N-Queens Combinatorial Problem

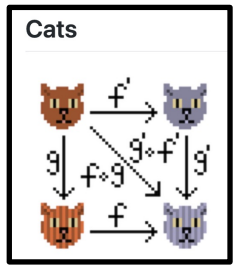
Polyglot FP for Fun and Profit – Haskell and Scala

see how the guard function has migrated from MonadPlus to Alternative and learn something about the latter

learn how to write a Scala program that draws an N-Queens solution board using the Doodle graphics library

see how to write the equivalent Haskell program using the Gloss graphics library

learn how to use Monoid and Foldable to compose images both in Haskell and in Scala



Scala Doodle

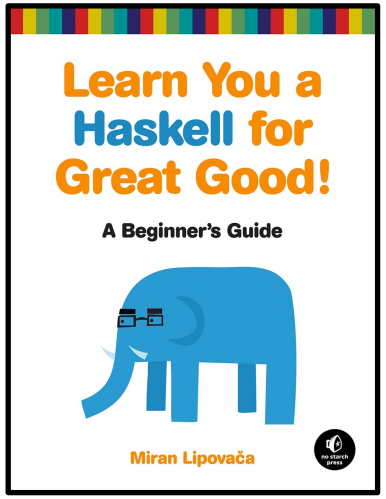
Part 2

with excerpts from

Haskell Gloss

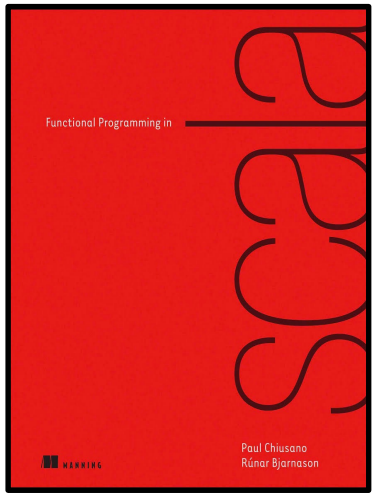
Cats Effect

Miran Lipovača



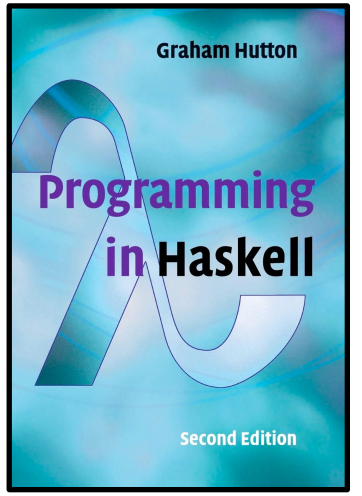
Runar Bjarnason

@runarorama



Paul Chiusano

@pchiusano



Graham Hutton

@haskellhutt

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

@philip_schwarz

After I finished working on Part 1, I realised that since **Miran Lipovača** wrote his book, the definition of the **guard** function has changed a bit.

For the purposes of this series, the behaviour of the function is unchanged, but it can be interesting to know what did change.

If you are not interested, then skip the next 14 slides.

The next slide reminds us of how **Miran Lipovača** explained that the **guard** function is defined for **MonadPlus** instances. In fact, due to some odd omission on my part, Part 1 says that the type of the **guard** function is this

```
Bool -> m ()
```

whereas the correct type includes a constraint, i.e. it is this

```
(MonadPlus m) => Bool -> m ()
```

That omission is another good reason for having the next slide.

MonadPlus and the guard Function

…

The `MonadPlus` type class is for **monads** that can also act as **monoids**. Here is its definition:

```
class Monad m => MonadPlus m where
  mzero :: ma
  mplus :: ma -> ma -> ma
```

`mzero` is synonymous with `mempty` from the `Monoid` type class, and `mplus` corresponds to `mappend`. Because lists are **monoids** as well as **monads**, they can be made an instance of this type class:

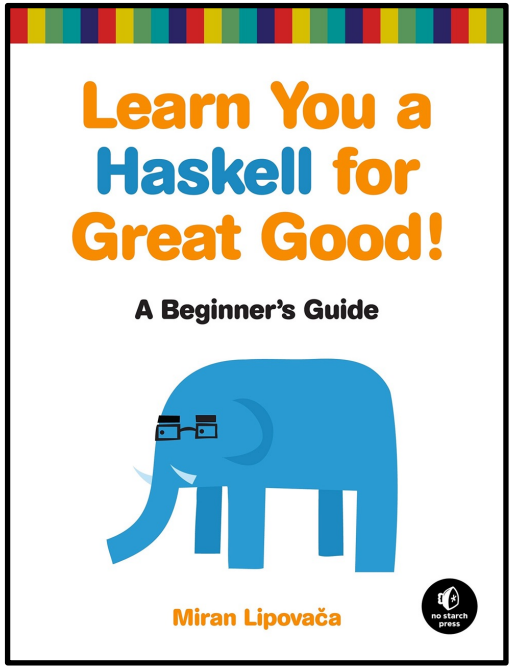
```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

For lists, `mzero` represents a nondeterministic computation that has no results at all – a failed computation. `mplus` joins two nondeterministic values into one. The `guard` function is defined like this:

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

`guard` takes a `Boolean` value. If that value is `True`, `guard` takes a `()` and puts it in a minimal default context that succeeds. If the `Boolean` value is `False`, `guard` makes a failed monadic value. Here it is in action:

```
ghci> guard (5 > 2) :: Maybe ()
Just ()

ghci> guard (1 > 2) :: Maybe ()
Nothing

ghci> guard (5 > 2) :: [()]
[()]

ghci> guard (1 > 2) :: [()]
[]
```



Miran Lipovača

It looks like currently it is not `MonadPlus` instances that provide a `guard` function, but `Alternative` instances.

Conditional execution of monadic expressions

```
guard :: Alternative f => Bool -> f ()
```

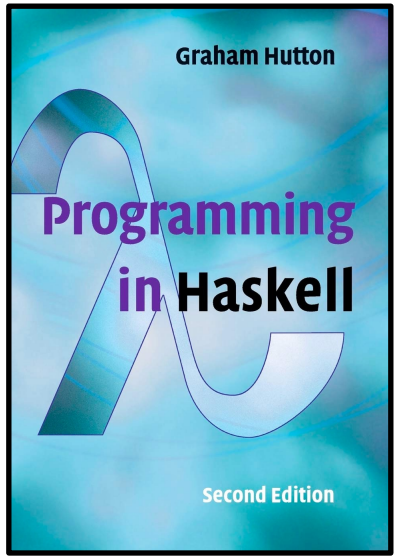
> Conditional failure of `Alternative` computations. Defined by
>
> ```
> guard True  = pure ()
> guard False = empty
> ```

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

See the next slide for the first part of **Graham Hutton**'s introduction to the `Alternative` type class.

Graham Hutton

@haskellhutt



13.5 Making choices

…

Making a choice between two alternatives isn't specific to parsers but can be generalised to a range of **applicative types**. This concept is captured by the following class declaration in the library `Control.Applicative`:

```
class Applicative f => Alternative f where
empty :: f a
(<|>) :: f a -> f a -> f a
```

That is, **for an applicative functor to be an instance of the `Alternative` class, it must support `empty` and `<|>` primitives of the specified types.** (The class also provides two further primitives, which will be discussed in the next section.) The intuition is that **`empty` represents an alternative that has failed, and `<|>` is an appropriate choice operator for the type. The two primitives are also required to satisfy the following identity and associativity laws**:

```
empty <|> x       = x
x <|> empty       = x
x <|> (y <|> z) = (x <|> y) <|> z
```

The motivating example of an `Alternative` type is the `Maybe` type, for which `empty` is given by the failure value `Nothing`, and `<|>` returns its first argument if this succeeds, and its second argument otherwise:

```
instance Alternative Maybe where
-- empty :: Maybe a
empty = Nothing
-- (<|>) :: Maybe a -> Maybe a -> Maybe a
Nothing <|> my = my
(Just x) <|> _ = Just x
```

Another instance of **Alternative** is the list. Here are examples of using both the **Maybe Alternative** and the list **Alternative**.

```haskell
instance Alternative Maybe where
  -- empty :: Maybe a
  empty = Nothing

  -- (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> my = my
  (Just x) <|> _ = Just x
```

```haskell
instance Alternative [] where
  -- empty :: [a]
  empty = []

  -- (<|>) :: [a] -> [a] -> [a]
  (<|>) = (++)
```

```
Haskell> Just 3 <|> Just 4
Just 3

Haskell> Nothing <|> Just 4
Just 4

Haskell> Just 3 <|> Nothing
Just 3

Haskell> Nothing <|> Nothing
Nothing

Haskell> empty <|> Just 4
Just 4

Haskell> Just 3 <|> empty
Just 3

Haskell> (empty::Maybe Int) <|> empty
Nothing
```

```
Haskell> [1,2,3] <|> [4,5,6]
[1,2,3,4,5,6]

Haskell> [] <|> [4,5,6]
[4,5,6]

Haskell> [1,2,3] <|> []
[1,2,3]

Haskell> [] <|> []
[]

Haskell> empty <|> [4,5,6]
[4,5,6]

Haskell> [1,2,3] <|> empty
[1,2,3]

Haskell> (empty::[Int]) <|> empty
[]
```

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

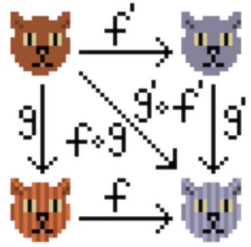
We have seen how **Haskell**'s **Alternative** is an **Applicative** that supports **empty** and **<|>**.

What about in **Scala**?

In the **Cats** library, **Alternative**'s **empty** function is provided by **MonoidK**, whose supertrait **SemigroupK** provides **combineK**, which is the equivalent of **Haskell**'s **<|>**, but whose operator alias is called **<+>**.

See next slide for usage examples of the **Alternative** instances for **List** and **Option**.

Cats



Alternative

Alternative extends `Applicative` with a `MonoidK`. Let's stub out all the operations just to remind ourselves what that gets us.

```scala
import cats.{Applicative, MonoidK}

trait Alternative[F[_]] extends Applicative[F] with MonoidK[F] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

  def pure[A](a: A): F[A]

  def empty[A]: F[A]

  def combineK[A](x: F[A], y: F[A]): F[A]
}
```

As you might recall, `pure` wraps values in the context; `ap` allows us to do calculations in the context; `combineK` allows us to combine, for any given type `A`, any two contextual values `F[A]`; and `empty` provides the identity element for the combine operation.

https://typelevel.org/cats/typeclasses/alternative.html

```
Haskell> empty <|> Just 3
Just 3

Haskell> Nothing <|> Just 3
Just 3

Haskell> Just 3 <|> Just 4
Just 3

Haskell> Just 3 <|> Nothing
Just 3
```

```
Haskell> empty <|> [4,5,6]
[4,5,6]

Haskell> [] <|> [4,5,6]
[4,5,6]

Haskell> [1,2,3] <|> [4,5,6]
[1,2,3,4,5,6]

Haskell> [1,2,3] <|> []
[1,2,3]
```

```
scala> val alt = Alternative[Option]
val alt: cats.Alternative[Option] = …

scala> alt.combineK(alt.empty[Int], 4.some)
val res0: Option[Int] = Some(4)

scala> alt.combineK(3.some, 4.some)
val res1: Option[Int] = Some(3)

scala> alt.combineK(3.some, None)
val res2: Option[Int] = Some(3)
```

```
scala> val alt = Alternative[List]
val alt: cats.Alternative[List] = …

scala> alt.combineK(alt.empty[Int], List(4,5,6))
val res0: List[Int] = List(4, 5, 6)

scala> alt.combineK(List(1,2,3), List(4,5,6))
val res1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> alt.combineK(List(1,2,3), List())
val res2: List[Int] = List(1, 2, 3)
```

```
scala> none <+> 3.some
val res0: Option[Int] = Some(3)

scala> 3.some <+> 4.some
val res1: Option[Int] = Some(3)

scala> 3.some <+> None
val res2: Option[Int] = Some(3)
```

```
scala> List() <+> List(4,5,6)
val res0: List[Int] = List(4, 5, 6)

scala> List(1,2,3) <+> List(4,5,6)
val res1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> List(1,2,3) <+> List()
val res2: List[Int] = List(1, 2, 3)
```

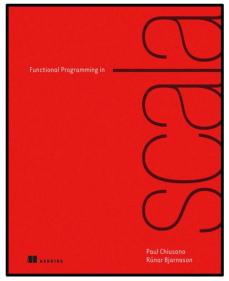
If you are asking yourself what's with the **K** in **MonoidK**, **SemigroupK** and **combineK**, see the first two slides of the following slide deck.

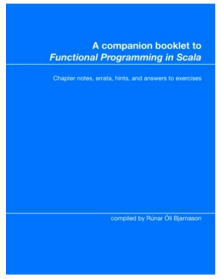
@philip_schwarz

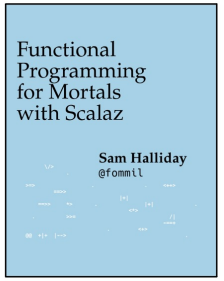
Monoids

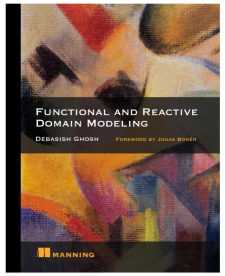
with examples using Scalaz and Cats

Part II - based on









slides by  @philip_schwarz

It turns out that the **Cats Alternative** provides a **guard** function!

On the next slide we take it for a ride.

```
import cats.Alternative
import cats.implicits

val alt = Alternative[List]
```

guard lets through **some** elements

```
[[1,2],[3,4]] >>= \nums ->
  [10,20] >>= \num ->
    guard (num > 15) >>
      return (num:nums)
```

```
List(List(1,2),List(3,4)).flatMap { nums =>
  List(10, 20).flatMap { num =>
    alt.guard(num > 15) >>
      alt.pure(num::nums)}}
```

```
[[20,1,2],[20,3,4]]
```

```
List(List(20,1,2),List(20,3,4))
```

guard lets through **all** elements

```
[[1,2],[3,4]] >>= \nums ->
  [10,20] >>= \num ->
    guard (True) >>
      return (num:nums)
```

```
List(List(1,2),List(3,4)).flatMap { nums =>
  List(10, 20).flatMap { num =>
    alt.guard(true) >>
      alt.pure(num::nums)}}
```

```
[[10,1,2],[20,1,2],[10,3,4],[20,3,4]]
```

```
List(List(10,1,2),List(20,1,2),
     List(10,3,4),List(20,3,4))
```

guard lets through **no** elements

```
[[1,2],[3,4]] >>= \nums ->
  [10,20] >>= \num ->
    guard (False) >>
      return (num:nums)
```

```
List(List(1,2),List(3,4)).flatMap { nums =>
  List(10, 20).flatMap { num =>
    alt.guard(false) >>
      alt.pure(num::nums)}}
```

```
[]
```

```
List()
```

failed non-deterministic computation

In the next five slides we mention the **IO monad**. If you are not familiar with it, feel free to skip the slides.

If you are interested in an introduction to the **IO monad** then see the following slide decks

Sierpinski's Triangle

Polyglot FP for Fun and Profit
Haskell and Scala

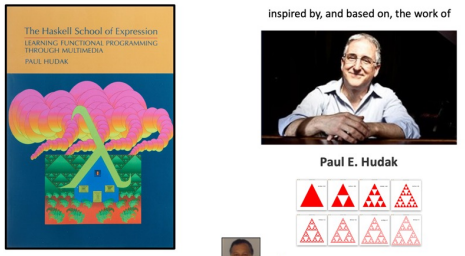
Take the very first baby steps on the path to doing **graphics** in **Haskell** and **Scala**

Learn about a simple yet educational **recursive algorithm** producing images that are pleasing to the eye

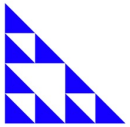
Learn how **functional programs** deal with the **side effects** required to draw images

See how libraries like **Gloss** and **Doodle** make drawing **Sierpinski's triangle** a doddle

inspired by, and based on, the work of



Paul E. Hudak



Cats Effect

Haskell Gloss

Scala Doodle

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

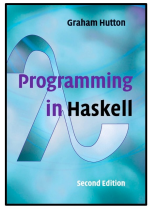
Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as **Game of Life** is first coded in **Haskell** and then translated into **Scala**, learning about the **IO monad** in the process

Also see how the program is coded in **Unison**, which replaces **Monadic Effects** with **Algebraic Effects**

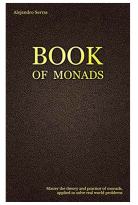
(Part 1)

through the work of



Graham Hutton
@haskellhutt

Will Kurt
@willkurt



Alejandro
Serrano Mena
@trupill

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

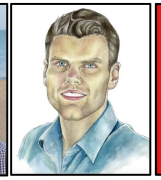
Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as the **impure functions** in the **Game of Life** are translated from **Haskell** into **Scala**, deepening you understanding of the **IO monad** in the process

(Part 2)

through the work of

Graham Hutton
@haskellhutt



Runar Bjarnason
@runarorama

FP in Scala

Paul Chiusano
@pchiusano

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

Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as **Trampolining** is used to overcome **Stack Overflow** issues with the simple **IO monad** deepening you understanding of the **IO monad** in the process

See **Game of Life IO actions** migrated to the **Cats Effect IO monad**, which is **trampolined** in its **flatMap** evaluation

(Part 3)

through the work of

Runar Bjarnason
@runarorama

FP in Scala

Paul Chiusano
@pchiusano

Graham Hutton
@haskellhutt

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

We have seen Alternative instances for Maybe/Option and for lists.

Out of curiosity, is there an Alternative instance for IO?

On the one hand, it looks like it:

The purpose of the `Alternative` instance for `IO` is to combine `IO` actions that might fail (by causing an IO error or otherwise throwing an exception) into a single `IO` action that "tries" multiple actions in turn, accepting the first successful one, or -- if all actions fail -- fails itself.

So, something like this would work to read one or more lines (using `some`) from standard input or else (using `<|>`) complain if no lines are available:

```
main = (print =<< some getLine) <|> putStrLn "No input!"
```

or you could write something like:

```
readConfig :: IO Config
readConfig = readConfigFile "~/.local/myapp/config"
         <|> readConfigFile "/etc/myapp/config"
         <|> return defaultConfig
```

On the next slide we have a look at an example of using the Alternative for IO.

@philip_schwarz

While not explicitly documented with `Alternative`, instances should essentially obey the following laws:

```
pure x <|> y = pure x
empty <|> x = x
```

You can intuit this as implementing some notion of "truthiness" and "falsiness", where `pure x` is always truthy and `empty` is always falsy.

For this to make any sense for `IO`, we need some notion of truthiness. There aren't many good ones, but `IO` has the ability to handle exceptions, so we can define truthy `IO` actions as actions that produce a value and falsy `IO` actions as actions that throw exceptions. Therefore, `(<|>)` for `IO` runs its first argument, and if it produces a value without throwing an exception, it returns the value; otherwise, it returns its second argument.

We now have a definition of `(<|>)` for `IO`, but what should `empty` be? Well, `empty` must be falsy, and we have defined falsiness on `IO` as "throwing an exception". Therefore, `empty` must be an action that throws an exception.

The `guard` function is very simple, since it is just `pure ()` when given `True` and `empty` when given `False`. This means your examples are really equivalent to the following:

```
empty <|> pure ()
empty <|> empty
```

https://stackoverflow.com/questions/48450826/what-is-the-purpose-of-instance-alternative-io

On the **LHS** of **Alternative**'s **<|>** operator we have a program that first asks the user to enter their name, and then tells them what their name is.

On the **RHS** of **<|>** we have a second program that we only want to execute if the first program encounters an error.

We are using **<|>** to ensure that either the first program executes successfully, or it encounters an error, in which case the second program is then executed.

See below for how the behaviour of the overall program changes depending on which part of the first program fails, if any.

```
(putStr "Enter Your Name:" >> getLine >>= \name -> putStrLn("Your Name is:" ++ name))
<|>
putStrLn("An IO Error Occurred!")
```

```
Enter Your Name:Fred
Your Name is:Fred
```

```
(empty >> getLine >>= \name -> putStrLn("Your Name is:" ++ name))
<|>
putStrLn("An IO Error Occurred!")
```

```
An IO Error Occurred!
```

```
(putStr "Enter Your Name:" >> empty >>= \name -> putStrLn("Your Name is:" ++ name))
<|>
putStrLn("An IO Error Occurred!")
```

```
Enter Your Name:An IO Error Occurred!
```

```
(putStr "Enter Your Name:" >> getLine >>= \name -> empty)
<|>
putStrLn("An IO Error Occurred!")
```

```
Enter Your Name:Fred
An IO Error Occurred!
```

But then on the other hand, it looks like the Alternative instance for IO is unlawful and/or broken.

Alejandro Serrano @trupill · Jul 11

Replying to @philip_schwarz @haskellhutt and 2 others

The usual problem with Monad and Alternative is how they are related to each other. Some people think that (x >>= empty) should be empty, which is violated in your third case (you get some side effects).

See the Haskell wiki on MonadPlus for more info.

1 1

Philip Schwarz @philip_schwarz · Jul 11

Thank you.

gitlab.haskell.org/ghc/ghc/-/issues/18871

Haskell Projects Groups Snippets Help

ghc GHC

Project overview

Repository

Issues 4,507

List

Boards

Labels

Service Desk

Milestones

Iterations

Glasgow Haskell Compiler › GHC › Issues › #18871

Open Created 8 months ago by **Isaac Berger**

Fix Alternative instance for IO

The Alternative instance for IO is currently defined in GHC.Base, using

```
(<|>) = mplusIO
```

Hunting further, mplusIO is defined in GHC.IO as

```
mplusIO :: IO a -> IO a -> IO a
mplusIO m n = m `catchException` \ (_ :: IOError) -> n
```

Why this is (possibly) wrong

hackage.haskell.org/package/alternative-io-0.0.1/docs/Data-Alternative-IO.html

alternative-io-0.0.1: IO as Alternative instance (deprecated)

Data.Alternative.IO

`IO` as `Alternative` instance.

If the left `IO` monad of (`<|>`) causes an error or `goNext` is used, the right `IO` monad is executed.

Of course, side effects cannot be rolled back. This means that this `Alternative` instance breaks the `Alternative` laws. But it's common in parsers.

What about in **Cats**?

Is there an **Alternative** instance for **IO**?

I could not find one.

It looks like there was an attempt to introduce an instance for some form of **IO**, and for fibers, but in the end only the proposal for fibers succeeded.

https://github.com/typelevel/cats-effect/issues/405

Search or jump to... / Pull requests Issues Marketplace Explore

typelevel / cats-effect

Code Issues 106 Pull requests 11 Discussions Actions Security Insights

Add an Alternative instance for Fiber/IO.Par #405

Closed **LukaJCB** opened this issue on 5 Nov 2018 · 35 comments

LukaJCB commented on 5 Nov 2018 Member

`Alternative` basically allows us to combine two `F` into `F[(A, B)]` or `F[Either[A, B]]` along with identities for each. This maps perfectly onto racing (where `IO.never` is the identity) and running in parallel (where `IO.unit` is the identity) An instance for `IO.Par` could allow us to make use of existing abstractions using the `SemigroupK` family. I.e. racing a list of IOs could be as simple `list.foldMapK(f)`.

github.com/typelevel/cats-effect/pull/576

Search or jump to... / Pull requests Issues Marketplace Explore

typelevel / cats-effect

Code Issues 106 Pull requests 11 Discussions Actions Security Insights

Add an Alternative instance for Fiber #576

Merged **djspiewak** merged 4 commits into `typelevel:master` from `theiterators:alternative-fiber` on 19 Jul 2019

Conversation 12 Commits 4 Checks 0 Files changed 4

@philip_schwarz

FWIW though, using **Cats** I *was* able to reproduce the same behaviour as in **Haskell**.

```
import cats.effect.IO
import cats.implicits._

def getLine(): IO[String] = IO{ scala.io.StdIn.readLine }
def putStr(s: String): IO[Unit] = IO{ print(s) }
def putStrLn(s: String): IO[Unit] = IO{ println(s) }
def empty: IO[Unit] = IO.raiseError(RuntimeException("bang"))
```

```
putStr("Enter Your Name:") >> getLine().flatMap{ name => putStrLn(s"Your Name is $name") }
<+>
putStrLn("An IO Error occurred!")
```

```
Enter Your Name:Fred
Your Name is:Fred
```

```
empty >> getLine().flatMap{ name => putStrLn(s"Your Name is $name") }
<+>
putStrLn("An IO Error occurred!")
```

```
An IO Error Occurred!
```

```
putStr("Enter Your Name:") >> empty.flatMap{ name => putStrLn(s"Your Name is $name") }
<+>
putStrLn("An IO Error occurred!")
```

```
Enter Your Name:An IO Error Occurred!
```

```
putStr("Enter Your Name:") >> getLine().flatMap{ name => empty }
<+>
putStrLn("An IO Error occurred!")
```

```
Enter Your Name:Fred
An IO Error Occurred!
```

Now that we have seen how the **guard** function has changed since **Miran Lipovača** wrote his book, let's turn to the task of displaying a solution to the **N-Queens** problem using graphics.

In the last slide of Part 1 we saw a **show** function for turning a solution into a string.

Functional Programming Principles in Scala

1.18K subscribers

6.3 Combinatorial Search Example

830 views • 11 Sept 2017

YouTube

```
def show(queens: List[Int]) = {
  val lines =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("* ").updated(col, "X ").mkString
  "\n" + (lines mkString "\n")
}                                                 > show: (queens: List[Int])java.lang.Stri

queens(4) map show                                > res0: scala.collection.immutable.Set[ja
                                                  | * * X *
                                                  | X * * *
                                                  | * * * X
                                                  | * X * * ", "
                                                  | * X * *
                                                  | * * * X
                                                  | X * * *
                                                  | * * X * ")
```

Here we wire the function up in a small program.

```
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

```
def showQueens(solution: List[Int]): Unit =
  println(show(solution))
```

```
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("* ")
                .updated(col, "X ")
                .mkString
  "\n" + (lines mkString "\n")
```

If, after printing a solution, we change the font to **Courier**, then the resulting board even takes on a square shape.

```
X * * * * * * *
* * * * X * * *
* * * * * * * X
* * * * * X * *
* * X * * * * *
* * * * * * X *
* X * * * * * *
* * * X * * * *
```

We are going to draw a solution board using the **Doodle** library. **Doodle**, adopts the concepts of **picture composition** and of the **separation** between, on the one hand, creating a picture, a **description** of something to be drawn, and on the other hand, doing the actual drawing, by processing/interpreting the picture.

Doodle

https://www.creativescala.org/doodle/

https://github.com/creativescala/doodle

Principles

Doodle: Compositional Vector Graphics

A few principles guide the design of **Doodle**, and differentiate it from other graphics libraries. The section explains these principles.

Pictures are Created by Composition

In Doodle a picture is constructed by combining together smaller pictures. For example, **we can create a row by putting pictures beside each other**. This idea of creating complex things from simpler things is known as ***composition***.

There are several implications of this, which means that **Doodle** operates differently to many other graphics libraries. This first is that **Doodle does not draw anything on the screen until you explicitly ask it to, say by calling the `draw` method**. **A picture represents a description of something we want to draw**. **A backend turns this description into something we can see (which might be on the screen or in a file)**. **This separation of description and action is known as the *interpreter pattern***. **The description is a "program" and a backend is an "interpreter" that runs that program.** In the graphics world the approach that **Doodle** takes is sometimes known as retained mode, while the approach of drawing immediately to the screen is known as immediate mode.

Another implication is that **Doodle can allow relative layout of objects**. **In Doodle we can say that one picture is next to another and Doodle will work out where on the screen they should be**. This requires a retained mode API as you need to keep around information about a picture to work out how much space it takes up.

A final implication is that **pictures have no mutable state**. **This is needed for composition** so you can, for example, put a picture next to itself and have things render correctly.

All of these ideas are core to functional programming, so you may have seen them in other contexts if you have experienced with functional programming. If not, don't worry. You'll quickly understand them once you start using **Doodle**, as **Doodle** makes the ideas very concrete.

Image

Doodle

Doodle: Compositional Vector Graphics

The `Image` library is the easiest way to create images using Doodle. The tradeoff the `Image` library makes is that it only support a (large but limited) subset of operations that are supported across all the backends.

`Image` is based on *composition* and the *interpreter pattern*.

Composition basically means that **we build big Images out of small Images**. For example, if we have an `Image` describing a red square and an Image describing a blue square

```
val redSquare = Image.square(100).fillColor(Color.red)
val blueSquare = Image.square(100).fillColor(Color.blue)
```

we can create an Image describing a red square next to a blue square by combining them together.

```
val combination = redSquare.beside(blueSquare)
```

The **interpreter pattern** means that **we separate describing the Image from rendering it**. **Writing**

```
Image.square(100)
```

doesn't draw anything. **To draw an image we need to call the `draw()` method**. This separation is important for **composition**; if we were to immediately draw we would lose **composition**.

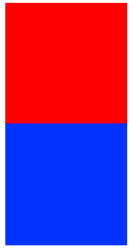
Basic Shapes

...

`Image.square(sideLength)` creates a **square** with the given side length.

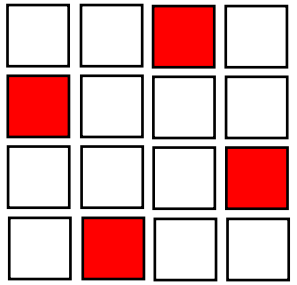
Just like we can use the **beside** function to create an image describing a red square next to a blue square, we can use the **above** function to create an image describing a red square above a blue square.

red beside blue



red above blue

Are you thinking what I am thinking? Yes, this should simplify things: we can can create a row of squares by combining the squares using the **beside** function, and we can create a board by combining rows using the **above** function.



Here again is the program that prints a solution board to the console, as text.

And here is a new program that uses **Doodle** to draw a solution board.

```
@main def main =
  val solution = List(4, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

```
def showQueens(solution: List[Int]): Unit =
  println(show(solution))
```

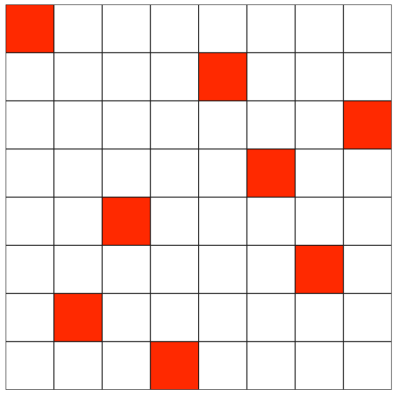
```
X * * * * * * *
* * * * X * * *
* * * * * * * X
* * * * * X * *
* * X * * * * *
* * * * * * X *
* X * * * * * *
* * * X * * * *
```

```
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

```
def showQueens(solution: List[Int]): Unit =
  val n = solution.length
  val frameTitle = s"{n}-Queens Problem – A solution"
  val frameWidth = 1000
  val frameHeight = 1000
  val frameBackgroundColour = Color.white
  val frame =
    Frame.size(frameWidth,frameHeight)
         .title(frameTitle)
         .background(frameBackgroundColour)
  show(solution).draw(frame)
```

Martin Odersky

```
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("* ")
                .updated(col, "X ")
                .mkString
  "\n" + (lines mkString "\n")
```



```
def show(queens: List[Int]): Image =
  val square = Image.square(100).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  val squareImageGrid: List[List[Image]] =
    for col <- queens.reverse
    yield List.fill(queens.length)(emptySquare)
              .updated(col,fullSquare)
  combine(squareImageGrid)
```

As suggested on the previous slide, the **combine** function on the right

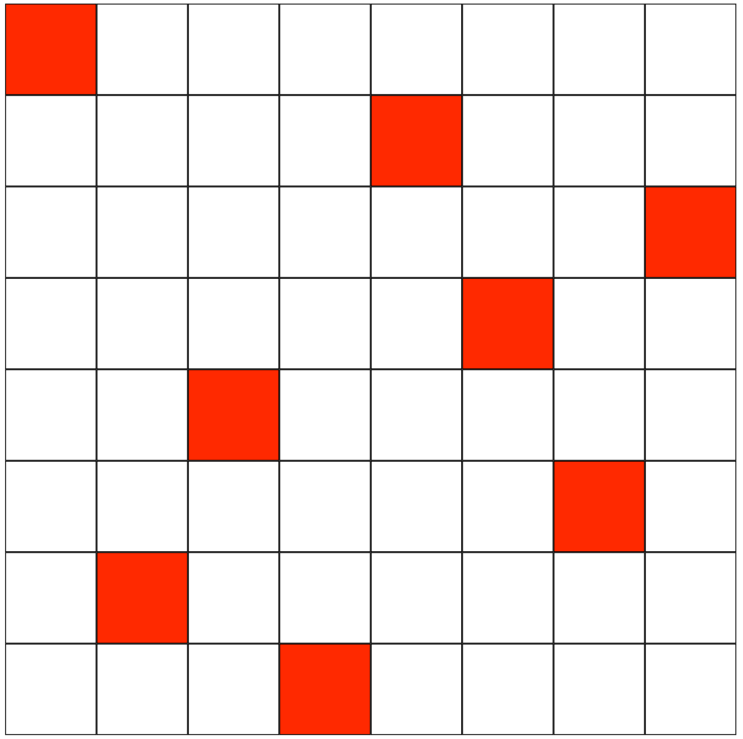
- creates the image of a row of squares by combining the images of the squares using the **beside** function
- creates the image of a grid of squares by combining the images of the rows using the **above** function

```
def combine(imageGrid: List[List[Image]): Image =
  imageGrid.map(_.reduce(_ beside _))
           .reduce(_ above _)
```

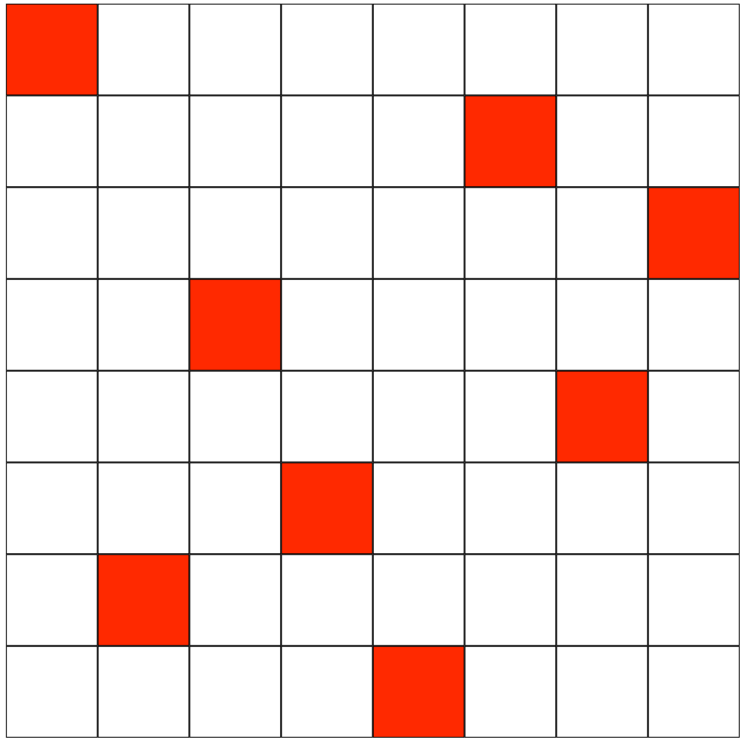
The boards drawn by our **Scala** program for the first three solutions of the **8-Queens Problem**.

@philip_schwarz

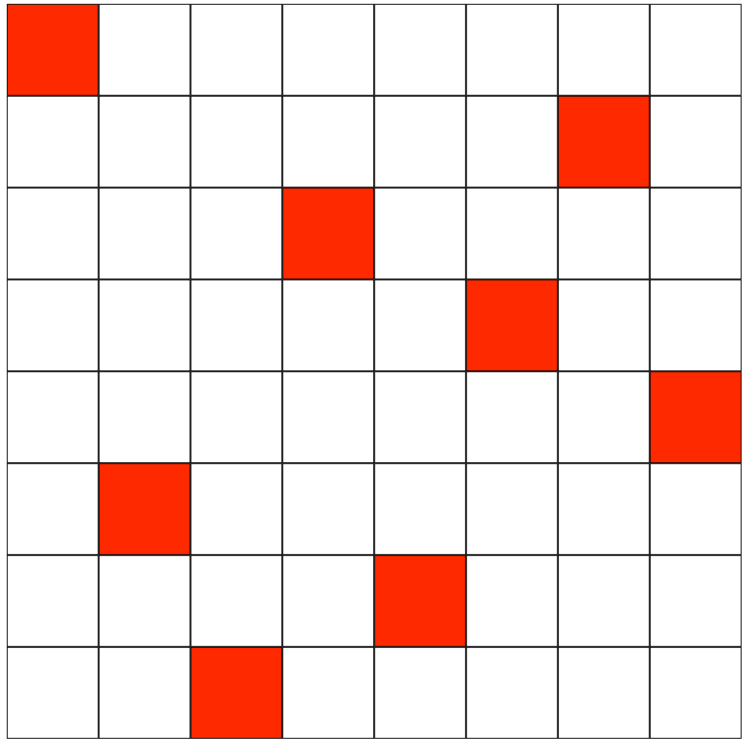
`List(3, 1, 6, 2, 5, 7, 4, 0)`



`List(4, 1, 3, 6, 2, 7, 5, 0)`



`List(2, 4, 1, 7, 5, 3, 6, 0)`



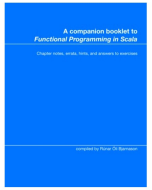
The next thing we are going to do is improve a bit the implementation of the **combine** function used by our program.

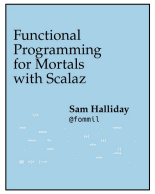
To do that, we are going to use the concepts of **Monoid** and **Foldable**. If you are new to them then see the next fifteen slides for a minimal introduction to the concepts, otherwise just skip the slides, which are partly based on the slide decks below.

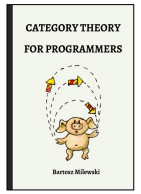
Monoids

with examples using Scalaz and Cats

based on







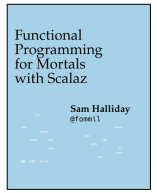
Part 1

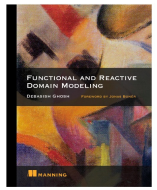
slides by  @philip_schwarz

Monoids

with examples using Scalaz and Cats

Part II - based on





slides by  @philip_schwarz

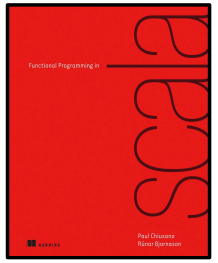
What is a monoid?

Let's consider the algebra of **string concatenation**. We can add `"foo" + "bar"` to get `"foobar"`, and the **empty string** is an **identity element** for that operation. That is, if we say `(s + "")` or `("" + s)`, the result is always `s`.

```
scala> val s = "foo" + "bar"
s: String = foobar

scala> assert( s == s + "" )

scala> assert( s == "" + s )

scala>
```

Furthermore, if we combine three strings by saying `(r + s + t)`, the operation is **associative** —it doesn't matter whether we parenthesize it: `((r + s) + t)` or `(r + (s + t))`.

```
scala> val (r,s,t) = ("foo","bar","baz")
r: String = foo
s: String = bar
t: String = baz

scala> assert( ( ( r + s ) + t ) == ( r + ( s + t ) ) )

scala> assert( ( ( r + s ) + t ) == "foobarbaz" )

scala>
```

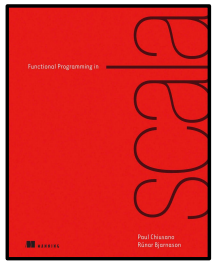


The exact same rules govern **integer addition**. It's **associative**, since `(x + y) + z` is always equal to `x + (y + z)`

```
scala> val (x,y,z) = (1,2,3)
x: Int = 1
y: Int = 2
z: Int = 3

scala> assert( ( ( x + y ) + z ) == ( x + ( y + z ) ) )

scala> assert( ( ( x + y ) + z ) == 6 )

scala>
```

and it has an **identity element**, **0** , which "does nothing" when added to another integer

```
scala> val s = 3
s: Int = 3

scala> assert( s == s + 0)

scala> assert( s == 0 + s)

scala>
```



Ditto for **integer multiplication**

```
scala> val (x,y,z) = (2,3,4)
x: Int = 2
y: Int = 3
z: Int = 4

scala> assert(( ( x * y ) * z ) == ( x * ( y * z ) ))

scala> assert(( ( x * y ) * z ) == 24)

scala>
```

whose **identity element** is **1**

```
scala> val s = 3
s: Int = 3

scala> assert( s == s * 1)

scala> assert( s == 1 * s)

scala>
```

The **Boolean** operators **&&** and **||** are likewise **associative**

```
scala> val (p,q,r) = (true,false,true)
p: Boolean = true
q: Boolean = false
r: Boolean = true

scala> assert(( ( p || q ) || r ) == ( p || ( q || r ) ))

scala> assert(( ( p || q ) || r ) == true )

scala> assert(( ( p && q ) && r ) == ( p && ( q && r ) ))

scala> assert(( ( p && q ) && r ) == false )
```

and they have **identity elements `true`** and **`false`**, respectively

```
scala> val s = true
s: Boolean = true

scala> assert( s == ( s && true ) )

scala> assert( s == ( true && s ) )

scala> assert( s == ( s || false ) )

scala> assert( s == ( false || s ) )
```

These are just a few simple examples, but **algebras like this are virtually everywhere**. The term for this kind of **algebra** is **monoid**.

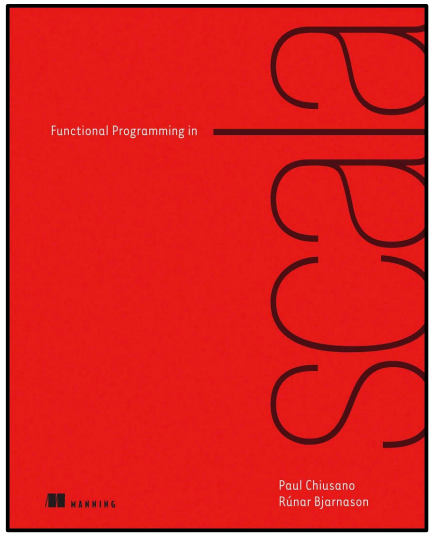
The **laws** of **associativity** and **identity** are collectively called the **monoid laws**.

A **monoid** consists of the following:

- Some type `A`
- An associative binary operation, `op`, that takes two values of type `A` and combines them into one: `op(op(x,y), z) == op(x, op(y,z))` for any choice of `x: A`, `y: A`, `z: A`
- A value, `zero: A`, that is an identity for that operation: `op(x, zero) == x` and `op(zero, x) == x` for any `x: A`

We can express this with a **Scala** trait:

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

An example instance of this trait is the `String` **monoid**:

```
val stringMonoid = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  val zero = ""
}
```

String concatenation function

`List` concatenation also forms a **monoid**:

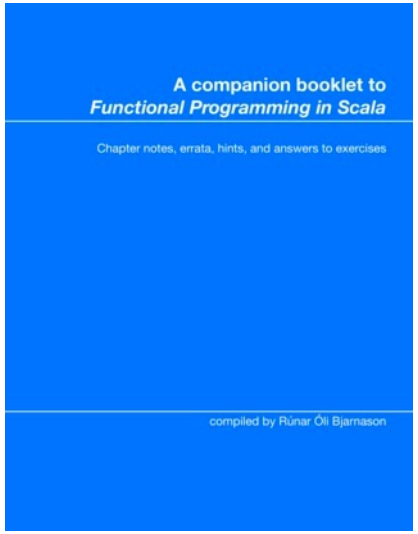
```
def listMonoid[A] = new Monoid[List[A]] {
  def op(a1: List[A], a2: List[A]) = a1 ++ a2
  val zero = Nil
}
```

List function returning a new list containing the elements from the left hand operand followed by the elements from the right hand operand

`Monoid` instances for **integer addition** and **multiplication** as well as the **Boolean operators**

```
implicit val intAdditionMonoid = new Monoid[Int] {
    def op(x: Int, y: Int) = x + y
    val zero = 0
}

implicit val intMultiplicationMonoid = new Monoid[Int] {
    def op(x: Int, y: Int) = x * y
    val zero = 1
}
```

```
implicit val booleanOr = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x || y
    val zero = false
}

implicit val booleanAnd = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x && y
    val zero = true
}
```

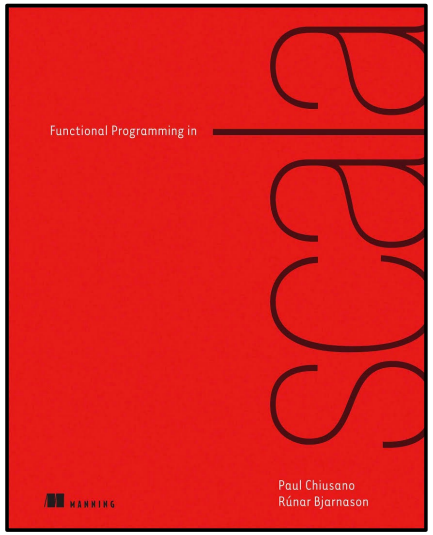


(by Runar Bjarnason)
@runarorama

Just what is a **monoid**, then? It's simply a type **`A`** and **an implementation of `Monoid[A]`** that satisfies the **laws**.

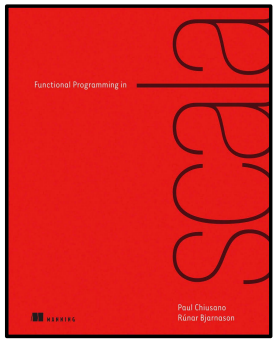
Stated tersely, a **monoid** is a **type** together with a **binary operation** (**`op`**) over that type, satisfying **associativity** and having an **identity element** (**`zero`**).

What does this buy us? **Just like any abstraction, a monoid is useful to the extent that we can write useful generic code assuming only the capabilities provided by the abstraction. Can we write any interesting programs, knowing nothing about a type other than that it forms a monoid? Absolutely!**



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Summary of the naming and location of a **Monoid**'s **associative binary operation** and **identity element** - simplified



FP in Scala

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```

```scala
trait Semigroup[F]  { self =>
  def append(f1: F, f2: => F): F
  …
}

final class SemigroupOps[F]…(implicit val F: Semigroup[F]) … {
  final def |+|(other: => F): F      = F.append(self, other)
  final def mappend(other: => F): F = F.append(self, other)
  final def ⊹(other: => F): F        = F.append(self, other)
  …
}

trait Monoid[F] extends Semigroup[F] { self =>
  def zero: F
  …
}
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```haskell
class Semigroup m where
(<>) :: m -> m -> m

class Semigroup m => Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

The **mappend** method is redundant and has the default implementation **mappend** = '(**<>**)'

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
  …
}

final class SemigroupOps[A: Semigroup](lhs: A) {
  def |+|(rhs: A): A = macro Ops.binop[A, A]
  def combine(rhs: A): A = macro Ops.binop[A, A]
  def combineN(rhs: Int): A = macro Ops.binop[A, A]
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
  …
}
```

Folding Right and Left

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
def sum(ints: List[Int]): Int =
  ints match {
    case Nil => 0
    case Cons(x, xs) => x + sum(xs)
}
```

```
def product(ds: List[Double]): Double =
  ds match {
    case Nil => 1.0
    case Cons(x, xs) => x * product(xs)
}
```

```
scala> sum(Cons(1,Cons(2,Cons(3,Nil))))
res0: Int = 6
scala> product(Cons(1.0,Cons(2.5,Cons(3.0,Nil))))
res1: Double = 7.5
scala>
```

Note how similar these two definitions are. They're operating on different types (`List[Int]` versus `List[Double]`), but aside from this, **the only differences are the value to return in the case that the list is empty** (`0` in the case of `sum`, `1.0` in the case of `product`), **and the operation to combine results** (`+` in the case of `sum`, `*` in the case of `product`).

Whenever you encounter duplication like this, you can generalize it away by pulling subexpressions out into function arguments…
Let's do that now. Our function will take as arguments the value to return in the case of the empty list, and the function to add an element to the result in the case of a nonempty list.

```
def foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B =
  as match {
    case Nil => z
    case Cons(x, xs) => f(x, foldRight(xs, z)(f))
  }
```

```
def sum(ns: List[Int]) =
  foldRight(ns, 0)((x,y) => x + y)

def product(ns: List[Double]) =
  foldRight(ns, 1.0)(_ * _)
```

`foldRight` is not specific to any one type of element, and we discover while generalizing that the value that's returned doesn't have to be of the same type as the elements of the list!

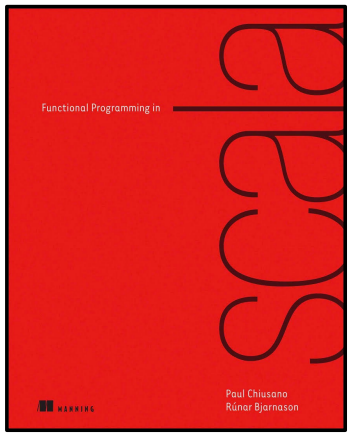
Our implementation of `foldRight` is not tail-recursive and will result in a `StackOverflowError` for large lists (we say it's not stack-safe). Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that is tail-recursive

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
1 + (2 + (3 + (foldRight(Nil, 0)((x,y) => x + y))))
1 + (2 + (3 + (0)))
6
```

```
@annotation.tailrec
def foldLeft[A,B](l: List[A], z: B)(f: (B, A) => B): B = l match {
  case Nil => z
  case Cons(h,t) => foldLeft(t, f(z,h))(f)
}
```

```
def foldRightViaFoldLeft[A,B](l: List[A], z: B)(f: (A,B) => B): B =
  foldLeft(reverse(l), z)((b,a) => f(a,b))
```

Implementing `foldRight` via `foldLeft` is useful because it lets us implement `foldRight` tail-recursively, which means it works even for large lists without overflowing the stack.

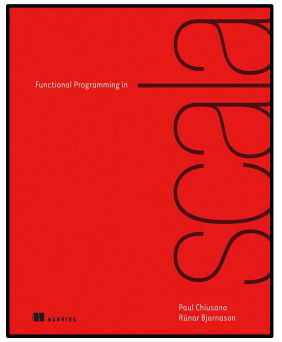


Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

footnotes

[9] In the Scala standard library, `foldRight` is a method on `List` and its arguments are curried similarly for better type inference.

[10] Again, `foldLeft` is defined as a method of `List` in the Scala standard library, and it is curried similarly for better type inference, so you can write `mylist.foldLeft(0.0)(_ + _)`.



FP in Scala

Scala Standard Library 2.12.8 Search

scala.collection.immutable
List

```
def foldLeft[B](z: B)(op: (B, A) ⇒ B): B
```

Applies a binary operator to a start value and all elements of this sequence, going left to right.

Note: will not terminate for infinite-sized collections.

B	the result type of the binary operator.
z	the start value.
op	the binary operator.
returns	the result of inserting op between consecutive elements of this sequence, going left to right with the start value z on the left:

```
op(...op(z, x_1), x_2, ..., x_n)
```

where $x_1, \ldots, x_n$ are the elements of this sequence. Returns z if this sequence is empty.

Definition Classes LinearSeqOptimized → TraversableOnce → GenTraversableOnce

```
def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

Applies a binary operator to all elements of this list and a start value, going right to left.

B	the result type of the binary operator.
z	the start value.
op	the binary operator.
returns	the result of inserting op between consecutive elements of this list, going right to left with the start value z on the right:

```
op(x_1, op(x_2, ... op(x_n, z)...))
```

where $x_1, \ldots, x_n$ are the elements of this list. Returns z if this list is empty.

Definition Classes List → LinearSeqOptimized → IterableLike → TraversableOnce → GenTraversableOnce

```
assert( List(1,2,3,4).foldLeft(0)(_+_) == 10 )
assert( List(1,2,3,4).foldRight(0)(_+_) == 10 )

assert( List(1,2,3,4).foldLeft(1)(_*_) == 24 )
assert( List(1,2,3,4).foldRight(1)(_*_) == 24 )

assert( List("a","b","c","d").foldLeft("")(_+_) == "abcd" )
assert( List("a","b","c","d").foldRight("")(_+_) == "abcd" )
```

Foldable data structures

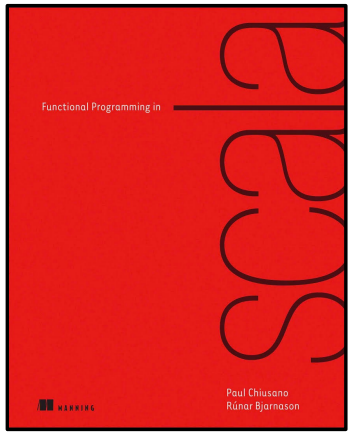
In chapter 3, we implemented the **data structures** `List` and `Tree`, both of which could be **folded**. In chapter 5, we wrote `Stream`, a lazy structure that also can be **folded** much like a `List` can, and now we've just written a **fold** for `IndexedSeq`.

When we're writing code that needs to process data contained in one of these **structures**, **we often don't care about** the **shape** of the **structure** (whether it's a tree or a list), or whether it's **lazy** or not, or provides **efficient random access**, and so forth.

For example, if we have a **structure** full of integers and want to calculate their sum, we can use `foldRight`:

```
ints.foldRight(0)(_ + _)
```

Looking at just this code snippet, **we shouldn't have to care about the type of `ints`**. It could be a `Vector`, a `Stream`, or a `List`, **or anything at all with a `foldRight` method**. **We can capture this commonality in a** `trait`:



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

```
trait Foldable[F[_]] {

  def foldRight[A,B](as: F[A])(z: B)(f: (A,B) => B): B

  def foldLeft[A,B](as: F[A])(z: B)(f: (B,A) => B): B

  def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
      foldLeft(as)(m.zero)(m.op)
}
```

Here we're abstracting over a type constructor `F`, much like we did with the `Parser` type in the previous chapter. We write it as `F[_]`, where the underscore indicates that `F` is not a type but a **type constructor** that takes one type argument. Just like functions that take other functions as arguments are called **higher-order functions**, something like `Foldable` is a **higher-order type constructor** or a **higher-kinded type** .[7]

[7] Just like values and functions have types, types and **type constructors** have **kinds**. Scala uses **kinds** to track how many type arguments a type constructor takes, whether it's co- or contravariant in those arguments, and what the kinds of those arguments are.

Don’t spend too much time comparing the **Scala** and **Haskell** implementations of the various **folding functions** on the next three slides.

There are many different ways of implementing the functions.

While looking at some of the implementations can reinforce our understanding of the functions, such comparisons are not relevant for our current purposes.

EXERCISE 10.12

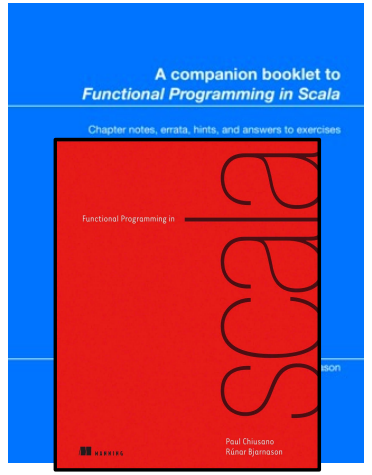
Implement `Foldable[List]`, `Foldable[IndexedSeq]`, and `Foldable[Stream]`. Remember that **`foldRight`**, **`foldLeft`**, and **`foldMap`** can all be implemented in terms of each other, but that might not be the most efficient implementation.

```scala
trait Foldable[F[_]] {

  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}

object ListFoldable extends Foldable[List] {
  override def foldRight[A, B](as:List[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:List[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
  override def foldMap[A, B](as:List[A])(f:A=>B)(mb:Monoid[B]):B =
    foldLeft(as)(mb.zero)((b, a) => mb.op(b, f(a)))
}

object IndexedSeqFoldable extends Foldable[IndexedSeq] {…}

object StreamFoldable extends Foldable[Stream] {
  override def foldRight[A, B](as:Stream[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:Stream[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
}
```

If you are new to **monoids**, don't worry about the implementation of **foldRight** and **foldLeft** except for the fact that it is possible to define them using **foldMap**.

Let's use the methods of **`ListFoldable`** and **`StreamFoldable`** to fold **Lists**/**Streams** of **Int**s and **String**s.

A Companion booklet to FP in Scala



FP in Scala

```scala
assert( ListFoldable.foldLeft(List(1,2,3))(0)(_+_) == 6)
assert( ListFoldable.foldRight(List(1,2,3))(0)(_+_) == 6)

assert( ListFoldable.concatenate(List(1,2,3))(intAdditionMonoid) == 6)
assert( ListFoldable.foldMap(List("1","2","3"))(_ toInt)(intAdditionMonoid) == 6)

assert( StreamFoldable.foldLeft(Stream(1,2,3))(0)(_+_) == 6)
assert( StreamFoldable.foldRight(Stream(1,2,3))(0)(_+_) == 6)

assert( StreamFoldable.concatenate(Stream(1,2,3))(intAdditionMonoid) == 6)
assert( StreamFoldable.foldMap(Stream("1","2","3"))(_ toInt)(intAdditionMonoid) == 6)
```

```scala
assert( ListFoldable.foldLeft(List("a","b","c"))("")(_+_) == "abc")
assert( ListFoldable.foldRight(List("a","b","c"))("")(_+_) == "abc")

assert( ListFoldable.concatenate(List("a","b","c"))(stringMonoid) == "abc")
assert( ListFoldable.foldMap(List(1,2,3))(_ toString)(stringMonoid) == "123")

assert( StreamFoldable.foldLeft(Stream("a","b","c"))("")(_+_)) == "abc")
assert( StreamFoldable.foldRight(Stream("a","b","c"))("")(_+_) == "abc")

assert( StreamFoldable.concatenate(Stream("a","b","c"))(stringMonoid) == "abc")
assert( StreamFoldable.foldMap(Stream(1,2,3))(_ toString)(stringMonoid) == "123")
```

The four fundamental functions of the Foldable trait in FPiS, Scalaz and Cats

fold

FPiS
```
def concatenate[A](as: F[A])(m: Monoid[A]): A =
  foldLeft(as)(m.zero)(m.op)
```

Scalaz
```
def fold[M:Monoid](t:F[M]):M =
  foldMap[M, M](t)(x => x)
```
```
def sumr[A](fa:F[A])(implicit A:Monoid[A]):A =
  foldRight(fa, A.zero)(A.append)
```
```
def suml[A](fa:F[A])(implicit A: Monoid[A]): A =
  foldLeft(fa, A.zero)(A.append(_, _))
```

Cats
```
def fold[A](fa: F[A])(implicit A: Monoid[A]): A =
  foldLeft(fa, A.empty) { (acc, a) => A.combine(acc, a) }
```
```
def combineAll[A: Monoid](fa: F[A]): A =
  fold(fa)
```

foldMap

FPiS
```
def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
  foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))
```

Scalaz
```
def foldMap[A,B](fa: F[A])(f: A => B)(implicit F: Monoid[B]): B
```

Cats
```
def foldMap[A, B](fa: F[A])(f: A => B)(implicit B: Monoid[B]): B =
  foldLeft(fa, B.empty)((b, a) => B.combine(b, f(a)))
```

foldRight

FPiS
```
def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
  foldMap(as)(f.curried)(endoMonoid[B])(z)
```

Scalaz
```
def foldRight[A, B](fa: F[A], z: => B)(f: (A, => B) => B): B
```

Cats
```
def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```

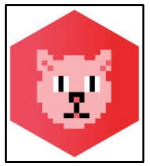
foldLeft

FPiS
```
def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
  foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)
```

Scalaz
```
def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B = {
  import Dual._, Endo._, syntax.std.all._
  Tag.unwrap(foldMap(fa)((a: A) =>
    Dual(Endo.endo(f.flip.curried(a))))(dualMonoid)) apply (z)
}
```

Cats
```
def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B
```

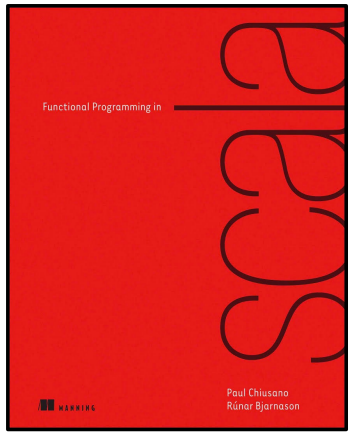
If you are new to **monoids**, don't worry about **endoMonoid** and the **dual** function, they are not important in our context.



concatenate	fold,suml,sumr	fold,combineAll
foldMap	foldMap	foldMap
foldLeft	foldLeft	foldLeft
foldRight	foldRight	foldRight

```scala
trait Foldable[F[_]] {

 def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

 def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def toList[A](as: F[A]): List[A] =
    foldRight(as)(List[A]())(_ :: _)
}
```

```scala
object ListFoldable extends Foldable[List] {

 override def foldMap[A, B](as:List[A])(f:A=>B)(mb:Monoid[B]):B =
    foldLeft(as)(mb.zero)((b, a) => mb.op(b, f(a)))

  override def foldRight[A, B](as:List[A])(z:B)(f:(A,B)=>B) =
    as match {
      case Nil => z
      case Cons(h, t) => f(h, foldRight(t, z)(f))
    }

  override def foldLeft[A, B](as:List[A])(z:B)(f:(B,A)=>B) =
    as match {
      case Nil => z
      case Cons(h, t) => foldLeft(t, f(z,h))(f)
    }
}
```

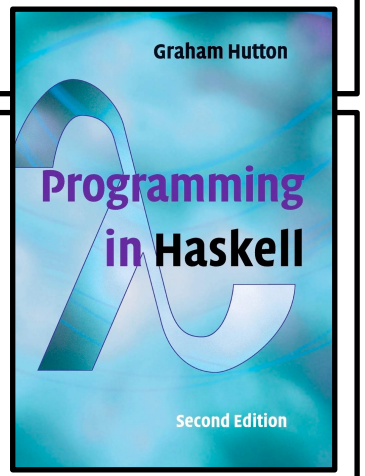



```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```

```haskell
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

  toList  :: t a -> [a]
  …
```

Default `Foldable` Definitions

```haskell
foldMap f = foldr (mappend.f) mempty
foldr f v = foldr f v . toList

fold = foldMap id
foldl f v = foldl f v . toList

toList = foldMap (\x -> [x])
…
```




```haskell
instance Foldable [] where

-- foldMap :: Monoid b => (a -> b) -> [a] -> b
foldMap _ [] = mempty
foldMap f (x:xs) = f x `mappend` foldMap f xs

-- foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

-- fold    :: Monoid a => t a -> a
fold = foldMap id

-- foldl   :: (a -> b -> a) -> a -> t b -> a
foldl _ v [] = v
foldl f v (x:xs) = foldl f (f v x) xs

toList  :: t a -> [a]
toList = id
…
```

		
concatenate	fold,combineAll	fold,mconcat
foldMap	foldMap	foldMap
foldLeft	foldLeft	foldl
foldRight	foldRight	foldr
op	combine	mappend
zero	empty	mempty

```haskell
instance Monoid [a] where
-- mempty :: [a]
mempty = []
 -- mappend :: [a] -> [a] -> [a]
mappend = (++)
```

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

The next two slides contain some examples of using **fold** and **foldMap**.

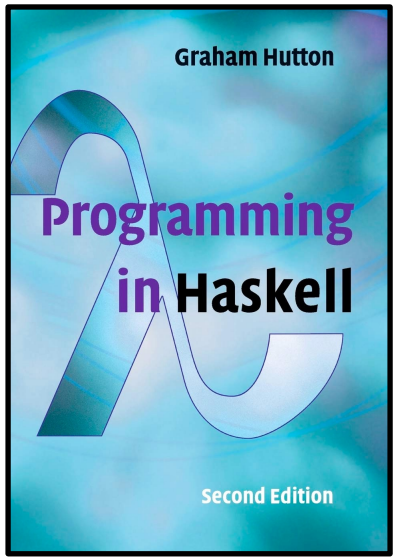
This slide is only here to help us understand upcoming examples using the **Haskell Monoids** for addition and multiplication, so feel free to skip this slide, at least on a first reading.

Numeric monoid for addition:

```
newtype Sum a = Sum a
          deriving (Eq, Ord, Show, Read)
getSum :: Sum a -> a
getSum (Sum x) = x

instance Num a => Monoid (Sum a) where
  -- mempty :: Sum a
  mempty = Sum 0

  -- mappend :: Sum a -> Sum a -> Sum a
  Sum x `mappend` Sum y = Sum (x+y)
```

Numeric monoid for multiplication:

```
newtype Product a = Product a
          deriving (Eq, Ord, Show, Read)

getProduct :: Product a -> a
getProduct (Product x) = x

instance Num a => Monoid (Product a) where
  -- mempty :: Product a
  mempty = Product 1

  -- mappend :: Product a -> Product a -> Product
  a
  Product x `mappend` Product y = Product (x*y)
```



Graham Hutton

@haskellhutt

fold Examples

(String, +, "") Monoid

```
Scala      List("a","b","c").combineAll == "abc"

Haskell    fold ["a","b","c"] == "abc"
```

(Integer, +, 0) Monoid

```
Scala      List(1,2,3).combineAll == 6

Haskell    getSum (fold (fmap Sum [1,2,3])) == 6
```

(Integer, *, 1) Monoid

```
Scala      val intProdMonoid = Monoid.instance[Int](1,_*_)
           List(1,2,3,4).combineAll(intProdMonoid) == 24

Haskell    getProduct (fold (fmap Product [1,2,3,4])) == 24
```

(List, ++, []) Monoid

```
Scala      List(List(1,2),List(3,4),List(5,6)).combineAll == List(1,2,3,4,5,6)

Haskell    fold [[1,2],[3,4],[5,6]] = [1,2,3,4,5,6]
```

(Option[m.type], m.op, m.zero) Monoid where m = (type=Integer, op=+, zero=0) Monoid

```
Scala      List(Some(1), None, Some(3), Some(4)).combineAll == Some(8)

Haskell    fmap getSum (fold [Just (Sum 1), Nothing, Just (Sum 3), Just (Sum 4)]) == Just 8
```

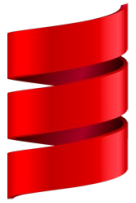
@philip_schwarz

In Scala, we are making the relevant Cats Foldable and Monoid instances available by importing cats._ and cats.implicits._

In Haskell, we are importing Data.Foldable and Data.Monoid.

If we look at the Haskell definitions of fold, foldMap and mconcat on the previous slide, we see that fold and mconcat are equivalent, so on this slide, where we use fold, we could have just as well used mconcat.

foldMap Examples



(String, +, “”) Monoid

```
Scala      List("a","b","c").foldMap(_ + "x") == "axbxcx"
Haskell    foldMap (\s -> s ++ "x") ["a","b","c"] == "axbxcx”
```

(String, +, “”) Monoid

```
Scala      List(12,34,56).foldMap(_.toString) == "123456"
Haskell    foldMap show [12,34,56] == "123456"
```

(Integer, +, 0) Monoid

```
Scala      List(1,2,3).foldMap(1 + _) == 9
Haskell    getSum (foldMap ((+) 1) (fmap Sum [1,2,3])) == 9
```

(List, ++, []) Monoid

```
Scala      List(Some(1), Some(2), None, Some(3)).foldMap(_.toList) == List(1, 2, 3)
Haskell    foldMap toList [Just 1, Just 2, Nothing, Just 3] == [1,2,3]
```

(Option[m.type], m.op, m.zero) Monoid where m = (type=Integer, op=+, zero=0) Monoid

```
Scala      List( Some(1), Some(3), None, Some(4) ).foldMap(x => x map (_ + 1 )) == Some(11)
Haskell    fmap getSum (foldMap (fmap ((+) 1)) [Just (Sum 1), Just (Sum 2), Nothing, Just (Sum 3)]) == Just 9
```

Following that refresher on **Monoid** and **Foldable**, let's turn to the task of improving the implementation of our program's **combine** function.

See below for the current implementation of the the function.

```
def combine(imageGrid: List[List[Image]): Image =
  imageGrid.map(_.reduce(_ beside _))
           .reduce(_ above _)
```

That's a nice way of coalescing all the grid's images into a single image, except that if/when the grid has no rows, or has an empty row, the **reduce** function will throw an **unsupported operation exception**.

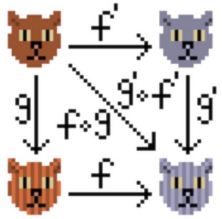
```
def combine(imageGrid: List[List[Image]): Image =
  imageGrid.map(_.foldLeft(Image.empty)(_ beside _))
           .foldLeft(Image.empty)(_ above _)
```

That's better: the **foldLeft** function now handles both an empty grid and a grid with empty rows.

We can make the **combine** function even simpler if we define the two **monoids** on the right.

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

Cats




```
def combine(imageGrid: List[List[Image]]): Image =
  Foldable[List].foldMap(imageGrid){ row =>
    Foldable[List].combineAll(row)(beside)
  }(above)
```

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

We can then pass the two **monoids** to the **foldMap** and **combineAll** functions provided by the **Cats Foldable** instance for **List**.

Cats allows us to call `xs.foldMap(m)` instead of `Foldable[List].foldMap(xs)(m)`. Similarly for **combineAll**.

Having written a **Scala** program that displays the chess board for an **N-Queens** solution, it would be nice to write the equivalent program in **Haskell**.

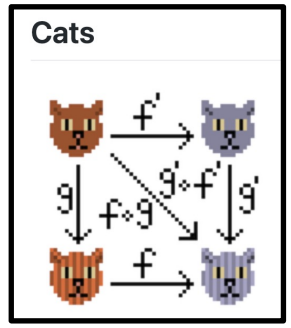
We'll write the program using a graphics library called **Gloss**.

For our purposes, the main way in which **Gloss** differs from **Doodle** is that while the latter makes our life easy by allowing us to create images without worrying about their position, and then to combine the images using their **beside** and **above** functions, **Gloss** requires us to position the images ourselves, and then combine them by stacking one above the other.

Because it is possible to build images with **Doodle** in the same way that they are built using **Gloss**, we are first going to create a modified version of our **Scala** program that does just that. We are then going to translate the program into **Haskell**.

The first thing we need to do is change the **combine** function so that rather than coalescing a grid of images by using the images' **beside** and **above** functions, which position an image **beside** or **above** another, it does so using the images' **on** function, which simply places one image **on** top of the other.

In the existing implementation of **combine**, we are using two **monoids**, one for composing images horizontally, and one for composing them vertically. Because of that, even though **Foldable**'s **foldMap** and **combineAll** functions are able to receive an implicit **monoid** parameter, we are having to pass the two different **monoids** into the functions explicitly.




```
import cats.Monoid

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

```
import cats.implicits._

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

In the new implementation however, we only need a single **monoid** for composing images, i.e. one that superimposes the images, so if we make the **monoid** implicit, it is automatically passed to the **foldMap** and **combineAll** functions behind the scenes.

@philip_schwarz

```
import cats.Monoid

given Monoid[Image] = Monoid.instance[Image](Image.empty, _ on _)
```

```
import cats.implicits._

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

Now that we have modified the **combine** function to superimpose images rather than **position** them **beside** or **above** each other, we need to **position** the images ourselves before **combining** them.

To do that, we no longer just create square images, we also use their **row** and **column** indices in the grid to compute their desired **position** in the drawing and then use **Image**'s **at** function to **position** the images.

On the left we see the current **show** and **combine** functions, and on the right, we see the modified ones.

```
def show(queens: List[Int]): Image =
  val square = Image.square(100).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  val squareImageGrid: List[List[Image]] =
    for col <- queens.reverse
    yield List.fill(queens.length)(emptySquare)
              .updated(col,fullSquare)
  combine(squareImageGrid)
```

```
def show(queens: List[Int], squareSize: Int): Image =
  val n = queens length
   val solution = queens reverse
   val (emptySquare, fullSquare) = createSquareImages(squareSize)
   val squareImages: List[Image] =
    for
      row <- List.range(0, n)
      col <- List.range(0, n)
      squareX = col * squareSize
      squareY = - row * squareSize
      squareImageAtOrigin = if solution(row) == col then fullSquare else emptySquare
      squareImage = squareImageAtOrigin.at(squareX,squareY)
    yield squareImage
  val squareImageGrid = (squareImages grouped n).toList
  combine(squareImageGrid)
```

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

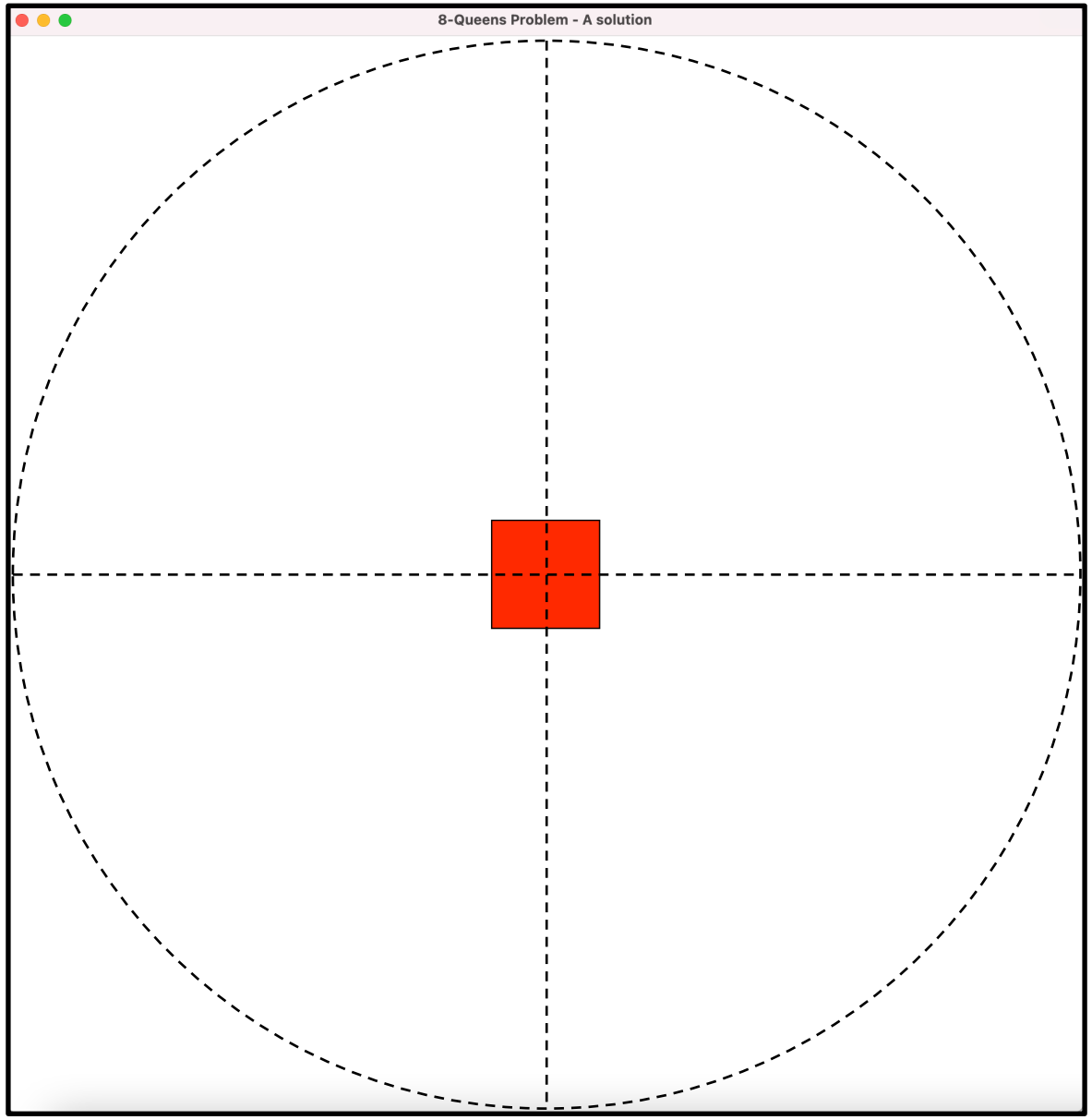
```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

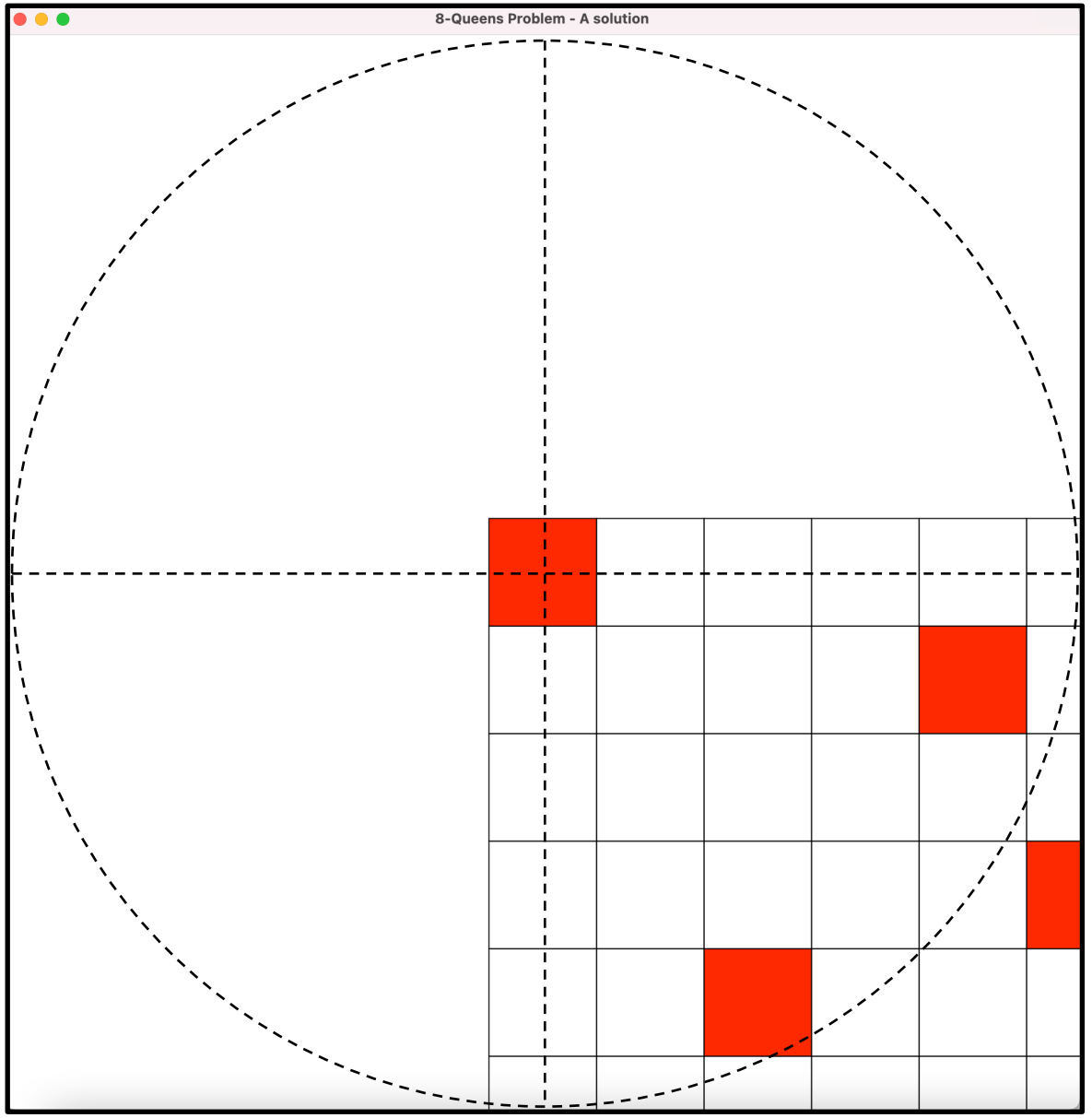
```
val beside = Monoid.instance[Image](Image.empty, _ beside _)
val above  = Monoid.instance[Image](Image.empty, _ above _)
```

```
given Monoid[Image] = Monoid.instance[Image](Image.empty, _ on _)
```

```
def createSquareImages(squareSize: Int): (Image,Image) =
  val square = Image.square(squareSize).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```

On the left is where the square images end up when we create them, i.e. all at the origin, the last one obscuring all the others, and on the right is where they end up after we position them relative to the first one, according to their position in the grid.





Now let's move on to the remaining functions in the program: **main** and **showQueens**. While **main** doesn't need to change, **showQueens** needs to do, for the whole board, what the **show** function does for individual squares in the board, i.e. it has to **position** the whole board image so that it is in the middle of the **Frame**, rather than where it is as a result of creating individual squares and **positioning** them relative to each other, i.e. at the coordinate origin (0,0).

On the left we see the current **showQueens** function, and on the right, we see the modified one.

```
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

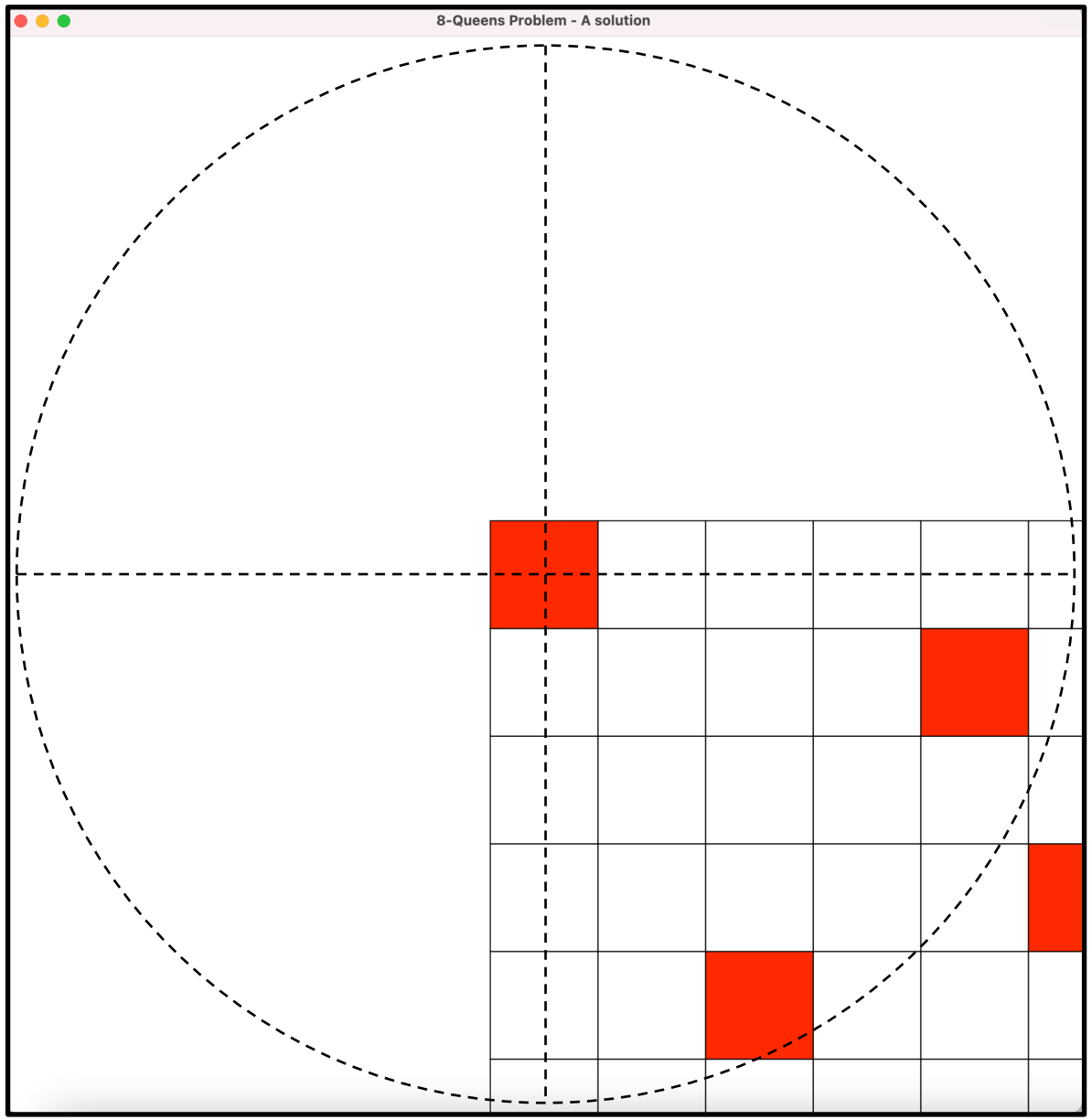
```
def showQueens(solution: List[Int]): Unit =
  val n = solution.length
  val frameTitle = s"{n}-Queens Problem – A solution"
  val frameWidth = 1000
  val frameHeight = 1000
  val frameBackgroundColour = Color.white
  val frame =
    Frame.size(frameWidth,frameHeight)
         .title(frameTitle)
         .background(frameBackgroundColour)
  show(solution).draw
```

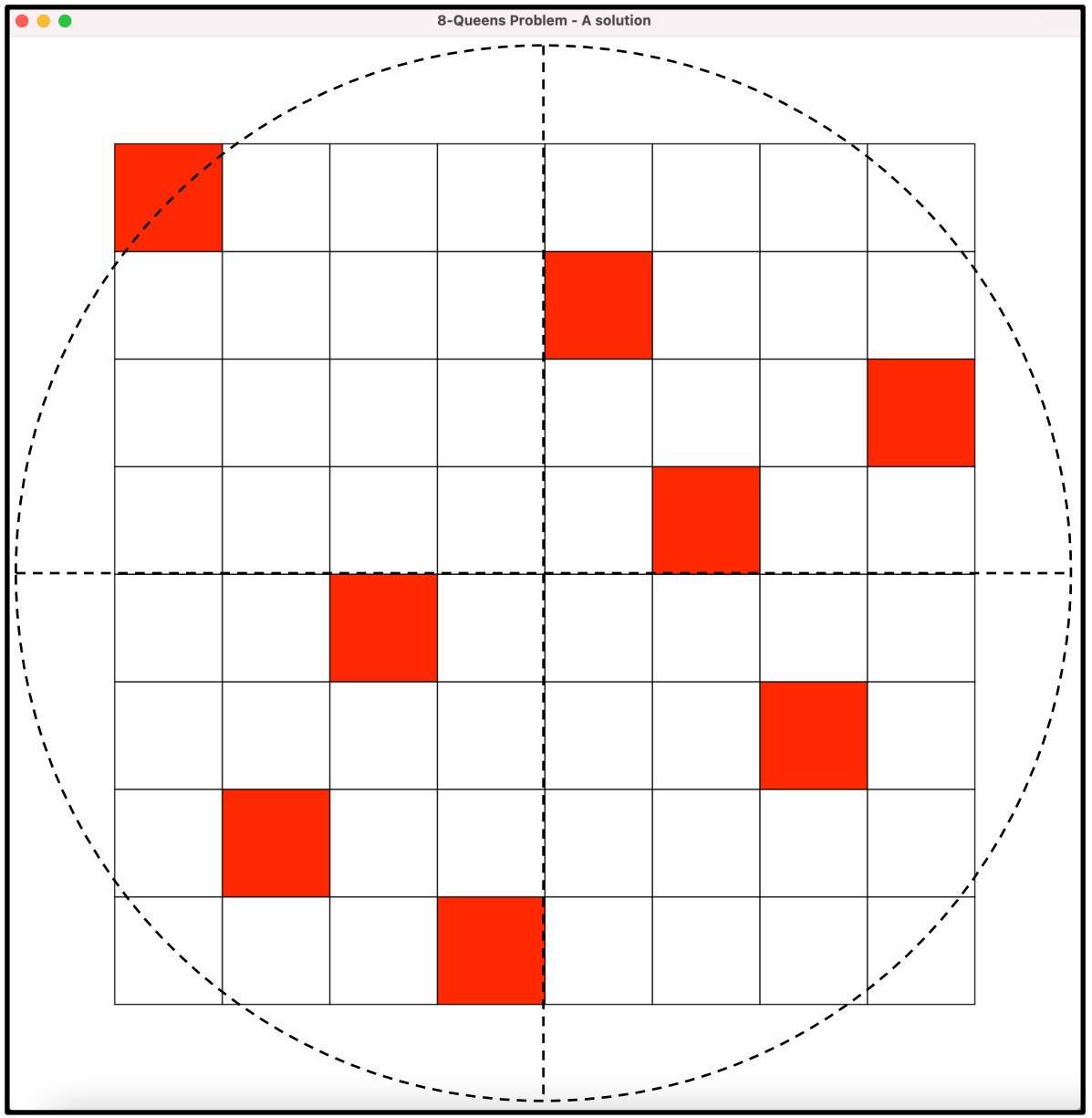
```
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

```
def showQueens(solution: List[Int]): Unit =
  val n = solution length
  val squareSize = 100
  val boardSize = n * squareSize
  val boardX = - (boardSize - squareSize) / 2
  val boardY = - boardX
  val frame = createFrame(n, squareSize)
  val boardImageAtOrigin = show(solution, squareSize)
  val boardImage = boardImageAtOrigin.at(boardX,boardY)
  boardImage.draw(frame)
```

```
def createFrame(n: Int, squareSize: Int): Frame =
  val title = s"${n}-Queens Problem - A solution"
  val backgroundColour = Color.white
  val boardSize = n * squareSize
  val width = boardSize + (squareSize * 2)
  val height = width
  Frame.size(width, height)
       .title(title)
       .background(backgroundColour)
```

On the left we see where the **show** function places the square images before combining them into a board image, and on the right, we see that the **showQueens** function repositions the board image so that it is in the middle of the frame.





```scala
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

Here is the modified **Scala** program in its entirety.

@philip_schwarz

```scala
def showQueens(solution: List[Int]): Unit =
  val n = solution length
  val squareSize = 100
  val boardSize = n * squareSize
  val boardX = - (boardSize - squareSize) / 2
  val boardY = - boardX
  val frame = createFrame(n, squareSize)
  val boardImageAtOrigin = show(solution, squareSize)
  val boardImage = boardImageAtOrigin.at(boardX,boardY)
  boardImage.draw(frame)
```

```scala
def createFrame(n: Int, squareSize: Int): Frame =
  val title = s"${n}-Queens Problem - A solution"
  val backgroundColour = Color.white
  val boardSize = n * squareSize
  val width = boardSize + (squareSize * 2)
  val height = width
  Frame.size(width, height)
       .title(title)
       .background(backgroundColour)
```

```scala
def show(queens: List[Int], squareSize: Int): Image =
  val n = queens length
  val solution = queens reverse
  val (emptySquare, fullSquare) = createSquareImages(squareSize)
  val squareImages: List[Image] =
    for
      row <- List.range(0, n)
      col <- List.range(0, n)
      squareX = col * squareSize
      squareY = - row * squareSize
      squareImageAtOrigin = if solution(row) == col then fullSquare else emptySquare
      squareImage = squareImageAtOrigin.at(squareX,squareY)
    yield squareImage
  val squareImageGrid = (squareImages grouped n).toList
  combine(squareImageGrid)
```

```scala
def createSquareImages(squareSize: Int): (Image,Image) =
  val square = Image.square(squareSize).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```

```scala
given Monoid[Image] = Monoid.instance[Image](Image.empty, _on_)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

Let's now turn to the task of translating our **Scala** program into **Haskell**.

To do the graphics, we are going to use a library called **`Gloss`**, whose documentation says the following:

- Gloss **hides the pain** of drawing simple vector graphics behind a nice data type and a few display functions.
- Get something cool on the screen in **under 10 minutes**.

The next slide is a very brief introduction to some of the few concepts that we'll need to draw a solution board.

hackage.haskell.org/package/gloss-1.13.2.1/docs/Graphics-Gloss.html

gloss-1.13.2.1: Painless 2D vector graphics, animations and simulations. Instances

Graphics.Gloss

Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions.

Getting something on the screen is as easy as:

```
import Graphics.Gloss
main = display (InWindow "Nice Window" (200, 200) (10, 10)) white (Circle 80)
```

Graphics.Gloss

Safe Haskell None
Language Haskell2010

Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions.

Getting something on the screen is as easy as:

```
import Graphics.Gloss
main = display (InWindow "Nice Window" (200, 200) (10, 10)) white (Circle 80)
```

Graphics.Gloss.Interface.Pure.Display

Safe Haskell None
Language Haskell2010

Display mode is for drawing a static picture.

Documentation

```
module Graphics.Gloss.Data.Display
```

```
module Graphics.Gloss.Data.Picture
```

```
module Graphics.Gloss.Data.Color
```

display

```
:: Display      Display mode.
-> Color        Background color.
-> Picture      The picture to draw.
-> IO ()
```

Open a new window and display the given picture.

Graphics.Gloss.Data.Display

Safe Haskell Safe-Inferred
Language Haskell2010

Documentation

```
data Display                                                    # Source
```

Describes how Gloss should display its output.

Constructors

`InWindow String (Int, Int) (Int, Int)`	Display in a window with the given name, size and position.
`FullScreen`	Display full screen.

First we create a `Picture`, e.g. a `Circle` with a radius of 80 pixels.

A picture is drawn on a `Display`, so we create a `Display` that consists of a window that has the desired title and is of the desired size (width an height) and is located at the desired position (x and y coordinates).

To draw a picture, we call the `display` function with a `Display`, the desired background colour for the `Display`, and the `Picture` to be drawn on the `Display`.

Just like the `drawInWindow` function in the `SOEGraphics` library, the `display` function in the `Gloss` library does not produce any **side effects**: it returns an **`IO action`**.

We can translate the Scala createSquareImages function into Haskell by using Gloss functions rectangleWire, rectangleSolid, and pictures.

```
def createSquareImages(squareSize: Int): (Image,Image) =
  val square = Image.square(squareSize).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```

rectangleWire :: Float -> Float -> Picture

A wireframe rectangle centered about the origin.

rectangleSolid :: Float -> Float -> Picture

A solid rectangle centered about the origin.

Gloss

pictures :: [Picture] -> Picture

A picture consisting of several others.

color :: Color -> Picture -> Picture

A picture drawn with this color.

```
createSquareImages :: Int -> (Picture, Picture)
createSquareImages squareSize = (emptySquare, fullSquare)
  where square = rectangleSolid (fromIntegral squareSize) (fromIntegral squareSize)
        squareBorder = rectangleWire (fromIntegral squareSize) (fromIntegral squareSize)
        emptySquare = pictures [color white square, color black squareBorder]
        fullSquare = pictures [color red square, color black squareBorder]
```

As for the **combine** function, it is very easy to translate it into **Haskell**, because while when using **Doodle** we had to define a **monoid** that combines images by superimposing them, **Gloss** already defines such a **monoid**, so there is even less code to write.

```
given Monoid[Image] = Monoid.instance[Image](Image.empty, _on_)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

▿ Monoid Picture

Defined in Graphics.Gloss.Internals.Data.Picture

Methods

```
mempty :: Picture

mappend :: Picture -> Picture -> Picture

mconcat :: [Picture] -> Picture
```

```
instance Monoid Picture where
  mempty       = Blank
  mappend a b = Pictures [a, b]
  mconcat      = Pictures
```

Gloss

Pictures [Picture]	A picture consisting of several others.

fold,combineAll	fold,mconcat
foldMap	foldMap
foldLeft	foldl
foldRight	foldr
combine	mappend
empty	mempty

```
combine :: [[Picture]] -> Picture
combine imageGrid = foldMap fold imageGrid
```

fold and **mconcat** are equivalent, so we could have used **mconcat** if we wanted to.

@philip_schwarz

Now let's turn to the heart of the program. Here is the **Scala** **show** function, together with its **Haskell** equivalent.

While in **Doodle** we position an image by using an image's **at** function, in **Gloss**, we do that using the **translate** function.

```scala
def show(queens: List[Int], squareSize: Int): Image =
  val n = queens length
  val solution = queens reverse
  val (emptySquare, fullSquare) = createSquareImages(squareSize)
  val squareImages: List[Image] =
    for
      row <- List.range(0, n)
      col <- List.range(0, n)
      squareX = col * squareSize
      squareY = - row * squareSize
      squareImageAtOrigin = if solution(row) == col then fullSquare else emptySquare
      squareImage = squareImageAtOrigin.at(squareX,squareY)
    yield squareImage
  val squareImageGrid = (squareImages grouped n).toList
  combine(squareImageGrid)
```

```haskell
show' :: [Int] -> Int -> Picture
show' queens squareSize = combine squareImageGrid
  where n = length queens
        solution = reverse queens
        (emptySquare, fullSquare) = createSquareImages squareSize
        squareImages =
          do
            row <- [0..n-1]
            col <- [0..n-1]
            let squareX = col * squareSize
                squareY = - row * squareSize
                squareImageAtOrigin = if (solution !! row) == col then fullSquare else emptySquare
                squareImage = translate (fromIntegral squareX) (fromIntegral squareY) squareImageAtOrigin
            return squareImage
        squareImageGrid = chunksOf n squareImages
```

Gloss

translate :: Float -> Float -> Picture -> Picture

A picture translated by the given x and y coordinates.

And here is the translation of the rest of the program from **Scala** to **Haskell**.

```scala
@main def main =
  val solution = List(3, 1, 6, 2, 5, 7, 4, 0)
  showQueens(solution)
```

```haskell
main :: IO ()
main = showQueens solution
  where solution = [3, 1, 6, 2, 5, 7, 4, 0]
```

```scala
def showQueens(solution: List[Int]): Unit =
  val n = solution length
  val squareSize = 100
  val boardSize = n * squareSize
  val boardX = - (boardSize - squareSize) / 2
  val boardY = - boardX
  val frame = createFrame(n, squareSize)
  val boardImageAtOrigin = show(solution, squareSize)
  val boardImage = boardImageAtOrigin.at(boardX,boardY)
  boardImage.draw(frame)
```

```haskell
showQueens :: [Int] -> IO ()
showQueens solution = display window backgroundColour boardImage
  where n = length solution
        squareSize = 100
        boardSize = n * squareSize
        boardX = - fromIntegral (boardSize - squareSize) / 2
        boardY = - boardX
        window = createWindowDisplay n squareSize
        backgroundColour = white
        boardImageAtOrigin = show' solution squareSize
        boardImage = translate boardX boardY boardImageAtOrigin
```

```scala
def createFrame(n: Int, squareSize: Int): Frame =
  val title = s"${n}-Queens Problem - A solution"
  val backgroundColour = Color.white
  val boardSize = n * squareSize
  val width = boardSize + (2 * squareSize)
  val height = width
  Frame.size(width, height)
       .title(title)
       .background(backgroundColour)
```

```haskell
createWindowDisplay :: Int -> Int -> Display
createWindowDisplay n squareSize = InWindow title (width, height) position
  where title = (show n) ++ "-Queens Problem - A solution"
        boardSize = n * squareSize
        width = boardSize + (2 * squareSize)
        height = width
        position = (0,0)
```

```haskell
main :: IO ()
main = showQueens solution
where solution = [3, 1, 6, 2, 5, 7, 4, 0]
```

Here is the **Haskell** program in its entirety.

```haskell
showQueens :: [Int] -> IO ()
showQueens solution = display window backgroundColour boardImage
  where n = length solution
        squareSize = 100
        boardSize = n * squareSize
        boardX = - fromIntegral (boardSize - squareSize) / 2
        boardY = - boardX
        window = createWindowDisplay n squareSize
        backgroundColour = white
        boardImageAtOrigin = show' solution squareSize
        boardImage = translate boardX boardY boardImageAtOrigin
```

```haskell
createWindowDisplay :: Int -> Int -> Display
createWindowDisplay n squareSize = InWindow title (width,height) position
  where title = (show n) ++ "-Queens Problem - A solution"
        boardSize = n * squareSize
        width = boardSize + (2 * squareSize)
        height = width
        position = (0,0)
```

```haskell
show' :: [Int] -> Int -> Picture
show' queens squareSize = combine squareImageGrid
  where n = length queens
        solution = reverse queens
        (emptySquare, fullSquare) = createSquareImages squareSize
        squareImages =
          do
            row <- [0..n-1]
            col <- [0..n-1]
            let squareX = col * squareSize
                squareY = - row * squareSize
                squareImageAtOrigin = if (solution !! row) == col then fullSquare else emptySquare
                squareImage = translate (fromIntegral squareX) (fromIntegral squareY) squareImageAtOrigin
            return squareImage
        squareImageGrid = chunksOf n squareImages
```

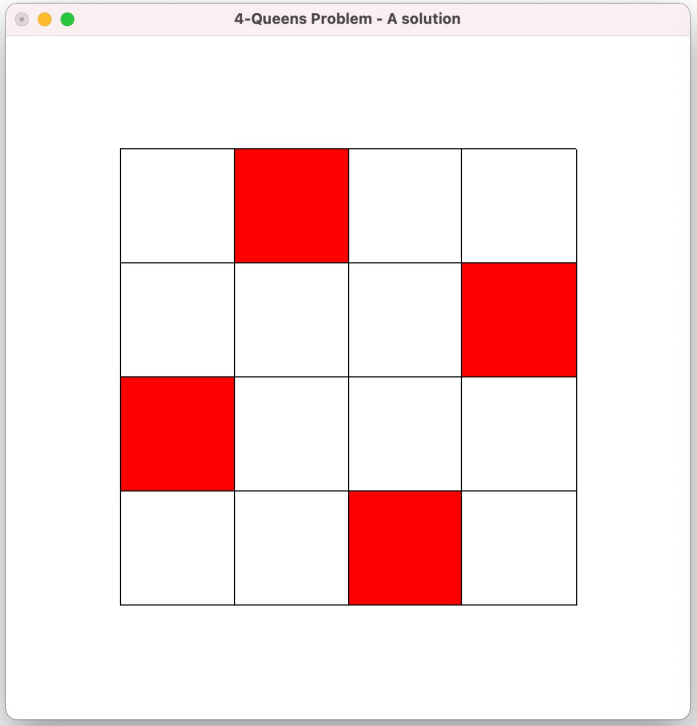
```haskell
combine :: [[Picture]] -> Picture
combine imageGrid = foldMap fold imageGrid
```

```haskell
createSquareImages :: Int -> (Picture, Picture)
createSquareImages squareSize = (emptySquare, fullSquare)
  where square = rectangleSolid (fromIntegral squareSize) (fromIntegral squareSize)
        squareBorder = rectangleWire (fromIntegral squareSize) (fromIntegral squareSize)
        emptySquare = pictures [color white square, color black squareBorder]
        fullSquare = pictures [color red square, color black squareBorder]
```

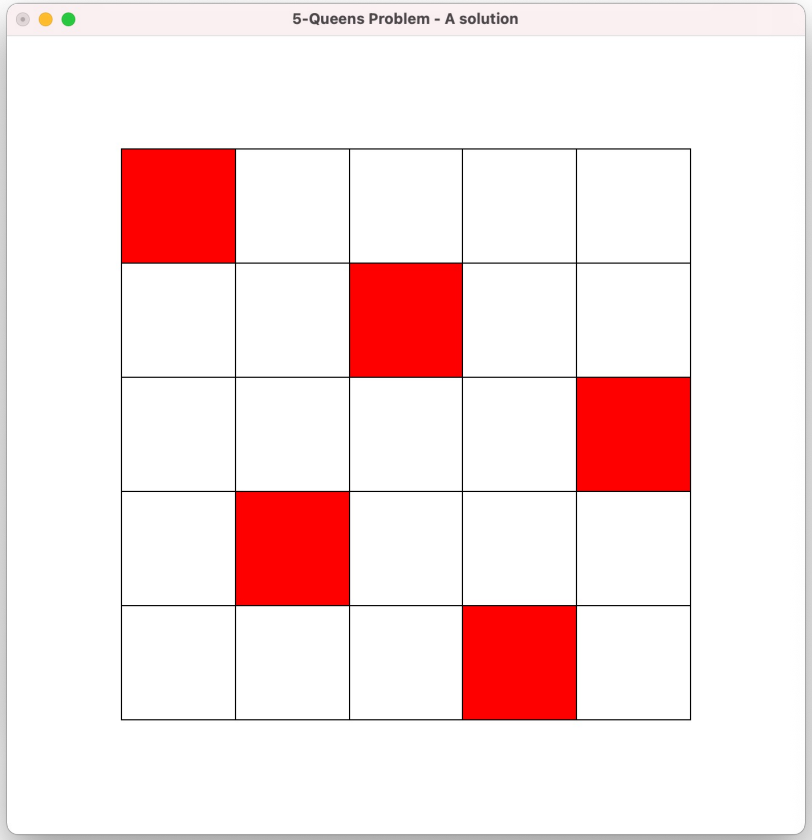
Here are the results of the **Haskell** program for the first solution for each of N=4,5,6.

@philip_schwarz

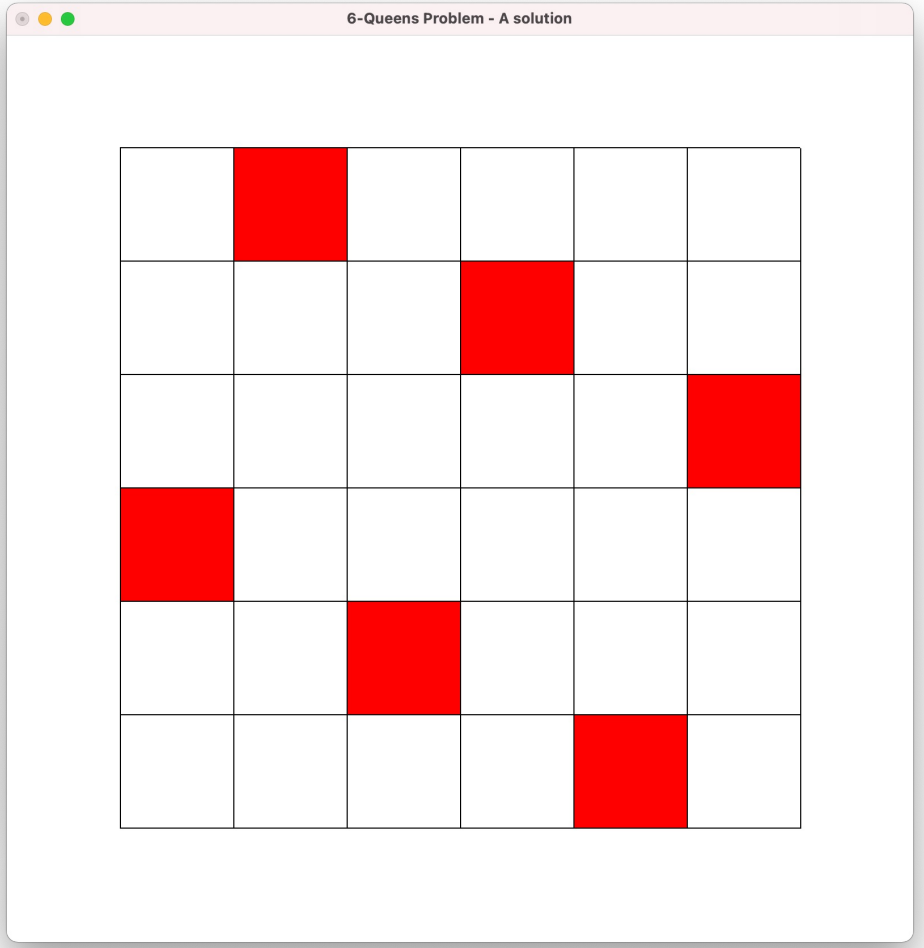
[2, 0, 3, 1]



[3, 1, 4, 2, 0]



[4, 2, 0, 5, 3, 1]



To conclude Part 2, here is a reminder, from Part 1, of the translation of the program for computing the **N-Queens Problem**.

```scala
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if safe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```

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

```scala
def safe(queen: Int, queens: List[Int]): Boolean =
  val (row, column) = (queens.length, queen)
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)
  zipWithRows(queens) forall safe
```

```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
```

```scala
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =
  math.abs(row - otherRow) ==  math.abs(column - otherColumn)
```

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

```scala
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =
  val rowCount = queens.length
  val rowNumbers = rowCount - 1 to 0 by -1
  rowNumbers zip queens
```

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

And here is a reminder of the output it produces.

```
scala> queens(4)
val res0: List[List[Int]] = List(List(3, 1, 4, 2), List(2, 4, 1, 3))

scala> queens(5)
val res1: List[List[Int]] = List(List(4, 2, 5, 3, 1), List(3, 5, 2, 4, 1), List(5, 3, 1, 4, 2), List(4, 1, 3, 5, 2),
                                 List(5, 2, 4, 1, 3), List(1, 4, 2, 5, 3), List(2, 5, 3, 1, 4), List(1, 3, 5, 2, 4),
                                 List(3, 1, 4, 2, 5), List(2, 4, 1, 3, 5))

scala> queens(6)
val res2: List[List[Int]] = List(List(5, 3, 1, 6, 4, 2), List(4, 1, 5, 2, 6, 3),
                                 List(3, 6, 2, 5, 1, 4), List(2, 4, 6, 1, 3, 5))
```

```
haskell> queens 4
[[3,1,4,2],[2,4,1,3]]

haskell> queens 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],
 [5,2,4,1,3],[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],
 [3,1,4,2,5],[2,4,1,3,5]]

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

@philip_schwarz

That's all for Part 2. I hope you found it interesting.

The first thing we are going to do in part 3 is write code that displays, all together, the results of **queens**(N) for N = 4, 5, 6, 7, 8.

We are then going to look at some alternative ways of coding the **N-Queens Problem**.

See you then.