Applying Combinatory Logic Synthesis To Work With Existing Software Frameworks

A Major Qualifying Project report:

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the degree of

Bachelor of Science

by

Jacob Bortell

Date: October 17 2017

Approved:
Professor George Heineman, Major Advisor

1 Introduction

When a developer wants to use a large and complex framework, before they can write one line of code, what is their first step? Reading documentation, skimming tutorials, and laboring through examples. Hours are spent trying to understand how the application logic is embedded throughout the application domain—how the framework's designers took a large, abstract problem and implemented a partial, subjective solution with objects, design patterns, packages, and libraries. As the framework grew, the creators' original vision and intention, which seemed so obvious when the framework was small, became diluted in the name of modularity and extensibility. Now, to use it at all, its complexity requires hours of just "catching up", which no developer wants to do.

Typically there have been three approaches to developing large frameworks:

(1) Become an expert in the domain. As lamented, becoming an expert requires hours of preparation. Even when finally the developer gains a mental model, while they can explain how they understand it to another, the next developer still cannot use it without themselves internalizing the intricacies of the framework's protocol and forming their own mental model.

(2) Generate extensions from logical specification. Logically specifying a framework's usage is far too difficult, especially when its constructs cannot be defined mathematically.

(3) Develop units, which can be assembled. A framework of modular units describes any object-oriented domain. However, application logic is diluted as complexity increases.

Each approach focuses on learning code rather than on learning the abstractions of the application for which the code was written. Instead, we want to program directly with the abstractions of the application as an *interface* to the application domain, thereby completely hiding its complexity and totally ignoring implementation details. Precisely, we want to enable developers to program with typed abstractions, from which a compiler synthesizes into fully working native code, compilable and runnable against the framework, without developers becoming experts in that framework.

Combinatory Logic Synthesis (CLS) then is our solution to the large framework problem. Application logic is represented and typed in a functional language as *semantic types*; the developer requests synthesis of certain application features; the compiler searches for a type safe solution and when found, unboxes the semantic types into native types, synthesizing a fully working native application.

Our CLS proof-of-concept we demonstrate through the Archway variation of Solitaire, based on a Solitaire domain [2], which allows for development of feature-rich variations from a single product line. I will discuss CLS through example and personal experience, offering a practical overview of its use, its power for flexibility, and its potential for improvement. First, I will review the history of modern CLS and how the Solitaire application followed its development, then discuss its design and how to develop, debug, and test such a program, and conclude with benchmarks and final recommendations for future work.

2 Background

CLS is a type-based approach to component-oriented synthesis using types as interface specifications [5]. It is composed by a repository of *combinators*, which represent atomic parts of the application logic. Each combinator has a *semantic type* which promises an implementation type. For example, a combinator may have the semantic type `Celsius` and at synthesis give the native type `float`. Formally, CLS has combinatory repository Γ and type assumptions $x : \tau$, where x is a combinator's type and τ is the implementation type. Given a repository Γ and type τ, CLS uses an inhabitation algorithm to find a combinatory expression E such that $\Gamma \vdash E : \tau$. If the inhabitation algorithm can find E, then E is an *inhabitant* of type τ.

In the beginning, CLS' Γ repository was a flat collection of combinators whose implementation types were statically defined as strings. Combinators were coupled to the implementation language without any ability to manipulate it, leaving it impossible to model adequately a large domain. To increase flexibility, synthesis was separated into two domains. In the first domain, combinators are written in a functional meta-language which can manipulate native code. In the second domain is a lightly defined object model, providing structure but flexible for synthesis.

CLS proof-of-concepts have now been using Scala as the functional language to synthesize Java code. Scala runs within the JVM, so it is practical for synthesizing and generating Java. Combinators are represented as specially annotated Scala objects or classes, which take arguments. Each Scala combinator returns a *semantic type*, represented by an arbitrary Scala symbol like `'Celsius`, and has an `apply()` method, which when called returns a Java type, a Java AST type, or void, if it just performs an action upon the Java domain.

Before beginning with the CLS design of Archway, let us look at an example which demonstrates the construction and application of CLS in a small but complete application.

2.1 CLS and a Temperature Interface Application

Listing 1 exemplifies the "main" file of a CLS application. The repository of written combinators is loaded, and in order to synthesize the application, each semantic type we want to include in the final application is requested as a compilation unit. Listing 1 has only one, but the `addJob()` method, as shown on line 14, can add compilation units.

Listing 2 shows the Scala trait `TemperatureCombinators`, which is this application's main collection of combinators. Greater explanation will come in Section 3, but for now, notice the distinction between the return type of `apply()` method and of `semanticType` variable. As CLS specifies, the application logic is separated from the application domain. The semantic type represents the application logic—Celsius, Fahrenheit, Interface—rather than the implementation type—float, class, getter method. Instead, as the inhabitation algorithm searches for a combinatory solution, it examines the semantic types, and if it

```
class TemperatureApplication {

    // Load an already written repository of combinators
    lazy val repository = new TemperatureCombinators

    // Instantiate a dynamic repository of combinators.
    // We don't have any yet, but we'll get there.
    lazy val Gamma = ReflectedRepository(repository,
      classLoader = this.getClass.getClassLoader)

    // Request compilation units through semantic types.
    lazy val jobs =
      Gamma.InhabitationBatchJob[CompilationUnit]('TemperatureInterface)
      // .addJob[CompilationUnit]('SemanticType)

    // Synthesize!
    lazy val results = Results.addAll(jobs.run())
}
```

Listing 1: Main File of CLS Application

finds a path through a series of semantic types ending in the requested one, then it calls all the `apply()` methods, receiving their implementation types in order to construct the program.

If we synthesize the Temperature Application according to the requested compilation unit in Listing 1 `('TemperatureInterface)`, then CLS will offer *two* programs after the inhabitation algorithm finishes, because it will have found two routes to `TemperatureInterface`:

- `'TemperatureInterface` (returns temperature) ←
  `'TemperatureAPI` (returns Celsius temperature)
- `'TemperatureInterface` (returns temperature) ←
  `'FahrenheitToCelsius` (converts Fahrenheit from Celsius) ←
  `'TemperatureAPI` (returns Celsius temperature)

In the first program, the generated `Temperature.getCurrentTemperature()` will return Celsius, and in the second it will return Fahrenheit. However, if we change our request to `('TemperatureInterface :&: 'Fahrenheit)`, then the inhabitation algorithm will return one program using all three combinators.

CLS is so powerful, because from a library of prepared combinators it automatically deduces the correct sequences to produce a working native application. However, it became apparent that having a static collection of combinators was not sufficient to model a complex application. It also became apparent, as more combinators were written, that between certain sets of combinators there were only minute differences. CLS then evolved to allow *dynamic combinators*, parameterized Scala classes, which can be instantiated dynamically as combinators at time of synthesis and added to the Γ repository. They can be shared easily between repositories, saving the developer and the project from duplicate code

```
trait TemperatureCombinators {

  @combinator object FahrenheitToCelsius {
    def apply(expr: Expression): Expression = {
      Java("(9.0/5.0) * $expr.toString + 32.0").expression()
    }
    val semanticType: Type = 'Temperature :&: 'Celsius =>:
                             'Temperature :&: 'Fahrenheit
  }

  @combinator object TemperatureAPI {
    def apply: Expression = {
      Java("Temperature.getCurrentTemperature()").expression()
    }
    val semanticType: Type = 'Temperature :&: 'Celsius
  }

  @combinator object TemperatureInterface {
    def apply(exp: Expression): CompilationUnit = {
      val s = exp.toString
      Java(s"""|public class TemperatureAdapter {
               |  float getTemperature() {
               |    return $s;
               |  }
               |}
            """.stripMargin).compilationUnit()
    }
    val semanticType: Type = 'Temperature =>: 'TemperatureInterface
  }

}
```

Listing 2: Temperature Combinators

and empowering CLS to model applications of greater complexity. Examples and greater explanation on dynamic combinators come in Section 3.2.

My exercise as a proof-of-concept was to build the solitaire variation "Archway" using an existing domain model and a repository of combinators. I have built Archway before but manually. My first step? Reading the documentation, skimming tutorials, and laboring through examples. There was a precise, incremental process identical among all solitaire variations built in this framework. The task was to form a mental model, and then just copy and paste the same constructs in order to satisfy the requisites of the domain. I had to follow approach (1): become an expert in the domain. After 425 lines of code in eight Java classes (not including the underlying framework from which I extended my classes), my implementation looked like Figure 1.

There are three main physical elements of the game: the Tableau, which is the four columns in the middle; the Reserve, which is the arch the Tableau from 2 to Queen; and the Foundation, the piles of Aces and Kings on the left- and right-hand side of the window, respectively. The rules are few:

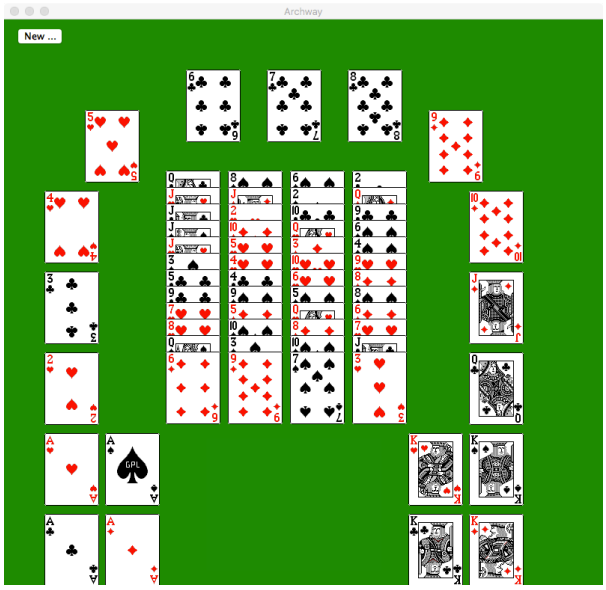

Figure 1: Original Implementation of Archway

- A card from the Reserve or the Tableau can be moved to the Aces Foundation if the card has the same suit and is one rank higher.
- A card from the Reserve or the Tableau can be moved to the Kings Foundation if the card has the same suit and is one rank lower.
- A card from the Reserve can be moved to the Tableau if the Tableau column is empty.
- The game is finished when all the cards from the Reserve and the Tableau are placed in the Foundations.

My task was to implement Archway again, this time with CLS.

3 Design

The Solitaire framework is designed after the Entity-Boundary-Controller model. Comprising the entity model are thirty-five Java classes, which among variations define and relate solitaire constructs constant among variations, such as cards, decks, piles, columns, gameplay spaces (foundation, tableau, reserve, waste pile), and moves (source, destination, moving element). Two Java classes support the Boundary, and combinators completely synthesize the Controller. Before CLS, when developing any new solitaire variation, the framework required that I manually defined and associated the following model elements:

- Containers classes (foundation, tableau, etc.) defined.
- View classes defined and associated with Containers.
- Controller classes defined and associated with Views.
- Move classes defined and associated with Controllers.

The framework still requires these definitions and associations, but because CLS separates the application logic from its domain implementation, *the developer need not learn any of the domain classes or their members*. Only in my development of Archway, which featured a card layout with a custom input of x-y positions, did I add about fifteen lines to one Java domain class. Instead of interfacing with the underlying framework, I needed only to write four Scala files (Figure 2):

- `Archway`: Loads repository of combinators, specifies compilation units to include in synthesis, and requests synthesis.
- `game`: Defines containers and move rules.
- `gameDomain`: Defines literal field members, their views, their views' placements, and any extra methods.
- `controllers`: Defines controllers.

Once `Archway.scala` requests synthesis, combinators generate all the classes, fields, definitions, associations, instantiations, and logic necessary to the model in real Java code.

Archway is a solitaire variation with many edge-case features:

- Two foundations for Aces and Kings respectively (framework allows definition of only one foundation).
- Non-rectangular card layout. Around the Tableau arches the Reserve, where at its left base is the Aces Foundation and at its right base is the Kings Foundation.
- Cards can be moved to and from the Tableau, but not to or from itself.

In section 4, I will cover each edge-case in more detail, as they demonstrate combinators' versatility in overcoming domain-stretching features.

3.1 Combinators

To represent and to process the application logic, the Solitaire CLS Framework uses Scala combinators, which generate compilable Java code with Twirl templates and static strings and which perform actions upon the Java domain in order to construct a runnable Java program.

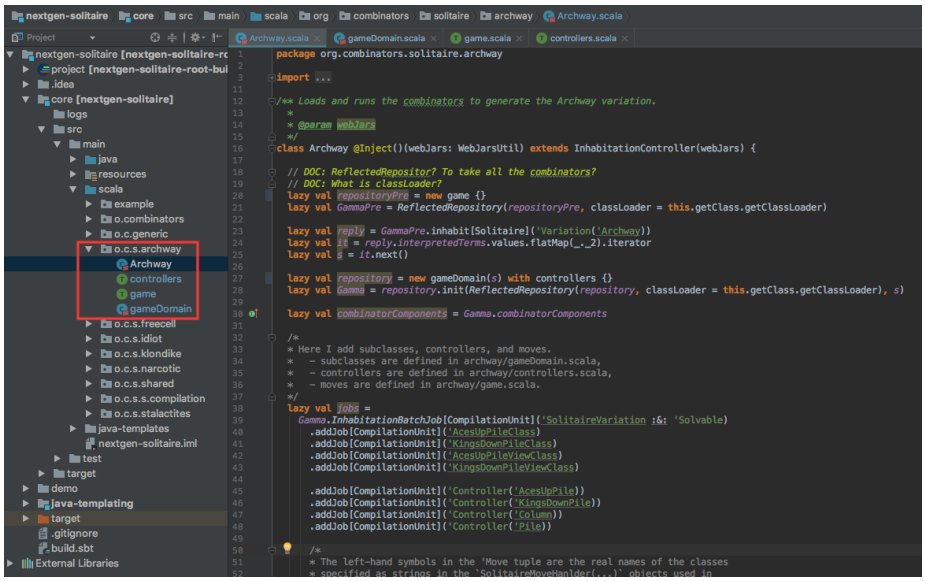

Figure 2: Directory structure in IntelliJ.

Every combinator has a *semantic type*, which the inhabitation algorithm inspects as it searches for a solution, and an *apply()* method, which returns a Java type, Java AST type, Scala type, or void if just performs an action upon the Java domain. If the inhabitation algorithm can use the combinator, then it calls the combinator's `apply()` method, which provides the concrete type or action promised by the semantic type. Listing 3 shows an example of a basic combinator.

```
@combinator object RootPackage {
  def apply: Name = Java("org.combinators.solitaire.archway").name()
  val semanticType: Type = 'RootPackage
}
```

Listing 3: Combinator to Provide Root Package of Variation

The semantic type is `'RootPackage`, an arbitrary Scala symbol that has no official declaration or specification. When the user requests synthesis of the Archway variation, the inhabitation algorithm along the way realizes that it needs something called a `'RootPackage`, and it will look for a combinator that promises `'RootPackage` until it finds this one. It then calls `RootPackage.apply()` and receives `org.combinators.solitaire.archway`, which is passed to a Twirl template with the line

```
package @Java(rootPackage);
```

where `rootPackage` is the parameter, and `@Java(...)` indicates a real Java statement, expression, or compilation unit. The synthesized Java line becomes

```
package org.combinators.solitaire.archway;
```

Listing 4 shows a more complex example by using a class of two combinators. When we pass `"archway"` to `ArchwayDomain`, together the combinators give

```
package org.combinators.solitaire.archway;
public class Archway extends Solitaire {/*...*/}
```

```
1  class ArchwayDomain(gameName: String) {
2
3    // Normalize string: Package name should be lower case,
4    // and class name should be capitalized.
5    val gameNamePackage = gameName.toLowerCase
6    val gameNameClass = gameName.toLowerCase.capitalize
7
8    @combinator object RootPackage {
9      def apply: Name = Java("org.combinators.solitaire.$gameNamePackage").name()
10     val semanticType: Type = 'RootPackage
11   }
12
13   @combinator object GameName {
14     def apply: SimpleName = Java(gameNameClass).simpleName()
15     val semanticType: Type = 'GameName
16   }
17 }
```

Listing 4: Combinator Class to Provide Root Package and Game Name

The Aces Foundation requires a different action than the Kings Foundation, so they both must be defined as subclasses of `Foundation`. Listing 5 shows a class combinator which generates a subclass with a Java string block. The semantic type on line 16 means that this combinator, when given a `'RootPackage` can give an `outSymbol`, which is whatever symbol the developer specifies, such as `'AcesUpPileClass`. All that I need to write in order to create the subclass is in the following line:

```
@combinator object AcesUpPile
  extends ExtendModel("Pile", "AcesUpPile", 'AcesUpPileClass)
```

When I request in `Archway.scala` the synthesis of `'AcesUpPileClass`, it will produce `AcesUpPile.java` as shown in Listing 6.

Therefore, instead of creating and writing two model classes for Aces and Kings Foundation and two more view classes to represent their views, I wrote four lines in `gameDomain.scala` like the single-line combinator above and another four in `Archway.scala` to request their synthesis.

```
class ExtendModel(parent: String, subclass: String, outSymbol: Symbol) {

  def apply(rootPackage: Name): CompilationUnit = {
    val name = rootPackage.toString()
    Java(
      s"""
         |package $name;
         |import ks.common.model.*;
         |public class $subclass extends $parent {
         |  public $subclass (String name) {
         |    super(name);
         |  }
         |}
       """.stripMargin).compilationUnit()
    }
    val semanticType : Type = 'RootPackage =>: outSymbol
  }
```

Listing 5: Class Combinator to Generate Subclass

```
package org.combinators.solitaire.archway;
import ks.common.model.*;

public class AcesUpPile extends Pile {
  public AcesUpPile (String name) {
    super(name);
  }
}
```

Listing 6: Generated Java Subclass

3.2 Dynamic Combinators

When the Solitaire/CLS proof-of-concept was first developed, there were only static combinators in a given Γ repository, processed at time of synthesis by the inhabitation algorithm in search of a solution. As certain sets of combinators were written, it was discovered that there were only minor differences between them, observed much in the same way as in the differences between solitaire variations. For example, two combinators would have to been written in order to synthesize one controller which handled `Pile` objects, and another to handle `Column` objects. To abstract the similarities, *dynamic combinators* were created, which can take parameters to dynamically define new combinators. Listing 7 demonstrates how the `WidgetController` dynamic combinator, which when given a Scala symbol, can at synthesis create a previously undefined combinator.

```
1  // Given a Scala symbol, construct a Controller.
2  class WidgetController(elementType: Symbol) {
3  def apply(
4    rootPackage: Name,
5    designate: SimpleName,
6    nameOfTheGame: SimpleName,
7    mouseClicked: Seq[Statement],
8    mouseReleased: Seq[Statement],
9    mousePressed: (SimpleName, SimpleName) => Seq[Statement]
10   ): CompilationUnit = {
11        // Call/render the Twirl template, generating these statements.
12        shared.controller.java.Controller.render(
13          RootPackage = rootPackage,
14          Designate = new SimpleName(elementType.name),
15          NameOfTheGame = nameOfTheGame,
16          AutoMoves = Seq.empty,
17          MouseClicked = mouseClicked,
18          MousePressed = mousePressed,
19          MouseReleased = mouseReleased
20        ).compilationUnit()
21      }
22 val semanticType: Type =
23   'RootPackage =>:
24   'NameOfTheGame =>:
25   elementType (elementType, 'ClassName) =>:
26   elementType (elementType, 'Released) =>:
27   elementType (elementType, 'Clicked) :&: 'NonEmptySeq =>:
28   elementType (elementType, 'Pressed) :&: 'NonEmptySeq) =>:
29   ('Pair ('WidgetVariableName, 'IgnoreWidgetVariableName) =>:
30   'Controller (elementType)
31 }
```

Listing 7: Dynamic Combinator to Create Controllers

`WidgetController` requires from the developer only a Scala symbol representing the controller to create, such as `'AcesController`. The parameters to `apply()` are passed from the repository. Note line 12: the call to `render` is referencing a Twirl template, in which all of `apply()`'s parameters are added. Recall that each controller requires three actions to be satisfied: Click, Press, and Release. The semantic type specifies explicitly the root package, the variation name, and that Click/Press/Release were satisfied, finally returning `'Controller('AcesController)`, a symbol which the developer can request for synthesis in `Archway.scala`.

Note that this dynamic combinator is missing the `@combinator` annotation seen in other combinators. Without the annotation, this class is not considered as a combinator in the Γ repository. To earn its name, a `WidgetController` object is instantiated at time of synthesis and added to the Γ repository. Therefore, instead of being statically defined, the Γ repository can be updated inside Scala traits which have an `init` method taking the Γ repository as input and updating it (Listing 8).

```scala
trait ArchwayControllers {
  // Receive and update the Gamma Repository at time of synthesis
  override def init[G <: SolitaireDomain](gamma : ReflectedRepository[G],
    s: Solitaire) : ReflectedRepository[G] = {

  // Add new combinator to Repository.
  val gamma = super.init(gamma, s)
  val updatedGamma = gamma.addCombinator(
    new WidgetController('AcesController)
  )

  // Return updated Repository.
  updatedGamma
}
```

Listing 8: Scala trait receives and updates Γ repository.

3.3 Scala Functions

In addition to combinators, Scala functions can easily generate commonly used Java blocks, such as associating model elements with view elements. In Listing 9, `loopConstructorGen()` takes a Java container object, the type of the model element, its name, and the view's name, and it constructs a loop which associates all container elements with a view, generated as shown in Listing 10.

```scala
val reserve = loopConstructGen(reserve, "Pile", "fieldReservePiles",
   "fieldReservePileViews")

def loopConstructGen(cont: Container, modelName: String,
  viewName : String, typ:String): Seq[Statement] = {
  Java(
  s"""
     |for (int j = 0; j < $cont.size(); j++) {
     |  $modelName[j] = new $typ(${modelName}Prefix + (j+1));
     |  addModelElement ($modelName[j]);
     |  $viewName[j] = new ${typ}View($modelName[j]);
     |}""".stripMargin).statements()
```

Listing 9: Scala function to construct a Java loop.

4 Development

Here I will describe the application environment and the development cycle which I used to code Archway, to synthesize the project, to view the results, to debug, and to fix errors.

JetBrain's "IntelliJ IDEA" is a professional IDE which has a Scala plugin allowing seamless integration between Java and Scala, providing sbt-compilation, auto-completion, symbol lookup, and reverse symbol lookup. In any Scala file,

```
for (int j = 0; j < 11; j++) {
  fieldReservePiles[j] = new Pile(fieldReservePilesPrefix + (j + 1));
  addModelElement(fieldReservePiles[j]);
  fieldReservePileViews[j] = new PileView(fieldReservePiles[j]);
}
```

Listing 10: Generated Loop Block

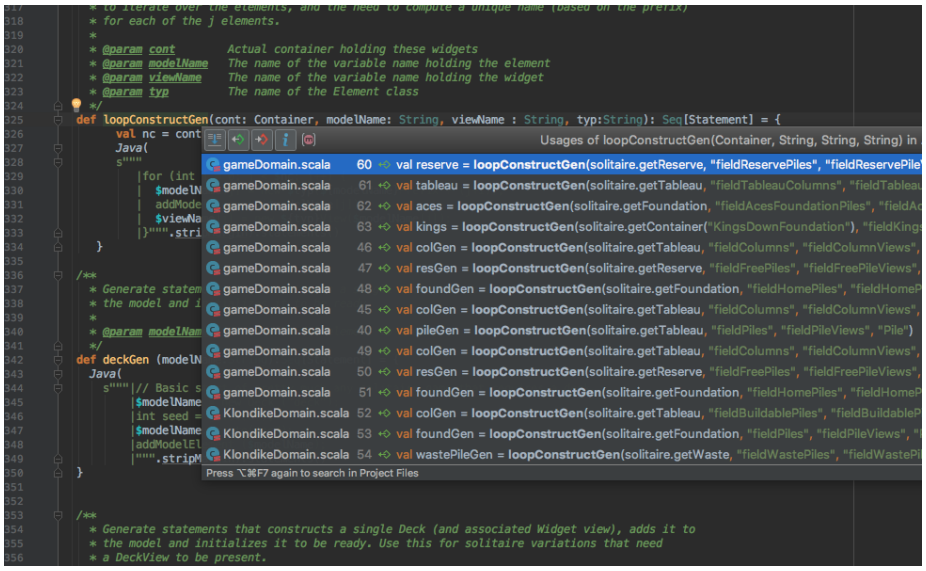

Figure 3: IntelliJ Reverse Lookup of Scala Function

I could instantly find the definition of a combinator, Scala function, or Java class, as well as all of their usages in the project (Figure 3). One of our original project goals was to find a way to view and to organize combinators, however it seemed largely fulfilled once we began using IntelliJ. Additionally, a second goal was to visualize the project's collection of semantic types, which are Scala symbols. Scala symbols are arbitrarily defined, so it is important that when the developer means to use a certain symbol, they can find it if already exists and which combinators use it.

IntelliJ has a "Find-all" search, which can reveal all instances of any symbol in the project 4), but the convenience is just a partial solution. Development in CLS revolves around semantic types, and how they are used among combinators. It would be useful, especially for large frameworks, to have a special organization and visualization of combinators, besides their location on disk, where the developer could see which combinators return what Scala symbols and implementation types, as well as the Scala symbols required for input.

To synthesize Archway, I would open a session in iTerm, a MacOS teriminal

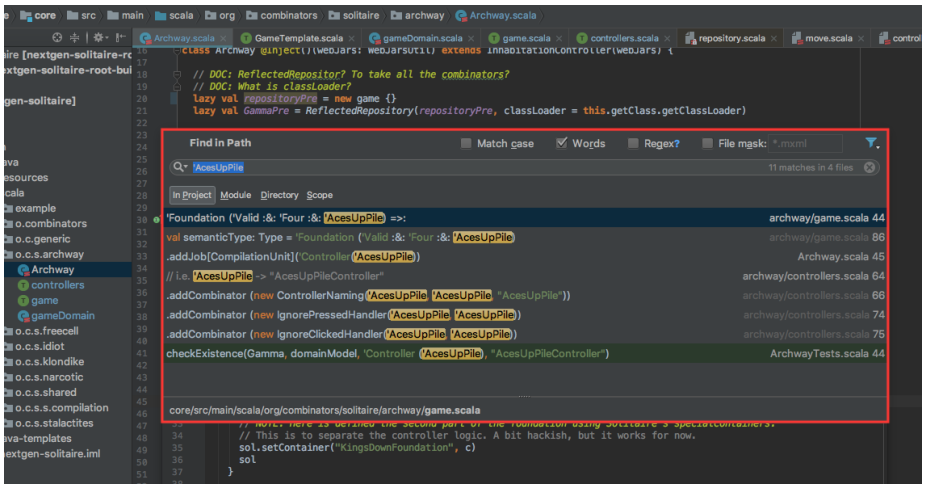

Figure 4: IntelliJ can find all instances of a symbol.

emulator, start `sbt`, run the CLS application (Figure 5), and request synthesis in the Safari browser with `localhost:9000/archway`.

The next step would be to clone the git repository of the native code, which I then would review in the Atom text editor (Figure 6), and compile and run in the terminal.

One can see that CLS intuits a new iterative development cycle. In a traditional project the cycle might be Edit, Compile, Run. In CLS the cycle is Edit Scala, Synthesize, Generate, Compile, Run, Edit Java (Figure 9).

(1) Edit Scala combinatory logic. In CLS, applications are built not from the top-down, but from the bottom-up. Already in place are the domain abstractions and methods necessary for a full-featured variation to function. Instead, programming is more like configuration than construction. Configure the controllers, the move rules, the cards' positions in the window, and CLS will write the classes, statements, loops, and methods to fulfill the configuration.

- Controllers in `controllers.scala`: specify their names and types, and whether the user will Click/Press/Release this controller.
- Moves in `game.scala`: specify their names and logical constraints, and to which Containers they belong.
- Views in `gameDomain.scala`: associate Containers with views and specify their x-y coordinates.
- Extra Fields and Methods in `gameDomain.scala`: specify any helper methods or fields not provided by default. Archway has a particular card setup, so I had to write a helper method, generated at synthesis, to place and order the cards. When writing helper methods, I made the most Java syntax

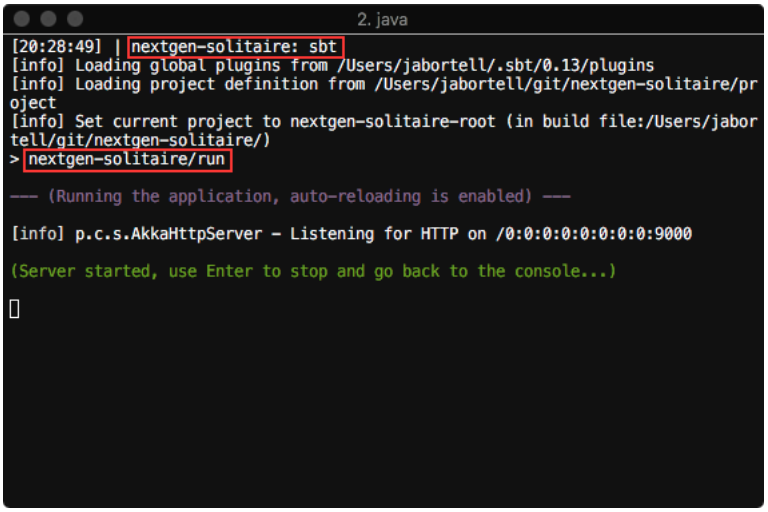

Figure 5: Running CLS in the terminal.

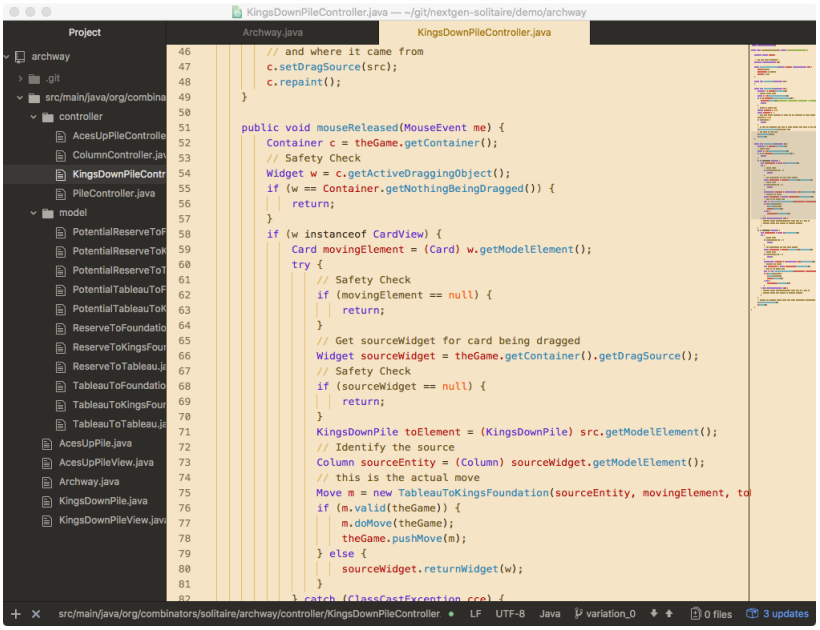

Figure 6: Native code in Atom text editor.

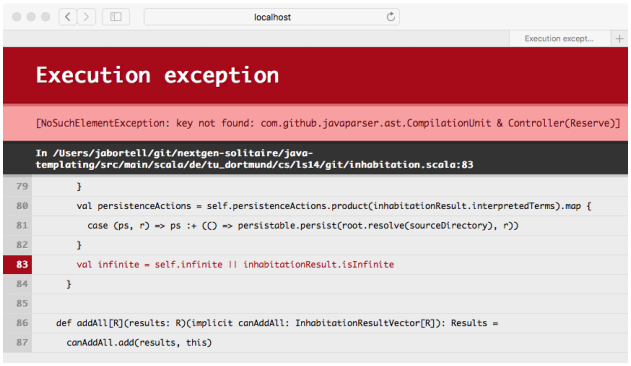

Figure 7: Failed Synthesis

mistakes, which CLS only reports after synthesis, when code generation is requested, a problem described in step (3) below.

- Feature selection in `Archway.scala`. Each Controller, Move, and extra class must be requested for synthesis as shown in Listing 11.

```scala
class Archway {

  lazy val jobs =
      Gamma.InhabitationBatchJob[CompilationUnit]('AcesUpPileClass)
      .addJob[CompilationUnit]('Controller('AcesUpPile)
      .addJob[CompilationUnit]('Controller('Reserve))
      .addJob[CompilationUnit]('Controller('Tableau))
      .addJob[CompilationUnit]('Move('ReserveToTableau))
      .addJob[CompilationUnit]('Move('ReserveToAcesFoundation))
      // ... more combinators to request...
  // Find solution.
  lazy val results = Results.addAll(jobs.run())
}
```

Listing 11: Request Synthesis of Components

(2) Synthesize project. First the CLS application is started with SBT and run with `nextgen-solitaire/run`. The application waits for a request through `localhost:9000`, such as `localhost:9000/archway`, then begins the synthesis process, usually taking about two minutes for a feature-rich variation such as Archway. If synthesis fails, the reason usually a missing combinatory inhabitant, then the missing compilation unit is reported and synthesis is cancelled (Figure 7). then the developer has to pull back features and return to step (1) until synthesis succeeds.

If synthesis is successful, then a page will open displaying the requested compilation units, a list of available variations with an option to generate a git

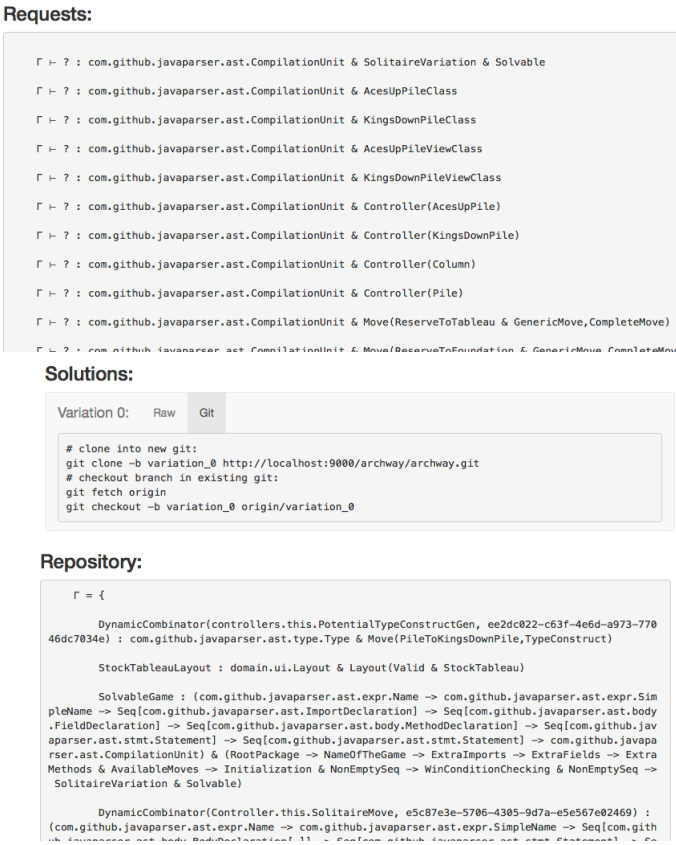

Figure 8: Successful synthesis returns the requested compilation units, the available variations, and the contents of the Γ repository.

repository of the native code, and the contents of the Γ combinatory repository (Figure 8).

Looking through the repository, one can find the names of the requested compilation units and the sequence of semantic types and combinators which produce combinatory types. When synthesis fails from a missing Controller inhabitant, the developer, after pulling back features until it succeeds, can deduce what inhabitant combinatory type is missing. For example, if the repository shows that it satisfied a controller's Click and Press but not release, then the developer can narrow the debugging scope.

(3) Request generation of Java code (middle figure in Figure 8). If generation fails, then there was a parsing error when constructing the Java AST (Figure 10). The developer must return to fix the violating Java string in the project and synthesize again.

(4) Compile and run the project. If compilation fails, it is most likely due to a

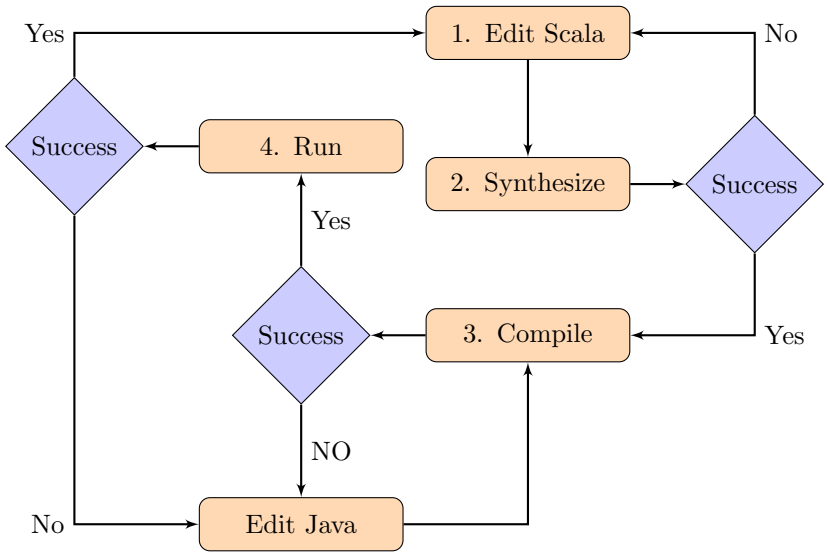

Figure 9: CLS Development Cycle

misspelled, generated Java name. Another common reason is that a field may be successfully generated before its type is defined in a class. In my development, I generated the necessary associations between View and Controller must sooner than successfully synthesizing the Controllers.

(5) Edit combinatory logic which produced the errors. Fixes usually include fixing Java strings and missing-inhabitant errors. Once errors are fixed, then the developer continues with new features to the project.

CLS has no complimentary IDE plugin or any UI features which would ease and quicken the cycle. Such helpful features would be:

- Synthesis directly requested from and displayed in IntelliJ.
- Reverse-lookup from synthesized Java code to Scala combinator.
- Inhabitation trace on failure.

4.1 Refactoring

There are three types of refactorings typical of CLS:

4.1.1 Static Combinator Object to Dynamic Combinator Class

When a combinator's logic can be abstracted to produce many combinators useful to other domains, then it is refactored to become its own class with parameters for variation. As a small example, review the the static combinator in Listing 12.

```
@combinator object ArchwayReserve {
def apply(): Reserve = {
  val reserve = new Reserve()
    for (_ <- 1 to 11)
      reserve.add(new Pile())
    reserve
  }
  val semanticType: Type = 'Reserve ('Valid :&: 'Eleven :&: 'Pile)
}
```

Listing 12: Scala code to instantiate `Pile`s for the Reserve

The `ArchwayReserve` static combinator merely instantiates `Pile`'s in the Reserve Container. However, because it is a static combinator, defined for one variation's repository, combinators initializing the Reserve would have to be written for each variation. The only difference would be the number of `Pile`'s in the Reserve. We can refactor this combinator to a dynamic combinator, as shown in Listing 13

```
class NPileReserve(n: Int, nAsType: Type) {
  def apply(): Reserve = {
    val reserve = new Reserve()
    for (_ <- 1 to n)
      reserve.add(new Pile())
    reserve
  }
  val semanticType: Type = 'Reserve ('Valid :&: nAsType :&: 'Pile)
}
// Combinator specific for Archway
@combinator object ElevenPileReserve extends NPileReserve(11, 'Eleven)
```

Listing 13: Static to Dynamic Combinator

Now, any variation can extend `NPileReserve` and instantiate the Reserve in synthesis without rewriting the full combinator.

4.1.2 Scala Code to Scala Method

`loopConstructGen` (Listing 9), is a primary example of Scala code moving to a method. It is called four times in the Archway variation, for each container, and many more times in other variations. Refactoring code to methods is not only for convenience but to assure correctness. Developing solitaire variations requires much string manipulation, so the less copying-and-pasting, the less likely a Java AST error will stall actual development.

4.1.3 Framework Edit

To produce the Archway variation, I added about 15 lines to a Java domain class and added one Scala method to provide a completely custom layout for solitaire

elements, when previously, elements could only be laid out in iteratively defined rectangles. Editing the underlying framework is rarely necessary. Nearly all the of the application logic, as CLS specifies, is separated to Scala combinators. However, for domain-stretching variations such as Archway, it could be necessary.

5 Debugging

There is no mechanism to debug combinators directly by stepping through the synthesized code. Instead, debugging is an iterative process focused around synthesis. If synthesis is unsuccessful, then the developer debugs the Scala code until it is successful. Then they debug any errors in the Java code, return to the combinator producing the errors, and resynthesize to confirm the fix.

In developing Archway, I frequently made the following types of mistakes and errors:

(1) Misspelled Java name, which produces an undefined reference. For example, in order to synthesize a model element, such as the Aces Foundation, correctly associating itself with controllers and views, and defining their placement, the string `fieldAcesFoundationPileViews` is passed to three Scala methods, and the string `fieldAcesFoundationPiles` is passed to one Scala method and referenced three times in a generated Java block. A misspelling anywhere will produce undefined references after synthesis. Although annoying, it is not too difficult to correct the problem. If I had misspelled the former string as `fieldAcesFoundtionPileViews` in a certain Scala method, and after synthesizing the project, found an undefined reference in the following block:

```
for (int j = 0; j < 4; j++) {
    fieldAcesFoundationPiles[j] =
      new AcesUpPile(fieldAcesFoundationPilesPrefix + (j + 1));
    addModelElement(fieldAcesFoundationPiles[j]);
    // Uh oh... undefined reference.
    fieldAcesFoundtionPileViews[j] =
      new AcesUpPileView(fieldAcesFoundationPiles[j]);
}
```

First I would correct the string and compile the project to verify the fix. I would then copy the undefined reference and in IntelliJ open a Find-All window, locating the string as a parameter to `loopConstructGen()`, which associates model elements with their views. Correcting the string in the Scala method and resynthesizing the project would fix the error.

(2) Grammar mistake in a Java string block, which produces an AST error, still allowing synthesis but preventing code generation. When the user selects `Compute` to generate the Java code and the git repository, as in Figure 8, generation will fail, and CLS will return the AST error in the terminal as in Figure 10. The only solution really is to return to the most recently added Java string block, and look for syntax mistakes. An on-the-fly Java AST parser

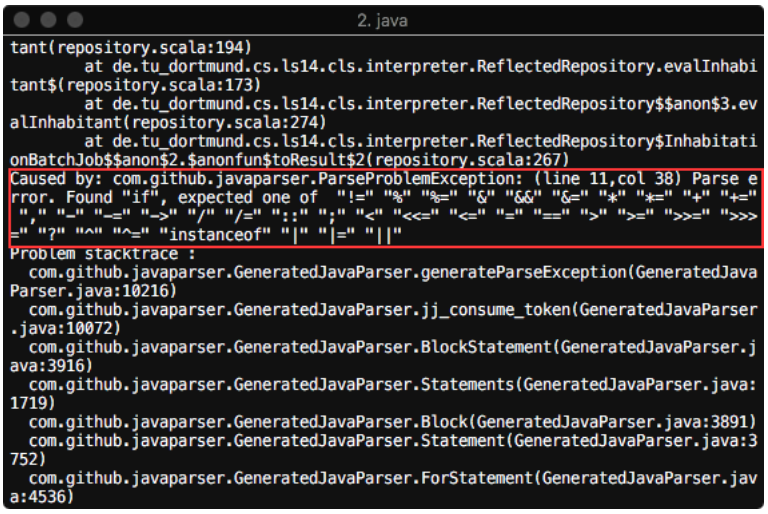

Figure 10: Java AST error after synthesis.

as an IntelliJ plugin would save many round-trips and much stress over missing semi-colons and parentheses.

(3) Unclear use of string in combinator, which at its best produces an undefined reference and at its worst halts synthesis. In a combinator, a string:

- May reference a Java field, class, or type which exists in the Solitaire domain or in a Twirl template. In

  ```
  val sameSuit = new SameSuit("movingCard", "destination.peek()")
  ```

  `SameSuit()` is a Constraint object, used in defining moves, which expects the user to reference real field names defined in the Controller Twirl template. Unfortunately, this breaks the fourth wall of the domain, requiring the user to have some understanding of existing fields and their Java types.

- May become a class or field itself. For example, when defining moves, the first argument to `SingleCardMove()` becomes the class name of that move.

- May reference a field or class which will be created as specified in another combinator. See paragraph (1) for an example.

While versatile, string synthesis can have unexpected manifestations, but with documentation and IntelliJ's symbol-lookup, I learned its usages. As the Solitaire CLS domain is under development, its maintainers are actively refactoring combinators to remove ambiguity by synthesizing more elements with fewer user-specified strings.

(4) Missing inhabitant, which prevents synthesis (Figure 7). This is the most common CLS-specific error. It means that in the inhabitation algorithm's

search for a path of semantic types, it failed to find the final requested type. The developer then has to pull pack iteratively the requests, combinators, and expressions related to the final type until synthesis succeeds, then add them back until finding the suspect combinator.

(5) Domain modeling gap. For edge-case variations such as Stalactites or Archway, there are features which force the developer to either refactor the underlying Java domain or to find a solution through the Scala domain. These features which expose modeling gaps are like those in a traditional software project: a user finds an error at runtime exposing a use case not articulated earlier in development.

In Archway I encountered two domain modeling gaps. In the first, I had explicitly defined move rules between all containers: from Reserve to Aces/Kings Foundation, Tableau to Aces/Kings Foundation, and Reserve to Tableau. For each controller, CLS generates a Java expression which casts the source container, passed in as the base class, to its actual class. For example, when a card is moved from the Reserve, the variable referencing the reserve is casted to `Pile`. At runtime, I found that when a card was moved from the Tableau and then released on the Tableau, the card disappeared: there was no Tableau-to-Tableau rule, because there is no such move in Archway Solitaire. Because I had not explicitly defined a rule where the Tableau was a source *and* the destination, the logic fell through to another Java block which threw an exception when trying to cast the Tableau, a `Column`, to a `Pile`. Normally, in a non-CLS project, I would have to add additional logic *somewhere*: maybe refactoring the controller logic, the move classes, or hard-coding a hack to ignore the problem. However, with CLS, the problem was completely solved in two lines of code: one to add a rule specifying that any move from Tableau to Tableau always returned false, and another to request synthesis of this move.

I found a similar problem when developing the Reserve, from which cards are never moved, only placed. Recall that each controller must have satisfied three actions: Click, Press, and Release. Archway has no deck, so the combinator `IgnoreClickedHandler` is added to all controllers. If a container should not have cards removed from itself, such as the Aces/Kings Foundation, then it gets the `IgnorePressedHandler`. However, for the Reserve, I needed a combinator which denied the player from releasing the card on the container. Before, there had been no need for this combinator, but adding it only required five lines of code for the new combinator. I associated it with the Reserve, resynthesized, and the feature was immediately present.

6 Testing

In IntelliJ we can unit test CLS combinators with coverage on both Scala and Java code. Variations are constructed and tested piece-wise: a feature is added, such as a controlle,r synthesis is requested, and then on success, the test checks the existence of the generated controller class. As shown in Listing 14, Archway is synthesized iteratively, first testing the existence of domain model elements

such as the Reserve, then after adding the controllers, their existence as generated classes.

Because dynamic combinators are shared between applications, it is important to know that when one is changed, other applications or variations of a single application correctly synthesize after the change. IntelliJ makes this easy, as all test suites can be run after any change by the developer.

```
class ArchwayTests {

  // Initiate synthesis.
  describe("Inhabitation") {
    lazy val domainModelRepository = new Archway

    lazy val GammaDomainModel =
      ReflectedRepository(domainModelRepository,
        classLoader = this.getClass.getClassLoader)

    lazy val possibleDomainModels: InhabitationResult[Solitaire] =
      GammaDomainModel.inhabit[Solitaire]('Variation('Archway))

    // Synthesis successful?
    it("Not infinite.") {
      assert(!possibleDomainModels.isInfinite)
    }

    // Now test existence of model elements.
    describe("Domain Model") {
      lazy val domainModel = possibleDomainModels.interpretedTerms.index(0)
      it("Reserve is size 11.") {
        assert(domainModel.getReserve.size == 11)
      }
      // ... test existence of other elements...

      // Add controllers, synthesize them, and test for existence.
      lazy val archway_repository = new gameDomain(domainModel) with ArchwayControllers
      lazy val Gamma = archway_repository.init(
        ReflectedRepository(archway_repository,
          classLoader = this.getClass.getClassLoader), domainModel)

      checkExistence(Gamma, domainModel, 'SolitaireVariation :&: 'Solvable, "Archway")
      checkExistence(Gamma, domainModel, 'Controller ('Pile), "ReserveController")
      // ...
   }
  }
}
```

Listing 14: Unit Testing in CLS

6.1 CLS and Correct-by-Construction

With dynamically generated Java code from Twirl templates and static strings, in addition to the semantic type specification of CLS, the question is, "Can we verify that the synthesized product is correct if we verify the combinators are

correct?" At this time the answer is no, for the following reasons:

1. Java strings are not verifiable, syntactically or otherwise, before synthesis. Many times compilation failed because I referenced a field incorrectly or missed a semicolon. One solution might be to integrate the Twirl template system with IntelliJ, then reference field names as components instead of as strings. However, for dynamically specified fields, prior reference is impossible.

2. Scala symbols have infinite semantic flexibility, but they have no real association with real Java types. A combinator may promise a semantic type of `'Integer` or even more imprecisely `'Temperature`, but it may actually give a method returning `String`. Should the final, concrete type be an integer or a string telling the temperature? Allowing a tighter specification between application logic and the domain model might improve the development cycle.

7 Conclusion

Combinatory Logic Synthesis is a novel way of separating a domain's application logic from its object model, allowing the synthesizing of application variations with minimal edits to the underlying framework. Dynamic combinators enable the developer to model complex applications and code duplication by instantiating Scala classes as combinators at time of synthesis. Although at this time developing with CLS involves more steps than in a typical development cycle, with improved IDE integration, CLS can become a serious and powerful tool for software engineering.

7.1 Archway

To synthesize the Archway variation, I wrote about 450 lines of code in 5 scala files, added two Java class placeholders, and refactored one Java class, to produce a Solitaire variation with 963 lines of code in 20 Java class files. With a 2.5 GHz Intel Core i5 and 4GB of RAM, synthesis took about 1 minute 38 seconds, and computing the Java AST took about 11 seconds.

As shown in Figure 11, synthesized Archway is identical to the original. Notice, that there is an additional button next to `New...` called `Solve`. Unimplemented originally but in synthesized Archway, `availableMoves()` was added as an extra method to find the sequence of moves which would win the game. All that was necessary was to write the method, and synthesis properly integrated it into the application. Unfortunately, after attempting in several new games, the depth-first search found no solutions, suggesting that Archway is not the solitaire game to play if you are interested in winning.

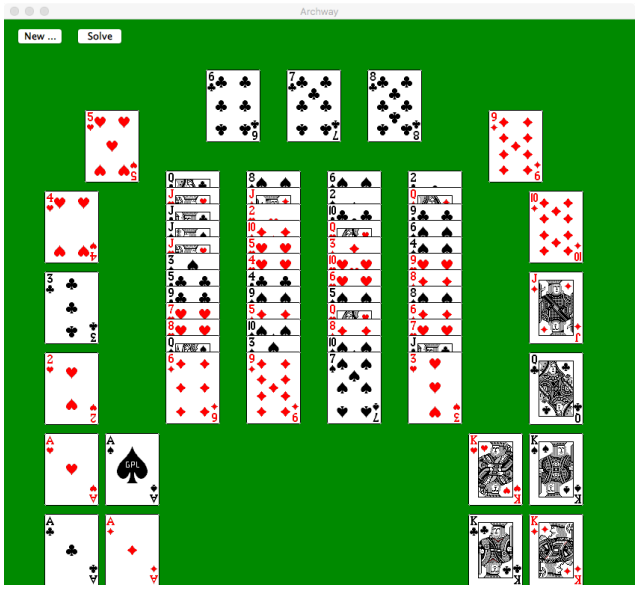

Figure 11: Synthesized Archway

7.2 Future Work

To produce one variation, CLS is a tool too complex to use, but to produce four, ten, fifty variations — it is indispensable. Pairing with an IntelliJ plugin, CLS could become a serious software engineering tool. The most needed features are organization of combinatory symbols and linting for Java strings in Scala code.

Because CLS separates synthesis between a functional language and the native object-oriented language, CLS is under development to fully decouple the languages from each other. Abstracting out the Γ repository as an interface to a concrete repository would allow seamless swapping of the native language. In other words, one of the components requested for synthesis could be the native language itself. The application logic would become completely independent of the domain, such that the domain could be Java, Python, C++, or Rust, but the developer would write combinators in just a single repository.

References

[1] Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, pages 67–86, 2014.

[2] George Heineman, Armend Hoxha, Boris Düdder, and Jakob Rehof. Towards migrating object-oriented frameworks to enable synthesis of product line members. In *Proceedings of the 19th International Conference on Software Product Line*, pages 56–60. ACM, 2015.

[3] Dan Li, Xiaoshan Li, Zhiming Liu, and Volker Stolz. *Interactive Transformations from Object-Oriented Models to Component-Based Models*, pages 97–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] David J Pearce. A calculus for constraint-based flow typing. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2013.

[5] Jakob Rehof and Moshe Y. Vardi. Design and Synthesis from Components (Dagstuhl Seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014.

List of Listings