Chapter 1

Evaluation

1.1 Experiment DSL Formal Specification

1.1.1 Introduction

An experiment comprises a set of research hypotheses, each of which is a statement on the measured effects of treatments. To determine the effect of treatments, a research design defines how to apply them to experimental objects. When such application is performed during experiment execution, the effect on dependent variables is measured by the corresponding instrumentation. This generates a series of data points that are analyzed to confirm or refute the hypotheses according to statistical tests corresponding to the type of statement on the research hypotheses. The semantics of an experiment consists of the confirmation/dismissal of its hypotheses.

The overall experiment semantics is despicted in Figure 1.1. In order to test the research hypotheses, first of all an experiment must be specified. Then this specification is used to compile an execution script and to generate an analysis script. Next, the infrastructure uses the execution script to execute all its applications producing a series of data points. Finally, the data points are analyzed by the analysis script to confirm or refute the hypotheses specified in the experiment specification.

Execution script compilation takes an experiment specification and produces an execution script according to the hypotheses and the experimental design. An execution script contains a series of applications in which each application contains an instrument, related to the dependent variable; an execution command, related to the treatment; and an argument, related to the experimental object.

During execution, each application command is executed by the infrastructure using the related argument. In addition, the instrument is applyed to collect the value for the

dependent variable. The outcome is a series of values in which each element corresponds to the result of the execution of an application described in the execution script.

Analysis script generation takes an experiment specification and generates an analysis script according to the hypotheses and the experimental design. An analysis script contains a series of statistical tests. Each test contains an analysis function and two arguments. The arguments contain a dependent variable name, a treatment name, and an object name, which are used filter execution results.

Finally, execution results are analyzed by the generated analysis script. During analysis, each argument is applied to a filter function to filter the results related to the argument. Next, the analysis function is applyed using two sets of data corresponding to each argument. The result of analysis is a set containing the results of each hypothesis.
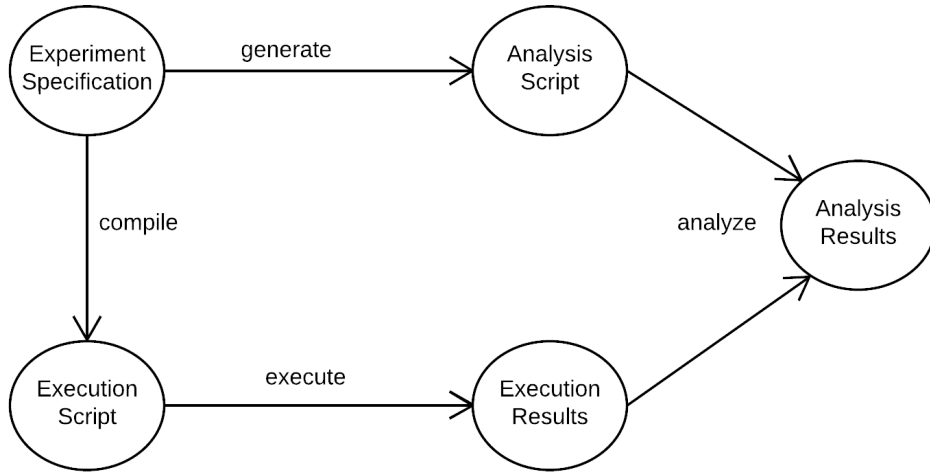

Figure 1.1: DSL semantics

1.1.2 DSL Model

DSL Model is listed in Listing 1. We represent types as records and we use e.hypotheses to access the field hypotheses of a given experiment $e \in E$, for example. In addition, we use overline to represent lists. For instance, hypotheses are represented by type H. So, $\overline{H}$ represents a list of hypotheses. Experiments contain a list of hypotheses, a design, a list of treatments, a list of objects, and a list of dependent variables. We also assume the existence of primitive types, such as String, Int, and Float.

Listing 1 DSL Model

$E ::= \{hypotheses : \overline{H}, design : D, treatments : \overline{T}, objects : \overline{O}, dependentVariables : \overline{DV}\}$

$H ::= \{name : String, dependentVariable : DV, treatment1 : T, operator : String,$
$treatment2 : T\}$

$D ::= \{runs : Int, designFunction : \overline{T} \times \overline{O} \to \overline{(T, O)}\}$

$T ::= \{name : String, command : String\}$

$O ::= \{name : String, argument : String\}$

$DV ::= \{name : String, instrument : String\}$

$ES ::= \{applications : \overline{A}\}$

$A ::= \{instrument : String, command : String, argument : String\}$

$ER ::= \{dependentVariableName : String, treatmentName : String, objectName : String,$
$value : Float\}$

$AS ::= \{hypothesesTests : \overline{HT}\}$

$HT ::= \{hypothesisName : String, analysisTests : \overline{AT}\}$

$AT ::= \{analysisFunction : \overline{ER} \times \overline{ER} \to AR, argument1 : ARG, argument2 : ARG\}$

$ARG ::= \{dependentVariableName : String, treatmentName : String,$
$objectName : String\}$

$HR ::= \{hypothesisName : String, testResults : \overline{TR}\}$

$TR ::= \{objectName : String, analysisResult : AR\}$

$AR ::= \{result : String\}$

$compileExecutionScript : E \to ES$

$generateAnalysisScript : E \to AS$

$execute : ES \to \overline{Float}$

$generateListOfExecutions : E \to \overline{(T, O, DV)}$

$zip : \overline{(T, O, DV)} \times \overline{Float} \to \overline{ER}$

$compileApplication : (T, O, DV) \to A$

$analyze : \overline{ER} \times AS \to \overline{HR}$

$suitableFunction : H \to (\overline{ER} \times \overline{ER} \to AR)$

1.1.3 Haskell Model

The model presented in the previous section has a correspondent model specified in Haskell language, which is presented in Appendix A. Implementation of functions is presented only in Haskell model. The correspondence between types specified in Section 1.1.2 and in Appendix A is showed next.

Listing 2 Models correspondence

$E \Leftrightarrow Experiment$

$H \Leftrightarrow ResearchHypothesis$

$D \Leftrightarrow ExperimentalDesign$

$T \Leftrightarrow Treatment$

$O \Leftrightarrow ExperimentalObject$

$DV \Leftrightarrow DependentVariable$

$ES \Leftrightarrow ExecutionScript$

$A \Leftrightarrow Application$

$ER \Leftrightarrow ExecutionResult$

$AS \Leftrightarrow AnalysisScript$

$HT \Leftrightarrow HypothesisTest$

$AT \Leftrightarrow AnalysisTest$

$ARG \Leftrightarrow Argument$

$HR \Leftrightarrow HypothesisResult$

$TR \Leftrightarrow TestResult$

$AR \Leftrightarrow AnalysisResult$

1.1.4 Functions Type Checking

In this section we show the correctnes of functions listed in Appendix A regarding their types, despite their having been already type checked by Haskell compiler.

1.1.4.1 compileExecutionScript

`compileExecutionScript :: Experiment -> ExecutionScript`

The return type of this function is `ExecutionScript` since it calls the constructor `ExecutionScript` passing an argument of type `[Application]` resulting from `map compileApplication (generateListOfExecutions experiment)`. The `map` function has the following signature: `map :: (a -> b)-> [a] -> [b]`. The first argument is a function, `compileApplication`. Type `a` is `((Treatment,ExperimentalObject, DependentVariable))` and type `b` is `Application`. So, the result of applying this `map` function is of type `[Application]`, which is the expected type for the constructor `ExecutionScript`. The second argument of the `map` function results from applying the function `generateListOfExecutions` to the argument of type `Experiment`.

1.1.4.2 compileApplication

```
compileApplication ::(Treatment, ExperimentalObject, DependentVariable)->Application
```

    The return type of this function is `Application` since it calls its constructor using arguments that come from accessor functions `instrumentCommand` from `DependentVariable`, `treatmentCommand` from `Treatment`, and `argument` from `ExperimentalObject`.

1.1.4.3 generateListOfExecutions

```
generateListOfExecutions :: Experiment -> [(Treatment, ExperimentalObject, DependentVariable)]
```

1.1.4.4 removeDuplicates

```
removeDuplicates :: (Eq a)=> [a] -> [a]
```

1.1.4.5 applyDesign

```
applyDesign :: ExperimentalDesign -> [ExperimentalObject] -> ResearchHypothesis -> [(Treatment,Experi
```

1.1.4.6 generateAnalysisScript

```
generateAnalysisScript :: Experiment -> AnalysisScript
```

1.1.4.7 generateHypothesisTests

```
generateHypothesisTests :: [ExperimentalObject] -> ExperimentalDesign -> ResearchHypothesis
-> HypothesisTest
```

1.1.4.8 createAnalysisTest

```
createAnalysisTest :: ResearchHypothesis -> ExperimentalObject -> AnalysisTest
```

1.1.4.9 execute

```
execute :: ExecutionScript -> [IO Float]
```

1.1.4.10 executeApplication

```
executeApplication :: Application -> IO Float
```

1.1.4.11 createExecutionResult

```
createExecutionResult :: ((Treatment,ExperimentalObject, DependentVariable),IO Float)->
ExecutionResult
```

1.1.4.12 analyze

`analyze :: [ExecutionResult] -> AnalysisScript -> [HypothesisResult]`

The result of this function is of type `[HypothesisResult]` since a map function is applied to `[HypothesisTest]` contained in `AnalysisScript`, which means the function `analyzeHypothesis` is applied to each `[HypothesisTest]` with the argument of type `[ExecutionResult]`. The result of function `analyzeHypothesis` is of type `HypothesisResult`. So, the result of this map function is of type `[HypothesisResult]`.

1.1.4.13 analyzeHypothesis

`analyzeHypothesis :: [ExecutionResult] -> HypothesisTest -> HypothesisResult`

The return type of this function is `HypothesisResult` since it calls the constructor `HypothesisResult` passing as arguments a `String` obtained from the accessor function `hypName` from `HypothesisTest` and a `[TestResult]` obtained from applying a map function with the function `applyAnalysisFunction` to `[AnalysisTest]`. `applyAnalysisFunction` takes arguments of type `[ExecutionResult]` and `AnalysisTest` and returns `TestResult`. So, the result of applying this map function is of type `[TestResult]`, which is the expected type for the constructor `HypothesisResult`.

1.1.4.14 applyAnalysisFunction

`applyAnalysisFunction :: [ExecutionResult] -> AnalysisTest -> TestResult`

The return type of this function is `TestResult` since it calls the constructor `TestResult` passing as arguments a `String` obtained from the accessor function `argObjectName` from `Argument` wich is obtained from `AnalysisTest` and a `AnalysisResult` obtained from applying `analysisFunction` to two arguments of type `[ExecutionResult]` resulting of applying `filterResults` function to the argument of type `[ExecutionResult]`.

1.1.4.15 filterResults

`filterResults :: [ExecutionResult] -> Argument -> [ExecutionResult]`

This function applies a filter to `[ExecutionResult]`, so the return type is the same, `[ExecutionResult]`.

1.1.5 Definitions

Definition 1. Experiment semantics:

- $\forall e : E \cdot wf(e) \implies$

  $\llbracket e \rrbracket = analyze(zip(generateListOfExecutions(e),$

  $execute(compileExecutionScript(e))), generateAnalysisScript(e))$

Definition 2. Reproducibility: an experiment $e : E$ is reproducible if and only if $\llbracket e \rrbracket$ is a function.

Definition 3. Infrastructure semantics:

- $\forall es : ES \cdot wf(es) \implies \llbracket es \rrbracket = execute(es)$

Definition 4. Experiment specification well-formedness: An experiment specification is well-formed if all elements used in its hypotheses are defined in the experiment specification with unique names and each hypothesis compares distinct treatments.

- An experiment specification is well-formed if and only if all treatments and dependent variables referred in its hypotheses are specified, and each hypothesis compares distinct treatments. In addition, each hypothesis, treatment, object, and dependent variable is specified with a unique name.

- $\forall e : E \cdot wf(e) \iff (\forall h \in e.hypotheses \cdot$

  $h.dependentVariable \in e.dependentVariables \wedge h.treatment1 \in e.treatments \wedge$

  $h.treatment2 \in e.treatments \wedge$

  $h.treatment1 \neq h.treatment2) \wedge$

  $(\forall rh1, rh2 \in e.hypotheses \cdot rh1 \neq rh2 \implies$

  $rh1.name \neq rh2.name) \wedge$

  $(\forall tr1, tr2 \in e.treatments \cdot tr1 \neq tr2 \implies$

  $tr1.name \neq tr2.name) \wedge$

  $(\forall o1, o2 \in e.objects \cdot o1 \neq o2 \implies$

  $o1.name \neq o2.name) \wedge$

  $(\forall dv1, dv2 \in e.dependentVariables \cdot dv1 \neq dv2 \implies$

  $dv1.name \neq dv2.name)$

Definition 5. Execution script well-formedness: Every execution script is well-formed.

- $\forall es : ES \cdot wf(es) = true$

Definition 6. Analysis script well-formedness: An analysis script is well-formed if and only if each distinct hypothesis test refers to a distinct hypothesis; each analysis test compares distinct treatments but the same object and dependent variable; and, for each hypothesis, each analysis test is related to a distinct object.

- $\forall as : AS \cdot wf(as) \iff (\forall ht1, ht2 \in as.hypothesesTests \cdot$
  $ht1 \neq ht2 \implies ht1.hypothesisName \neq ht2.hypothesisName) \land$
  $(\forall ht \in as.hypothesesTests \cdot$
  $(\forall at \in ht \cdot$
  $at.argument1.dependentVariableName =$
  $at.argument2.dependentVariableName \land$
  $at.argument1.objectName = at.argument2.objectName \land$
  $at.argument1.treatmentName \neq at.argument2.treatmentName) \land$
  $(\forall at1, at2 \in ht \cdot at1 \neq at2 \implies$
  $at1.argument1.objectName \neq at2.argument1.objectName))$

1.1.6 Properties

1. Execution script compilation well-formedness: The result of compiling a well-formed experiment specification is a well-formed execution script.

   - $\forall e : E \cdot wf(e) \implies wf(compileExecutionScript(e))$

     Proof. According to Definition 5, every execution script of type `ES` is well formed, and `compileExecutionScript(e)` returns an execution script of type `ES`. Thus, if `e` is well-formed, `wf(compileExecutionScript(e))= true`.

     $\square$

2. Execution script compilation soundness: The infrastructure runs required commands to evaluate all the hypotheses according to the design of the experiment.

   - For each hypothesis of the experiment, all of its treatments are applied n times to each experimental object, according to the experimental design and using the corresponding instrumentation. The number of repetitions n is specified in the experimental design.
   - $\forall e : E \cdot wf(e) \implies \forall h \in e.hypotheses \cdot$
     $\forall (t, o) \in e.design.designFunction(\{t1, t2\}, e.objects) \cdot$
     $\exists_{=n} a \in compileExecutionScript(e).applications \mid$
     $a.instrument = h.dependentVariable.instrument \land$
     $a.command = t.command \land$
     $a.argument = o.argument$

     Proof. Execution script is created by `compileExecutionScript` function (Listing 3 lines 18-19) with an argument `e:E`. First, this function calls `generateListOfExecutions`.

Inside this function, `concatMap` is applied with the function `applyDesign` to `e.hypotheses` (Listing 3 line 26), which means the function `applyDesign` is applied to each hypothesis, and what follows holds for all hypotheses. Additional arguments to `applyDesign` are `e.design` and `e.objects`. For each hypothesis h, inside `applyDesign` function, `designFunction` is applied to (`[h.treatment1,h.treatment2],e.objects`) (Listing 3 line 30). The result is of type $\overline{(T,O)}$, where T and O are related by `designFunction`. To each result, `h.dependentVariable` is joined by a `map` function (Listing 3 line 29). The result is of type $\overline{(T,O,DV)}$, where T and O are related by the design function, and DV is related to the same hypothesis as T. To this list, the function `removeDuplicates` is applied, and then `replicate` (Listing 3 line 25) with argument `e.design.runs`. As a result, in the resulting list, each element of type (T,O,DV) is repeated `e.design.runs` times. Next, a `map` function is applied to this list using the function `compileApplication` (Listing 3 line 19). So, what follows holds for every $(t,o) \in e.design.designFunction(\{t1,t2\},e.objects)$. Inside `compileApplication` function, an Application $a : A$ is created (Listing 3 line 22). Given an element $(t : T, o : O, dv : DV)$, `a.instrument = dv.instrument`, `a.command=t.command`, and `a.argument=o.argument`. $\qquad\square$

3. Execution resource optimization: The infrastructure only runs commands required to evaluate the hypotheses according to the design of the experiment and nothing else.

- Each application executed by the infrasctructure applies a treatment to an object according to the design of the experiment. The treatment is related to one hypothesis specified in the experiment and the instrument used to measure the dependent variable is related to the same hypothesis. In addition, the experimental object is related to the treatment according to the experimental design.

- $\forall e : E \cdot wf(e) \implies \forall a \in compileExecutionScript(e).applications \cdot$
  $\exists h \in e.hypotheses, \{dv, t\} \subseteq h, o \in e.objects \mid$
  $a.instrument = dv.instrument \wedge$
  $a.command = t.command \wedge$
  $a.argument = o.argument \wedge$
  $(t, o) \in e.design.designFunction(\{h.treatment1, h2.treatment2\}, e.objects)$
  $\wedge\ (\forall \{dv', t'\} \subseteq h, o' \in e.objects \cdot$
  $a.instrument = dv'.instrument \wedge$
  $a.command = t'.command \wedge$

$$a.argument = o'.argument \wedge$$
$$(t', o') \in e.design.designFunction(\{h.treatment1, h.treatment2\}, e.objects)$$
$$\implies (dv, t, o) = (dv', t', o'))$$

Proof. As showed in the previous property, every Application $a : A$ is created by `compileApplication`. Given an input (`t:T,o:O,dv:DV`), `a.instrument = dv.instrument`, `a.command=t.command`, and `a.argument=o.argument`. `compileApplication` is called by a `map` function, so the number of applications generated is the same as the number of elements in $\overline{(T, O, DV)}$. Since the input list $\overline{(t : T, o : O, dv : DV)}$ is generated after applying `applyDesign` function to some hypothesis `h`, for each element in this list, `t` and `o` are related by `designFunction`, and `t` and `dv` are related to the same hypothesis `h`.

□

4. Analysis script generation well-formedness: The result of generating an analyis script from a well-formed experiment specification is a well-formed analysis script.

- $\forall e : E \cdot wf(e) \implies wf(generateAnalysisScript(e))$

Proof. An analysis script $as : AS$ is generated by `generateAnalysisScript` function (Listing 3 lines 44-45). `as.hypothesesTests` are generated by applying `map` with `generateHypothesisTests` to `e.hypotheses`. Each $ht : HT$ is generated from a distinct $h : H$, and $ht.hypothesisName$ is taken from $h.name$ (Listing 3 line 48). Since `e` is well-formed, each hypothesis has a distinct name, and, consequently, each `ht` has a distinct hypoyhesisName. `ht.analysisTests` are created by applying `map` with function `createAnalysisTest` to commonObjects (Listing 3 line 48), where commonObjects are the objects related by `designFunction` to both `h.treatment1` and `h.treatment2`. `createAnalysisTest` function (Listing 3 lines 52-55) retrieves the suitable function for that hypothesis and creates two arguments. Considering `h:H`, `o:O` the arguments of this function, `at.argument1.dependentVariableName = h.dependentVariable.name` and `at.argument2.dependentVariableName = h.dependentVariable.name`, so `at.argument1.dependentVariableName` = `at.argument2.dependentVariableName`.

`at.argument1.objectName = o.name` and `at.argument2.objectName = o.name`, so `at.argument1.objectName` = `at.argument2.objectName`.

`at.argument1.treatmentName = h.treatment1.name` and `at.argument2.treatmentName = h.treatment2.name`. e is well-formed, so `at.argument1.treatmentName` $\neq$`at.argument2.treatmentName`. ■

Each $at : AT$ is created from a distinct object. Since e is well-formed, every object has a distinct name. So, every at has a distinct argument1.objectName.

$\square$

5. Experiment soundness: Analysis is performed by using a suitable analysis function for each hypothesis and using correct arguments in the correct order. In addition, execution data are produced by executing a sound execution script compiled from the experiment specification.

- For each hypothesis, the analysis function is suitable to analyze it and each argument of the analysis function corresponds to a set of data resulting of applying each treatment to an object, according to the experimental design, and measured by the corresponding instrument. In addition, the arguments are provided to the analysis function in the correct order. Moreover, execution data are produced by executing a sound execution script compiled from a well-formed experiment specification.

- $\forall e : E \cdot wf(e) \implies \forall hr \in \llbracket e \rrbracket \cdot$
  $\forall tr \in hr \cdot tr = suitableFunction(h)(argument1, argument2)$
  where
  $argument1 = execute($
  $compileApplication(h.dependentVariable, h.treatment1, o)$
  $)$
  $argument2 = execute($
  $compileApplication(h.dependentVariable, h.treatment2, o)$
  $)$
  $h = b_{HR \leftrightarrow H} hr$
  $o = b_{TR \leftrightarrow hObjects} tr$
  $hObjects = e.design.designFunction(\{h.treatment1, h.treatment2\},$
  $e.objects).objects$

  $b_{HR \leftrightarrow H}$ is a bijection between hypotheses results HR and hypotheses H. With $b_{HR \leftrightarrow H} hr$, we get the $h : H$ corresponding to a $hr : HR$.

  Likewise, $b_{TR \leftrightarrow hObjects}$ is bijection between test results TR and the objects resulting of applying the design function to the treatments of a given hypothesis and the objects. We also use a helper function $objects : \overline{(T, O)} \rightarrow \overline{O}$.

  Proof. An analysis script $as : AS$ is generated from an experiment specification $e : E$ by `generateAnalysisScript` function (Listing 3 lines 44-45). `as.hypothesesTests` are generated by applying `map` with `generateHypothesisTests` to `e.hypotheses`. This creates a bijection between `e.hypotheses` and `as.analysisTests`: ▮
  $H \leftrightarrow HT$ `generateHypothesisTests(H,...):HT`.

11

Inside `generateHypothesisTests` function, for a given hypothesis `h`, `designFunction` ▬ is first applied to `[h.treatment1,h.treatment2]` and `e.objects` (Listing 3 line 49). The result is then filtered to extract objects that are related to both `h.treatment1` and `h.treatment2`. In what follows, we will refer to this list as `hObjects`. Next, `map` is applied with `createAnalysisTest` function to `hObjects`. This ceates a bijection between `hObjects` and AT:

$hObjects \leftrightarrow AT$ `createAnalysisTest(O,...):AT`.

Each $at : AT$ contains an `analysisFunction`, which is retrieved by calling `suitableFunction` using `h` as argument (Listing 3 line 53). We assume the result of this function is always a suitable function to analyze that hypothesis. `at.argument1` and `at.argument2` are used to filter execution results before applying `analysisFunction`. Considering the argument `o:O`, `at.argument1.dependentVariableName` is taken from `h.dependentVariable.name`, `at.argument1.treatmentName` is taken from `h.treatment1.name`, and `at.argument1.objectName` ▬ is taken from `o.name`. Likewise, `at.argument2.dependentVariableName` is taken from `h.dependentVariable.name`, `at.argument2.treatmentName` is taken from `h.treatment2.name`, ▬ and `at.argument2.objectName` is taken from `o.name`.

Considering an analysis script `as:AS`, in `analyze` function, hypotheses results $\overline{HR}$ are created by `map` with function `analyzeHypothesis` applied to `as.hypothesesTests` (Listing 6 line 12). Each $hr : HR$ is created by `analyzeHypothesis` ▬ from a $ht : HT$. This creates a bijection between HT and HR:

$HT \leftrightarrow HR$ `analyzeHypothesis(HT,...):HR`.

By transitivity, or composition of functions, there is also a bijection between H and HR:

$H \leftrightarrow HR$ `analyzeHypothesis(generateHypothesisTests(H,...),...):HR`

In `analyzeHypothesis` function, hr.testResults are created by `map` with `applyAnalysisFunction` ▬ applied to `ht.analysisTests` (Listing 6 line 15). This creates a bijection between AT and TR:

$AT \leftrightarrow TR$ `applyAnalysisFunction(AT,...):TR`.

By transitivity, or composition of functions, there is also a bijection between $hObjects$ and TR:

$hObjects \leftrightarrow TR$ `applyAnalysisFunction(createAnalysisTest(O,...),...):TR`

In `applyAnalysisFunction`, `at.analysisFunction` is applied to two arguments (Listing 6 line 18). Each argument is a subset of execution results corresponding to a dependent variable measured when applying each treatment to

an object. Execution results are filtered by `filterResults` function using the arguments `at.argument1` and `at.argument2`.

$\square$

Appendices

A   Haskell Model

```haskell
1  module Experiment where
2  import Infrastructure
3  import Analysis
4  import ExecutionResult
5
6  data Experiment = Experiment {researchHypotheses ::
       [ResearchHypothesis], design :: ExperimentalDesign, treatments::
       [Treatment], objects:: [ExperimentalObject], dependentVariables ::
       [DependentVariable]}
7  data ResearchHypothesis = ResearchHypothesis {hypothesisName :: String,
       dependentVariable :: DependentVariable, treatment1 :: Treatment,
8  operator::String, treatment2 :: Treatment} deriving (Show, Eq, Ord)
9  data DependentVariable = DependentVariable {dvName :: String,
       instrumentCommand :: String} deriving (Show, Eq, Ord)
10 data ExperimentalDesign =  ExperimentalDesign {runs :: Int,
       designFunction :: [Treatment]->[ExperimentalObject]
       ->[(Treatment,ExperimentalObject)] }
11 data Treatment = Treatment {treatmentName :: String, treatmentCommand
       :: String} deriving (Show, Eq, Ord)
12 data ExperimentalObject = ExperimentalObject {objectName :: String,
       argument:: String} deriving (Show, Eq, Ord)
13
14 experiment :: Experiment -> [HypothesisResult]
15 experiment exp= analyze  executionResults (generateAnalysisScript exp)
16   where executionResults = map createExecutionResult (zip
       (generateListOfExecutions exp) (execute (compileExecutionScript
       exp)))
17
18 compileExecutionScript :: Experiment -> ExecutionScript
19 compileExecutionScript experiment = ExecutionScript (map
       compileApplication (generateListOfExecutions experiment))
20
21 compileApplication ::(Treatment,ExperimentalObject, DependentVariable)
       -> Application
22 compileApplication (treatment,object, depVariable) = Application
       (instrumentCommand depVariable) (treatmentCommand treatment)
       (argument object)
23
```

```haskell
24  generateListOfExecutions :: Experiment ->
        [(Treatment,ExperimentalObject,DependentVariable)]
25  generateListOfExecutions experiment = concatMap (replicate (runs
        (design experiment))) (removeDuplicates
        treatmentApplicationDependentVariable)
26    where treatmentApplicationDependentVariable = (concatMap (applyDesign
          (design experiment) (objects experiment)) (researchHypotheses
          experiment))
27
28  applyDesign :: ExperimentalDesign -> [ExperimentalObject]->
        ResearchHypothesis ->
        [(Treatment,ExperimentalObject,DependentVariable)]
29  applyDesign design objects hypothesis = map ((\a (b,c) -> (b,c,a))
        (dependentVariable hypothesis)) treatmentApplication
30    where treatmentApplication = (designFunction design) [treatment1
          hypothesis, treatment2 hypothesis] objects
31
32  removeDuplicates :: (Eq a) => [a] -> [a]
33  removeDuplicates [] = []
34  removeDuplicates (x:xs) | x `elem` xs   = removeDuplicates xs
35                          | otherwise     = x : removeDuplicates xs
36
37
38  createExecutionResult :: ((Treatment,ExperimentalObject,
        DependentVariable),IO Float)->ExecutionResult
39  createExecutionResult ((treat, obj, depVariable), value) =
        ExecutionResult (dvName depVariable) (treatmentName treat)
        (objectName obj) value
40
41  cartesianProductDesign :: [Treatment] -> [ExperimentalObject] ->
        [(Treatment,ExperimentalObject)]
42  cartesianProductDesign treatments objects =[(treatment,object) |
        treatment <- treatments, object<-objects]
43
44  generateAnalysisScript :: Experiment -> AnalysisScript
45  generateAnalysisScript experiment = AnalysisScript (map
        (generateHypothesisTests (objects experiment) (design experiment))
        (researchHypotheses experiment))
46
47  generateHypothesisTests :: [ExperimentalObject] -> ExperimentalDesign
        -> ResearchHypothesis -> HypothesisTest
48  generateHypothesisTests objects design hypothesis = HypothesisTest
        (hypothesisName hypothesis) (map (createAnalysisTest hypothesis)
        commonObjects)
```

```
49      where treatmentApplication = (designFunction design) [treatment1
            hypothesis, treatment2 hypothesis] objects
50          commonObjects =[object | object<-objects, elem ((treatment1
                hypothesis),object) treatmentApplication && elem
                ((treatment2 hypothesis),object) treatmentApplication]
51
52  createAnalysisTest :: ResearchHypothesis -> ExperimentalObject ->
        AnalysisTest
53  createAnalysisTest  rh object = AnalysisTest (suitableFunction rh)
        argument1 argument2
54    where argument1 = Argument (dvName (dependentVariable rh))
          (treatmentName (treatment1 rh)) (objectName object)
55          argument2 = Argument (dvName (dependentVariable rh))
              (treatmentName (treatment2 rh)) (objectName object)
56
57  suitableFunction :: ResearchHypothesis ->
        ([ExecutionResult]->[ExecutionResult]->AnalysisResult)
58  suitableFunction rh = wilcoxTest
59
60  wilcoxTest :: [ExecutionResult]->[ExecutionResult]->AnalysisResult
61  wilcoxTest sample1 sample2 = AnalysisResult "some result"
```

Listing 3: Experiment.hs

```
1  module ExecutionResult where
2  data ExecutionResult = ExecutionResult {resDependentVariableName ::
       String, resTreatmentName :: String, resObjectName :: String, value
       :: IO Float}
```

Listing 4: ExecutionResult.hs

```
1  module Infrastructure where
2  import System.Random
3  data ExecutionScript = ExecutionScript {applications :: [Application]}
       deriving (Show, Eq, Ord)
4  data Application = Application {appInstrument :: String, appCommand ::
       String, appArgument:: String} deriving (Show, Eq, Ord)
5
6  execute :: ExecutionScript -> [IO Float]
7  execute applicationDescriptor =map executeApplication (applications
       applicationDescriptor)
8  executeApplication :: Application -> IO Float
9  executeApplication application = getStdRandom (randomR (1,1000))
```

```haskell
1  module Analysis where
2  import ExecutionResult
3  data AnalysisScript = AnalysisScript {hypothesesTests ::
       [HypothesisTest]}
4  data HypothesisTest = HypothesisTest {hypName::String, analysisTests ::
       [AnalysisTest]}
5  data AnalysisTest = AnalysisTest {analysisFunction ::
       [ExecutionResult]->[ExecutionResult]->AnalysisResult, argument1 ::
       Argument, argument2::Argument}
6  data Argument = Argument {argDvName :: String, argTreatmentName ::
       String, argObjectName ::String}
7  data HypothesisResult = HypothesisResult {hrHypothesisName::String,
       testResults::[TestResult]} deriving (Show, Eq, Ord)
8  data TestResult = TestResult {trObjectName::String, analysisResult ::
       AnalysisResult} deriving (Show, Eq, Ord)
9  data AnalysisResult = AnalysisResult {result :: String} deriving
       (Show, Eq, Ord)
10
11 analyze :: [ExecutionResult] -> AnalysisScript -> [HypothesisResult]
12 analyze results (AnalysisScript hypothesesTests) = map
       (analyzeHypothesis results) hypothesesTests
13
14 analyzeHypothesis :: [ExecutionResult] -> HypothesisTest ->
       HypothesisResult
15 analyzeHypothesis executionResults ht = HypothesisResult (hypName ht)
       (map (applyAnalysisFunction executionResults) (analysisTests ht))
16
17 applyAnalysisFunction :: [ExecutionResult] -> AnalysisTest -> TestResult
18 applyAnalysisFunction results test = TestResult (argObjectName
       (argument1 test)) ((analysisFunction test) (filterResults results
       (argument1 test)) (filterResults results (argument2 test) ))
19
20 filterResults :: [ExecutionResult] -> Argument -> [ExecutionResult]
21 filterResults results (Argument dependentVariableName treatmentName
       objectName) = ([ result | result <- results,
       (resDependentVariableName result) == dependentVariableName &&
       (resTreatmentName result)==treatmentName && (resObjectName
       result)==objectName])
```

Listing 6: Analysis.hs