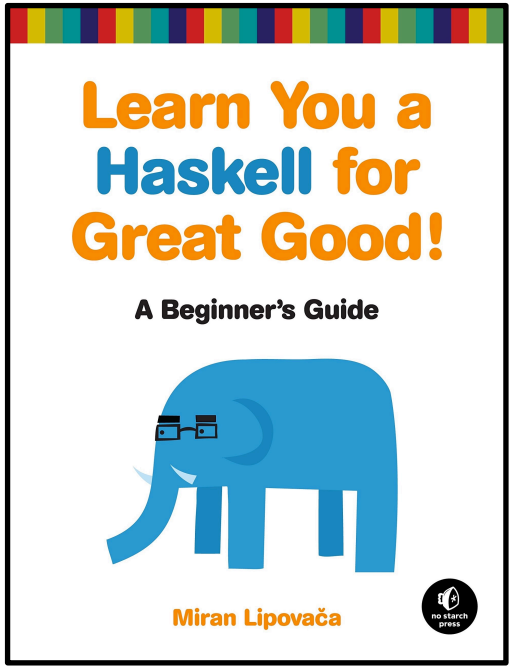
```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

The predicate returns a monadic value whose result is a Bool, but because it's a monadic value, its context can be anything from a possible failure to non-determinism and more! To ensure that the context is reflected in the final result, the result is also a monadic value.

Let's take a list and only keep those values that are smaller than 4. To start, we'll just use the regular filter function:

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

That's pretty easy.



Miran Lipovača

While the next five slides provide a useful example of using the **filterM** function, the example involves the **Writer monad**, so if you are not familiar with that **monad** you may want to skip the slides for now.

If you could do with an introduction to the **Writer monad**, then you might want to check out the slide deck on the right.

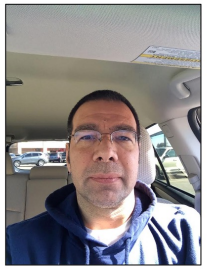
Whether you go through the next five slides or skip them, the two slides after that provide a nice and simple example of using the **filterM** function on the **List monad**.

@philip_schwarz

Writer Monad

Learn how to use the Writer monad to log (trace) the execution of functions through the work of

Bartosz Milewski
@BartoszMilewski



Alvin Alexander
@alvinalexander

slides by @philip_schwarz

Now, let's make a predicate that, aside from presenting a True or False result, also provides a log of what it did. Of course, we'll be using the Writer monad for this:

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
    | x < 4 = do
        tell ["Keeping " ++ show x]
        return True
    | otherwise = do
        tell [show x ++ " is too large, throwing it away"]
        return False
```

Here is the signature of filterM again, for reference.

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

Instead of just returning a Bool, this function returns a Writer [String] Bool. It's a monadic predicate. Sounds fancy, doesn't it? If the number is smaller than **4** we report that we're keeping it and then **return** True. Now, let's give it to filterM along with a list. Because the predicate returns a Writer value, the resulting list will also be a Writer value.

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

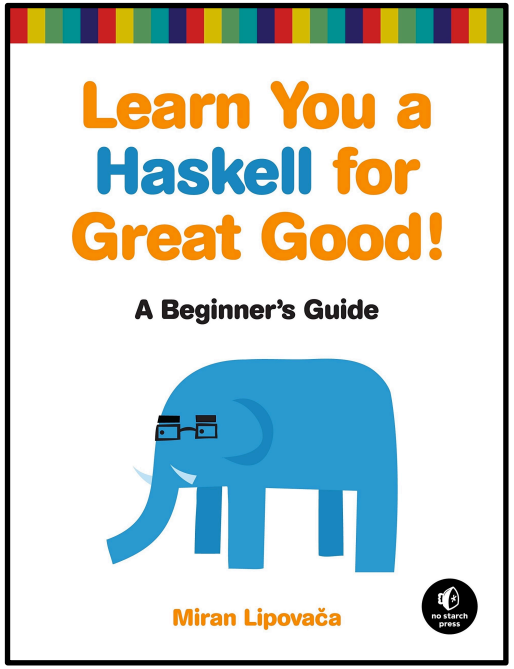
Examining the result of the resulting Writer value, we see that everything is in order. Now, let's print the log and see what we got:

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 is too large, throwing it away
Keeping 1
5 is too large, throwing it away
Keeping 2
10 is too large, throwing it away
Keeping 3
```

Awesome. So just by providing a monadic predicate to filterM, we were able to filter a list while taking advantage of the monadic context that we used.

mapM_ is a variant of the mapM function that we saw earlier.

```
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```
Map each element of a structure to a monadic action, evaluate these actions from left to right, and ignore the results. For a version that doesn't ignore the results see mapM.



Miran Lipovača

The next slide shows the **Scala** equivalent of the **keepSmall** function, together with some tests.

```haskell
import Control.Monad.Writer
```

```haskell
keepSmall :: Int -> Writer [String] Bool
keepSmall x
    | x < 4 = do
        tell ["Keeping " ++ show x]
        return True
    | otherwise = do
        tell [show x ++ " is too large, throwing it away"]
        return False
```

```haskell
(assertEqual "keepSmall test 1"
             (keepSmall 2)
             (writer (True, ["Keeping 2"])))

(assertEqual "keepSmall test 2"
             (keepSmall 5)
             (writer (False, ["5 is too large, throwing it away"])))

(assertEqual "keepSmall test 3"
             (runWriter (keepSmall 2))
             (True, ["Keeping 2"]))

(assertEqual "keepSmall test 4"
             (runWriter (keepSmall 5))
             (False, ["5 is too large, throwing it away"]))

(assertEqual "keepSmall test 5"
             (fst (runWriter (keepSmall 2)))
             True)
```

```scala
import cats.data.Writer
import cats.instances._
```


```scala
def keepSmall(x: Int): Writer[[String],Boolean] =
  if x < 4
  then
    for
      _ <- Writer.tell(List("Keeping " + x))
    yield true
  else
    for
      _ <- Writer.tell(List(x + " is too large, throwing it away"))
    yield false
```

```scala
assert(keepSmall(2)
       ==
       Writer(List("Keeping 2"),true))

assert(keepSmall(5)
       ==
       Writer(List("5 is too large, throwing it away"),false))

assert(keepSmall(2).run
       ==
       (List("Keeping 2"),true))

assert(keepSmall(5).run
       ==
       (List("5 is too large, throwing it away"),false))

assert(keepSmall(2).value
       ==
       true)
```

Now let's look at the **Scala** equivalent of passing the **keepSmall** function to **filterM**.

While the **Scala Cats** library doesn't provide **filterM**, which operates on **monads**, it provides **filterA**, which operates on **applicatives**, and is a generalization of **filterM**.

cats

TraverseFilter Companion object TraverseFilter

```
trait TraverseFilter[F[_]] extends FunctorFilter[F]
```

TraverseFilter, also known as Witherable, represents list-like structures that can essentially have a traverse and a filter applied as a single combined operation (traverseFilter).

Based on Haskell's Data.Witherable

Cats

hackage.haskell.org/package/witherable-0.4.1/docs/Witherable.html

witherable-0.4.1: filterable traversable

Witherable

```
class (Traversable t, Filterable t) => Witherable t where
```

```
filterA[G[_], A](fa: F[A])(f: (A) ⇒ G[Boolean])(implicit G: Applicative[G]): G[F[A]]
```

Filter values inside a G context.

This is a generalized version of Haskell's filterM. This StackOverflow question about filterM may be helpful in understanding how it behaves.

Example:

```
scala> import cats.implicits._
scala> val l: List[Int] = List(1, 2, 3, 4)
scala> def odd(i: Int): Eval[Boolean] = Now(i % 2 == 1)
scala> val res: Eval[List[Int]] = l.filterA(odd)
scala> res.value
res0: List[Int] = List(1, 3)

scala> List(1, 2, 3).filterA(_ => List(true, false))
res1: List[List[Int]] = List(List(1, 2, 3), List(1, 2), List(1, 3), List(1), List(2, 3), List(2), List(3), List())
```

See the slide after next for more on the bottom example.

```
filterA :: Applicative f => (a -> f Bool) -> t a -> f (t a)
```

rename

```
filterA :: Applicative g => (a -> g Bool) -> f a -> g (f a)
```

Applicative	G	g
function parameter list-like parameter	A ⇒ G[Boolean] F[A]	a -> g Bool f a
result	G[F[A]]	g (f a)

```
def filterA[G[_], A](fa: F[A])(f: (A) ⇒ G[Boolean])(implicit G: Applicative[G]): G[F[A]]
```

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

Note that while **filterM** operates on a list, **filterA** operates on any list-like **F**.

```
assertEqual
  "keepSmall test 6"
  (fst (runWriter (filterM keepSmall [9,1,5,2,10,3])))
  [1,2,3])
```

```
assert(
  List(9,1,5,2,10,3).filterA(keepSmall).value
  ==
  List(1,2,3))
```

F = List

Cats

```
assertEqual
  "keepSmall test 7"
  (fst (runWriter (listen (filterM keepSmall [9,1,5,2,10,3]))))
  ([1,2,3],["9 is too large, throwing it away",
            "Keeping 1",
            "5 is too large, throwing it away",
            "Keeping 2",
            "10 is too large, throwing it away",
            "Keeping 3"])
```

```
assert(
  List(9,1,5,2,10,3).filterA(keepSmall).listen.value
  ==
  (List(1,2,3),List("9 is too large, throwing it away",
                    "Keeping 1",
                    "5 is too large, throwing it away",
                    "Keeping 2",
                    "10 is too large, throwing it away",
                    "Keeping 3")))
```

F = List

Cats

```
assert( Vector(9,1,5,2,10,3).filterA(keepSmall).value == List(1,2,3))

assert( Vector(9,1,5,2,10,3).filterA(keepSmall).listen.value ==
  (Vector(1,2,3),List("9 is too large, throwing it away",
    "Keeping 1",
    "5 is too large, throwing it away",
    "Keeping 2",
    "10 is too large, throwing it away",
    "Keeping 3")))
```

F = Vector

Cats

```
assert( Option(3).filterA(keepSmall).listen.value ==
  (Option(3),List("Keeping 3")))

assert( Option(9).filterA(keepSmall).listen.value ==
  (None,List("9 is too large, throwing it away")))
```

F = Option

Cats

As promised, the next two slides show a nice and simple example of using the **filterM** function on the **List monad**.

@philip_schwarz

A very cool Haskell trick is using filterM to get the powerset of a list (if we think of them as sets for now). The powerset of some set is a set of all subsets of that set. So if we have a set like **[1,2,3]**, its powerset would include the following sets:

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

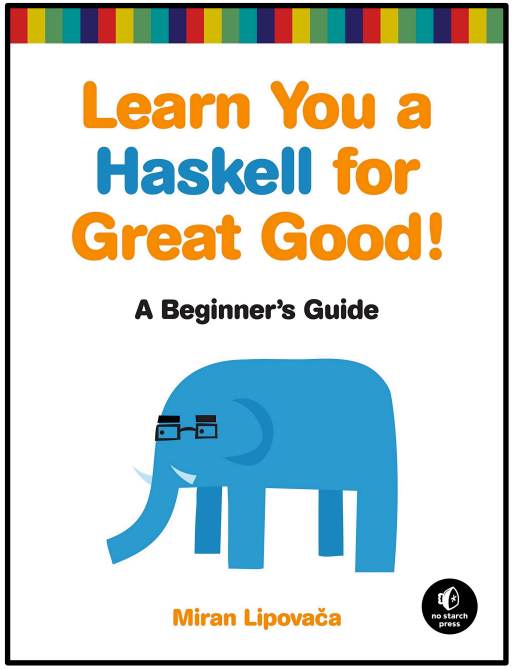
In other words, getting a powerset is like getting all the combinations of keeping and throwing out elements from a set. **[2,3]** is like the original set, only we excluded the number **1**. To make a function that returns a powerset of some list, we're going to rely on non-determinism. We take the list **[1,2,3]** and then look at the first element, which is **1** and we ask ourselves: should we keep it or drop it? Well, we'd like to do both actually. So we are going to filter a list and we'll use a predicate that non-deterministically both keeps and drops every element from the list. Here's our powerset function:

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

Wait, that's it? Yup. We choose to drop and keep every element, regardless of what that element is. We have a non-deterministic predicate, so the resulting list will also be a non-deterministic value and will thus be a list of lists. Let's give this a go:

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

This takes a bit of thinking to wrap your head around, but if you just consider lists as non-deterministic values that don't know what to be so they just decide to be everything at once, it's a bit easier.



Miran Lipovača

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

Graham Hutton

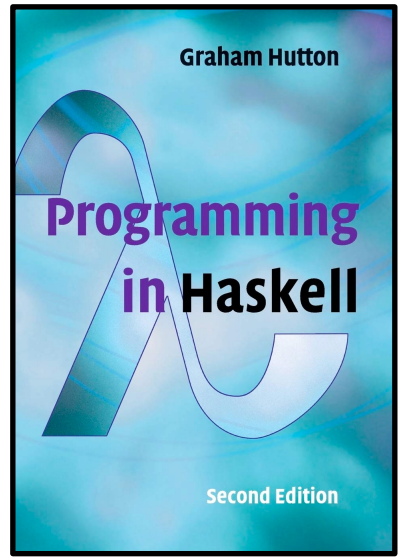
@haskellhutt

A **monadic** version of the **filter** function on lists is defined by generalizing its type and definition in a similar manner to **mapM**:

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```

For example, in the case of the **list monad**, **using filterM provides a particularly concise means of computing the powerset of a list**, which is given by **all possible ways of including or excluding each element of the list**:

```
> filterM (\x -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```



After familiarising (or reacquainting) ourselves with **mapM** and **filterM**, it is finally time to look at **foldM**.

Folding Unfolded

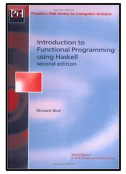
Polyglot FP for Fun and Profit
Haskell and Scala

See how **recursive functions** and **structural induction** relate to **recursive datatypes**

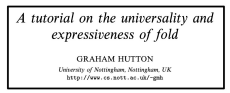
Follow along as the **fold abstraction** is introduced and explained

Watch as **folding** is used to simplify the definition of **recursive functions** over **recursive datatypes**

Part 1 - through the work of







Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

Graham Hutton
@haskellhutt

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Folding Unfolded

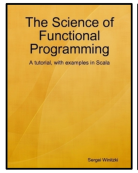
Polyglot FP for Fun and Profit
Haskell and Scala

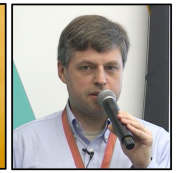
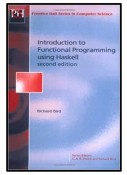
See **aggregation functions** defined **inductively** and implemented using **recursion**

Learn how in many cases, **tail-recursion** and the **accumulator trick** can be used to avoid **stackoverflow errors**

Watch as **general aggregation** is implemented and see **duality theorems** capturing the relationship between **left folds** and **right folds**

Part 2 - through the work of



Sergei Winitzki
sergei-winitzki-11a6431

Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

If you could do with an introduction to (or refresher on) **folding**, then maybe have a look at one or more of the first three decks in this series.

Folding Unfolded

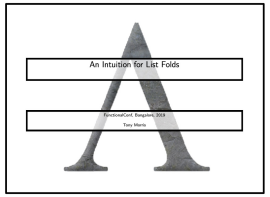
Polyglot FP for Fun and Profit
Haskell and Scala

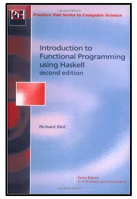
Develop the **correct intuitions** of what **fold left** and **fold right** actually do, and how different these two functions are

Learn other important concepts about **folding**, thus reinforcing and expanding on the material seen in parts 1 and 2

Includes a brief introduction to (or refresher of) **asymptotic analysis** and θ**-notation**

Part 3 - through the work of



Tony Morris
@dibblego

https://presentations.tmorris.net/

Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Folding Unfolded

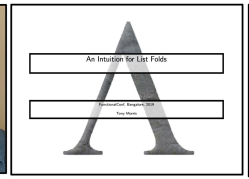
Polyglot FP for Fun and Profit
Haskell and Scala

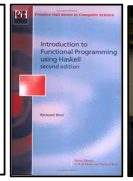
Can a **left fold** ever work over an **infinite list**? What about a **right fold**? Find out.

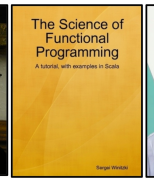
Learn about the other two functions used by functional programmers to implement **mathematical induction**: **iterating** and **scanning**.

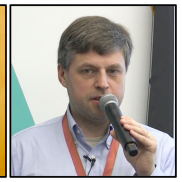
Learn about the limitations of the **accumulator technique** and about **tupling**, a technique that is the dual of the **accumulator trick**.

Part 4 - through the work of





Tony Morris
@dibblego

https://presentations.tmorris.net/

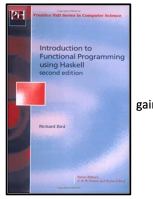
Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

Sergei Winitzki
sergei-winitzki-11a6431

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

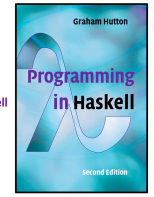


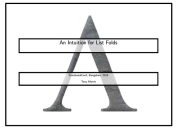
gain a deeper understanding of why **right folds** over very large and **infinite lists** are sometimes possible in **Haskell**

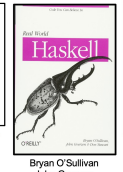
see how **lazy evaluation** and **function strictness** affect **left** and **right folds** in **Haskell**

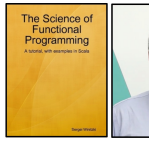
learn when an ordinary **left fold** results in a **space leak** and how to avoid it using a **strict left fold**

Part 5 - through the work of








Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

Tony Morris
@dibblego

https://presentations.tmorris.net/

Bryan O'Sullivan
John Goerzen
Donald Bruce Stewart

Sergei Winitzki
sergei-winitzki-11a6431

Graham Hutton
@haskellhutt

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

foldM

The monadic counterpart to foldl is foldM. If you remember your folds from the folds section, you know that foldl takes a binary function, a starting accumulator and a list to fold up and then folds it from the left into a single value by using the binary function. foldM does the same thing, except it takes a binary function that produces a monadic value and folds the list up with that. Unsurprisingly, the resulting value is also monadic. The type of foldl is this:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Whereas foldM has the following type:

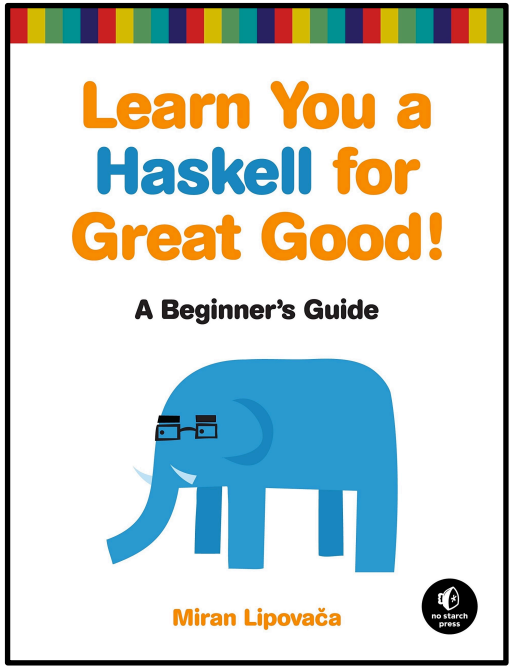
```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

The value that the binary function returns is monadic and so the result of the whole fold is monadic as well. Let's sum a list of numbers with a fold:

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

The starting accumulator is **0** and then **2** gets added to the accumulator, resulting in a new accumulator that has a value of **2**. **8** gets added to this accumulator resulting in an accumulator of **10** and so on and when we reach the end, the final accumulator is the result.

Now what if we wanted to sum a list of numbers but with the added condition that if any number is greater than **9** in the list, the whole thing fails? It would make sense to use a binary function that checks if the current number is greater than **9** and if it is, fails, and if it isn't, continues on its merry way. Because of this added possibility of failure, let's make our binary function return a Maybe accumulator instead of a normal one.



Miran Lipovača

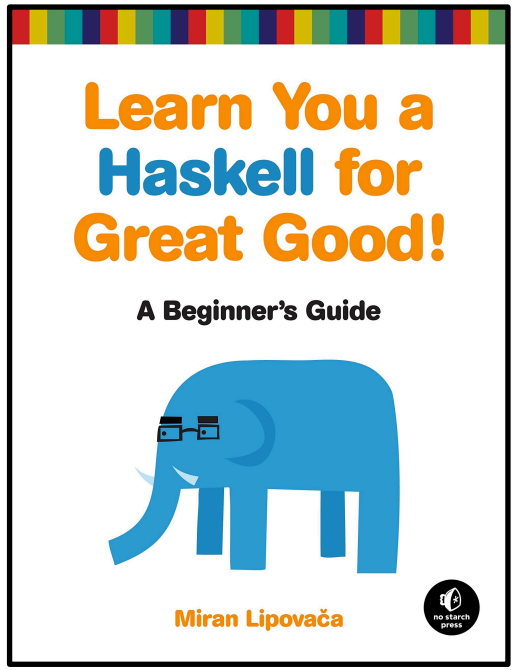
Here's the binary function:

```haskell
binSmalls  :: Int -> Int -> Maybe Int
binSmalls  acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)
```

Because our **binary function** is now a **monadic function**, we can't use it with the normal **foldl**, but we have to use **foldM**. Here goes:

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14

ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

Excellent! Because one number in the list was greater than **9**, the whole thing resulted in a **Nothing**. Folding with a **binary function** that returns a **Writer** value is cool as well because then you log whatever you want as your **fold** goes along its way.



Here is the **Scala** equivalent of the above example using **Cats**.


```scala
def binSmalls(acc: Int, x: Int): Option[Int] = x match
  case n if n > 9 => None
  case otherwise  => Some(acc + x)
```

alternatively →

```scala
def binSmalls(acc: Int, x: Int): Option[Int] =
  Option.unless(x > 9)(acc + x)
```

```scala
import cats.syntax.foldable._
assert(  List(2,8,3,1).foldM(0)(binSmalls) == Some(14) )
assert( List(2,11,3,1).foldM(0)(binSmalls) == None )
```

@philip_schwarz

While that example of using **foldM** with a **binary function** that returns an **optional value** is useful, things get a bit harder to understand when the **binary function** returns a **list of values**.

Since the way that we are going to solve the **N-Queens combinatorial problem** using **foldM** is by passing the latter a **binary function** returning a **list of values**, in upcoming slides we are going to look at a number of examples that do just that, in order to strengthen our understanding of the **foldM** function.

Before we do that though, let's take another look at the definition of **foldM**.

If we look back at **Martin Lipovača**'s definition of the **foldM** function, we see that it doesn't explain much. Rather, it is his example that helps a bit to understand how **foldM** works.

The nice thing about the official definition of **foldM** (on the next slide) is that while on the one hand it explains even less, on the other hand, it is accompanied by a very helpful **equivalence** that gives us a very concrete way of understanding what **foldM** does. We'll be reminding ourselves of this **equivalence** a few times in upcoming slides.

By the way: while the signature of the **foldM** function explained by **Martin Lipovača** operates exclusively on lists

```
foldM :: (Monad m) => (b -> a -> m b) -> b -> [a] -> m b
```

the definition of **foldM** on the next slide is more generic and operates on any **Foldable**:

```
foldM :: (Foldable t, Monad m)          => (b -> a -> m b) -> b -> t a -> m b
```

Miran Lipovača

The monadic counterpart to foldl is foldM...

foldl takes a binary function, a starting accumulator and a list to fold up and then folds it from the left into a single value by using the binary function.

foldM does the same thing, except it takes a binary function that produces a monadic value and folds the list up with that.

Unsurprisingly, the resulting value is also monadic.

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b   # Source
```

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the `folded function' are not commutative.

```
foldM f a1 [x1, x2, ..., xm]

==

do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

If right-to-left evaluation is required, the input list should be reversed.

Note: `foldM` is the same as `foldlM`

https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Monad.html

As planned, we now turn to examples of **folding** a list using **foldM** with a **binary function** that returns a list.

As shown in the example below, the length of the list returned by **foldM** is a function of the lengths of both its list parameter and the list returned by its function parameter..

In this first example, we deliberately use a **binary function** that ignores both its parameters, to stress the point that the above **property** holds regardless of the particular values contained in the lists.

If **foldM** is applied to
- a function returning a list of length **m**
- an initial accumulator **z**
- a list of length **n**

Then **foldM** returns a list of length **m ^ n**

```
e.g. (foldM (\_ -> \_ -> [0,0]) 9 [1,2,3]) ⟹ [0,0,0,0,0,0,0,0]
                          m=2   z   n=3          m^n = 2^3 = 8
```

If **foldM** is applied to
- no matter which function
- an initial accumulator **z**
- an empty list (i.e. a list with length **n**=0)

Then **foldM** returns list [**z**].

```
                                  z n=0      z
e.g. (foldM (\_ -> \_ -> [0,0]) 9 []) ⟹ [9]
```

```
>
:{
   let f = (\_ -> \_ -> [0,0])
       [x1,x2,x3] = [1,2,3]
       a1 = 9
   in do
     a2 <- f a1 x1
     a3 <- f a2 x2
     f a3 x3
:}
[0,0,0,0,0,0,0,0]
```

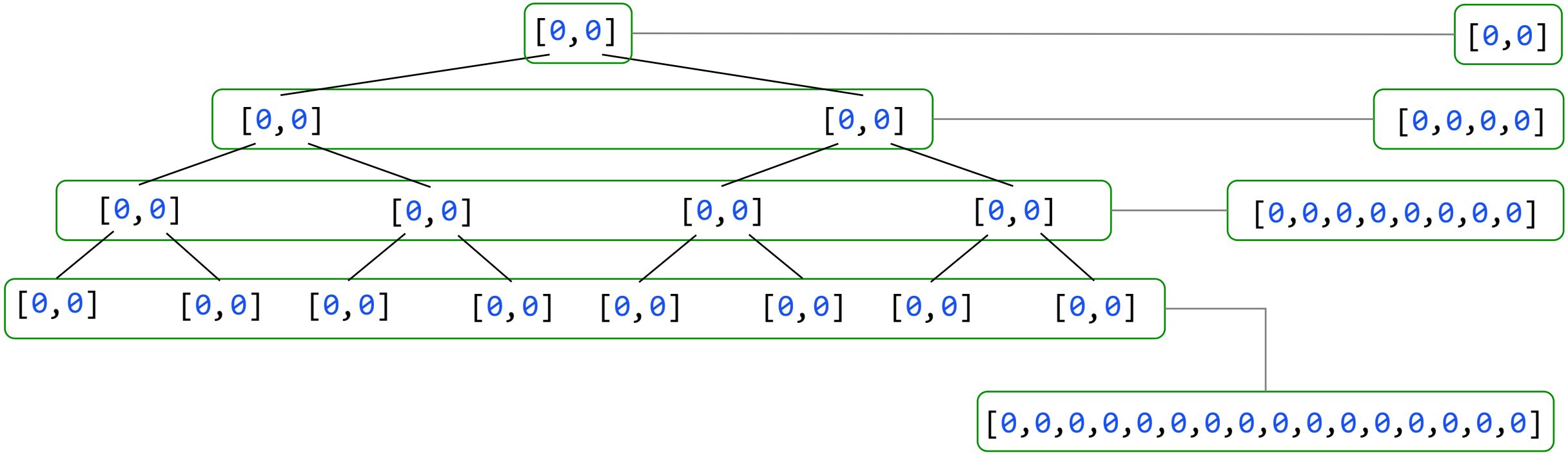
```
foldM f a1 [x1, x2, ..., xm]

==

do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

Invocation	Result	Result length
foldM (_ -> _ -> [0,0]) 9 []	[9]	2^0=1
foldM (_ -> _ -> [0,0]) 9 [1]	[0,0]	2^1=2
foldM (_ -> _ -> [0,0]) 9 [1,2]	[0,0,0,0]	2^2=4
foldM (_ -> _ -> [0,0]) 9 [1,2,3]	[0,0,0,0,0,0,0,0]	2^3=8
foldM (_ -> _ -> [0,0]) 9 [1,2,3,4]	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]	2^4=16

@philip_schwarz

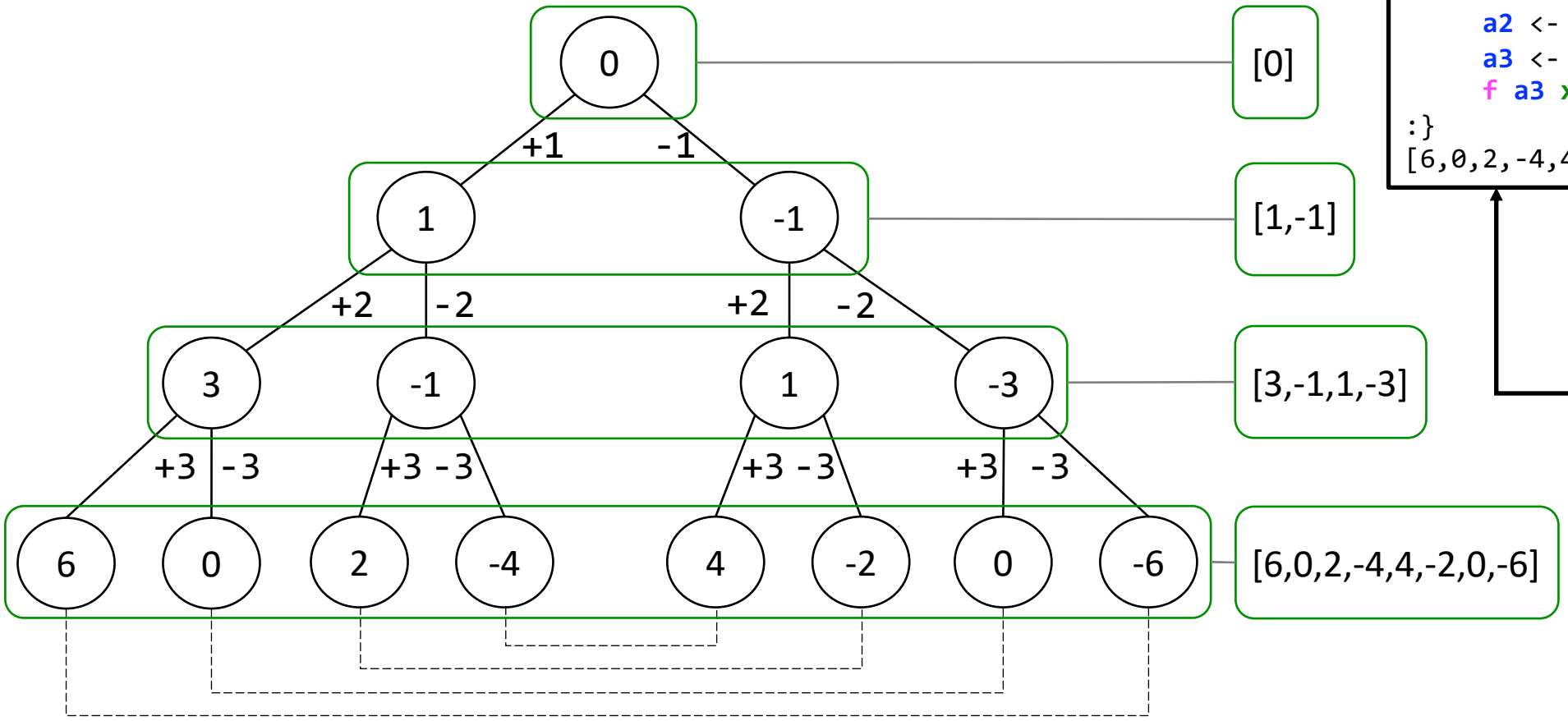
Same example as on the previous slide, but with a diagram that helps us understand how **foldM** builds its result when its list parameter is non-empty.

Invocation	Result	Result length
`foldM (\_ -> \_ -> [0,0]) 9 []`	`[9]`	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	`[0,0]`	2^1=2
`foldM (\_ -> \_ -> [0,0]) 9 [1,2]`	`[0,0,0,0]`	2^2=4
`foldM (\_ -> \_ -> [0,0]) 9 [1,2,3]`	`[0,0,0,0,0,0,0,0]`	2^3=8
`foldM (\_ -> \_ -> [0,0]) 9 [1,2,3,4]`	`[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]`	2^4=16



In this second example we change the **binary function** that we pass to **foldM** so that rather than ignoring its two parameters (the **accumulator** **acc** and the current list element **x**) and always returning the same two-element list `[0,0]`, the function now returns a two-element list containing (a) the result of adding the current element to the **accumulator** and (b) the result of subtracting the current element from the **accumulator**.

Invocation	Result	Result length
`foldM (\acc x -> [acc + x, acc - x]) 0 []`	`[0]`	2^0=1
`foldM (\acc x -> [acc + x, acc - x]) 0 [1]`	`[1,-1]`	2^1=2
`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2]`	`[3,-1,1,-3]`	2^2=4
`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2,3]`	`[6,0,2,-4,4,-2,0,-6]`	2^3=8




```
>
:{
   let f = \acc x -> [acc + x, acc - x]
       [x1,x2,x3] = [1,2,3]
       a1 = 0
   in do
     a2 <- f a1 x1
     a3 <- f a2 x2
     f a3 x3
:}
[6,0,2,-4,4,-2,0,-6]
```

```
foldM f a1 [x1, x2, ..., xm]

==

do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

In this third example we change the **binary function** that we pass to **foldM** so that rather than returning a two-element list containing (a) the result of adding the current element **x** to the **accumulator** and (b) the result of subtracting **x** from the **accumulator**, it returns a two element list containing (a) the result of adding **x** to the front of the **accumulator list** and (b) the **accumulator list**.

As a result, **foldM** computes the **powerset** of its list parameter.

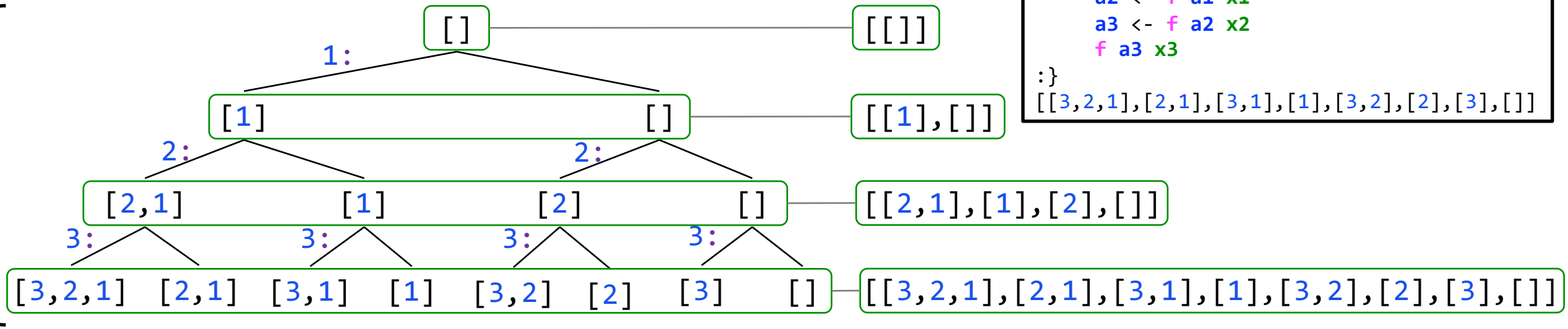
Invocation	Result	Result length
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1
`foldM (\acc x -> [x:acc,acc]) [] [1]`	`[[1],[]]`	2^1=2
`foldM (\acc x -> [x:acc,acc]) [] [1,2]`	`[[2,1],[1],[2],[]]`	2^2=4
`foldM (\acc x -> [x:acc,acc]) [] [1,2,3]`	`[[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]`	2^3=8

```
foldM f a1 [x1, x2, ..., xm]

==

do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

By the way, here on the right is a reminder of how we computed a **powerset** earlier on.

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True,False]) xs

> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

```
>
:{
   let f = \acc x -> [x:acc,acc]
       [x1,x2,x3] = [1,2,3]
       a1 = []
   in do
     a2 <- f a1 x1
     a3 <- f a2 x2
     f a3 x3
:}
[[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]
```



Here are some tests for the three **foldM** examples that we have just gone through.

```haskell
import Control.Monad
```

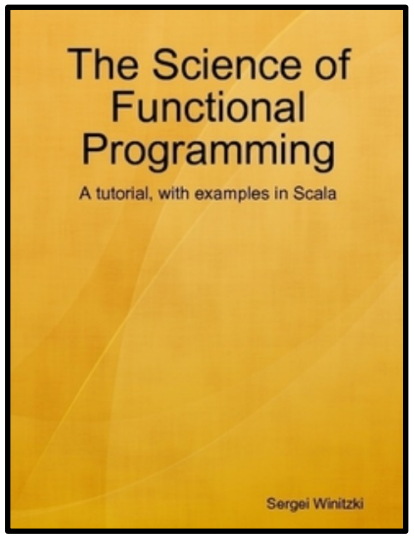
```haskell
assertEqual
  "foldM test 1"
  (foldM (\_ _ -> [0,0]) 9 [1,2,3,4])
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

assertEqual
  "foldM test 2"
  (foldM (\acc x -> [acc+x,acc-x]) 0 [1,2,3])
  [6,0,2,-4,4,-2,0,-6]

assertEqual
  "foldM test 3"
  (foldM (\acc x -> [x:acc,acc])[] [1,2,3])
  [[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]
```

```scala
import cats.syntax.foldable._
```

Cats

```scala
assert(
  List(1,2,3,4).foldM(9)((_,_) => List(0, 0))
  ==
  List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0))

assert(
  List(1,2,3).foldM(0)((acc,x) => List(acc+x, acc-x))
  ==
  List(6,0,2,-4,4,-2,0,-6))

assert(
  List(1,2,3).foldM(List.empty)((acc,x) => List(x::acc,acc))
  ==
  List(List(3,2,1),List(2,1),List(3,1),List(1),List(3,2),List(2),List(3),List()))
```

In the rest of this deck we'll refer to the function passed to **foldM** as an **updater function**. The idea comes from **Sergei Winitzki**, who gives that name to the function passed to **foldLeft**. See the next slide for more details (if you are in a hurry then just see its first three lines and its last two lines).




```
@tailrec def leftFold[A, B](s: Seq[A], b: B, g: (B, A) => B): B =
  if (s.isEmpty) b
  else leftFold(s.tail, g(b, s.head), g)
```

We call this function a "left fold" because it aggregates (or "folds") the sequence starting from the leftmost element.

In this way, we have defined a general method of computing any inductively defined aggregation function on a sequence.

The function `leftFold` implements the logic of aggregation defined via mathematical induction.

Using **`leftFold`**, we can write concise implementations of methods such as **.sum**, **.max**, and many other aggregation functions.

The method **`leftFold`** already contains all the code necessary to set up the **base case** and the **inductive step**.

The programmer just needs to specify the expressions for the **initial value `b`** and for the **updater function `g`**.

Sergei Winitzki
sergei-winitzki-11a6431

@philip_schwarz

Now that we have gained some familiarity with the **foldM** function, let's begin to see how it can be used to solve the **N-Queens combinatorial problem**.

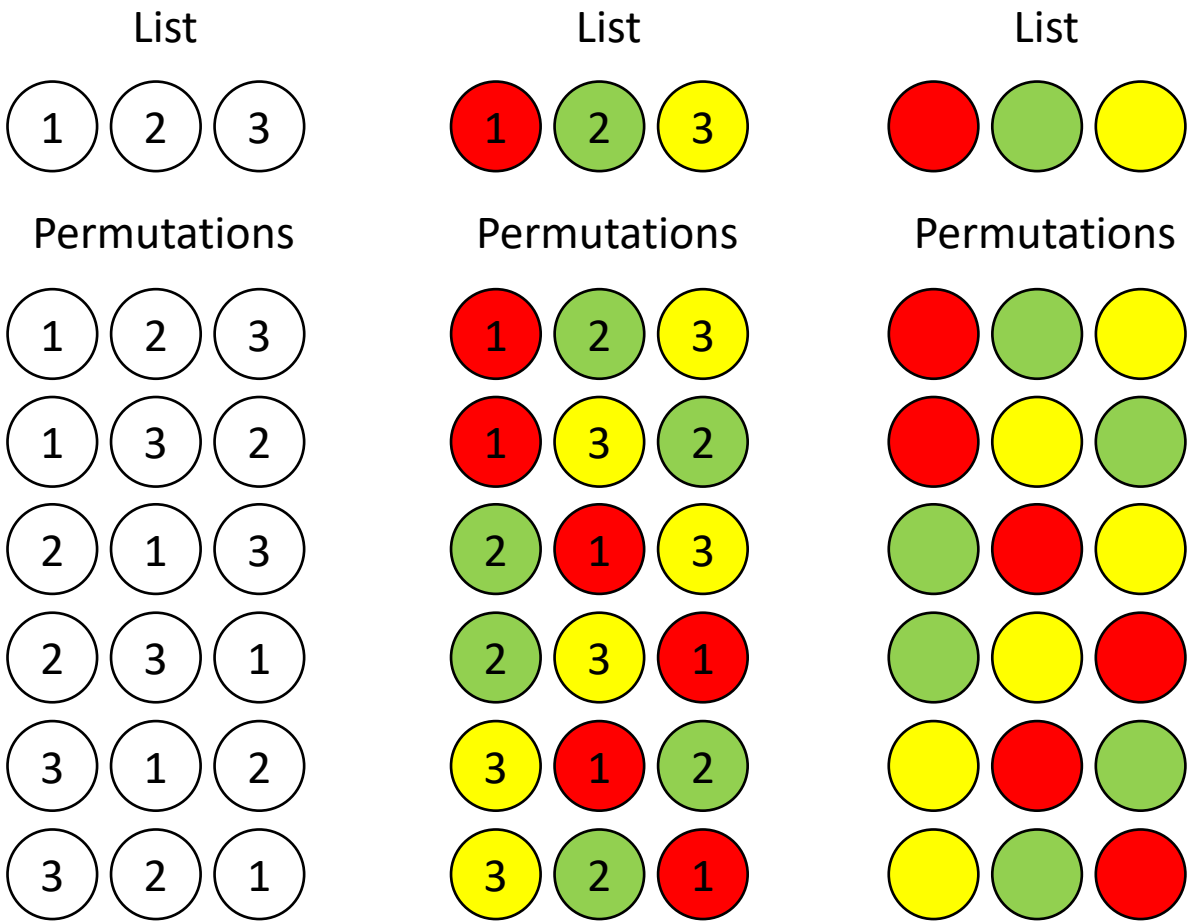
Let's refer to the solution that uses **foldM** as the **folding queens solution**.

The **folding queens solution** needs to generate the **permutations** of a list of integers.

The number of **permutations** of a list of length n is $n!$, the **factorial** of n.

e.g. the number of **permutations** of a list of three elements is

$$3! = 3 * 2 * 1 = 6$$



Let's take a look at the **updater functions** that we have seen so far in our **foldM** usage examples.

#	Invocation	Result
1	`foldM (\_ _ -> [0,0]) 9 [1,2,3]`	`[0,0,0,0,0,0,0,0]`
2	`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2,3]`	`[6,0,2,-4,4,-2,0,-6]`
3	`foldM (\acc x -> [x:acc,acc]) [] [1,2,3]`	`[[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]`

Here are some of the characteristics of the above **updater functions**:

- The first **updater function** ignores both its parameters. It doesn't really manage an **accumulator** and doesn't care about the particular elements that are in the input list. The role of the input list is purely to control the number of iterations, so the only thing that matters is it length.
- The second and third **updater functions** use both of their parameters. They do manage the **accumulator** and they do care about the particular elements that are in the input list, as they affect the final result.
- In all three **updater functions**, the **accumulator** is a single value, i.e. a number or a list.
- Since all three **updater functions** return a two-element list, the length of the list returned by **folding** an input list of length **n** is is **2^n**.

Now let's take a look at an **updater function** that can be used to generate the **permutations** of a list.

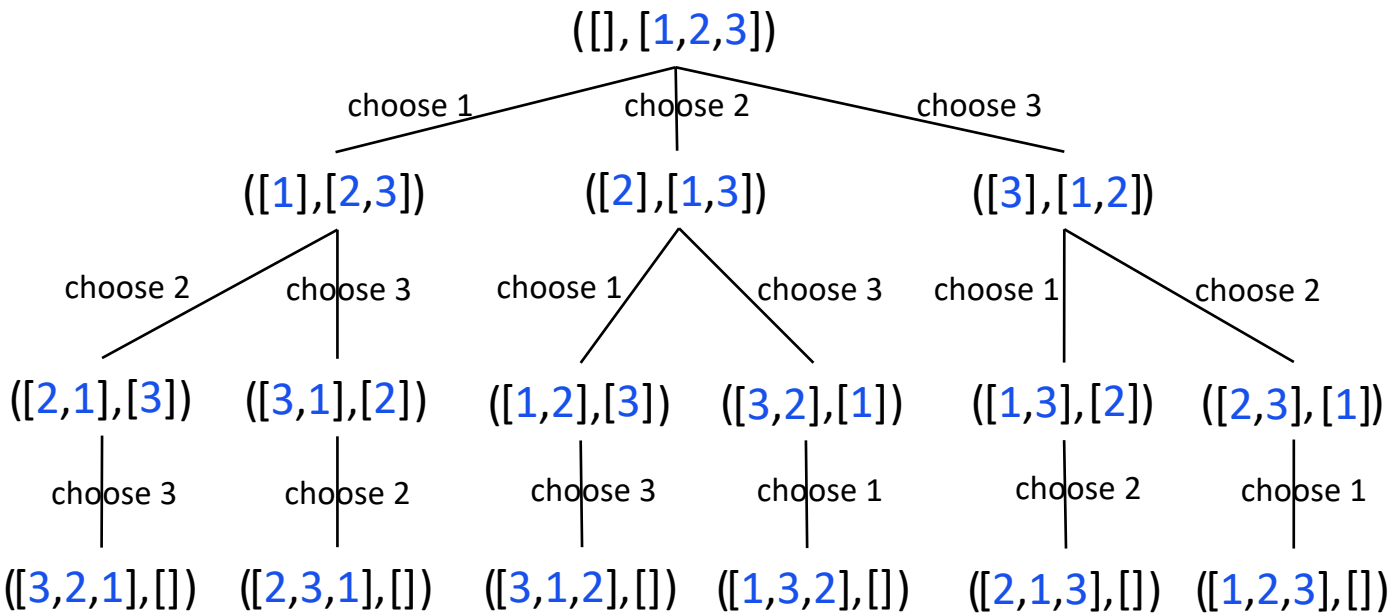
```
update :: Eq a => ([a], [a]) -> p -> [([a], [a])]
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

- While this **updater function** does not ignore its first parameter, it does ignore its second one. While it does manage an **accumulator**, it doesn't care about the particular elements that are in the input list. The role of the input list is purely to control the number of iterations, so the only thing that matters is it length.
- This **updater function** also has a more complex **accumulator** parameter which consists of a pair of values. The first **accumulator** value is a partial **permutation**. The second **accumulator** value is a list of the input list elements that have not yet been chosen (picked) in the creation of the partial **permutation**. Each remaining (not yet chosen) input list elements is chosen in turn to grow the partial **permutation** into a new, more complete partial **permutation** by prefixing it with the chosen element, which is removed from the available choices.
- See below for a sample invocation of the **updater function**. See the next slide for more examples and a diagram clarifying how things work.

Invocation	Result
`foldM update ([],[1,2,3]) [1,2,3]`	`[([3,2,1],[]),([2,3,1],[]),([3,1,2],[]),([1,3,2],[]),([2,1,3],[]),([1,2,3],[])]`

```haskell
update :: Eq a => ([a], [a]) -> p -> [([a], [a])]
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

Invocation	Result
`foldM update ([],[1,2,3]) []`	`[([],[1,2,3])]`
`foldM update ([],[1,2,3]) [1]`	`[([1],[2,3]), ([2],[1,3]), ([3],[1,2])]`
`foldM update ([],[1,2,3]) [1,2]`	`[([2,1],[3]), ([3,1],[2]), ([1,2],[3]), ([3,2],[1]), ([1,3],[2]), ([2,3],[1])]`
`foldM update ([],[1,2,3]) [1,2,3]`	`[([3,2,1],[]), ([2,3,1],[]), ([3,1,2],[]), ([1,3,2],[]), ([2,1,3],[]), ([1,2,3],[])]`



The result of **foldM** is not exactly a list of **permutations**, but rather, a list of a pair of a **permutation** and the empty list.

Let's defined a couple of **helper functions** to make it more convenient to get hold of the desired list of **permutations**.

```haskell
permute :: Eq a => [a] -> [([a], [a])]
permute xs = foldM update ([],xs) xs

permutations :: (Eq a) => [a] -> [[a]]
permutations xs = map fst (permute xs)
```

```
> permutations []
[[]]

> permutations [1]
[[1]]
```

```
> permutations [1,2]
[[2,1],[1,2]]

> permutations [1,2,3]
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
```

```
> permutations [1,2,3,4]
[[4,3,2,1],[3,4,2,1],[4,2,3,1],[2,4,3,1],[3,2,4,1]
,[2,3,4,1],[4,3,1,2],[3,4,1,2],[4,1,3,2],[1,4,3,2]
,[3,1,4,2],[1,3,4,2],[4,2,1,3],[2,4,1,3],[4,1,2,3]
,[1,4,2,3],[2,1,4,3],[1,2,4,3],[3,2,1,4],[2,3,1,4]
,[3,1,2,4],[1,3,2,4],[2,1,3,4],[1,2,3,4]]
```

```
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

We want to use **foldM** to solve the **N-Queens combinatorial problem**, so let's rename the variables of the **update** function to reflect the fact that the **permutations** that we want to generate are the possible lists of positions (columns) of n queens on an $n \times n$ board.

```
oneMoreQueen (queens, emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns]
```

Not all **permutations** are valid though: we need to filter out unsafe **permutations**.

Just like we did in **Part 1**, the way we are going to determine if a **permutation** is safe is by using a **safe** function.

Let's add to **oneMoreQueen** a filter that invokes the **safe** function.

```
oneMoreQueen (queens, emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns, safe queen]
```

The **safe** function that we are going to use this time round is much more concise that the one that we used in **Part 1** (reproduced here on the right).

Since the **safe** function is not the focus of **Part 4**, and since we have already defined one such function in **Part 1**, we are not going to spend any time explaining it, except for saying that it takes as parameter the candidate queen position x and relies both on **queens** (the board to which we want to add the next queen), and on n, the size of the board.

```
safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] queens]
```

```
safe queen queens = all safe (zipWithRows queens) where
  safe (r,c) = c /= col && not (onDiagonal col row c r)
  row = length queens
  col = queen
```

```
onDiagonal row column otherRow otherColumn =
  abs (row - otherRow) == abs (column - otherColumn)
```

```
zipWithRows queens = zip rowNumbers queens
  where
    rowCount = length queens
    rowNumbers = [rowCount-1,rowCount-2..0]
```

Earlier we tried out our **update** function as follows

Invocation	Result
`foldM update ([],[1,2,3]) [1,2,3]`	`[([3,2,1],[]), ([2,3,1],[]), ([3,1,2],[]), ([1,3,2],[]), ([2,1,3],[]), ([1,2,3],[])]`

@philip_schwarz

The **queens** function that we need to implement is this:

```
queens :: Int -> [[Int]]
queens n = ???
```

Given that we have renamed **update** to **oneMoreQueen**, here is how we need to call **foldM**:

```
foldM oneMoreQueen ([],[1..n]) [1..n]
```

We saw earlier that in order to extract the list of **permutations**/queens from the result of **update**/**oneMoreQueen**, we need to map over the result list a function that takes the first element of each pair in the list.

So here is how we implement **queens**:

```
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n])
```

On the next slide we add **oneMoreQueen** and **safe** to **queens** and see the final result.

Here is the **queens** function that we have been building up to.

```
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n])  where
  oneMoreQueen (queens, emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns, safe queen]  where
    safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] queens]
```

```
> queens 8
[[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],[3,5,2,8,6,4,7,1],[3,6,4,2,8,5,7,1],[5,7,1,3,8,6,4,2]
,[4,6,8,3,1,7,5,2],[3,6,8,1,4,7,5,2],[5,3,8,4,7,1,6,2],[5,7,4,1,3,8,6,2],[4,1,5,8,6,3,7,2]
,[3,6,4,1,8,5,7,2],[4,7,5,3,1,6,8,2],[6,4,2,8,5,7,1,3],[6,4,7,1,8,2,5,3],[1,7,4,6,8,2,5,3]
,[6,8,2,4,1,7,5,3],[6,2,7,1,4,8,5,3],[4,7,1,8,5,2,6,3],[5,8,4,1,7,2,6,3],[4,8,1,5,7,2,6,3]
,[2,7,5,8,1,4,6,3],[1,7,5,8,2,4,6,3],[2,5,7,4,1,8,6,3],[4,2,7,5,1,8,6,3],[5,7,1,4,2,8,6,3]
,[6,4,1,5,8,2,7,3],[5,1,4,6,8,2,7,3],[5,2,6,1,7,4,8,3],[6,3,7,2,8,5,1,4],[2,7,3,6,8,5,1,4]
,[7,3,1,6,8,5,2,4],[5,1,8,6,3,7,2,4],[1,5,8,6,3,7,2,4],[3,6,8,1,5,7,2,4],[6,3,1,7,5,8,2,4]
,[7,5,3,1,6,8,2,4],[7,3,8,2,5,1,6,4],[5,3,1,7,2,8,6,4],[2,5,7,1,3,8,6,4],[3,6,2,5,8,1,7,4]
,[6,1,5,2,8,3,7,4],[8,3,1,6,2,5,7,4],[2,8,6,1,3,5,7,4],[5,7,2,6,3,1,8,4],[3,6,2,7,5,1,8,4]
,[6,2,7,1,3,5,8,4],[3,7,2,8,6,4,1,5],[6,3,7,2,4,8,1,5],[4,2,7,3,6,8,1,5],[7,1,3,8,6,4,2,5]
,[1,6,8,3,7,4,2,5],[3,8,4,7,1,6,2,5],[6,3,7,4,1,8,2,5],[7,4,2,8,6,1,3,5],[4,6,8,2,7,1,3,5]
,[2,6,1,7,4,8,3,5],[2,4,6,8,3,1,7,5],[3,6,8,2,4,1,7,5],[6,3,1,8,4,2,7,5],[8,4,1,3,6,2,7,5]
,[4,8,1,3,6,2,7,5],[2,6,8,3,1,4,7,5],[7,2,6,3,1,4,8,5],[3,6,2,7,1,4,8,5],[4,7,3,8,2,5,1,6]
,[4,8,5,3,1,7,2,6],[3,5,8,4,1,7,2,6],[4,2,8,5,7,1,3,6],[5,7,2,4,8,1,3,6],[7,4,2,5,8,1,3,6]
,[8,2,4,1,7,5,3,6],[7,2,4,1,8,5,3,6],[5,1,8,4,2,7,3,6],[4,1,5,8,2,7,3,6],[5,2,8,1,4,7,3,6]
,[3,7,2,8,5,1,4,6],[3,1,7,5,8,2,4,6],[8,2,5,3,1,7,4,6],[3,5,2,8,1,7,4,6],[3,5,7,1,4,2,8,6]
,[5,2,4,6,8,3,1,7],[6,3,5,8,1,4,2,7],[5,8,4,1,3,6,2,7],[4,2,5,8,6,1,3,7],[4,6,1,5,2,8,3,7]
,[6,3,1,8,5,2,4,7],[5,3,1,6,8,2,4,7],[4,2,8,6,1,3,5,7],[6,3,5,7,1,4,2,8],[6,4,7,1,3,5,2,8]
,[4,7,5,2,6,1,3,8],[5,7,2,6,3,1,4,8]]

> length (queens 8)
92
```

On the next slide we compare the above function with how it looks on the **Rosetta Code** site where it originates from.

```haskell
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n])  where
  oneMoreQueen (safeQueens,emptyColumns) _ = [(queen:safeQueens, delete queen emptyColumns) | queen <- emptyColumns, safe queen]
    where safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] safeQueens]
```

https://rosettacode.org/wiki/N-queens_problem

```haskell
-- given n, "queens n" solves the n-queens problem, returning a list of all the
-- safe arrangements. each solution is a list of the columns where the queens are
-- located for each row
queens :: Int -> [[Int]]
queens n = map fst $ foldM oneMoreQueen ([],[1..n]) [1..n]  where

  -- foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
  -- foldM folds (from left to right) in the list monad, which is convenient for
  -- "nondeterminstically" finding "all possible solutions" of something. the
  -- initial value [] corresponds to the only safe arrangement of queens in 0 rows

  -- given a safe arrangement y of queens in the first i rows, and a list of
  -- possible choices, "oneMoreQueen y _" returns a list of all the safe
  -- arrangements of queens in the first (i+1) rows along with remaining choices
  oneMoreQueen (y,d) _ = [(x:y, delete x d) | x <- d, safe x]  where

    -- "safe x" tests whether a queen at column x is safe from previous queens
    safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] y]
```

Iterative Solution

Understanding the code on the left is straightforward now that we have a firm understanding of how **foldM** works, although renaming **x**, **y** and **d** the way we did above eases comprehension imho.

Note that while the **recursive** implementation of **queens** from **Part 1** (shown below), blindly tries to place the next queen in every column 1..n, the implementation of **queens** that uses **foldM** only tries to place the queen in columns which are known not to be already occupied.

```haskell
queens n = placeQueens n where
  placeQueens 0 = [[]]
  placeQueens k = [queen:queens | queens <- placeQueens(k-1),
                                  queen <- [1..n],
                                  safe queen queens]
```

```haskell
safe queen queens = all safe (zipWithRows queens)
  where
    safe (r,c) = c /= col && not (onDiagonal col row c r)
    row = length queens
    col = queen
```

```haskell
onDiagonal row column otherRow otherColumn =
  abs (row - otherRow)
  ==
  abs (column - otherColumn)
```

```haskell
zipWithRows queens = zip rowNumbers queens
  where
    rowCount = length queens
    rowNumbers = [rowCount-1,rowCount-2..0]
```

Recursive Solution

Let's translate our **foldM**-based **N-Queens** program from **Haskell** to **Scala** with **Cats**.

See the next slide for the same code but spread over several lines, to aid comprehension.

```haskell
import Control.Monad
```

```haskell
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n])  where
  oneMoreQueen (safeQueens,emptyColumns) _ = [(queen:safeQueens, delete queen emptyColumns) | queen <- emptyColumns, safe queen]
    where safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] safeQueens]
```

```scala
import cats.syntax.foldable._
```


```scala
def queens(n: Int): List[List[Int]] =
  def oneMoreQueen(acc:(List[Int],List[Int]),x:Int): List[(List[Int],List[Int])] = acc match { case (queens, emptyColumns) =>
    def safe(x:Int): Boolean = { for (c,n) <- queens zip (1 to n) yield x != c + n && x != c - n } forall identity
    for queen <- emptyColumns if safe(queen) yield (queen::queens, emptyColumns diff List(queen)) }
  List.range(1, n + 1).foldM(Nil, List.range(1, n + 1))(oneMoreQueen) map (_.head)
```

```haskell
import Control.Monad
```

```haskell
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n]) where
  oneMoreQueen (safeQueens,emptyColumns) _ =
    [(queen:safeQueens, delete queen emptyColumns) |
      queen <- emptyColumns,
      safe queen]
    where safe x = and [x /= c + n && x /= c - n |
                        (n,c) <- zip [1..] safeQueens]
```

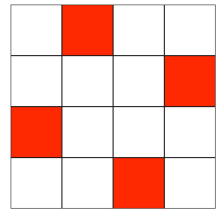
```scala
import cats.syntax.foldable._
```

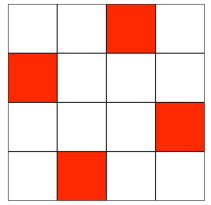

```scala
def queens(n: Int): List[List[Int]] =
  def oneMoreQueen(acc:(List[Int],List[Int]),x:Int): List[(List[Int],List[Int])] =
    acc match { case (queens, emptyColumns) =>
      def safe(x:Int): Boolean = {
        for (c,n) <- queens zip (1 to n)
        yield x != c + n && x != c - n
      } forall identity
      for
        queen <- emptyColumns
        if safe(queen)
      yield (queen::queens, emptyColumns diff List(queen)) }
  List.range(1, n + 1).foldM(Nil, List.range(1, n + 1))(oneMoreQueen) map (_.head)
```

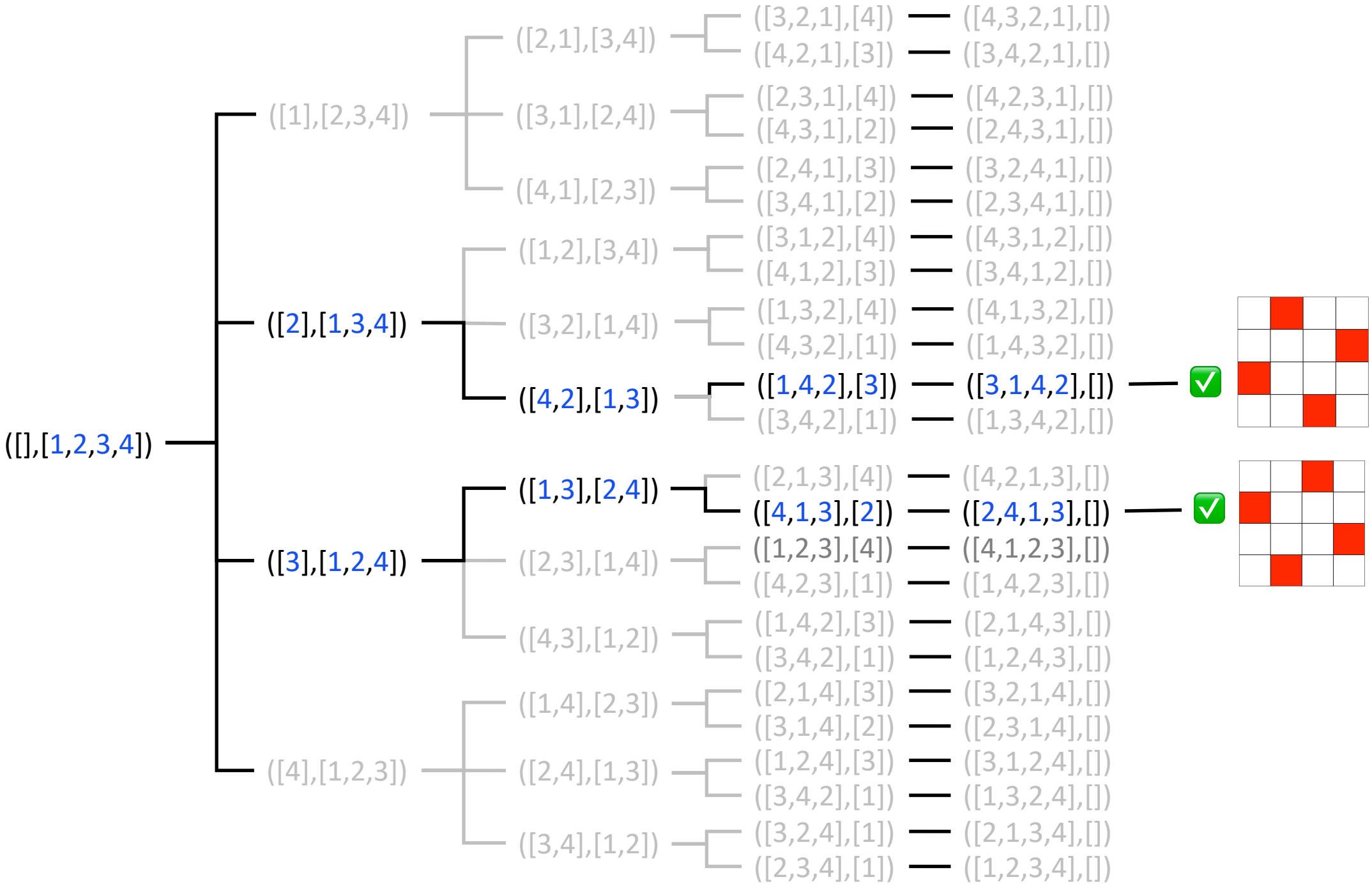
The leafs of the tree on the right represent all the possible permutation of 4 queens on a 4 x 4 board.

The only two solutions to the **4-Queens Problem** are marked by a tick box and accompanied by an image of the corresponding board.

The next slide shows the same tree after greying out those portions that are not part of the two success paths.






([],[1,2,3,4])
([1],[2,3,4])
([2],[1,3,4])
([3],[1,2,4])
([4],[1,2,3])
([2,1],[3,4])
([3,1],[2,4])
([4,1],[2,3])
([1,2],[3,4])
([3,2],[1,4])
([4,2],[1,3])
([1,3],[2,4])
([2,3],[1,4])
([4,3],[1,2])
([1,4],[2,3])
([2,4],[1,3])
([3,4],[1,2])
([3,2,1],[4])
([4,3,2,1],[])
([4,2,1],[3])
([3,4,2,1],[])
([2,3,1],[4])
([4,2,3,1],[])
([4,3,1],[2])
([2,4,3,1],[])
([2,4,1],[3])
([3,2,4,1],[])
([3,4,1],[2])
([2,3,4,1],[])
([3,1,2],[4])
([4,3,1,2],[])
([4,1,2],[3])
([3,4,1,2],[])
([1,3,2],[4])
([4,1,3,2],[])
([4,3,2],[1])
([1,4,3,2],[])
([1,4,2],[3])
([3,1,4,2],[])
([3,4,2],[1])
([1,3,4,2],[])
([2,1,3],[4])
([4,2,1,3],[])
([4,1,3],[2])
([2,4,1,3],[])
([1,2,3],[4])
([4,1,2,3],[])
([4,2,3],[1])
([1,4,2,3],[])
([1,4,2],[3])
([2,1,4,3],[])
([3,4,2],[1])
([1,2,4,3],[])
([2,1,4],[3])
([3,2,1,4],[])
([3,1,4],[2])
([2,3,1,4],[])
([1,2,4],[3])
([3,1,2,4],[])
([3,4,2],[1])
([1,3,2,4],[])
([3,2,4],[1])
([2,1,3,4],[])
([2,3,4],[1])
([1,2,3,4],[])

@philip_schwarz

To conclude this slide deck, let's create a tree which, rather than being greyed out in all its portions that lead to a **failed permutation**, it is greyed out only in **subtrees** whose **root** represents the **failed attempt to add an unsafe queen to a board**.

To help us work out which portions to grey out, we are going to take the **accumulator** parameter of the **Scala oneMoreQueen** function and change its type from

```
List[Int]
```

to

```
Writer[List[String],List[Int]]
```

so that we can get **oneMoreQueen** to log events informing us **when adding a queen to a board is safe and when it isn't**.

```scala
def queens(n: Int): List[List[Int]] =
  def oneMoreQueen(acc:(List[Int],List[Int]),x:Int): List[(List[Int],List[Int])] =
    acc match { case (queens, emptyColumns) =>
      def safe(x:Int): Boolean = {
        for (c,n) <- queens zip (1 to n)
        yield x != c + n && x != c - n
      } forall identity
      for
        queen <- emptyColumns
        if safe(queen)
      yield (queen::queens, emptyColumns diff List(queen)) }
  List.range(1, n + 1).foldM(Nil, List.range(1, n + 1))(oneMoreQueen) map (_.head)
```


```scala
def queens(n: Int): List[Writer[List[String],List[Int]]] =
  def oneMoreQueen(acc:(Writer[List[String],List[Int]],List[Int]),x:Int): List[(Writer[List[String],List[Int]],List[Int])] =
    acc match { case (queens, emptyColumns) =>
        def safe(x:Int): Boolean = {
          for (c,n) <- queens.value zip (1 to n)
          yield x != c + n && x != c - n
        } forall identity
        for
          queen <- emptyColumns
          newQueens = queens.tell(List(s"\n$queen is ${if safe(queen) then "safe" else "unsafe" } for ${queens.value}"))
          if safe(queen)
        yield (newQueens map (queen::_), emptyColumns diff List(queen)) }
    List.range(1, n + 1).foldM(Writer(List.empty[String],Nil), List.range(1, n + 1))(oneMoreQueen) map (_.head)
```

As it stands, the modified **oneMoreQueen** function is not very useful because **queens** only returns boards that are **solutions**, and so the logged events that accompany the solution boards simply tell us that all the queens added to the **solution boards** were **safe to add**.

```
assert(
  queens(4).map(_.value)
  ==
  List(
    List(3, 1, 4, 2),
    List(2, 4, 1, 3)))


queens(4) map (_.listen.value) foreach println

(List(3, 1, 4, 2),List(
2 is safe for List(),
4 is safe for List(2),
1 is safe for List(4, 2),
3 is safe for List(1, 4, 2)))
(List(2, 4, 1, 3),List(
3 is safe for List(),
1 is safe for List(3),
4 is safe for List(1, 3),
2 is safe for List(4, 1, 3)))
```

In order to get **oneMoreQueens** to log information about **queens that are unsafe to add to a board**, we are going to comment out the **guard** (constraint) that causes **oneMoreQueens** to **backtrack** whenever an attempt is made to **add an unsafe queen**.

```
  if safe(queen)
```

The next slide shows the results that are returned by the **queens** function after doing that (with some spacing added to aid comprehension).

(List(4, 3, 2, 1),List(
1 is safe for List(),
2 is unsafe for List(1),
3 is unsafe for List(2, 1),
4 is unsafe for List(3, 2, 1)))

(List(3, 4, 1, 2),List(
2 is safe for List(),
1 is unsafe for List(2),
4 is unsafe for List(1, 2),
3 is unsafe for List(4, 1, 2)))

(List(4, 1, 2, 3),List(
3 is safe for List(),
2 is unsafe for List(3),
1 is unsafe for List(2, 3),
4 is unsafe for List(1, 2, 3)))

(List(1, 3, 2, 4),List(
4 is safe for List(),
2 is safe for List(4),
3 is unsafe for List(2, 4),
1 is unsafe for List(3, 2, 4)))

(List(3, 4, 2, 1),List(
1 is safe for List(),
2 is unsafe for List(1),
4 is safe for List(2, 1),
3 is unsafe for List(4, 2, 1)))

(List(4, 1, 3, 2),List(
2 is safe for List(),
3 is unsafe for List(2),
1 is safe for List(3, 2),
4 is safe for List(1, 3, 2)))

(List(1, 4, 2, 3),List(
3 is safe for List(),
2 is unsafe for List(3),
4 is safe for List(2, 3),
1 is safe for List(4, 2, 3)))

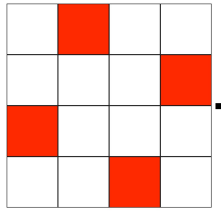
(List(2, 1, 3, 4),List(
4 is safe for List(),
3 is unsafe for List(4),
1 is safe for List(3, 4),
2 is unsafe for List(1, 3, 4)))

(List(4, 2, 3, 1),List(
1 is safe for List(),
3 is safe for List(1),
2 is unsafe for List(3, 1),
4 is unsafe for List(2, 3, 1)))

(List(1, 4, 3, 2),List(
2 is safe for List(),
3 is unsafe for List(2),
4 is unsafe for List(3, 2),
1 is unsafe for List(4, 3, 2)))

(List(2, 1, 4, 3),List(
3 is safe for List(),
4 is unsafe for List(3),
1 is unsafe for List(4, 3),
2 is unsafe for List(1, 4, 3)))

(List(1, 2, 3, 4),List(
4 is safe for List(),
3 is unsafe for List(4),
2 is unsafe for List(3, 4),
1 is unsafe for List(2, 3, 4)))



(List(2, 4, 3, 1),List(
1 is safe for List(),
3 is safe for List(1),
4 is unsafe for List(3, 1),
2 is safe for List(4, 3, 1)))

(List(3, 1, 4, 2),List(
2 is safe for List(),
4 is safe for List(2),
1 is safe for List(4, 2),
3 is safe for List(1, 4, 2)))

(List(1, 2, 4, 3),List(
3 is safe for List(),
4 is unsafe for List(3),
2 is safe for List(4, 3),
1 is unsafe for List(2, 4, 3)))

(List(3, 2, 4, 1),List(
1 is safe for List(),
4 is safe for List(1),
2 is safe for List(4, 1),
3 is unsafe for List(2, 4, 1)))

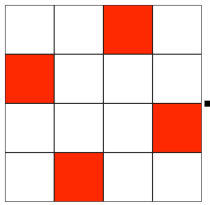
(List(1, 3, 4, 2),List(
2 is safe for List(),
4 is safe for List(2),
3 is unsafe for List(4, 2),
1 is safe for List(3, 4, 2)))

(List(3, 2, 1, 4),List(
4 is safe for List(),
1 is safe for List(4),
2 is unsafe for List(1, 4),
3 is unsafe for List(2, 1, 4)))

(List(2, 3, 4, 1),List(
1 is safe for List(),
4 is safe for List(1),
3 is unsafe for List(4, 1),
2 is unsafe for List(3, 4, 1)))

(List(4, 2, 1, 3),List(
3 is safe for List(),
1 is safe for List(3),
2 is unsafe for List(1, 3),
4 is safe for List(2, 1, 3)))

(List(2, 3, 1, 4),List(
4 is safe for List(),
1 is safe for List(4),
3 is safe for List(1, 4),
2 is unsafe for List(3, 1, 4)))



(List(4, 3, 1, 2),List(
2 is safe for List(),
1 is unsafe for List(2),
3 is safe for List(1, 2),
4 is unsafe for List(3, 1, 2)))

(List(2, 4, 1, 3),List(
3 is safe for List(),
1 is safe for List(3),
4 is safe for List(1, 3),
2 is safe for List(4, 1, 3)))

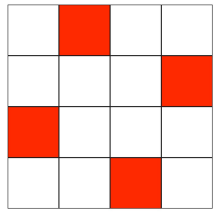
(List(3, 1, 2, 4),List(
4 is safe for List(),
2 is safe for List(4),
1 is unsafe for List(2, 4),
3 is safe for List(1, 2, 4)))

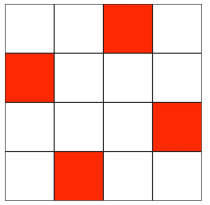
Armed with the information in those logging events, we are now able to update our tree so that **unsafe queens** are highlighted in red and the **unfruitful paths** associated with to those **unsafe queen additions** (paths normally ignored by the **queens** function) are greyed out.

@philip_schwarz

([],[1,2,3,4])

- ([1],[2,3,4])
  - ([2,1],[3,4])
    - ([3,2,1],[4]) — ([4,3,2,1],[])
    - ([4,2,1],[3]) — ([3,4,2,1],[])
  - ([3,1],[2,4])
    - ([2,3,1],[4]) — ([4,2,3,1],[])
    - ([4,3,1],[2]) — ([2,4,3,1],[])
  - ([4,1],[2,3])
    - ([2,4,1],[3]) — ([3,2,4,1],[])
    - ([3,4,1],[2]) — ([2,3,4,1],[])
- ([2],[1,3,4])
  - ([1,2],[3,4])
    - ([3,1,2],[4]) — ([4,3,1,2],[])
    - ([4,1,2],[3]) — ([3,4,1,2],[])
  - ([3,2],[1,4])
    - ([1,3,2],[4]) — ([4,1,3,2],[])
    - ([4,3,2],[1]) — ([1,4,3,2],[])
  - ([4,2],[1,3])
    - ([1,4,2],[3]) — ([3,1,4,2],[]) — ✅
    - ([3,4,2],[1]) — ([1,3,4,2],[])
- ([3],[1,2,4])
  - ([1,3],[2,4])
    - ([2,1,3],[4]) — ([4,2,1,3],[])
    - ([4,1,3],[2]) — ([2,4,1,3],[]) — ✅
  - ([2,3],[1,4])
    - ([1,2,3],[4]) — ([4,1,2,3],[])
    - ([4,2,3],[1]) — ([1,4,2,3],[])
  - ([4,3],[1,2])
    - ([1,4,2],[3]) — ([2,1,4,3],[])
    - ([3,4,2],[1]) — ([1,2,4,3],[])
- ([4],[1,2,3])
  - ([1,4],[2,3])
    - ([2,1,4],[3]) — ([3,2,1,4],[])
    - ([3,1,4],[2]) — ([2,3,1,4],[])
  - ([2,4],[1,3])
    - ([1,2,4],[3]) — ([3,1,2,4],[])
    - ([3,4,2],[1]) — ([1,3,2,4],[])
  - ([3,4],[1,2])
    - ([3,2,4],[1]) — ([2,1,3,4],[])
    - ([2,3,4],[1]) — ([1,2,3,4],[])





That’s all for **Part 4**.

I hope you found it useful.

See you in **Part 5**.