

Synthesizing Mobile Games with Combinators

A Major Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Joseph Blackman

Dean Kiourtsis

Bailey Sheridan

Date: March 17, 2017

Professor George Heineman, Advisor

Table of Contents

Table of Contents	1
Abstract	2
Introduction	3
Background	3
2.1 Logic Specification Languages	3
2.2 Combinatory Logic Synthesis and the Inhabitation Problem	7
2.3 Scala Implementation of CLS	11
2.4 Kombat Solitaire	13
2.5 Ionic Framework	14
Methodology	15
3.1 Codebase	15
3.2 Source Control	17
3.3 Domain Model	18
Results	18
4.1 Combinators and Kinding	18
4.2 Games	23
4.3 Tutorial	30
Evaluation	30
Future Work	31
References	33
Appendix A: Table of Combinators	35
Appendix B: Tutorial of Process	38

Abstract

Developers rarely create entire software systems from scratch, instead they use resources provided by other developers such as software frameworks. Software frameworks are reusable code environments designed for specific tasks^[1]. For example, the Ionic framework was designed for creating platform-independent mobile applications^[2]. Learning to use a framework takes a long time, and not all of its nuances may be immediately apparent through documentation and code examples. Our project sets out to reduce the amount of time and effort required to use the Ionic framework to create mobile puzzle games. We used combinators to successfully synthesize several games and produce a library of Ionic components to expedite the Ionic mobile game creation process.

Introduction

One of the dreams of computer science is to automatically generate a complete working software program from a simple logical specification. Despite decades of research, this problem still does not have a clear solution. The difficulty of this problem may be due to its overall complexity, a lack of computing power, or the amount of effort required for current logical specification languages. In this project we attempt to automatically generate code for an Ionic mobile application with the use of Combinatory Logic Synthesis (CLS).

Background

2.1 Logic Specification Languages

Logic Specification is a powerful tool for programmers since it allows one to model and formally prove code to be correct with regards to a given specification. One of the earliest such languages is Hoare Logic^[3]. Hoare Logic works as follows: Given a program in initial state P , after executing a command c , the program ends in state Q ^[4]. This is called a Hoare Triple, traditionally written as $\{P\} c \{Q\}$. Within the context of programming, P and Q represent a state of memory, and c a block of code. An example of a Hoare Triple for a variable assignment:

$$\{X=1\} Y=X \{Y=1\}$$

Figure 1: Hoare Triple 1

In English, this says “If the program starts with X as 1, then running $Y=X$ means the program will end with Y equal to 1.”^[4] It is worth noting that the initial state can be made stronger, and the final state weaker, and the triple is still correct. Thus

$$\{X=1 \wedge Z=5\} Y=X \{Y>0\}$$

Figure 2: Hoare Triple 2

is a logical continuation of Hoare Triple 1. In English, this says “If the program starts with X as

1 and Z as 5, then running $Y=X$ will cause the program to end with Y greater than 0.” Obviously, Z has no bearing on the code, and specifying $Y=1$ is more strict than $Y > 0$. This triple can be rewritten as

$$\{X=1 \wedge Z=5\} \rightarrow \{X=1\} \ Y=X \ \{Y=1\} \rightarrow \{Y>0\}$$

Figure 3: Hoare Triple 3

where the implication arrow (\rightarrow) represents a simplification of state.

However, there are other control structures in programming than simple assignment statements, and these require more complex logic. `if` statements are the simplest kind of branching logic, and their representation is fairly simple. Consider the following code:

```

if (X > 3) {
    Y = X;
} else {
    Y = X+3;
}

```

Figure 4: Simple IF statement

At the end of execution, Y will be greater than 3 as long as X was initially greater than 0.

Therefore, we can prove the Hoare Triple $\{X>0\} \subset \{Y>3\}$:

	H1. $\{X > 0\}$
L1. if ($X > 3$) {	
	H2. $\{X > 0 \wedge X > 3\}$
L2. Y = X;	
	H3a. $\{Y > 0 \wedge Y > 3\} \rightarrow$ H3b. $\{Y > 3\}$
L3. } else {	
	H4a. $\{X > 0 \wedge X \leq 3\} \rightarrow$ H4b. $\{X + 3 > 3 \wedge X + 3 \leq 6\}$
L4. Y = X + 3;	
	H5a. $\{Y > 3 \wedge Y \leq 6\} \rightarrow$ H5b. $\{Y > 3\}$
L5. }	
	H6. $\{Y > 3\}$

Figure 5: Proof of Hoare Triple

First, note that there are many extra states in this code. By adding in intermediate code states, each individual triple can be validated, and thus the entire execution can be validated.

Second, consider the two triples $\{H1\} \text{ L1 } \{H2\}$ and $\{H1\} \text{ L3 } \{H4a\}$:

$\{X > 0\}$ if ($X > 3$)	$\{X > 0 \wedge X > 3\}$
$\{X > 0\}$ else	$\{X > 0 \wedge X \leq 3\}$

Figure 6: Hoare Triples 4 and 5

Each of these represents a branch in the `if` statement. H2 is the state if the condition is true, so it contains the initial state and the condition. H4a is the state if false, so it contains the negation of the condition. Finally, H3b and H5b must match H6 for this proof to be valid.

The last kind of control structure is loops. Loops are similar to `if` statements, except that when the condition is true, a loop will execute and then evaluate the condition again. Thus, the

syntax is very similar, except that the state after the loop executes must be the same as state before execution. We call this state the *loop invariant*, and finding a statement of this is crucial to finding a formal proof for a loop. Here is a simple loop which takes some initial $X=m$, $Y=0$ and ends with $X=0$, $Y=m$. In this code, after each loop iteration the sum of $X+Y$ is always a constant, so this is our loop invariant.

	H1a. $\{X = m \wedge Y = 0\} \rightarrow$
	H1b. $\{X + Y = m\}$
L1. while (X != 0) {	
	H2a. $\{X \neq 0 \wedge X + Y = m\} \rightarrow$
	H2b. $\{X + (Y + 1) = m + 1\}$
L2. Y = Y + 1 ;	
	H3a. $\{X + Y = m + 1\} \rightarrow$
	H3b. $\{(X - 1) + Y = m\}$
L3. x = x - 1 ;	
	H4. $\{X + Y = m\}$
L4. }	
	H5a. $\{X + Y = m \wedge X = 0\} \rightarrow$
	H5b. $\{Y = m \wedge X = 0\}$

Figure 7: Hoare Logic loop invariant 1

Again, this proof is mostly comprised of assignments, but observe that the loop invariant is present throughout the code, particularly in H1b, H2a, and H4, so that not only is $\{H1b\} \text{ L1 } \{H2a\}$ a valid triple, but $\{H4\} \text{ L1 } \{H2a\}$ is as well. Finally, when we exit the loop, our loop invariant is still true, and we can also assume the inverse of our looping condition, which proves the final state.

Let us consider one last example of Hoare logic, again using a while loop. This example computes $Y=m!$ given an initial $X=m$:

	H1a. $\{X = m\} \rightarrow$
	H1b. $\{1 = m! / X!\}$
L1. <code>int Y = 1;</code>	
	H2. $\{Y = m! / X!\}$
L2. <code>while (X != 0) {</code>	
	H3a. $\{Y = m! / X! \wedge X \neq 0\} \rightarrow$
	H3b. $\{Y * X = (m! * X) / X!\} \rightarrow$
	H3c. $\{Y * X = m! / (X - 1)!\}$
L3. <code>Y = Y * X;</code>	
	H4. $\{Y = m! / (X - 1)!\}$
L4. <code>X = X - 1;</code>	
	H5. $\{Y = m! / X!\}$
L5. <code>}</code>	
	H6a. $\{Y = m! / X! \wedge X = 0\} \rightarrow$
	H6b. $\{Y = m!\}$

Figure 8: Hoare Logic loop invariant 2

Let us walk through the execution. The code is fairly straightforward, it multiplies Y repeatedly by X , as X decreases, so $Y = m * (m-1) * (m-2) * \dots * 2 * 1 = m!$. To prove its correctness, we start off with our initial $X=m$ (H1a) and make a mathematical rearrangement. Next, we initialize Y , and get our loop invariant (H2). As we enter the while loop, we assume the condition is true (H3a) and simplify (H3b, H3c). We multiply Y by the current value of X to reach H4, then decrease X by 1 to reach H5 and re-obtain our loop invariant. Finally, we assume the loop condition to be false (H6a) and simplify to find that $Y = m!$ (H6b).

As one works more with Hoare Logic, two things become clear: First, proving the correctness of code forces the programmer to have a deep understanding of it, and second, creating and verifying such proofs takes a considerable amount of time.

2.2 Combinatory Logic Synthesis and the Inhabitation Problem

There is an inherent difficulty in using Hoare Logic as a means of proving code correctness. Other systems like Linear Temporal Logic^[5], Büchi Automata^[6], and Kripke Structures^[7] still place a great burden on engineers by forcing them to logically specify each and every line of code or functional implementation. One strategy for dealing with this complexity is to analyze the program in smaller, more manageable modules which are assembled to form a proof of the entire system. This strategy is the theory behind Combinatory Logic Synthesis^[8].

Combinatory Logic Synthesis is a type-based approach to code synthesis. Formally typed blocks of code, called *combinators*, are automatically combined using an inhabitation algorithm to form a final product^[8]. Much like Hoare Logic, each combinator starts with an input type, contains a block of code, and exits with another type. For example, consider a Java program which takes in a list and returns its largest element. Such a program might be broken up into three combinators, one to input the list, one to sort it, and one to return the largest element:

```

InputList : True  $\rightarrow$  list  $\cap$  unsorted {
    Scanner input = new Scanner(System.in);
    String line = input.nextLine();
    Scanner parser = new Scanner(line).useDelimiter("[,\\s]+");
    ArrayList<Integer> list = new ArrayList<Integer>();
    while (parser.hasNextInt()) {
        list.add(parser.nextInt());
    }
}

BubbleSort : list  $\cap$  unsorted  $\rightarrow$  list  $\cap$  sorted {
    for (int i=0; i<list.size(); i++) {
        for (int j=0; j<list.size()-1; j++) {
            if (list.get(j) < list.get(j+1)) {
                int val = list.get(j);
                list.set(j, list.get(j+1));
                list.set(j+1, val);
            }
        }
    }
}

PrintList : list  $\cap$  sorted  $\rightarrow$  print {
    System.out.println("Sorted list: " + list);
    System.out.println("Largest value: " + list.get(0));
}

```

Figure 9: List Sorting Combinators

To generate Java code which satisfies this specification, we ask CLS to generate an object of type `print`. There is only one function that generates this, `PrintList`, and it takes as input `list \cap sorted`, so CLS searches for that type. It finds only one valid combinator, `BubbleSort`, which takes a `list \cap unsorted` as input, and CLS searches for this. Finally, it finds `InputList`, which has no required input, so the execution traces back `InputList \rightarrow BubbleSort \rightarrow PrintList`, merges the contents of these functions and

generates the final program.^[9]

One advantage CLS has over Hoare Logic is the ease of updating code. In Hoare Logic, implementing a new sorting algorithm would require a full formal proof of that code, which might demand updates to other branches in the logic. In CLS, this is as simple as writing another combinator:

```

InsertionSort : list  $\cap$  unsorted  $\rightarrow$  list  $\cap$  sorted {
  for (int i=1; i<list.size(); i++) {
    int j = i;
    while (j > 0 && list.get(j-1) < list.get(j)) {
      int val = list.get(j);
      list.set(j, list.get(j-1));
      list.set(j-1, val);
      j--;
    }
  }
}

```

Figure 10: Insertion Sort Combinators

When the inhabitation code runs, it looks for functions that generate `list \cap sorted`, finds two valid paths and generates two complete programs: one using bubble sort, and another using insertion sort.

Another benefit of combinators is renaming variables. In the above examples, the input variable is named `list`. Not only is this generally not a good name, it also doesn't specify anything about its contents. Using CLS, we can change this variable's name to be in a combinator:

```

ListName : listName { prices }
InputList2 : listName → list ∩ unsorted {
    Scanner input = new Scanner(System.in);
    String line = input.nextLine();
    Scanner parser = new Scanner(line).useDelimiter("[,\\s]+");
    ArrayList<Integer> [listName] = new ArrayList<Integer>();
    while (parser.hasNextInt()) {
        [listName].add(parser.nextInt());
    }
}
BubbleSort : listName → list ∩ unsorted → list ∩ sorted {
    for (int i=0; i<[listName].size(); i++) {
        for (int j=0; j<[listName].size()-1; j++) {
            if ([listName].get(j) < [listName].get(j+1)) {
                int val = [listName].get(j);
                [listName].set(j, [listName].get(j+1));
                [listName].set(j+1, val);
            }
        }
    }
}
PrintList : listName → list ∩ sorted → print {
    System.out.println("Sorted list: " + [listName]);
    System.out.println("Largest value: " + [listName].get(0));
}

```

Figure 11: Value Replacing Combinator Example

The generated code will replace `[listName]` with a value from a combinator, in this case the `ListName` combinator. We can easily change this variable's name, and the change will automatically propagate to all of the combinators that need it.

However, CLS does not protect against poor programing. Here is a combinator which randomizes the order of the list:

```

BogoSort : list  $\cap$  unsorted  $\rightarrow$  list  $\cap$  sorted {
    Random r = new Random();
    for (int i=0; i<list.size()-1; i++) {
        int j = r.nextInt(list.size()-i)+i;
        int val = list.get(j);
        list.set(j, list.get(i));
        list.set(i, val);
    }
}

```

Figure 12: Combinator with incorrect code

This code does not check if the list is sorted afterward. Thus, the final program will not necessarily print out the largest element, since the combinator will rarely sort the list properly. The root issue here is that the type of the combinator is incorrect, it should be `list \cap unsorted \rightarrow list \cap unsorted`. As opposed to a full formal specification, CLS relies on the programmer to correctly type their code fragments.

2.3 Scala Implementation of CLS

Our project is based on a Scala implementation by Jan Bessai. Combinators are defined natively as Scala objects and the inhabitation algorithm runs natively^[9]. The Scala implementation differs from the Java one in complexity. The Java system uses Abstract Syntax Trees^[9] which provide more power and flexibility when assembling and manipulating Java source code. In contrast, the Scala implementation is only capable of concatenating strings. However, this is sufficient for these simple examples. Here's the sorting combinator in Scala, which returns working Java code:

```

@combinator object BubbleSort {
  def apply(listName:String) : String = {
    return s"""
      for (int i=0; i<${listName.size()}; i++) {
        for (int j=0; j<${listName.size()-1}; j++) {
          if (${listName.get(j)} < ${listName.get(j+1)}) {
            int val = ${listName.get(j)};
            ${listName.set(j, ${listName.get(j+1)});
            ${listName.set(j+1, val)};
          }
        }
      }
    """
  }
  val semanticType:Type = 'listName =>: 'list :&: 'unsorted =>:
  'list :&: 'sorted
}

```

Figure 13: Scala Sorting Combinator

Mostly, this looks the same as the Java combinators, with the key difference being that parameters, such as the name of the list, are also declared within the `apply()` statement.

The Scala system also allows the user to create classes, which look identical to combinators, except that the type can be specified later. In our code, we created generic classes to handle different languages, one for HTML and another for JavaScript. Here is an example of same sorting code, created with a class:

```

class GenericSort(inputType:Type, outputType:Type, listName:String) {
  def apply() : String = {
    s"""
      for (int i=0; i<${listName.size()}; i++) {
        for (int j=0; j<${listName.size()-1}; j++) {
          if (${listName.get(j)} < ${listName.get(j+1)}) {
            int val = ${listName.get(j)};
            ${listName.set(j, ${listName.get(j+1)})};
            ${listName.set(j+1, val)};
          }
        }
      }
    """
  }
  val semanticType:Type = 'listName =>: inputType =>: outputType
}
@combinator object BubbleSort extends GenericSort('list :&: 'unsorted, 'list :&:
'sorted)

```

Figure 14: Combinator object example

Implementing combinators and classes through Scala CLS is reasonably straightforward, and for our project is just as powerful as the Java system. Furthermore, as long as the combinators are typed correctly, the system has the same level of probability as Hoare Logic, without the complexity and fragility.

2.4 Kombat Solitaire

The Kombat Solitaire framework is an object-oriented framework for developing variants of solitaire. It contains sixty-seven thousand lines of code with the goal of generating dozens to hundreds of solitaire variants^[10]. Typically, creating a solitaire variation from this framework is difficult due to its complexity, and the amount of knowledge required to work with it. In particular, the ability to implement features, such as rules and win conditions, requires in depth

experience with the framework. Rather than asking users to learn the framework, Heineman and Hoxha combine the Kombat Solitaire framework with CLS to simplify the process of creating a solitaire variation with the framework. This ensures that the resulting code is correct, and readable to anyone familiar with the framework.

Starting from the original Kombat Solitaire tutorial, Heineman and Hoxha created a repository of combinators necessary for implementing a solitaire variation. From this point, they selected different abstractions from the framework to extract into CLS combinators. These combinators are used to build a solitaire variation.

```

WinRuletype: (alpha.gameType  $\cap$  pilerule  $\cap$  homePile)  $\rightarrow$ 
    (alpha.gameType  $\cap$  winrule)
WinRuleterm: {
   $\lambda$  piles. {letbox NumPiles = {piles} in {
    box["
    boolean won = true;
    for (int i = 0; i < " NumPiles "; i++) {
      if (fieldHomePiles[i].count() != 13) {
        won = false;
      }
    }
    if (won) { return true; }
    "]
  }}
}

```

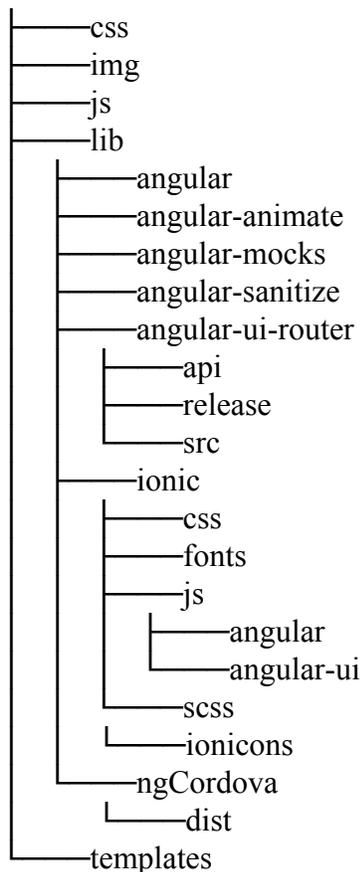
Figure 15: Kombat Solitaire WinRule Combinator

In the above example, the `WinRule` combinator contains the code to check if the game is in a winning state. The specific type will place this block of code into the correct location in the game, and the precise code is provided by the programmer. This system allows for easy, configurable creation of solitaire games, analogous to the purpose of this MQP which is simple creation of Ionic apps.

2.5 Ionic Framework

Our goal for the project was to reduce or eliminate the need for a programmer to learn a software framework. Our framework of choice was the Ionic framework, with which we've created mobile games. We selected Ionic as a framework because of its ease of use: it is platform-independent, requires little setup, and can be quickly tested in a browser. Ionic apps can be deployed to iOS and Android, as the underlying system uses Node.js, relying on libraries such as Angular.js and Cordova^[2]. At the time we began, Ionic v2 was in beta and there was not yet a stable release, so we worked with Ionic v1 instead.

The file structure for an Ionic application is represented by the following tree. The base directory for this tree is in a folder named www. Our game code resides in the top level js folder (all JavaScript code), the top level templates folder (all HTML code) and the top level css folder. The top level img folder would contain any images for our games if we used any. The lib folder and its subfolders contain any source packages and modules required for Ionic to run.



Methodology

3.1 Codebase

Our first step was to learn about the technologies involved in creating an Ionic app, which mainly consists of JavaScript and HTML5. Using these, we created a skeleton app with a main menu, options, level select, and individual levels. We also implemented a SQLite database to save level status in between sessions, and wrote a sample game for testing the app. We separated the code into combinators and wrote a CLS program with Scala to automate its creation. The first step was to convert the output of the CLS program into an Ionic application using Java. Next, we wrote our combinators to pair the file contents with their locations. At this point the program was able to successfully synthesize and run our sample game. After we could generate end-to-end code, each of us made our own puzzle game: Hangman, Lights Out, and Word Mastermind.

Since our combinator code uses only string concatenation, the CLS file became excessively long. In order to shorten this file and simplify our code, we took advantage of templating to move large strings to other files. Due to the Scala based toolset we were using, Twirl was the templating language of choice^[11]. This is because Twirl uses the Play framework, which works with Java and Scala without any additional setup. We extracted all of our JavaScript, HTML, and CSS files into Twirl templates, which can be called from our Scala code and rendered into strings, allowing them to be concatenated with the rest of our inhabited code. Using Twirl greatly increased the readability of our CLS code by extracting all of the game code into files.

Each Twirl template is written in the file format *name.scala.type* where *name* is some identifying filename and *type* is the file extension, such as *.js* or *.html*. Twirl was able to support *.js* and *.html* natively, but not the *.css* files. We solved this by using the *.xml* template format for the *.css* files. Since neither *.css* nor *.xml* files are interpreted, they are functionally equivalent.

Some of our templates are simple blocks of code, but others are configurable via input variables. Typed parameters can be passed into Twirl templates when rendered, and expressed in the code with the @ symbol. Our variables were mostly strings and functions, but some of the

templating uses the `@for` syntax, which creates loops within the code. Additionally, the Scala types for the rendered templates use the Twirl Play types since they will correctly interpret escape sequences.

One of the issues we experienced with Scala was generating each game's unique pages, while still generating shared files such as the main page and level select. We resolved this issue by associating every page with a game via `kinding`, a CLS directive. Thus, combinators which were game-sensitive would reference the `kinding` and generate code for a specific game when running the inhabitation. For example, although the main page is shared between all games, it takes in the the game type to specify the game's title when the inhabitation is run.

While working on this project it was helpful to occasionally integrate Scala features into the base software. This was particularly helpful when we implemented Ionic framework elements by using complex types and built in Scala functions like `mkString()`. This allowed us to keep the end user result simple while still allowing complex and configurable systems. We also used string interpolation for the Ionic framework components, making these sections more readable. In general, working with Scala as a language rather than as another framework helped to keep our code readable and streamlined.

As an example, our SQL combinator did not originally include Scala features, but was modified to include a Scala Map object for ease of use. At first, the SQL combinator had static and unmodifiable columns. In order to make this more configurable, we modified the SQL columns to be represented by a Scala Map object. This allows users to add, remove, and access new columns in the database at will.

3.2 Source Control

We used Github for source control throughout the project. Our repository began with the `master` branch, where we experimented with the Ionic framework, and created our initial games. This branch contains only Ionic code, and continues to serve as a testing ground. This branch implements all of our games in parallel so that we can switch between them easily. Each game has its own JavaScript and HTML files, and we can render each game by changing the file reference in `app.js`, `states.js`, and `controllers.js`. The Ionic code can be run with

`ionic serve` from anywhere in the project, and Ionic will start the current game. Nothing in this branch is written in Scala nor implements CLS.

After we were satisfied with our games, we created a new branch called `sbt`. We extracted all of the code into combinators to be generated by CLS. Instead of keeping all the games in parallel, each game has its own set of combinators, and can be rendered by changing the inhabitation request. The `sbt` branch does not contain any pure Ionic code, all of it is extracted into combinators and Twirl templates. Some core Ionic code is still present in this branch, such as the style sheets and icons.

The `cls_sbt` folder contains our Scala code, our Java file handler, and Twirl templates. The main directory of this branch contains automated run scripts which launch `sbt`, compile and run the CLS inhabitation, and then calls `ionic serve`. The full process takes from 30 to 120 seconds, depending on computer architecture, so the batch file helps to streamline this process.

We created a new branch as a tutorial for our framework. This branch showcases the process of creating a new game, Nim. We created this branch based on our commits from initially writing this game, and created a companion text tutorial. While creating this game, we tracked our thoughts, order, and amount of time to complete each step.

3.3 Domain Model

Games created by our framework are made with a specific domain model in mind. We cannot claim to synthesize every type of mobile game since the possibilities are endless. Instead, our framework assists with creating only certain kinds of mobile games. It supports only mobile games that have a main menu, options, and level select screen. Players enter the game through the main menu, and can then access the level select or options. From the level select screen, players can access individual levels. Each level keeps an internal state which represents partial progress through the level, success, or failure. A player can exit a level, and return later the same state they left it in. Similarly, completing a level will mark it in the level select.

We chose to create puzzle games to test our framework, but theoretically any genre of game with state based levels can be implemented using our system. For example, one such usage would be flash cards, with each level containing a different card. However, this has no partial

state which results in it being a poor test case. The state is saved into a SQLite database, with columns modified by combinators, so the table could be used for many other purposes.

Results

4.1 Combinators and Kinding

The completed CLS Scala repository contains a total of 51 different combinators and is able to build 6 unique apps. In order to allow the generation of multiple games, a kinding must be included with the repository.

```
val gameVar = Variable("gameName")
lazy val kinding =
  Kinding(gameVar)
    .addOption('mastermind)
    .addOption('hangman)
    .addOption('lightsout)
    .addOption('dummy)
    .addOption('monster)
    .addOption('nim)
```

Figure 16: Kinding example

A kinding simply allows a number of different types to be accessed from one variable. The code above allows `gameVar` to stand for any of our games. This simplifies the entire game creation process as a majority of the combinators don't operate differently for different games. In order to add another game to the kinding, the programmer must simply append `.addOption('gameName)`, where `gameName` is the name of the game being added.

There are a number of different styles of combinators, each with having a different method of use and overall purpose. Below is a list of examples and descriptions for each type of combinator used in our software.

4.1.1 Static Value Combinators

```
@combinator object ScriptList {
  def apply(): Array[String] = {
    return Array("sql", "states", "controllers", "game")
  }
  val semanticType:Type = 'scriptList
}
```

Figure 17: ScriptList Combinator

The static value combinators are the simplest. These combinators keep track of values that are universally used in the Ionic application. The combinator above keeps track of the scripts that must be included in the Ionic application, so that the list can be reused multiple times throughout our code.

4.1.2 Title Combinators

```
class Title(game:Symbol, title:String) {
  def apply() : String = {
    return title
  }
  val semanticType:Type = game :&: 'gameTitle
}
@combinator object HangmanTitle extends Title('hangman, "Hangman")
@combinator object DummyTitle extends Title('dummy, "Dummy")
```

Figure 18: Title Combinator Objects

The title combinators keep track of the title names of the games. Unlike the static value combinators, title combinators are made by extending the Title class with values for the game type and the game's name. The game type must be one of the ones present in the `gameVar` kinding.

4.1.3 Function Combinators

```

type buttonType = (String, String, String) => Html
@combinator object Button {
  def apply(): buttonType = {
    return (callback:String, text:String, color:String) =>
      new Html(s""<button class="button button-$color
levelBtn" ng-click="$callback()">$text</button>""")
  }
  val semanticType:Type = 'button
}

```

Figure 19: Button Combinator

The function combinators produce functions for use in other combinators. The purpose of these combinators is to generate Ionic components in an HTML template with ease. In the example above, the function created simply substitutes three strings into the HTML for an Ionic button, and then labels the resulting string as HTML code.

4.1.4 Twirl Rendering Combinators

```

@combinator object HangmanHTML {
  def apply(button:buttonType): String = {
    return html.html.hangman.render(button).toString()
  }
  val semanticType:Type = 'button =>: 'hangman :&: 'html
}

```

Figure 20: HangmanHTML Combinator

The twirl render combinators can take other combinators as input. In general, the inputs for these combinators are functions or strings to be used when rendering the template. These combinators are used to convert a template HTML, JS, or CSS file into a string with Twirl. They are necessary since the base template code, which is written by the programmer, must be modified before it can be used in an Ionic app. In the above example, the combinator takes the

button combinator as an input argument and uses it when rendering the hangman HTML file. Other instances of the twirl render combinator take in strings as input, or even no input.

4.1.5 Final HTML and JavaScript Combinators

```
@combinator object GameHTML {
  def apply(contents:String): String = {
    return html.html.game.render(contents).toString()
  }
  val semanticType:Type = gameVar :&: 'html =>: gameVar :&:
  'gameHtml
}
```

Figure 21: GameHTML Combinator

The final HTML and JavaScript combinators produce code for the finished Ionic app. Each of these combinators takes in a string, and returns a finished string which contains an entire file. Depending on the file being created, the renderer will read a different Twirl template. In the above example, the combinator takes in a string with the type `gameVar :&: 'html`, and uses it to render the game HTML page with a string of type `gameVar :&: 'gameHtml`. The resulting string represents an Ionic-ready HTML file containing the code from an HTML render combinator.

4.1.6 File Binding Combinators

```
class Bind(inputType:Type, filePath:String){
    def apply(expr:String) : Tuple = {
        return new Tuple(expr, filePath)
    }
    val semanticType:Type = inputType => 'BoundFile :&: inputType
    :&: gameVar
}
@combinator object Bind5 extends Bind('gameHtml :&: gameVar,
"www/templates/game.html")
```

Figure 22: Bind Combinator Objects

The file bind combinators are the last set of combinators used when building an app. These take the HTML, JS, and CSS strings from the final HTML and JavaScript combinators, then bind them into a tuple along with the destination of the corresponding file. In the above example, the combinator takes the combinators with type 'gameHtml :&: gameVar, which contains the game HTML code, and groups it with the file location `www/templates/game.html`. The tuples from these combinators are used to place files where Ionic expects them to be, and populate them with the proper code.

4.2 Games

Each of us created our own game to learn the Ionic framework, and to have some code to convert into combinators by looking for similarities shared between the games. Here is a summary of each of our games, along with some of the difficulties presented in their implementation.

4.2.1 Hangman

In Hangman, the player is asked to guess an English word letter by letter. Each time they guess a correct letter, all instances of that letter are revealed at their locations in the word. In our implementation, after seven incorrect guesses the player loses the game, and their progress is reset. Each level contains a different word for the player to guess, with later levels having more challenging words.

Players guess a letters one at a time via a text box in the level. Due to an engine limitation, the enter key does not function for submitting a guess, and the player must click on the guess button. This button is generated via our button combinator. This combinator takes three inputs: a callback function to be called when the button is pressed, the text displayed on the button, and the button color. On the screen are blank boxes with invisible letters to indicate how long the word for the level is. When the player guesses a letter, a function checks all of the guessable letters in those boxes for that level and checks for matches. Any instance of the guessed letter is made visible, and after each guess there is a check to see if the amount of revealed letters matched the length of the word to see if the player has won.

The biggest challenge of writing this game was correctly revealing letters. Each letter reveal location had to be associated with the level it was on as well as what letter would be revealed there. The final solution involved getting all of the elements that had the level number and the "guessable" keyword in its class name. When a letter is guessed, the "discovered" keyword is added to its location and it is ignored in further searches.

Hangman uses the Button combinator and the Title combinator, as well as two combinators that generate the JavaScript and the HTML for the game. It shares other combinators with the other games, such as generating the main menu and level select.

Level 1

r
a

Gussed Letters:
d f c o a r

Tries Left: 3

🏠 BACK
RESTART

```

@combinator object HangmanJS extends RenderJS('hangman :&: 'js,
js.js.hangman.render())

@combinator object HangmanHTML {
  def apply(button:buttonType): String = {
    return html.html.hangman.render(button).toString()
  }
  val semanticType:Type = 'button =>: 'hangman :&: 'html
}

@combinator object HangmanTitle extends Title('hangman, "Hangman")

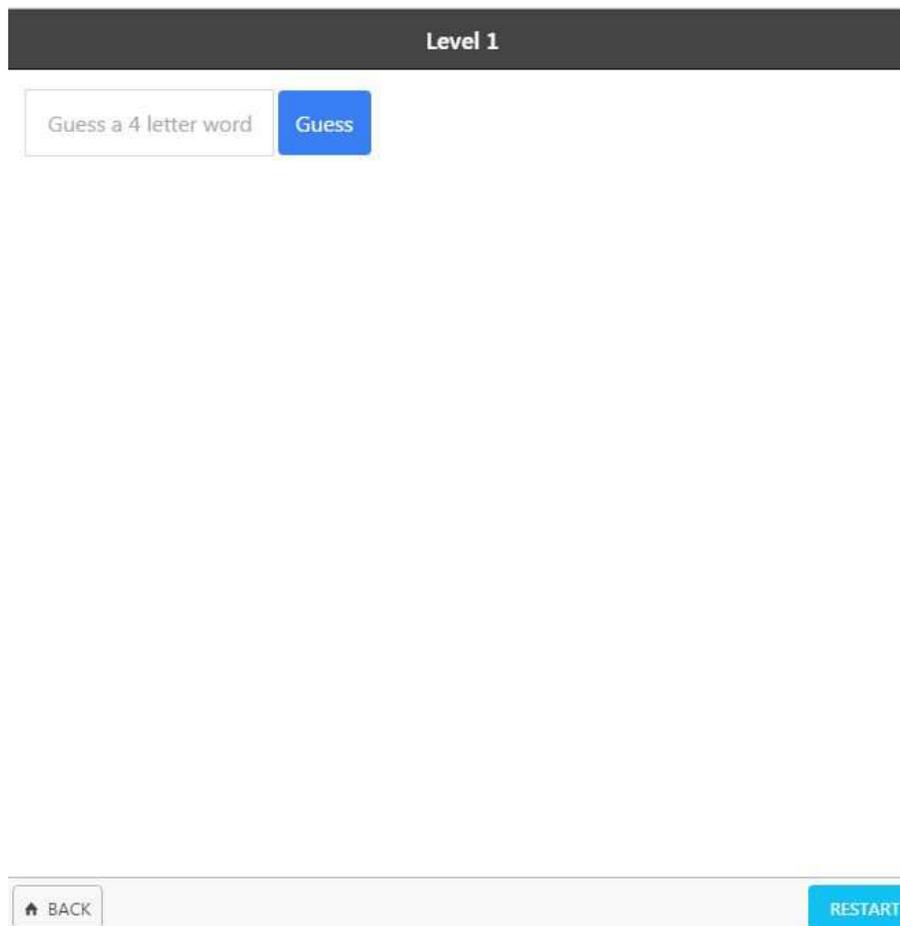
```

Figure 23: Hangman Combinators

4.2.2 Word Mastermind

Word Mastermind is word guessing game where the player tries to guess a hidden word. Much like mastermind, if the guessed word shares letters with the secret word, the number of shared letters will be revealed. Unlike mastermind, the position of the shared letters is not revealed. Additionally, guesses must be proper English words, not random collections of letters. Thus, the challenge lies in figuring out which letters in the guessed word are matching, and trying to anagram them into real words.

Implementing this game posed a few challenges, particularly verifying that the input strings are valid words. Initially, we pulled words from an online dictionary, but due to security restrictions of Javascript, this list was not accessible without explicit user permission. Next, we tried using a local dictionary file, but once again Javascript is not allowed to directly access the file system, so this list was refactored into a script so that it could be loaded.



The screenshot shows the game interface for Level 1. At the top, a dark grey header bar contains the text "Level 1". Below this, there is a text input field with the placeholder text "Guess a 4 letter word" and a blue "Guess" button to its right. At the bottom of the interface, there is a light grey footer bar containing a "BACK" button with a home icon on the left and a blue "RESTART" button on the right.

```

@combinator object MastermindJS extends RenderJS('mastermind :&: 'js,
js.js.mastermind.render())

@combinator object MastermindHTML {
  def apply(button:buttonType): String = {
    return html.html.mastermind.render(button).toString()
  }
  val semanticType:Type = 'button =>: 'mastermind :&: 'html
}

@combinator object MastermindTitle extends Title('mastermind,
"Mastermind")

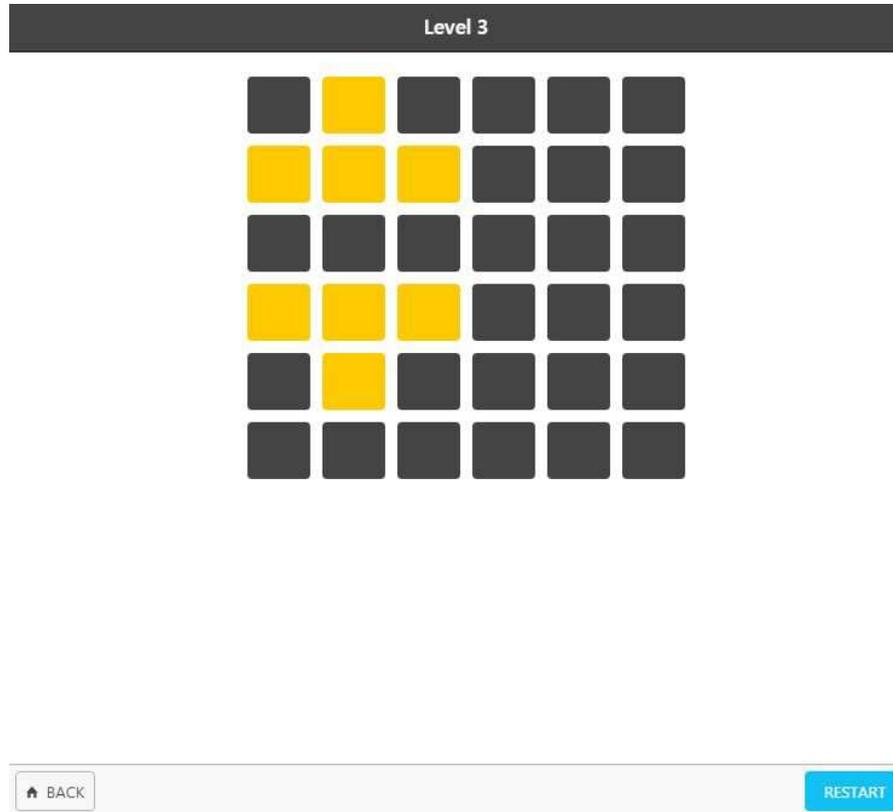
```

Figure 24: Mastermind Combinators

4.2.3 Lights Out

The goal of the Lights Out puzzle is to turn all buttons (lights) into the gray (off) state. For each move the player flips the status of a light and its four adjacent neighbors. Each level has a different pattern of lights, with more complicated levels requiring more moves to complete.

The main challenge with writing the Lights Out puzzle was setting up the partial state saving with the database. We added a level state column to the SQL database and modify the SQL combinators to account for the change. This led us to use a Scala Map object to represent database columns, making the database more configurable.



```
@combinator object LightsOutJS extends RenderJS('lightsout :&: 'js,
js.js.lightsout.render())
```

```
@combinator object LightsOutHTML {
  def apply(): String = {
    return html.html.lightsout.render().toString()
  }
  val semanticType:Type = 'lightsout :&: 'html
}
```

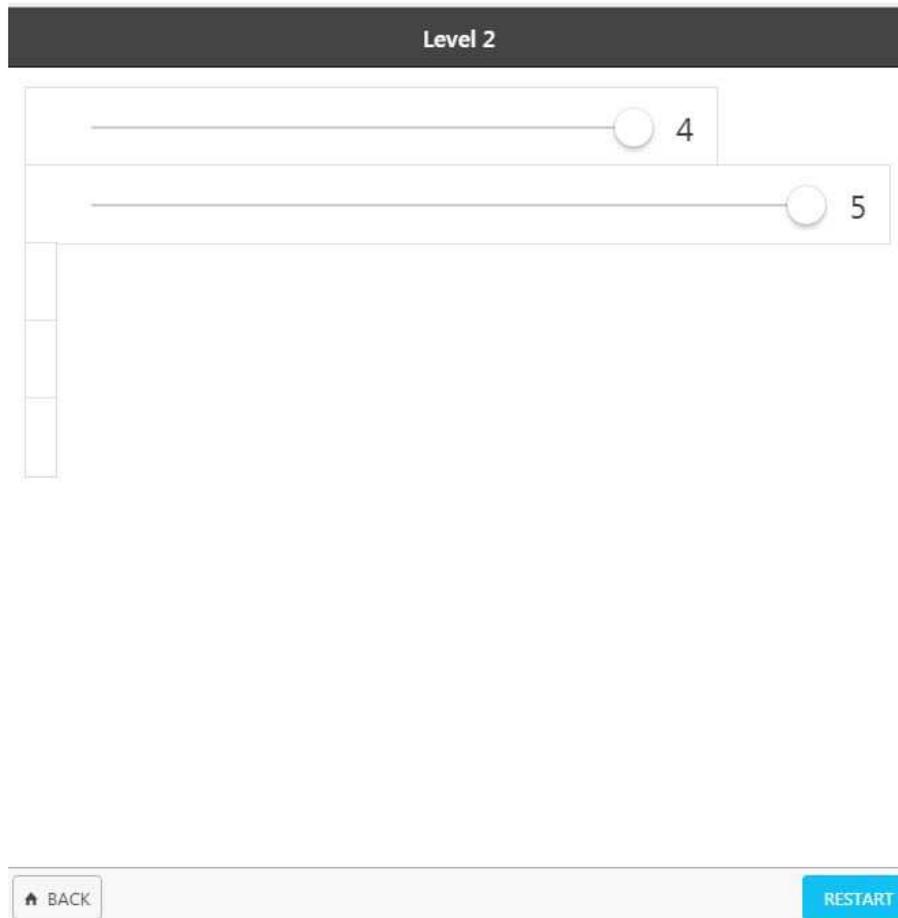
```
@combinator object LightsOutTitle extends Title('lightsout, "Lights
Out")
```

Figure 25: Lights Out Combinators

4.2.4 Nim

Nim is a game where two players take turns removing matches from piles. On their turn, each player can remove as many matches as they want from the pile, and the player who removes the last match wins. In Javascript, piles are represented with sliders, and removing matches is done by sliding the slider down. Later levels increase the number of piles and the number of matches in each pile.

Since this game requires two players, in our implementation we provide an AI to play against. Since nim is a solved game, and the winning move from any position can be calculated, the AI will occasionally make a random move instead of the optimal one. Thus, the challenge in implementing this game was entirely in coding the AI, and ironing out bugs in the decision making process.



```

@combinator object NimJS extends RenderJS('nim :&: 'js,
js.js.nim.render())

@combinator object NimHTML {
  def apply(range:rangeType): String = {
    return html.html.nim.render(range).toString()
  }
  val semanticType:Type = 'range =>: 'nim :&: 'html
}

@combinator object NimTitle extends Title('nim, "Nim")

```

Figure 26: Nim Combinators

4.3 Tutorial

Much of the setup and learning process for creating a game with combinators was confusing and complicated. We have condensed hours of bug fixing into a streamlined tutorial for our users to follow. This tutorial walks through game creation from start to finish to show users how simple it is to make their own game with the game of Nim (Appendix B).

The tutorial files contain minimal code, only enough to run our dummy game. The first step of the tutorial is copying the dummy files and combinators and changing the names to reflect the new game, Nim. From there, the user should write the HTML for the game. In our example, Nim, this also uses a combinator to add in Ionic range elements. Next, the user should implement the game logic in the JavaScript template file. The tutorial mentions a few provided functions in the game.js file as well as some optional functions the user can write that will be called automatically by our framework. Optional functions available are `initializeLevel()`, `beforeLeave()`, `onWin()`, `onLose()`, and `restartLevel()`, which will be called at appropriate times within the game.js code. Within the user's own JavaScript code, they can also call the provided game.js functions `$scope.restart()` and `$scope.completeLevel()`, the latter of which takes in a boolean argument indicating if the level completed in a win or a loss. Of course, the user can also

define as many functions as they would like to further customize the functionality and flow of their game.

Evaluation

After all of this tweaking and fine tuning, we are left with our final CLS program. This final product was evaluated using the original criterion: percentage of code generated by combinators and the total number of combinators.

Our main criteria is the percentage of code that can be generated by the CLS when given an arbitrary puzzle app. When generating one of the existing test games, CLS generates 100% of the code to make a working Ionic app. The results of running the program is the generation of blank Ionic app files and 11 CLS generated files, of which three are generic and eight are app specific or app sensitive. These files are all that are necessary to run the Ionic app for the specified game. These results have considerably passed the original goal of generating at least 80% of the code with CLS.

Another important factor for evaluating the project is the number of combinators used to generate the final product. With larger numbers of combinators, the CLS inhabitation algorithm will take a longer time to run. Our code contains a total of 51 combinators. (Appendix A) Of these, 18 are game specific combinators shared between six games. The remaining 33 are generic and used in some way for all of the games we created. Twelve of those generic combinators are used to bind our game files with the file paths they belong on, and five are for creating generic Ionic components. The rest create the files Ionic requires to run an application.

Another metric for evaluating our code is to consider how challenging this process would be without it. Generating Ionic apps from scratch requires a large amount of background knowledge with the framework, but so does our system. However, if a user is familiar with both systems, then writing a new app with combinators will require less code but more time, if only due to having to compile the code through Scala.

Even though we are successfully able to generate 100% of an Ionic game, we did not eliminate the need to read documentation and program properly. Our included tutorial will greatly reduce the time taken to create an Ionic app following our domain model, but anyone

who wants to add to our combinator repository will still need to fundamentally understand how combinators work. The system will fail if combinators are implemented with the wrong types.

Future Work

We used Ionic version 1 for our project during the beta period of Ionic version 2. We chose not to use v2 because it was more reliable to use a stable version. Towards the end of our project, Ionic v2 was officially released, so one avenue for future work would be to update our codebase to use v2 instead. This new version of Ionic has many new features that might appeal to anyone using our code. The task of converting the project to use Ionic v2 would not be complex. The structure of our combinators will remain the same, while the code in the Twirl templates should update to the Ionic v2 standards. Additionally, the combinators we have which provide specific Ionic components will need revising.

There is also room for expansion with regards to our domain model. We created our combinator library specifically for puzzle games, but future developers may want to create other kinds of Ionic apps, or alter our current model to create a wider variety of mobile games. More general future work also includes creating a combinator library for a different framework other than Ionic, but that would be an entirely new project.

Within our own domain model there is still work to do. Future programmers can add more combinators to expand upon the model. We currently have one kind of layout each for the main menu, level select, and options screen. Each of those screens can be expanded to be chosen from multiple layouts through combinators. More options can also be added to the options screen as well, similar to the way we added more columns to our SQL table.

References

- [1] Fayad, M. and Schmidt, D., “Object-Oriented Frameworks”, Guest editorial, *Communications of the ACM*, special issue on Object-oriented Frameworks, 40(10), Oct. 1997.
- [2] Drifty (2016). Welcome to Ionic - Ionic Framework. Retrieved from <http://ionicframework.com/docs/guide/preface.html>
- [3] Hoare, C.A.R (1969). An Axiomatic Basis for Computer Programming. Retrieved from <http://web.archive.org/web/20160304013345/http://www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf>
- [4] Pierce, B (2016). Hoare. Retrieved from <http://www.cis.upenn.edu/~bcpierce/sf/current/Hoare.html>
- [5] Stanford (1998). Linear-time Temporal Logic. <http://www-step.stanford.edu/tutorial/temporal-logic/temporal-logic.html>
- [6] Mukund, M. (1996). Finite-state Automata on Infinite Inputs. <http://www.imsc.res.in/~madhavan/papers/pdf/tcs-96-2.pdf>
- [7] Kripke, Saul A. (1963) Semantical Analysis of Modal Logic http://fitelson.org/142/kripke_1.pdf
- [8] Rehof, J. (2013). Towards Combinatory Logic Synthesis. Retrieved from http://www-seal.cs.tu-dortmund.de/seal/downloads/rehof/research_papers/Beat13rehof.pdf

- [9] Submitted by Döder, Rehof, Bessai, Heineman (2017).
<http://web.cs.wpi.edu/~heineman/tmp/icfp.pdf>
- [10] Heineman, G., Hoxha, A., Döder, B., & Rehof, J. (2015). Towards migrating object-oriented frameworks to enable synthesis of product line members. *Proceedings of the 19th International Conference on Software Product Line*, 56-60. Retrieved from <http://doi.acm.org/10.1145/2791060.2791076>
- [11] Play (2017). The template engine.
<https://www.playframework.com/documentation/2.5.x/ScalaTemplates>

Appendix A: Table of Combinators

	Combinator Name	Use
1	MastermindHTML	Mastermind
2	HangmanHTML	Hangman
3	LightsOutHTML	LightsOut
4	FrankensteinHTML	Frankenstein
5	NimHTML	Nim
6	DummyHTML	Dummy
7	GameHTML	Generic
8	MastermindJS	Mastermind
9	HangmanJS	Hangman
10	LightsOutJS	LightsOut
11	FrankensteinJS	Frankenstein
12	NimJS	Nim
13	DummyJS	Dummy
14	GameJs	Generic
15	MastermindTitle	Mastermind
16	HangmanTitle	Hangman
17	LightsOutTitle	LightsOut
18	FrankensteinTitle	Frankenstein
19	NimTitle	Nim
20	DummyTitle	Dummy
21	MainPage	Generic
22	LevelList	Generic
23	SQLColumns	Generic
24	ScriptList	Generic

25	StateList	Generic
26	IndexHTML	Generic
27	AppJs	Generic
28	CSS	Generic
29	States	Generic
30	Controllers	Generic
31	SQL	Generic
32	Dictionary	Generic
33	LevelSelect	Generic
34	Settings	Generic
35	BindMainPage	Generic, extends Bind
36	BindHTML	Generic, extends Bind
37	BindJS	Generic, extends Bind
38	BindAppJS	Generic, extends Bind
39	BindCtrlJS	Generic, extends Bind
40	BindSqlJS	Generic, extends Bind
41	BindDict	Generic, extends Bind
42	BindGameHTML	Generic, extends Bind
43	BindGameJS	Generic, extends Bind
44	BindLevelSelect	Generic, extends Bind
45	BindSettings	Generic, extends Bind
46	BindCSS	Generic, extends Bind
47	Button	Generic Ionic component
48	Toggle	Generic Ionic component
49	Range	Generic Ionic component
50	Checkboxes	Generic Ionic component

51	RadioButtons	Generic Ionic component
----	--------------	-------------------------

Appendix B: Tutorial

This tutorial will guide you through making a puzzle app for Nim (<https://en.wikipedia.org/wiki/Nim>). It will teach you all the necessary steps to create an Ionic puzzle app using the cls-sbt software.

Setup:

Prior to using this software, you need to install and setup all of the required software. You will also need to clone [the tutorial github branch](#). Both can be found here: <https://github.com/Launchpad-MQP/Ionic-Strawman/tree/tutorial>.

You will also need Google Chrome to view the generated Ionic application.

In addition, each step is linked to commits for the start and end of the step. We recommend that you pull from the starting commit, and compare your code with the changes in the ending commit.

Step 1: Copy the dummy game

Estim. time: 30 minutes

Starting commit: 063de6dc3d8dd1ccfc9552cd4aeaec6553a95179

Ending commit: 180d4d1fad5ee8ed532a0204bd626a2d23238b15

- 1) In order to start, we will want the bare minimum combinators and files. The result will look nearly identical to the dummy example in the tutorial branch.
- 2) Add your game name type to the kinding with `.addOption('gameName)`. This will be 'nim' in our example. All of your combinators need to share this name.
`.addOption('nim)`
- 3) Create an empty `.twirl.js` file in the templates directory.
- 4) Create an empty `.twirl.html` file in the templates directory.
- 5) Create combinator for the game name.

```
@combinator object NimTitle extends Title('nim, "Nim")
```

- 6) Create combinator to render the HTML templates.

```
@combinator object NimHTML {
  def apply(): String = {
```

```

        return
        html.html.nim.render().toString()
    }
    val semanticType:Type = 'nim :&: 'html
}

```

7) Create combinator to render the JavaScript.

```
@combinator object NimJS extends RenderJS('nim :&: 'js, js.js.nim.render())
```

8) Change query so that we generate your game.

```
val reply = reflectedRepository.inhabit[Tuple] ('BoundFile :&: 'nim)
```

9) Run the `run.bat` or `run.sh` and confirm that the following files are created properly.

```

settings.html, level_select.html, style.css,
controllers.js, app.js, states.js, index.html, game.html,
sql.js, game.js, main.html

```

10) If Ionic doesn't automatically open the app in a web browser, you can view your app at `localhost:8100`

Step 2: Write the HTML

Estim. time: dependent on game. (120 minutes for nim example)

Starting commit: `180d4d1fad5ee8ed532a0204bd626a2d23238b15`

Ending commit: `48f6682d5d0e25c42fdd9d51dc4b0f9ac8edd753`

- 1) The next step to creating your Ionic puzzle app is creating the HTML page for game levels. This page will serve as the skeleton of a level's visual components which can be modified per level by using JavaScript.
- 2) There are a number of tools at your disposal for implementing Ionic framework components. In this example we will be using a range slider.
 - a) We must include the range combinator as an input to the HTML combinator.

```

@combinator object NimHTML {
  def apply(range:rangeType): String = {
    return html.html.nim.render(range).toString()
  }
}

```

```

    val semanticType:Type = 'range =>: 'nim :&: 'html
  }

```

- b) We can now use the function from the rangeType combinator in the HTML template. Other details and helpful combinators can be found near the top of `AppCreator.scala`.
 - c) Note that `{{levelNum}}` can be used to reference the level number. It is important to use this when naming components as the HTML is simply hidden when not in a level. This style can also be used to reference any JavaScript variable defined in `$scope`.
 - d) AngularJS can be used for any features not covered by a combinator.
- 3) For our nim app HTML, we will use five sliders separated by divs. The completed HTML is as follows.

```

@(range:(String, String, String, String, String, String, String) => Html)
<div class="row">
  @range("0", "1", "", "", "slider_{{levelNum}}_0", "test.blah0", "callback")
</div><div class="row">
  @range("0", "1", "", "", "slider_{{levelNum}}_1", "test.blah1", "callback")
</div><div class="row">
  @range("0", "1", "", "", "slider_{{levelNum}}_2", "test.blah2", "callback")
</div><div class="row">
  @range("0", "1", "", "", "slider_{{levelNum}}_3", "test.blah3", "callback")
</div><div class="row">
  @range("0", "1", "", "", "slider_{{levelNum}}_4", "test.blah4", "callback")
</div>

```

- 4) If you run `run.bat` or `run.sh` now, your level should be properly populated with sliders.

Step 3a: Basic JavaScript

Estim. time: dependent on game. (150 minutes for nim example)

Starting commit: `48f6682d5d0e25c42fdd9d51dc4b0f9ac8edd753`

Ending commit: `15e4c8d680d44e21058152e6b6f344f585a93485`

- 1) Now that you have a visual component to the level, you need to write the code and logic that will run the puzzle. This will all be done in JavaScript.
- 2) There are several features that you may want to use in your JavaScript.

- a) Use `$scope.variableName` for global variables and functions.
 - b) `$stateParams.levelNum` holds the current level number, starting at 0.
 - c) `$scope.completeLevel(boolean)` is used to end a level when it is completed. *boolean* should be true if the player was successful, or false if they failed.
 - d) Create `initializeLevel()`, `restartLevel()`, and/or `beforeLeave()` functions for extra functionality.
- 3) For nim, we need to set the pile sizes (represented by slider size) on initialization and when the player makes a move. We use DOM manipulation for the latter task.

```
function setSlider(i) {
  var value = $scope.sliders[i]
  var slider =
document.getElementsByName('slider_'+$scope.levelNum+'_'+i)[0]
  slider.max = value
  slider.value = value
}

$scope.initializeLevel = function() {
  $scope.sliders = [1, 2, 3, 4, 5]

  for (var i=0; i<$scope.sliders.length; i++) {
    setSlider(i)
  }
}
```

- 4) If you run the sbt software now, your sliders should initialize properly. Do not worry if they all appear the same size, this is changed later.

Step 3b: Win Condition

Estim. time: dependent on game. (30 minutes for nim example)

Starting commit: 15e4c8d680d44e21058152e6b6f344f585a93485

Ending commit: f24bbfeb302fb909bc03cb6ef19ed24017448be8

- 1) We must now implement the callback function, a function we referenced when creating the HTML, and a completion test.
- 2) In order for the sliders to be able use the callback function, we must define the function in `$scope`.
- 3) We must call `$scope.completeLevel()` when the level is completed.

4) The added code can be seen below:

```

$scope.callback = function(slider) {
  var i = parseInt(slider.split('_')[2])
  var slider = document.getElementsByName('slider_'+$scope.levelNum+'_'+i)[0]
  var value = parseInt(slider.value)
  if ($scope.sliders[i] == value) {
    return // User didn't change the slider
  } else {
    $scope.sliders[i] = value
  }
  setSlider(i)

  if ($scope.checkComplete()) {
    $scope.completeLevel(true)
  }
}

$scope.checkComplete = function() {
  for (var i=0; i<$scope.sliders.length; i++) {
    if ($scope.sliders[i] != 0) {
      return false
    }
  }
  return true
}

```

Step 3c: AI and Final Polish

Estim. time: dependent on game. (30 minutes for nim example)

Starting commit: f24bbfeb302fb909bc03cb6ef19ed24017448be8

Ending commit: 47036d34d51392c7388abbb36e5113bde8596ad4

- 1) From here on, everything you implement is up to you. We will be implementing an AI player to add some challenge to the game, as well as some graphical polish for the sliders.
- 2) The style change is modification to the `setSlider()` function; we change the slider width and display the size of the slider.

```

function setSlider(i) {
  var value = $scope.sliders[i]

```

```
var slider =
document.getElementsByName('slider_'+$scope.levelNum+'_'+i)[0]
slider.max = value
slider.value = value
var div = slider.parentElement
div.style.width = 100 * value / $scope.max + "%"
var rightIcon = div.getElementsByTagName('i')[1]
rightIcon.innerHTML = value
}
```

- 3) Feel free to implement your own AI; in this tutorial we implement an optimal AI that will occasionally make mistakes.