N-Queens Combinatorial Problem

Polyglot **FP** for **F**un and **P**rofit – **Haskell** and **Scala**

Learn how to write **FP** code that displays a **graphical representation** of all the numerous **N-Queens** solutions for **N**=**4**,**5**,**6**,**7**,**8**

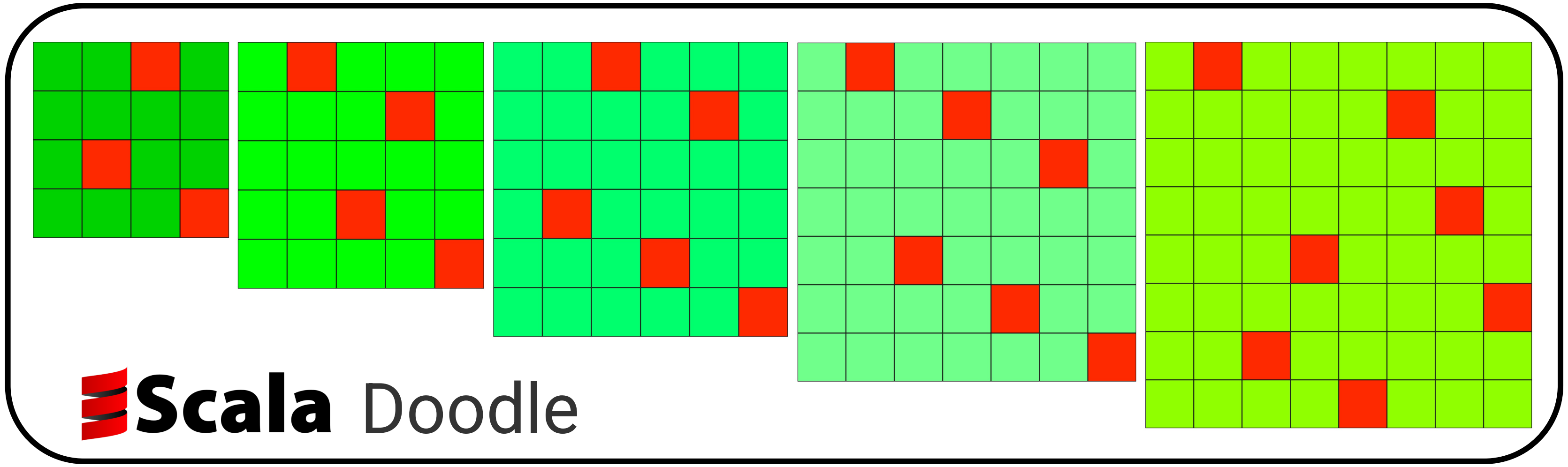
See how to neatly solve the problem by exploiting its **self-similarity** and using a **divide and conquer** approach

Make light work of assembling multiple images into a whole, by exploiting **Doodle**'s facilities for **combining** images using a **relative layout**

See relevant **FP** functions, like **Foldable**'s **intercalate** and **intersperse**, in action

Part 3

scala cats



SCALAZ

slides by

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

Welcome to **Part 3** of this series. In this part, we are going to write a new program that displays, all together, the results of **queens**(**N**) for **N** = **4**, **5**, **6**, **7**, **8**.

The next slide shows both the program (from **Part 2**) that displays the board for a single solution, and the beginnings of the new program, which will reuse some logic from both **Part 1** and **Part 2**.

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

```scala
def showQueens(solution: List[Int]): Int =
  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)
```

```scala
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)
```

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

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

The program on the left is from **Part 2**. It displays the board for a single solution. The program on the right uses the **queens** function (and ancillary functions) from **Part 1**. It displays, all together, the results of **queens**(N) for N = 4, 5, 6, 7, 8.

```scala
@main def main =
  val ns = List(4,5,6,7,8)
  ns map queens pipe makeResultsImage pipe display(ns)
```

See next slide for an explanation of **pipe**.

```scala
def display(ns: List[Int])(image: Image): Unit =
  val frameTitle = "N-Queens Problem - Solutions for N = ${ns.mkString(",")}"
  val frameWidth = 1800
  val frameHeight = 1000
  val frameBackgroundColour = Color.white
  val frame =
    Frame.size(frameWidth,frameHeight)
         .title(frameTitle)
         .background(frameBackgroundColour)
  image.draw(frame)
```

```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)
```

We are switching from the program on the left, to the one on the right. New code is on a green background.

In the new program, generating an image is the responsibility of **makeResultsImage**.

```scala
val makeResultsImage: List[List[List[Int]]] => Image = ??? // to be implemented
```

```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
```

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

@philip_schwarz

Scala's pipe function allows us to take an expression consisting of a number of nested function invocations, e.g. f(g(h(x))), and turn it into an equivalent expression in which the functions appear in the order in which they are invoked, i.e. h, g and f, rather than in the inverse order, i.e. f, g and h.

```
def pipe[B](f: (A) => B): B
```

Converts the value by applying the function `f`.

B	the result type of the function `f`.
f	the function to apply to the value.
returns	a new value resulting from applying the given function `f` to this value.

scala.util

ChainingOps

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

Adds chaining methods `tap` and `pipe` to every type.

Here is one example

```
def inc(n: Int): Int = n + 1
def twice(n: Int): Int = n * 2
def square(n: Int): Int = n * n
```

```
assert(square(twice(inc(3))) == 64)
```

```
assert ((3 pipe inc pipe twice pipe square) == 64)
```

We are using pipe to make our main function easier to understand

```
@main def main =
  val ns = List(4,5,6,7,8)
  display(ns)(makeResultsImage(ns map queens))
```

```
@main def main =
  val ns = List(4,5,6,7,8)
  ns map queens pipe makeResultsImage pipe display(ns)
```

Our current objective is to implement the **makeResultsImage** function invoked by **main**.

```
@main def main =
  val ns = List(4,5,6,7,8)
  ns map queens pipe makeResultsImage pipe display(ns)
```

Let's begin by introducing a couple of type aliases to aid comprehension.

```
val makeResultsImage: List[List[List[Int]]] => Image = ???
```

```
type Solution = List[Int]
```

```
val makeResultsImage: List[List[Solution]] => Image = ???
```

```
type Solutions = List[Solution]
```

```
val makeResultsImage: List[Solutions] => Image = ???
```

If at some point, while reading the next three slides, you feel a strong sense of **déjà vu**, that is to be expected.

There is a lot of **symmetry** between the slides, and the code that they contain.

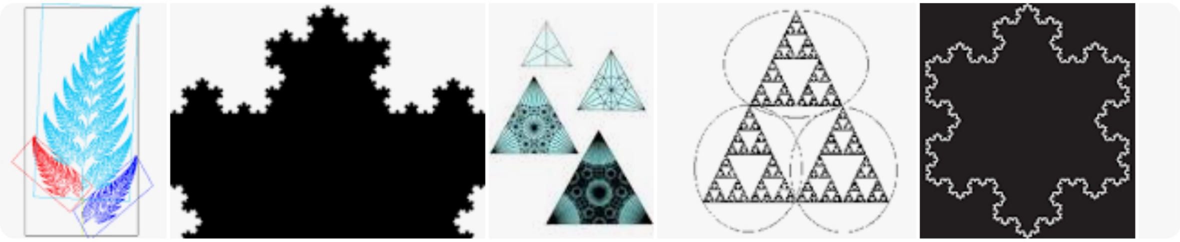
That's because the problem that we are working on exhibits a good degree of **self-similarity**.

```
type Solution  = List[Int]
type Solutions = List[Solution]
```

The problem looks very similar at three different levels:

- When we operate at the **single Solution** level, we need to create an image of a **grid** of **squares** (a **solution board**).
- When we operate at the **multiple Solution** level, we need to create an image of a **grid** of **boards** (the **solution boards** for some **N**).
- When we operate at the **multiple Solutions** level, we need to create an image of a **grid** of **grids** of **boards** (i.e. all the **solution boards** for **N**=**4**,**5**,**6**,**7**,**8**).

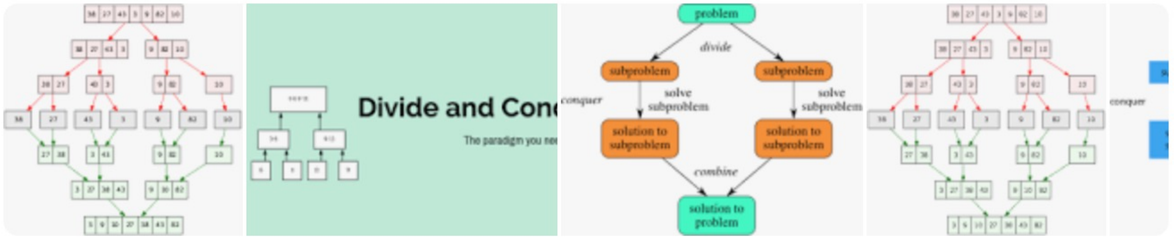
If things appear to get a bit confusing at times, keep a cool head by focusing on the function signatures at play, and by reminding yourself that all we are doing is **divide and conquer**.



In mathematics, a self-similar object is **exactly or approximately similar to a part of itself** (i.e., the whole has the same shape as one or more of the parts). ... Scale invariance is an exact form of self-similarity where at any magnification there is a smaller piece of the object that is similar to the whole.

https://en.wikipedia.org › wiki › Self-similarity

Self-similarity - Wikipedia



A divide-and-conquer algorithm recursively breaks down a **problem** into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

https://en.wikipedia.org › wiki › Divide-and-conquer_alg...

Divide-and-conquer algorithm - Wikipedia

To turn multiple Solutions elements into an image, we are going to first create an Image for each Solutions element, then arrange the resulting images in a Grid, and finally combine the images into a single compound image, adding padding around images as we combine them.

```scala
type Grid[A]    = List[List[A]]
type Solution   = List[Int]
type Solutions  = List[Solution]
```

```scala
val makeResultsImage: List[Solutions] => Image = makeSolutionsImageGrid andThen combineWithPadding
```

Creating the images, and arranging them into a grid, will be done by makeSolutionsImageGrid, whereas combining the images into a single compound image, inserting padding around the images, will be done by combineWithPadding.

```scala
makeSolutionsImageGrid : List[Solutions] => Grid[Image]
```

```scala
combineWithPadding : Grid[Image] => Image
```

In order to create a Grid[Image], makeSolutionsImageGrid must create an image for each Solutions element.

To help with that, on the next slide we define a function called makeSolutionsImage, which given a Solutions element, returns an Image.

This is analogous to makeResultsImage, but operates on an individual Solutions element rather than on a list of such elements, so it operates one level below makeResultsImage.

To turn multiple **Solution** elements into an image, we are going to first create an **Image** for each **Solution**, then arrange the images in a **Grid**, and finally **combine** the images into a single **compound** image, adding **padding** around images as we combine them.

```
type Grid[A]    = List[List[A]]
type Solution   = List[Int]
type Solutions  = List[Solution]
```

```
val makeSolutionsImage: List[Solution] => Image = makeBoardImageGrid andThen combineWithPadding
```

Creating the images, and arranging them into a grid, will be done by **makeBoardImageGrid**, whereas **combining** the images into a single **compound** image, inserting **padding** around the images, will be done by **combineWithPadding** (yes, we introduced it on the previous slide).

```
makeBoardImageGrid : List[Solution] => Grid[Image]
```

```
combineWithPadding : Grid[Image] => Image
```

In order to create a **Grid**[**Image**], **makeBoardImageGrid** must create an image for each **Solution** element.

To help with that, on the next slide we define a function called **makeBoardImage**, which given a **Solution** element, returns an **Image**.

This is analogous to **makeSolutionsImage**, but operates on an individual **Solution** element rather than on a list of such elements, so it operates one level below **makeSolutionsImage**.

To turn a Solution, which represents a chess board, into an image, we are going to first create an Image for each square in the Solution, then arrange the images in a Grid, and finally combine the images into a single compound image.

```
type Grid[A]     = List[List[A]]
type Solution    = List[Int]
type Solutions = List[Solution]
```

```
val makeBoardImage: Solution => Image = makeSquareImageGrid andThen combine
```

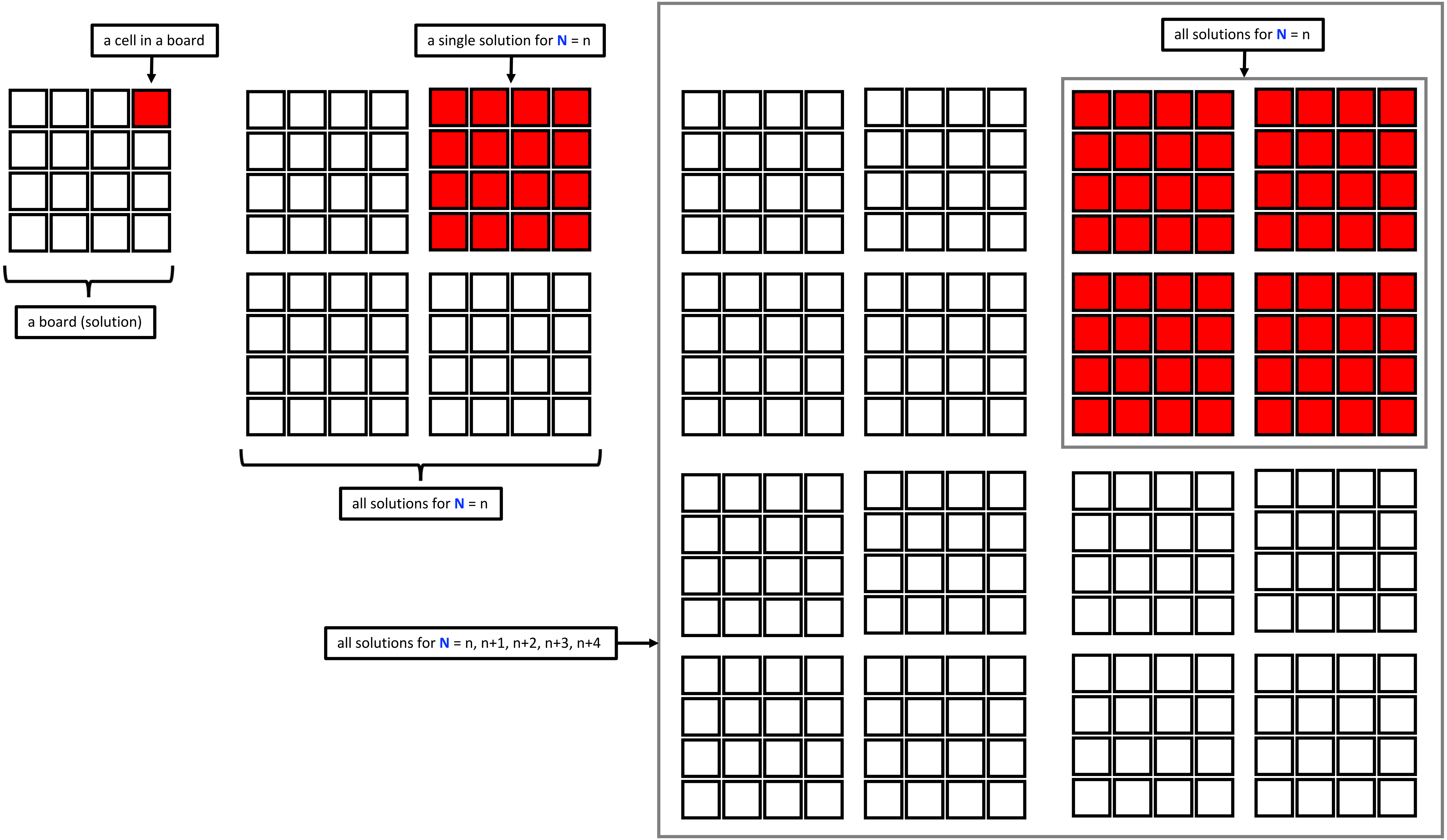
Creating the images, and arranging them into a grid, will be done by makeSquareImageGrid, whereas combining the images into a single compound image will be done by combine (yes, we implemented such a function in **Part 2**).

```
makeSquareImageGrid : Solution => Grid[Image]
```

```
combine : Grid[Image] => Image
```

The next slide visualises the self-similarity of the problem we are working on, and its amenability to a divide and conquer approach.

@philip_schwarz


a cell in a board
a board (solution)
a single solution for N = n
all solutions for N = n
all solutions for N = n, n+1, n+2, n+3, n+4
all solutions for N = n

Here are the functions that we have identified so far.

We already have implementations for the first three functions.

On the next slide, we start implementing the next three functions, which create grids of images.

```scala
type Grid[A]     = List[List[A]]
type Solution    = List[Int]
type Solutions = List[Solution]
```

```scala
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combineWithPadding
```

Create an image of a grid of grids of boards (i.e. all the solution boards for N=4,5,6,7,8)

```scala
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combineWithPadding
```

Create an image of a grid of boards (the solution boards for some N)

```scala
val makeBoardImage: Solution => Image =
  makeSquareImageGrid andThen combine
```

Create an image of a grid of squares (a solution board)

```scala
makeSolutionsImageGrid: List[Solutions] => Grid[Image]
```

Create a grid of images of grids of boards (i.e. all the solution boards for N=4,5,6,7,8)

```scala
makeBoardImageGrid: List[Solution] => Grid[Image]
```

Create a grid of board images (the solution boards for some N)

```scala
makeSquareImageGrid: Solution => Grid[Image]
```

Create a grid of square images (a solution board)

```scala
combineWithPadding: Grid[Image] => Image
```

Combine a grid of images into a composite image with padding around the images

```scala
combine: Grid[Image] => Image
```

Combine a grid of images into a composite image with no padding around the images

Since all three of these functions have to create a grid, they will have some logic in common.

Let's put that shared logic in a function called **makeImageGrid**.

```
makeSolutionsImageGrid: List[Solutions] => Grid[Image]
```

```
makeBoardImageGrid: List[Solution] => Grid[Image]
```

```
makeSquareImageGrid: Solution => Grid[Image]
```

```
def makeImageGrid[A](as: List[A], makeImage: A => Image, gridWidth: Int): Grid[Image] =
  as map makeImage grouped gridWidth toList
```

Implementing **makeSolutionsImageGrid** and **makeBoardImageGrid** is now simply a matter of invoking **makeImageGrid**.

```
type Grid[A]     = List[List[A]]
type Solution    = List[Int]
type Solutions = List[Solution]
```

```
def makeSolutionsImageGrid(queensResults: List[Solutions]): Grid[Image] =
  makeImageGrid(queensResults, makeSolutionsImage, gridWidth = 1)
```

We are creating a degenerate grid, one with just **1** column. We are doing so simply because it happens to result in an effective layout.

```
def makeBoardImageGrid(solutions: List[Solution]): Grid[Image] =
  makeImageGrid(solutions, makeBoardImage, gridWidth = 17)
```

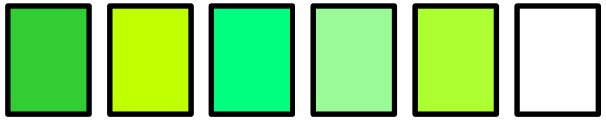
Again, it just happens that creating a grid with **17** columns results in a better layout of solution boards than is otherwise the case.

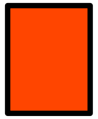
See the previous slide for implementations of **makeSolutionsImage** and **makeBoardImage**.

While Implementing **makeSolutionsImageGrid** and **makeBoardImageGrid** was simply a matter of invoking **makeImageGrid**, implementing **makeSquareImageGrid** is more involved.

```
def makeSquareImageGrid(columnIndices:Solution): Grid[Image] =
  val n = columnIndices.length
  val (emptySquare, fullSquare) = makeSquareImages(n)
  val occupiedCells: List[Boolean] =
    columnIndices.reverse flatMap { col => List.fill(n)(false).updated(col-1,true) }
  val makeSquareImage: Boolean => Image = if (_) fullSquare else emptySquare
  makeImageGrid(occupiedCells, makeSquareImage, gridWidth = n)
```

```
def makeSquareImages(n: Int): (Image,Image) =
  val emptySquareColour = n match
    case 4 => Color.limeGreen
    case 5 => Color.lime
    case 6 => Color.springGreen
    case 7 => Color.paleGreen
    case 8 => Color.greenYellow
    case other => Color.white
  val square: Image = Image.square(10).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(emptySquareColour)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```





To help distinguish the solution **boards** for different values of **N** (**4**,**5**,**6**,**7**,**8**), we are giving their **empty squares** of a different shade of green.

Looking back at the implementations of our three functions for creating images, now that we have implemented the functions that create image grids, it is time to implement **combine** and **combineWithPadding**.

```
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combineWithPadding
```

```
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combineWithPadding
```

```
val makeBoardImage: Solution => Image =
  makeSquareImageGrid andThen combine
```

```
combine: Grid[Image] => Image
```

```
combineWithPadding: Grid[Image] => Image
```

On the next slide we start implementing **combineWithPadding**.

@philip_schwarz

Remember the implementation of the **combine** function that we used, in **Part 2**, to take a grid of images and produce a **compound** image that is their **composition**?

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

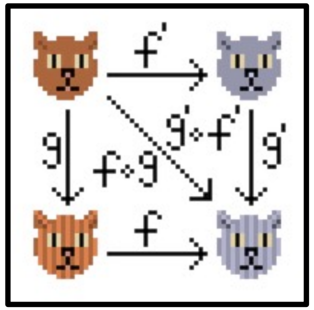
```scala
def combine(imageGrid: List[List[Image]]): Image =
  import cats.implicits._
  imageGrid.foldMap(_ combineAll beside)(above)
```

We need to implement **combineWithPadding**, a function that differs from **combine** in that instead of just **combining** the images contained in its image grid parameter, it also needs to insert a **padding image** between neighbouring images as it does that.

```scala
combineWithPadding: Grid[Image] => Image
```

The **combine** function first **folds** the images in a row (**combineAll** is just an alias for **fold**) using the **beside monoid**, and then **folds** the resulting row images using the **above monoid**. The **combineWithPadding** function needs to **fold** images in the same way, but in addition, it also needs to insert a **padding image** between each pair of such images.

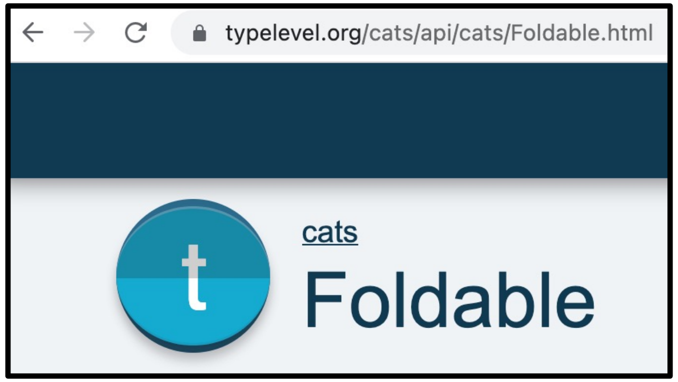
This is clearly a job for the **intercalate** function provided by **Cats**' **Foldable** type class!




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

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

```scala
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
```



Let's go ahead and implement **combineWithPadding** using the **intercalate** function provided by **Cats**' **Foldable** type class!

The **padding** consists of a **white square** with a width of **10** pixels.

```
def combineWithPadding(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage)

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

val paddingImage = Image.square(10).strokeColor(Color.white).fillColor(Color.white)
```

Here again, for reference, is how we implemented **combine** in **Part 2**.

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

What about the **combine** function, which doesn't do any **padding**? Why is it that a couple of slides ago we said that we need to implement it? We have already implemented it in **Part 2** (see above).

While we can certainly just use the above **combine** function, it is interesting to see how much simpler the implementation becomes if we leverage the **combineWithPadding** function that we have just introduced.

In case you find it useful, here is how the second **combineWithPadding** function looks like if we use **Foldable**[**List**]explicitly.

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

```
def combine(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage = Image.empty)
```

No **padding**, is just **padding** with an **empty image**. A bit silly maybe, but attractively simple.

@philip_schwarz

The next slide shows all of the code needed to display the **N-Queens** solutions for **N**=**4**,**5**,**6**,**7**,**8**.

While the **queens** function is included on the slide, this is purely to remind us of how the solutions are produced, and so its three subordinate functions are not shown.

```scala
@main def main =
  val ns = List(4,5,6,7,8)
  ns map queens pipe makeResultsImage pipe display(ns)
```

```scala
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combineWithPadding
```

```scala
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combineWithPadding
```

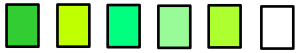
```scala
val makeBoardImage: Solution => Image =
  makeSquareImageGrid andThen combine
```

```scala
def makeSolutionsImageGrid(queensResults: List[Solutions]): Grid[Image] =
  makeImageGrid(queensResults, makeSolutionsImage, gridWidth = 1)
```

```scala
def makeBoardImageGrid(solutions: List[Solution]): Grid[Image] =
  makeImageGrid(solutions, makeBoardImage, gridWidth = 17)
```

```scala
def makeSquareImageGrid(columnIndices:Solution): Grid[Image] =
  val n = columnIndices.length
  val (emptySquare, fullSquare) = makeSquareImages(n)
  val occupiedCells: List[Boolean] = columnIndices.reverse flatMap { col =>
    List.fill(n)(false).updated(col-1,true) }
  val makeSquareImage: Boolean => Image = if (_) fullSquare else emptySquare
  makeImageGrid(occupiedCells, makeSquareImage, gridWidth = n)
```

```scala
def makeSquareImages(n: Int): (Image,Image) =
  val emptySquareColour = n match
    case 4 => Color.limeGreen
    case 5 => Color.lime
    case 6 => Color.springGreen
    case 7 => Color.paleGreen
    case 8 => Color.greenYellow
    case other => Color.white
  val square: Image = Image.square(10).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(emptySquareColour)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```



```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)
```

```scala
type Grid[A]     = List[List[A]]
type Solution    = List[Int]
type Solutions = List[Solution]
```

```scala
def display(ns: List[Int])(image: Image): Unit =
  val frameTitle = "N-Queens Problem - Solutions for N = ${ns.mkString(",")}"
  val frameWidth = 1800
  val frameHeight = 1000
  val frameBackgroundColour = Color.white
  val frame = Frame.size(frameWidth,frameHeight)
                   .title(frameTitle)
                   .background(frameBackgroundColour)
  image.draw(frame)
```

```scala
def makeImageGrid[A](as: List[A], makeImage: A => Image, gridWidth: Int): Grid[Image] =
  as map makeImage grouped gridWidth toList
```

```scala
def combine(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage = Image.empty)

def combineWithPadding(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage)

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

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

val paddingImage = Image.square(10)
                        .strokeColor(Color.white)
                        .fillColor(Color.white)
```

While the code on the previous slide works, I think the **combineWithPadding** function is doing too much. It is unnecessarily conflating two responsibilities: inserting **padding** between images, and **combining** the images.

Let's delete the **combine** and **combineWithPadding** functions on the left, and reinstate the earlier **combine** function, on the right.

```scala
def combine(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage = Image.empty)

def combineWithPadding(images: Grid[Image]): Image =
  combineWithPadding(images, paddingImage)

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

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

We must now stop **makeResultsImage** and **makeSolutionsImage** from using the **combineWithPadding** function that we have deleted.

```scala
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combineWithPadding
```

```scala
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combineWithPadding
```

```scala
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combine
```

```scala
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combine
```

Now that **padding** no longer gets introduced when images are **combined**, when is it going to be introduced? See the next slide.

Let’s define a function called **insertPadding**, that takes a grid of images, and inserts a **padding image** between each pair of neighbouring images. We can implement the function using the **intersperse** function provided by https://github.com/scala/scala-collection-contrib.

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

scala.collection.decorators

IteratorDecorator

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

Enriches Iterator with additional methods.

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

Inserts a separator value between each element.

```scala
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.

Except that when I try to use the **intersperse** function with **Scala 3**, I get a compilation error, so I have opened an issue: https://github.com/scala/scala-collection-contrib/pull/146.

Luckily, the very same **intersperse** function is also available in **Scalaz**.

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

```scala
List(1, 2, 3) intersperse 0 assert_=== List(1,0,2,0,3)
List(1, 2) intersperse 0 assert_=== List(1,0,2)
List(1) intersperse 0 assert_=== List(1)
nil[Int] intersperse 0 assert_=== nil[Int]
```

Now, remember **makeImageGrid**, the function that we use to turn a list into a grid of images?

```scala
def makeImageGrid[A](as: List[A], makeImage: A => Image, gridWidth: Int): Grid[Image] =
  as map makeImage grouped gridWidth toList
```

Now that we have defined **insertPadding**, we can use it to define a variant of **makeImageGrid** which, in addition to creating a grid of images, inserts **padding** between those images.

```scala
def makePaddedImageGrid[A](as: List[A], makeImage: A => Image, gridWidth: Int): Grid[Image] =
  makeImageGrid(as, makeImage, gridWidth) pipe insertPadding
```

Armed with the above function, we can now remedy the fact that we have eliminated the **combineWithPadding** function. The **padding** that was previously inserted by using **combineWithPadding**, will now be inserted by invoking **makePaddedImageGrid** rather than **makeImageGrid**. We need to make the switch in the following two functions:

```scala
def makeSolutionsImageGrid(queensResults: List[Solutions]): Grid[Image] =
  makeImageGrid(queensResults, makeSolutionsImage, gridWidth = 1)
```

```scala
def makeBoardImageGrid(solutions: List[Solution]): Grid[Image] =
  makeImageGrid(solutions, makeBoardImage, gridWidth = 17)
```

The next slide applies the additions/changes described on this slide, to the code needed to display the **N-Queens** solutions for **N**=**4**,**5**,**6**,**7**,**8**.

```scala
@main def main =
  val ns = List(4,5,6,7,8)
  ns map queens pipe makeResultsImage pipe display(ns)
```

```scala
val makeResultsImage: List[Solutions] => Image =
  makeSolutionsImageGrid andThen combineWithPadding
```

```scala
val makeSolutionsImage: List[Solution] => Image =
  makeBoardImageGrid andThen combineWithPadding
```

```scala
val makeBoardImage: Solution => Image =
  makeSquareImageGrid andThen combine
```

```scala
def makeSolutionsImageGrid(queensResults: List[Solutions]): Grid[Image] =
  makePaddedImageGrid(queensResults, makeSolutionsImage, gridWidth = 1)
```

```scala
def makeBoardImageGrid(solutions: List[Solution]): Grid[Image] =
  makePaddedImageGrid(solutions, makeBoardImage, gridWidth = 17)
```

```scala
def makeSquareImageGrid(columnIndices:Solution): Grid[Image] =
  val n = columnIndices.length
  val (emptySquare, fullSquare) = makeSquareImages(n)
  val occupiedCells: List[Boolean] = columnIndices.reverse flatMap { col =>
    List.fill(n)(false).updated(col-1,true) }
  val makeSquareImage: Boolean => Image = if (_) fullSquare else emptySquare
  makeImageGrid(occupiedCells, makeSquareImage, gridWidth = n)
```

```scala
def makeSquareImages(n: Int): (Image,Image) =
  val emptySquareColour = n match
    case 4 => Color.limeGreen
    case 5 => Color.lime
    case 6 => Color.springGreen
    case 7 => Color.paleGreen
    case 8 => Color.greenYellow
    case other => Color.white
  val square: Image = Image.square(10).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(emptySquareColour)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  (emptySquare, fullSquare)
```

```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)
```

```scala
type Grid[A]     = List[List[A]]
type Solution  = List[Int]
type Solutions = List[Solution]
```

```scala
def display(ns: List[Int])(image: Image): Unit =
  val frameTitle = "N-Queens Problem - Solutions for N = ${ns.mkString(",")}"
  val frameWidth = 1800
  val frameHeight = 1000
  val frameBackgroundColour = Color.white
  val frame = Frame.size(frameWidth,frameHeight)
                   .title(frameTitle)
                   .background(frameBackgroundColour)
  image.draw(frame)
```

```scala
def makePaddedImageGrid[A](as:List[A],makeImage:A => Image,gridWidth:Int):Grid[Image] =
  makeImageGrid(as, makeImage, gridWidth) pipe insertPadding
```

```scala
def makeImageGrid[A](as: List[A], makeImage: A => Image, gridWidth: Int): Grid[Image] =
  as map makeImage grouped gridWidth toList
```

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

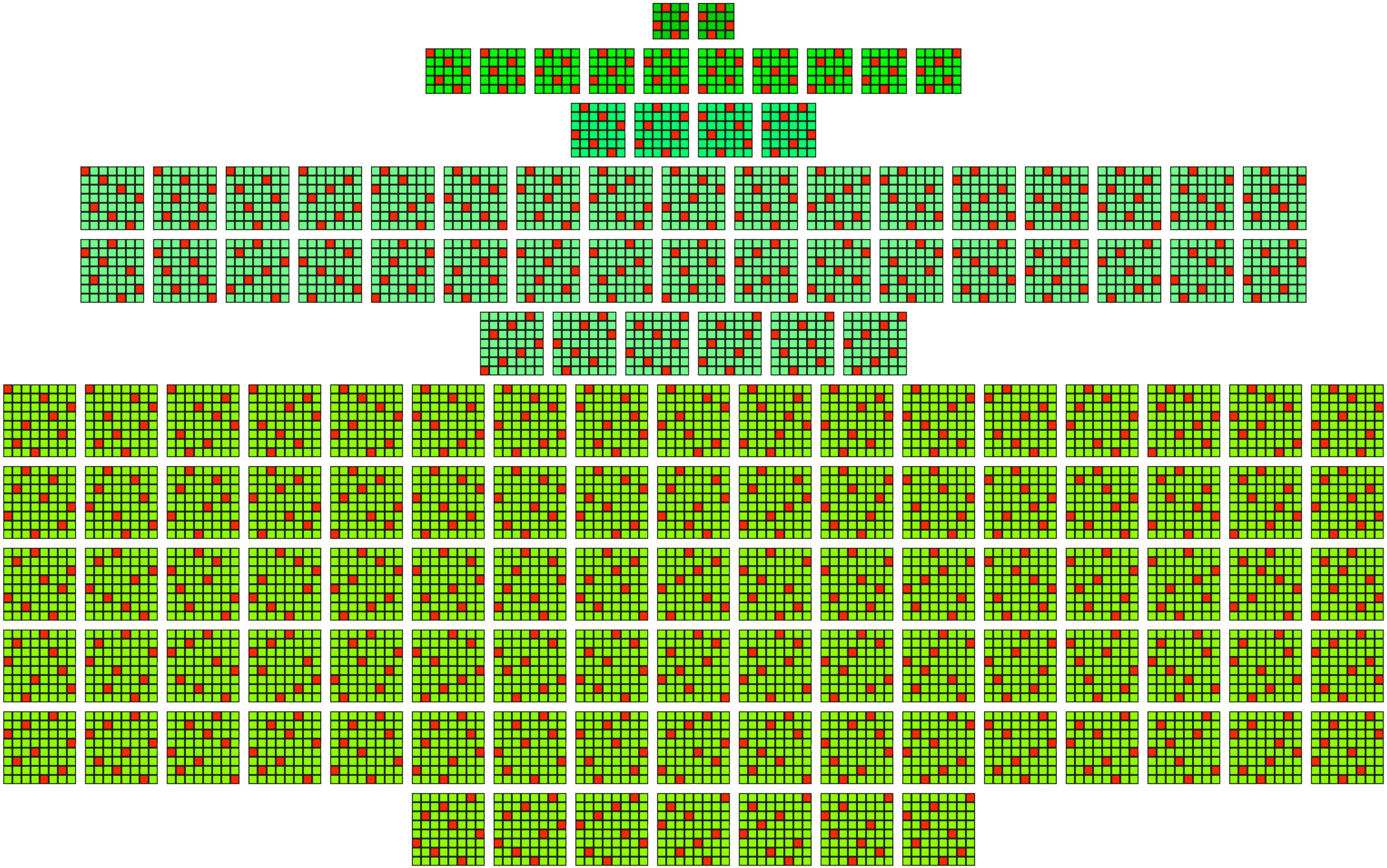
```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)

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

val paddingImage = Image.square(10)
                        .strokeColor(Color.white)
                        .fillColor(Color.white)
```

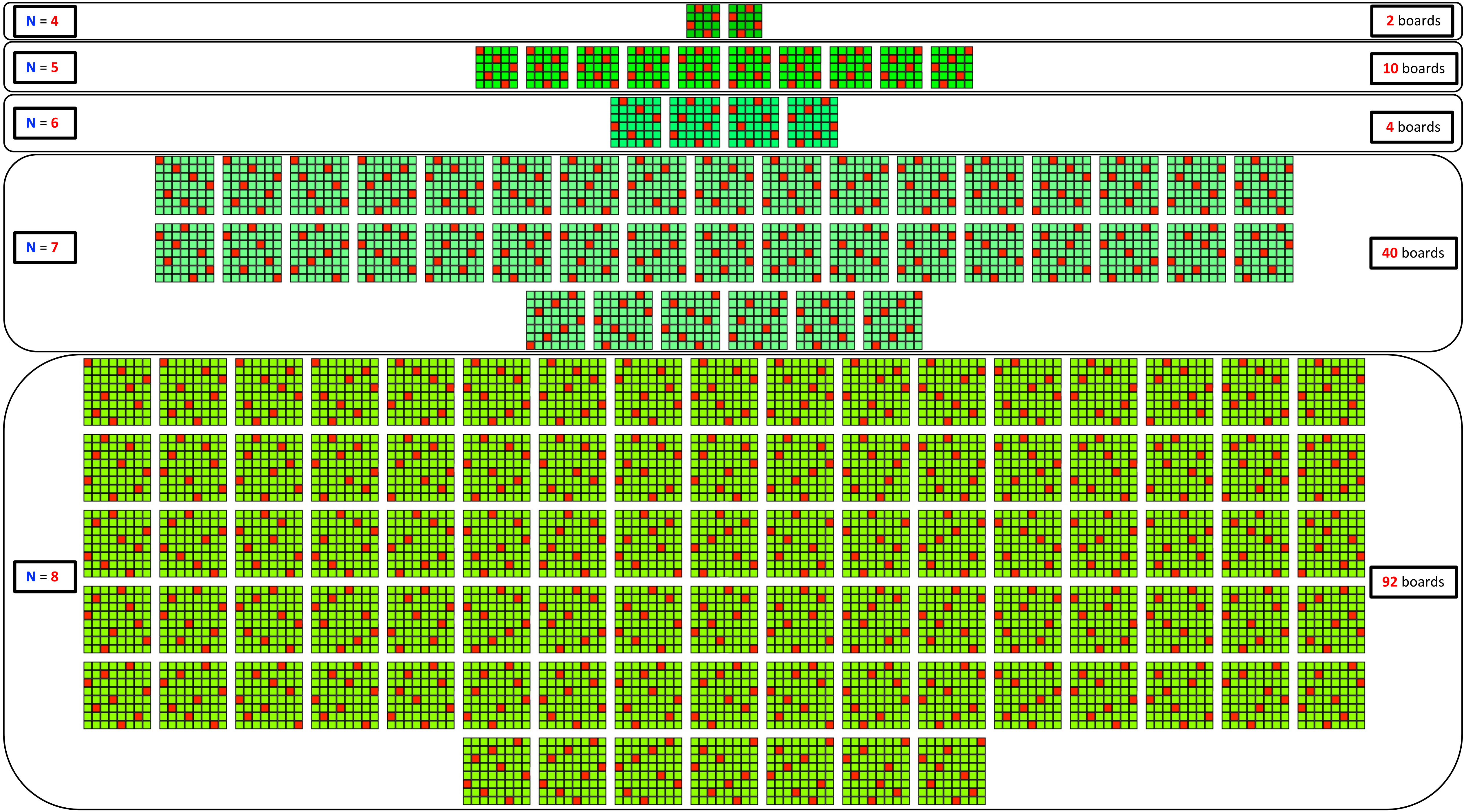
@philip_schwarz

We are now finally ready to run the program!

See the next slide for the results.

See the slide after that for the same results, but annotated with a few comprehension aids.




N = 4
2 boards
N = 5
10 boards
N = 6
4 boards
N = 7
40 boards
N = 8
92 boards

Remember in **Part 2**, when we changed the **Scala** program's logic for displaying a board, so that instead of exploiting **Doodle**'s ability to **automatically position** images relative to each other, by combining them with the **beside** and **above** functions, the logic had to first explicitly position the images by itself, and then combine the images using the **on** function?

We did that so that we could then translate the logic from **Scala** with **Doodle** to **Haskell** with **Gloss**.

Imagine doing the equivalent in order to display the **N-Queens** solutions for **N**=**4**,**5**,**6**,**7**,**8**!

While it could turn out to be relatively challenging, I don't think that if we did do it, we would feel a great sense of accomplishment.

While we might come across opportunities to use interesting **functional programming** techniques, I can imagine us being assailed by a growing sense that we are working on a fool's errand.

Let's do something more interesting/constructive instead. In **Part 4** we are going to first look at **Haskell**'s **intersperse** and **intercalate** functions, and then see an alternative way of solving the **N-Queens problem**, using the **foldM** function.

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.

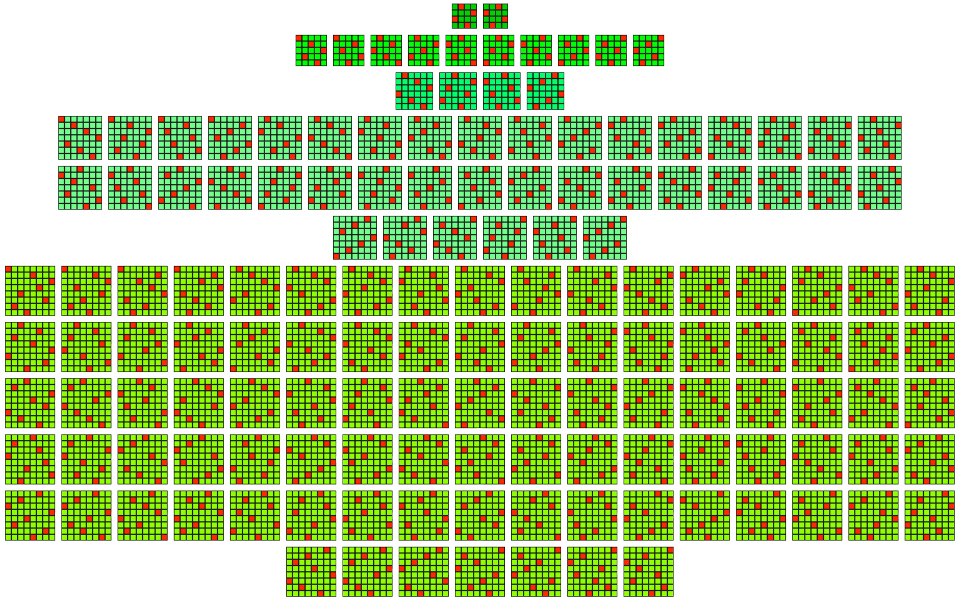
```
def show(queens: List[Int]): Image =
  val square = Image.square(100).strokeColor(Color.black)
  val emptySquare: Image = square.fillColor(Color.white)
  val fullSquare: Image = square.fillColor(Color.orangeRed)
  val squareImageGrid: List[List[Image]] =
    for col <- queens.reverse
    yield List.fill(queens.length)(emptySquare)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

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

```
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)
```

```
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)
```



I hope you enjoyed that.

See you in **Part 4**.

@philip_schwarz