User Documentation for StatWhy v.1.2.0

Yusuke Kawamoto[1], Kentaro Kobayashi[1,2], and Kohei Suenaga[3] *

[1] AIST, Tokyo, Japan
[2] University of Tsukuba, Ibaraki, Japan
[3] Kyoto University, Kyoto, Japan

Abstract. StatWhy is a software tool for automatically verifying the correctness of statistical hypothesis testing programs. Specifically, programmers are required to annotate the source code of the statistical programs with the requirements for the statistical analyses. Then our StatWhy tool automatically checks whether the programmers have properly specified the requirements for the statistical methods, thereby identifying any missing or incorrect requirements that need to be corrected. In this documentation, we first present how to install and use the StatWhy tool. We then demonstrate how the tool can avoid common errors in the application of a variety of hypothesis testing methods.

1 Foreword

Statistical methods have been widely misused and misinterpreted in various scientific fields, raising significant concerns about the integrity of scientific research. To mitigate this problem, we propose a new method for formally specifying and automatically verifying the correctness of statistical programs in our paper [3].

StatWhy is a software tool that implements our proposed method for automatically checking whether a programmer has properly specified the requirements for the statistical methods.

In Section 2, we first present how to install the StatWhy tool. In Section 3, we present how to execute StatWhy. In Section 4, we show an illustrating example to see how a tool user can verify a statistical program. In Section 5, we briefly explain notations used in StatWhy specifications. In Section 6, we present examples to demonstrate how we can prevent common errors in applying a variety of hypothesis testing programs.

1.1 Availability

The source code of the StatWhy tool is publicly available at https://github.com/fm4stats/statwhy including a range of example programs.

1.2 Contact

Report any bugs and requests to https://github.com/fm4stats/statwhy/issues.

* The authors are listed in alphabetical order.

1.3 Acknowledgments.

The authors are supported by JSPS KAKENHI Grant Number JP24K02924, Japan. Yusuke Kawamoto is supported by JST, PRESTO Grant Number JPMJPR2022, Japan. Kohei Suenaga is supported by JST CREST Grant Number JPMJCR2012, Japan.

2 Installation

In this section, we explain how to install StatWhy and our extension of Cameleer [6].

2.1 Installing StatWhy

Installing OCaml First, we need to install OCaml via the official package manager opam.

1. Install opam
   On Ubuntu:

```
$ sudo apt-get update
$ sudo apt-get install opam
```

   On macOS, install opam with Homebrew:

```
$ brew install opam
```

   Alternatively, if you use MacPorts:

```
$ port install opam
```

   After the installation of spam, you need to initialize it:

```
$ opam init -y
$ eval $(opam env)
```

2. Install OCaml 5.0
   To install OCaml 5.0, execute the following command[4]:

```
$ opam switch create 5.0.0
$ eval $(opam env)
```

Installing StatWhy and Cameleer Download the source code of StatWhy, including an extension of Cameleer. Install StatWhy and Cameleer by running:

```
$ unzip statwhy-1.2.0.zip
$ cd cameleer
$ opam pin add .
```

This will install Cameleer, StatWhy (included in "cameleer/src/statwhy"), and their dependencies. In the installation of Cameleer, the Why3 platform [2] is automatically installed [5].

[4] Using the versions later than OCaml 5.0, we encountered a dynamic link issue with Cameleer on macOS.

[5] We have tested StatWhy using the Why3 platform 1.7.1 and Cameleer 0.1.

Installing CVC5 On Ubuntu 24.04 LTS, you can install CVC5 by:

```
$ sudo apt install cvc5
```

Alternatively, you can directly download a binary from the GitHub repository:

```
$ wget https://github.com/cvc5/cvc5/releases/download/cvc5-1.2.0/cvc5-Linux-
    x86_64-static.zip
$ unzip cvc5-Linux-x86_64-static.zip
$ sudo cp ./cvc5-Linux-x86_64-static/bin/cvc5 /usr/local/bin
```

On macOS, a Homebrew Tap for CVC5 is also available:

```
$ brew tap cvc5/homebrew-cvc5
$ brew install cvc5
```

After installing the solver, run the following command to let Why3 detect it:

```
$ why3 config detect
```

3 Usage

You can verify an OCaml code via Cameleer by running the following:

```
$ statwhy <file-to-be-verified>.ml
```

If you want to verify a WhyML code:

```
$ statwhy <file-to-be-verified>.mlw
```

Note that in both cases, you need our extension of Cameleer to load StatWhy.

4 Getting Started

We show an example of an OCaml program annotated with preconditions and postconditions that we want to verify using StatWhy. In the code below, the function `example1` conducts a *two-sided t-test for a mean of a population*, given a dataset `d` as input.

doc/doc_examples/example1.ml

```
open CameleerBHL

module Example1 = struct
  open Ttest

  (* Declarations of a distribution and formulas *)
  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = Atom (Pred ("<", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))
  let fmlA_u = Atom (Pred (">", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))
  let fmlA = Atom (Pred ("=!", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))
```

```
  (* executes the t-test for a population mean *)
  let example1 (d : float list dataset) : float = exec_ttest_1samp t_n 1.0 d Two
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB (Eq p) fmlA
  *)

  ...
end
```

The specification of `example1` is written in the Gospel specification language [1]. The comment section starting with `(*@` denotes the specification of `example1` [6]. More information on the syntax of Gospel can be found on the following webpage: https://ocaml-gospel.github.io/gospel/language/syntax.

Now we explain the details of the function `example1` as follows.

```
  let example1 (d : float list dataset) : float = exec_ttest_1samp t_n 1.0 d Two
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB (Eq p) fmlA
  *)
```

In this program, given a dataset `d` as input, the command `exec_ttest_1samp t_n 1.0 d Two` computes the p-value of the one-sample t-test with the alternative hypothesis `fmlA` that the mean of the population distribution `(ppl d)` is not 1.0.

At the beginning of the specification, `p = example1 d` assigns the result of `example1 d` to the name `p`, which can be used throughout the specification.

The precondition in the `requires` clause expresses the requirement for correctly applying the t-test.

- `is_empty (!st)` specifies that the history `st` is empty. This requirement reminds the programmer to check that no hypothesis test has been conducted before this test.
- `sampled d t_n` means that the dataset `d` has been sampled from a population with a normal distribution `t_n`. This condition prevents the pro-

[6] Programmers are required to use the WhyML language to describe specifications.

grammers from forgetting to check whether the population follows a normal distribution.
- `(World (!st) interp) |= Possible fmlA_l` and `(World (!st) interp) |= Possible fmlA_u` represent that the analyst has a prior belief that the *lower-tail* hypothesis `fmlA_l` $\stackrel{\text{def}}{=}$ $(\mathsf{mean}(x) < 1.0)$ may be true, and that the *upper-tail* hypothesis `fmlA_u` $\stackrel{\text{def}}{=}$ $(\mathsf{mean}(x) > 1.0)$ may be true[7]. Thanks to this annotation, programmers are reminded to check whether the alternative hypothesis should be two-tailed or one-tailed, and thus whether the t-test command should be two-tailed or one-tailed.

The postcondition in the `ensures` clause expresses what the analyst wants to learn from the t-test in the world where the test has been performed.

- `(Eq p) = compose_pvs fmlA !st` represents that `p` is equal to the p-value obtained by the hypothesis test with the alternative hypothesis `fmlA`.
- `(World !st interp) |= StatB p fmlA` represents that the analyst obtains a statistical belief on the alternative hypothesis `fmlA` with the p-value `p`, in the world equipped with the record `st` of all hypothesis tests executed so far. This logical formula (`StatB p fmlA`) employs a statistical belief modality `StatB` introduced in belief Hoare logic (BHL) [4,5].
- The logical connective `&&` represents an *asymmetric conjunction* to control the goal-splitting transformation; the proof task for A `&&` B is split into those for A and $A \to B$.

For the instructions on verifying this code, see Section 6.1.

Remarks on Modules and Expressions

We present remarks on OCaml programs to be verified using StatWhy.

- We need to open `CameleerBHL` and `Ttest` (or any hypothesis testing modules) explicitly.
- We add an attribute `[@run]` to ignore `let` expressions that are used exclusively for the execution and should not be verified by StatWhy. This attribute is useful when we want to avoid verifying a particular function.
- We cannot use the "and" pattern `_` and the "unit" pattern `()` simultaneously in top-level definitions. The current version of Cameleer does not support these patterns in top-level definitions. For instance, we cannot use the following expression "`let[@run] _ = ...`."

5 Notations in StatWhy Specifications

In this section, we briefly describe notations used in StatWhy specifications. As shown in the previous section, we use formulas, such as `(World (!st) interp)`

[7] We remark that the disjunction `fmlA_l` $\vee$ `fmlA_u` is logically equivalent to `fmlA`.

`|= Possible fmlA_l`, to specify the requirements of a hypothesis testing program in the framework of Belief Hoare logic (BHL) [4,5]. These formulas represent the properties of populations or datasets, and each hypothesis test requires several conditions expressed by such formulas.

5.1 Overview of belief Hoare Logic (BHL)

Belief Hoare logic (*BHL*) [4,5] is a program logic equipped with epistemic modal operators for the statistical beliefs acquired via hypothesis testing. We briefly explain this logic using a simple example described in our paper [3] as follows.

In the framework of BHL, we express a procedure for statistical hypothesis testing as a program C using a programming language. Then, we use modal logic to describe the requirements for the tests as a *precondition* formula, e.g.,

$$\psi_{\mathsf{pre}} \stackrel{\text{def}}{=} y \rightsquigarrow N(\mu, \sigma^2) \wedge \mathbf{P}\varphi \wedge \kappa_{\emptyset},$$

where $y \rightsquigarrow N(\mu, \sigma^2)$ represents that a dataset y is sampled from the population that follows a normal distribution $N(\mu, \sigma^2)$ with an unknown mean μ and an unknown variance σ^2. The modal formula $\mathbf{P}\varphi$ represents that, before conducting the hypothesis test, we have the *prior belief* that the alternative hypothesis φ *may be* true. The formula $\kappa_{\emptyset}$ represents that no statistical hypothesis testing has been conducted previously.

The statistical belief we acquire from the hypothesis test is specified as a *postcondition* formula, e.g.,

$$\psi^{\mathsf{post}} \stackrel{\text{def}}{=} \mathbf{K}_{y,A}^{\leq 0.05}\varphi. \tag{1}$$

Intuitively, by a hypothesis test A on the dataset y, we believe φ with a p-value $\alpha \leq 0.05$. Since the result of the hypothesis test may be wrong, we use the belief modality $\mathbf{K}_{y,A}^{\leq 0.05}$ instead of the S5 knowledge modality $\mathbf{K}$. Using this logic, the interpretation of the result of statistical methods is regarded to be epistemic.

Finally, we combine all the above and describe the whole statistical inference as a *judgment*:

$$\Gamma \vdash \{\psi_{\mathsf{pre}}\}\ C\ \{\psi^{\mathsf{post}}\}, \tag{2}$$

representing that whenever the precondition ψ_{pre} is satisfied, the execution of the program C results in the satisfaction of the postcondition ψ^{post}. By deriving this judgment using derivation rules in BHL, we conclude that the procedure for the statistical inference is correct whenever the precondition is satisfied.

5.2 BHL Formulas

To describe the requirements and interpretations of statistical analyses in Gospel, we introduce types for *terms*, *atomic formulas*, and *logical formulas* of an extension of belief Hoare logic (BHL) as follows [8].

[8] The actual definition can be found in `logicalFormula.mlw`

```
type term = RealT real_term | PopulationT population | ...
type atomic_formula = Pred psymb (list term)
type formula = Atom atomic_formula | Not formula
             | Conj formula formula | Disj formula formula
             | Possible formula | Know formula | StatB pvalue formula
             | ...
```

where a term can express a real number and a population; an atomic formula consists of a predicate symbol and a list of terms; a BHL formula is built using modal epistemic operators `Possible`, `Know`, and `StatB`. Then, we introduce predicate symbols (e.g., `eq_variance` and `check_variance`), functions symbols (e.g., `mean` and `ppl`), and a Kripke semantics where BHL formulas are interpreted under (i) the record `st` of all hypothesis tests executed so far [4,5] and (ii) the interpretation `interp` of private variables.

The interpretation of a BHL formula, `fml`, in a possible world, `World !st interp`, is written as `(World !st interp) |= fml`. The two symbols `st` and `interp` are reserved words defined in `cameleerBHL.mlw`.

Predicate and function symbols are defined in `logicalFormula.mlw` and `atomicFormulas.mlw`. BHL and the Kripke semantics are implemented in `statBHL.mlw` and `statELHT.mlw`.

5.3 Useful Operators to Describe Hypothesis Testing Programs

We introduce useful operators to facilitate the specification of StatWhy programs. Since hypothesis testing programs often involve comparisons among multiple groups of data, the preconditions and postconditions can become lengthy by repeating similar conditions. To simplify such redundant specifications for clarity and efficiency, we have provided a set of folding operations, allowing programmers to abstract away repetitive parts.[9]

For example, to simplify the conditions for comparing each pair of groups, we can use the operator `for_all`. This higher-order function checks whether a given predicate holds for all elements in a list. In StatWhy programs, this syntax sugar is often used to describe assertions that must hold for combinations of populations or terms. In the following example, the formula asserts that all the populations have the same variance:

```
for_all
(fun t -> let (x, y) = t in
  (World !st interp) |= eq_variance x y)
(combinations_poly ppls)
```

In the above formula, `for_all` is used to describe the iteration over all possible pairs `(x, y)` of populations in a set `ppls`. Using the predicate `eq_variance`, this formula states that for each pair `(x, y)` of the populations, `x` and `y` have the same variance.

[9] These folding operations are defined in `hof.mlw` and `utility.mlw`.

To improve the performance of StatWhy, we implemented a custom proof strategy—a combination of proof tactics and transformations—to accelerate the proof search in the presence of folding operations. Specifically, by applying a transformation called `compute_in_goal`, StatWhy automatically *unfolds* assertions written with syntax sugar such as `for_all` before discharging a verification condition. For instance, the above example formula is first transformed into:

```
((World !st interp) |= eq_variance p_1 p_2)
/\ ((World !st interp) |= eq_variance p_1 p_3)
/\ ... /\ ((World !st interp) |= eq_variance p_(n-1) p_n)
```

Then, thanks to the removal of the folding operation, StatWhy can efficiently discharge the verification conditions even when the set `ppls` of populations is large.[10]

Programmers can also choose to use other predefined abbreviations available in `hof.mlw` and `utility.mlw`.

- `hof.mlw` provides typical higher-order operations on polymorphic lists, such as fold and map.
- `utility.mlw` file defines operations that are useful to handle combinations of terms and populations, which are frequently used in multiple comparison programs.

For example, the `combinations_poly` function in `utility.mlw` enables users to compute all possible pairings of elements in a polymorphic list. This function is particularly useful in StatWhy when multiple populations or terms need to be combined to describe specifications. A concrete example of using `combinations_poly` appeared in the previous example, where `(combinations_poly ppls)` represents the set of all possible pairs of the populations contained in `ppls`.

6 Examples of Analyses Using StatWhy

We show a variety of examples of how StatWhy can be used to avoid common errors in hypothesis testing.

- Section 6.1 demonstrates how to use StatWhy in an example of a one-sample t-test.
- Section 6.2 shows how StatWhy checks whether appropriate kinds of t-test are applied to different situations.
- Section 6.3 explains how StatWhy verifies a program for the Bonferroni correction in multiple comparisons and addresses p-value hacking problems.
- Section 6.4 shows how StatWhy checks the programs with ANOVA and other hypothesis tests under different requirements.

[10] Conventionally, the number of groups compared in a hypothesis test is usually less than 8. However, we have found that SMT solvers often get stuck when trying to discharge the verification conditions that involve such folding operations.

– Section 6.5 demonstrates how StatWhy verifies Tukey's HSD test—a method for multiple comparison.
– Section 6.6 presents an experimental evaluation of the execution times of StatWhy using a variety of hypothesis testing programs.

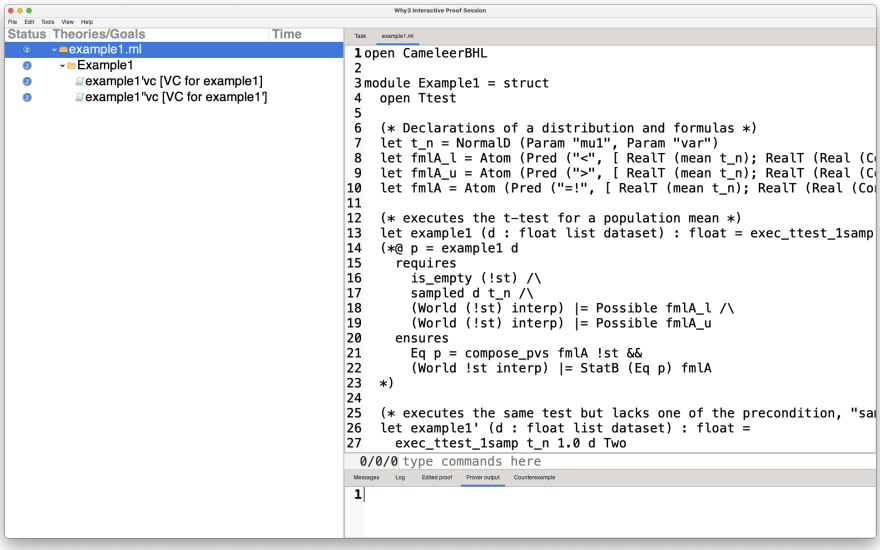
6.1 Simple *t*-test (One-Sample *t*-test)

We demonstrate how to use StatWhy through an example of verifying a program that performs the *t-test for a mean of a population*.

To verify the OCaml program `doc/doc_examples/example1.ml`, execute the following command:

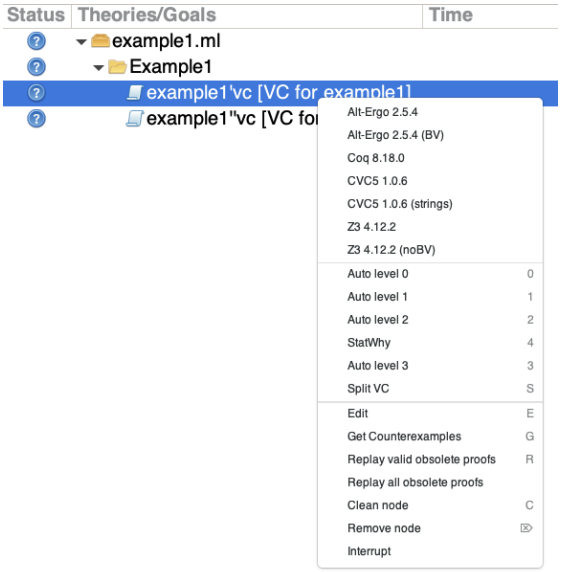
```
$ statwhy doc/doc_examples/example1.ml
```

This command transforms the OCaml code into WhyML code, generates the verification conditions (VC), and launches Why3 IDE as follows.



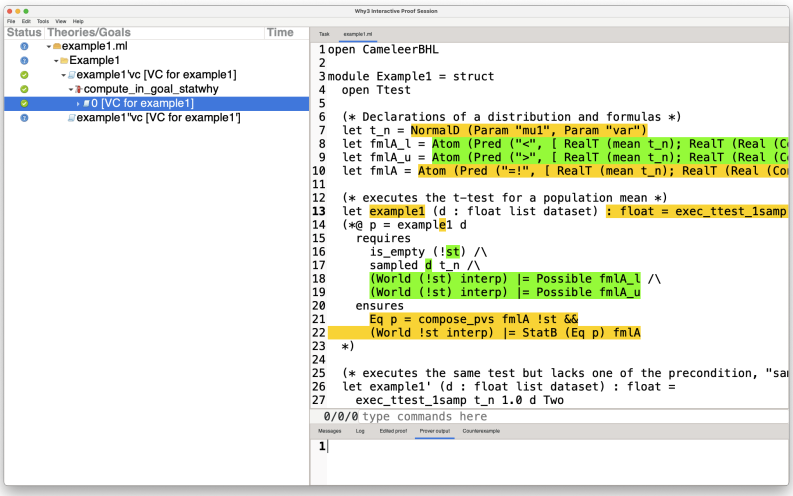
The Why3 IDE screen displays the code to be verified in the right panel and shows the VCs generated by the code in the left panel. Why3 generates two VCs from the source code file `example1.ml` in this case, `example1'vc` and `example1''vc`. The former is a VC for `example1`, which was explained in Section 4. The latter VC is similar to the former, but lacks one of the preconditions: `sampled d t_n`.

To discharge each VC, right-click on the VC, then click "StatWhy" or press '4' after selecting the VC.

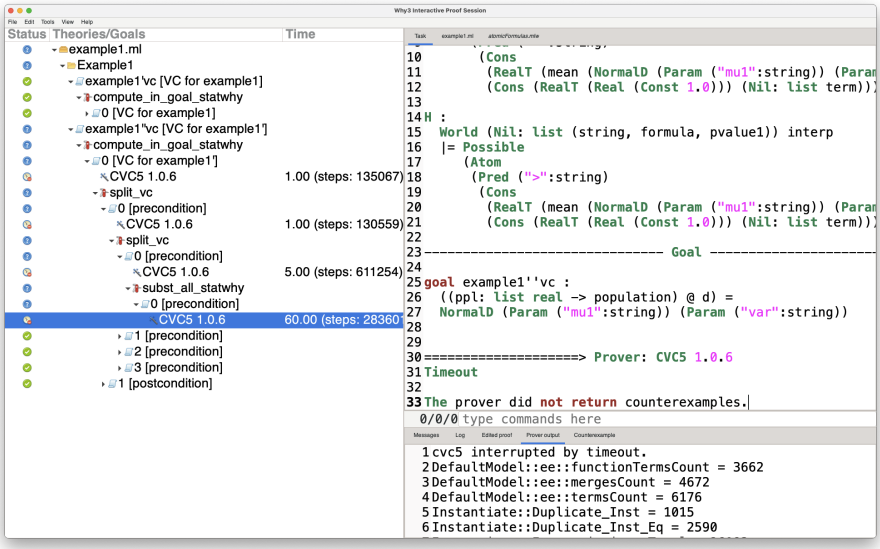


"StatWhy" is our custom proof strategy; it first applies Why3's default proof strategies (e.g., `split_vc` for splitting conjunctive verification conditions into smaller ones and `compute_in_goal` for applying computations and simplifications to proof goals). These invocations of the proof strategies are interleaved with calls to SMT solvers, whose timeouts are set to small values. If these applications of the default proof strategies fail to discharge the VCs, then we apply aggressive transformation strategies that unfold the definitions of the functions and predicates defined in StatWhy.

If one prover succeeds in verifying a VC, a check-mark (✓) will appear to the left of the VC, indicating that the VC is correct:



If Why3 fails to discharge the goal, by clicking a failed VC, the analyst finds the judgment that cannot be discharged on the right panel. For example, according to the goal on the right panel of the following figure, StatWhy has failed to prove that the population `(ppl d)` has the normal distribution type `NormalD`.



6.2 Several Variants of *t*-tests

StatWhy can distinguish between different preconditions for each hypothesis test command and remind users to make such conditions explicit. In this example, we consider several variants of t-tests, such as paired/non-paired t-tests and t-tests in the presence of populations with equal/unequal variance.

Paired *t*-test vs. Non-Paired *t*-test We use the *paired t-test* when there is a pairing or matching between the two samples. On the other hand, the *non-paired t-test* is applied otherwise, e.g., when two datasets are sampled from the population independently. StatWhy can check which of the tests should be applied to a situation that satisfies the precondition.

Paired *t*-test The specification of the paired t-test command `exec_ttest_paired` in StatWhy is as follows:

```
val exec_ttest_paired (p1 : population) (p2 : population) (y : dataset (list
    real, list real)) (alt : alternative) : real
writes { st }
requires {
  match (p1, p2) with
  | ((NormalD _ _), (NormalD _ _)) ->
```

```
  match y with
  | (y1, y2) ->
      paired y1 y2 /\
      sampled y1 p1 /\ sampled y2 p2 /\
      let r1 = mean p1 in
      let r2 = mean p2 in
      match alt with
      | Two ->
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Up ->
        (World !st interp) |= Not (Possible (r1 <' r2)) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Low ->
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Not (Possible (r1 >' r2))
      end
  end
  | _ -> false
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean p1 in
  let r2 = mean p2 in
  let h = match alt with
    | Two -> r1 =!' r2
    | Up -> r1 >' r2
    | Low -> r1 <' r2
  end in !st = Cons ("ttest_paired", h, Eq pv) !(old st)
}
```

`exec_ttest_paired` takes four arguments. `p1` and `p2` are the distributions to be tested, `y` is a pair of datasets, and `alt` denotes what type of the alternative hypothesis is; `Two` is for two-tailed tests, `Up` is for upper-tailed tests, and `Low` is for lower-tailed tests.

The precondition for the test is in the `requires` clause. `paired y1 y2` means that two samples `y1` and `y2` are obtained in pairs. The `ensures` clause specifies the postcondition of the test. The expression `!st = !(old st) ++ Cons ("ttest_ind", h, Eq pv) Nil` stores the resulting p-value `Eq pv` and the alternative hypothesis (`h`) in the store `st`.

Non-Paired *t*-test The code below shows the specification of the non-paired t-test (Student's t-test):

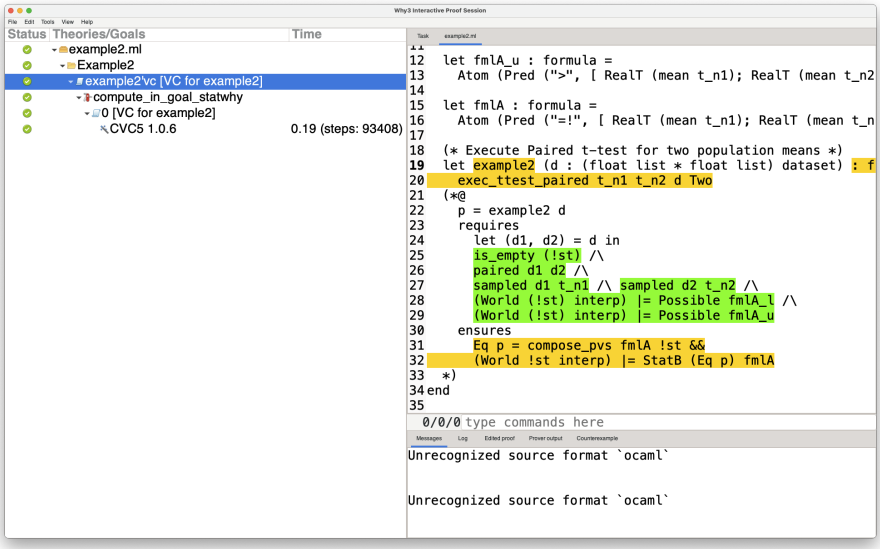
```
val exec_ttest_ind_eq (p1 : population) (p2 : population) (y : dataset (list
    real, list real)) (alt : alternative) : real
writes { st }
requires {
  match (p1, p2) with
  | ((NormalD _ _), (NormalD _ _)) ->
```

```
  match y with
  | (y1, y2) ->
      non_paired y1 y2 /\
      (World !st interp) |= eq_variance p1 p2 /\
      (World !st interp) |= Not (check_variance p1) /\
      (World !st interp) |= Not (check_variance p2) /\
      sampled y1 p1 /\ sampled y2 p2 /\
      let r1 = mean p1 in
      let r2 = mean p2 in
      match alt with
      | Two ->
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Up ->
        (World !st interp) |= Not (Possible (r1 <' r2)) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Low ->
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Not (Possible (r1 >' r2))
      end
  end
  | _ -> false
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean p1 in
  let r2 = mean p2 in
  let h = match alt with
    | Two -> r1 =!' r2
    | Up -> r1 >' r2
    | Low -> r1 <' r2
  end in !st = Cons ("ttest_ind_eq", h, Eq pv) !(old st)
}
```

The precondition of `exec_ttest_ind_eq` is as follows:

- `non_paired y1 y2` specifies that `y1` and `y2` must be sampled separately (not in pair).
- `(World !st interp) |= eq_variance p1 p2` specifies that `p1` and `p2` have the same variance.
- `(World !st interp) |= Not (check_variance p1)` and `(World !st interp) |= Not (check_variance p2)` specify that the variances of `p1` and `p2` have not been checked, i.e., *unknown* to the programmer.

In the file `doc/doc_examples/example2.ml`, we have an example that conducts the paired t-test command `exec_ttest_paired`. To verify this code, execute `statwhy example2.ml`, which will open the Why3 IDE as follows.



Equal vs. Unequal Variance We explain the *equal variance* assumption in the non-paired t-test as follows. In the non-paired t-test, we should use different t-tests according to this assumption. When we know or assume that two populations have equal variance, we usually use *Student's t-test*. In contrast, we apply *Welch's t-test* if we cannot assume equal variance. StatWhy distinguishes the difference by the precondition `eq_variance`.

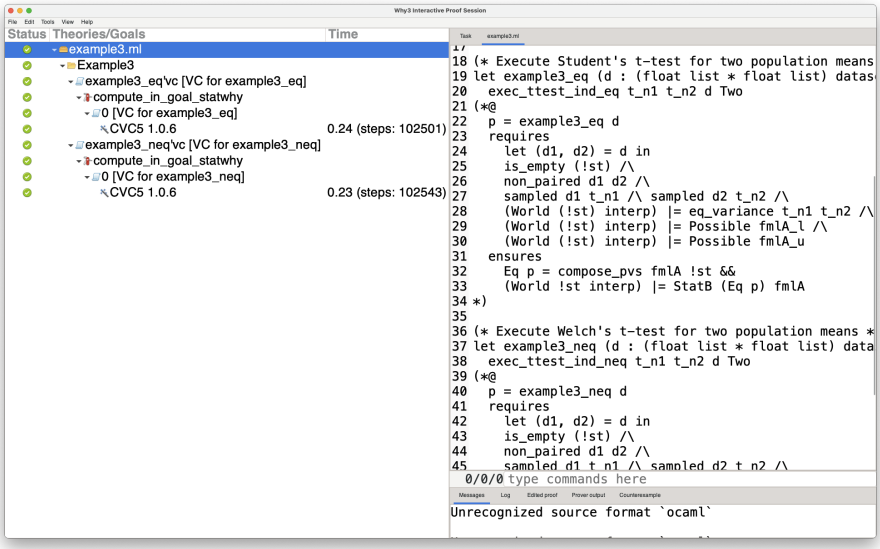
`exec_ttest_ind_eq` in the last section is our formalization of Student's t-test. `(World !st interp) |= eq_variance p1 p2` in the precondition denotes that the two populations that respectively draw the datasets `y1` and `y2` have equal variance.

In contrast, `exec_ttest_ind_neq` below assumes `Not (eq_variance p1 p2)`:

```
val exec_ttest_ind_neq (p1 : population) (p2 : population) (y : dataset (list
    real, list real)) (alt : alternative) : real
writes { st }
requires {
  match (p1, p2) with
  | ((NormalD _ _), (NormalD _ _)) ->
  match y with
  | (y1, y2) ->
      non_paired y1 y2 /\
      (World !st interp) |= Not (check_variance p1) /\
      (World !st interp) |= Not (check_variance p2) /\
      (World !st interp) |= Not (eq_variance p1 p2) /\
      sampled y1 p1 /\ sampled y2 p2 /\
      let r1 = mean p1 in
      let r2 = mean p2 in
      match alt with
      | Two ->
```

```
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Up ->
        (World !st interp) |= Not (Possible (r1 <' r2)) /\
        (World !st interp) |= Possible (r1 >' r2)
      | Low ->
        (World !st interp) |= Possible (r1 <' r2) /\
        (World !st interp) |= Not (Possible (r1 >' r2))
      end
  end
  | _ -> false
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean p1 in
  let r2 = mean p2 in
  let h = match alt with
    | Two -> r1 =!' r2
    | Up -> r1 >' r2
    | Low -> r1 <' r2
  end in !st = Cons ("ttest_ind_neq", h, Eq pv) !(old st)
}
```

The code examples for these t-tests are available in `doc/doc_examples/example3.ml`



6.3 Dealing with Combined Tests in StatWhy

Let A_{φ_0} and A_{φ_1} be two hypothesis tests with alternative hypotheses φ_0 and φ_1, respectively. There are two possible combinations of A_{φ_0} and A_{φ_1}. One is the

disjunctive combination $A_{\varphi_0 \vee \varphi_1}$ whose alternative hypothesis is $\varphi_0 \vee \varphi_1$, while the other is the conjunctive combination $A_{\varphi_0 \wedge \varphi_1}$ with alternative hypothesis $\varphi_0 \wedge \varphi_1$. StatWhy can check whether a program correctly calculates the p-value of such combined tests.

***P*-Values of Disjunctive Alternative Hypothesis** Assume that p-values of A_{φ_0} and A_{φ_1} are p_0 and p_1, respectively. It is known that the p-value p of $A_{\varphi_0 \vee \varphi_1}$ satisfies $p \leq p_0 + p_1$, which is called the Bonferroni correction. StatWhy automatically calculates the p-value of $A_{\varphi_0 \wedge \varphi_1}$ as $p_0 + p_1$ if A_{φ_0} and A_{φ_1} are performed independently.

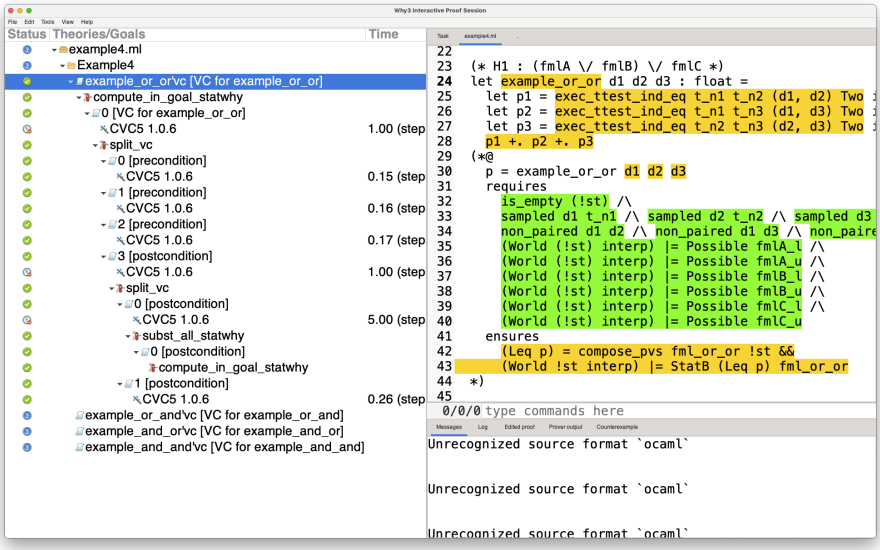
The code below defines a procedure `example_or_or`, which compares the dataset `d1` with `d2` and `d3`, and `d2` with `d3`, then calculates the overall p-value.

doc/doc_examples/example4.ml

```
module Example4 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var")
  let t_n2 = NormalD (Param "mu2", Param "var")
  let t_n3 = NormalD (Param "mu3", Param "var")
  let fmlA_l = Atom (Pred ("<", [ RealT (mean t_n1); RealT (mean t_n2) ]))
  let fmlA_u = Atom (Pred (">", [ RealT (mean t_n1); RealT (mean t_n2) ]))
  let fmlA = Atom (Pred ("=!", [ RealT (mean t_n1); RealT (mean t_n2) ]))
  let fmlB_l = Atom (Pred ("<", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlB_u = Atom (Pred (">", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlB = Atom (Pred ("=!", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlC_l = Atom (Pred ("<", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fmlC_u = Atom (Pred (">", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fmlC = Atom (Pred ("=!", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fml_or_or = Disj (Disj (fmlA, fmlB), fmlC)
  let fml_or_and = Disj (Conj (fmlA, fmlB), fmlC)
  let fml_and_or = Conj (Disj (fmlA, fmlB), fmlC)
  let fml_and_and = Conj (Conj (fmlA, fmlB), fmlC)

  (* H1 : (fmlA \/ fmlB) \/ fmlC *)
  let example_or_or d1 d2 d3 : float =
    let p1 = exec_ttest_ind_eq t_n1 t_n2 (d1, d2) Two in
    let p2 = exec_ttest_ind_eq t_n1 t_n3 (d1, d3) Two in
    let p3 = exec_ttest_ind_eq t_n2 t_n3 (d2, d3) Two in
    p1 +. p2 +. p3
  (*@
    p = example_or_or d1 d2 d3
    requires
      is_empty (!st) /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t_n3 /\
      non_paired d1 d2 /\ non_paired d1 d3 /\ non_paired d2 d3 /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u /\
      (World (!st) interp) |= Possible fmlB_l /\
      (World (!st) interp) |= Possible fmlB_u /\
      (World (!st) interp) |= Possible fmlC_l /\
      (World (!st) interp) |= Possible fmlC_u
```

```
    ensures
      (Leq p) = compose_pvs fml_or_or !st &&
      (World !st interp) |= StatB (Leq p) fml_or_or
  *)

  ...
end
```

To verify this code, run `statwhy doc/doc_examples/example4.ml` and apply the "StatWhy" strategy to `example_or_or'vc`.



***P*-Values of Conjunctive Hypotheses** To calculate the p-value of a conjunctive hypothesis (e.g., $\varphi_0 \wedge \varphi_1$), we take the minimum of the p-values of its subformulas φ_0 and φ_1. StatWhy can also verify these conjunctive combinations of hypothesis tests.

The code below compares `d1`, `d2`, and `d3`, as the code in the last section, but reports the smallest p-value among the three comparisons.

doc/doc_examples/example4.ml

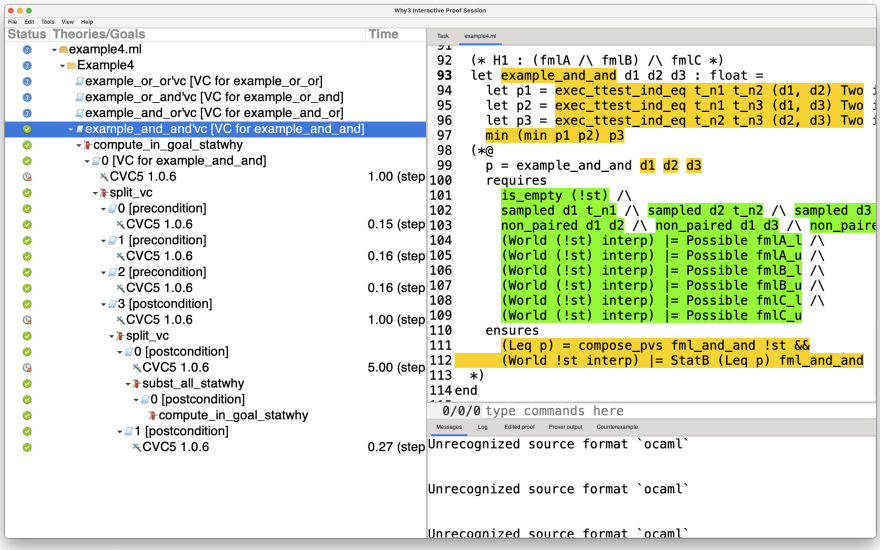
```
open CameleerBHL

module Example4 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var")
  let t_n2 = NormalD (Param "mu2", Param "var")
  let t_n3 = NormalD (Param "mu3", Param "var")
  let fmlA_l = Atom (Pred ("<", [ RealT (mean t_n1); RealT (mean t_n2) ]))
  let fmlA_u = Atom (Pred (">", [ RealT (mean t_n1); RealT (mean t_n2) ]))
```

```
  let fmlA = Atom (Pred ("=!", [ RealT (mean t_n1); RealT (mean t_n2) ]))
  let fmlB_l = Atom (Pred ("<", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlB_u = Atom (Pred (">", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlB = Atom (Pred ("=!", [ RealT (mean t_n1); RealT (mean t_n3) ]))
  let fmlC_l = Atom (Pred ("<", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fmlC_u = Atom (Pred (">", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fmlC = Atom (Pred ("=!", [ RealT (mean t_n2); RealT (mean t_n3) ]))
  let fml_or_or = Disj (Disj (fmlA, fmlB), fmlC)
  let fml_or_and = Disj (Conj (fmlA, fmlB), fmlC)
  let fml_and_or = Conj (Disj (fmlA, fmlB), fmlC)
  let fml_and_and = Conj (Conj (fmlA, fmlB), fmlC)

  ...

  (* H1 : (fmlA /\ fmlB) /\ fmlC *)
  let example_and_and d1 d2 d3 : float =
    let p1 = exec_ttest_ind_eq t_n1 t_n2 (d1, d2) Two in
    let p2 = exec_ttest_ind_eq t_n1 t_n3 (d1, d3) Two in
    let p3 = exec_ttest_ind_eq t_n2 t_n3 (d2, d3) Two in
    min (min p1 p2) p3
  (*@
    p = example_and_and d1 d2 d3
    requires
      is_empty (!st) /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t_n3 /\
      non_paired d1 d2 /\ non_paired d1 d3 /\ non_paired d2 d3 /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u /\
      (World (!st) interp) |= Possible fmlB_l /\
      (World (!st) interp) |= Possible fmlB_u /\
      (World (!st) interp) |= Possible fmlC_l /\
      (World (!st) interp) |= Possible fmlC_u
    ensures
      (Leq p) = compose_pvs fml_and_and !st &&
      (World !st interp) |= StatB (Leq p) fml_and_and
  *)
end
```

To verify the code above, execute the following command:

```
$ statwhy doc/doc_examples/example4.ml
```

And apply the "StatWhy" strategy to `example_and_and'vc`.



***P*-Value Hacking** The *p-value hacking* or *data dredging* is a method for manipulating statistical analysis to obtain a lower p-value than the actual one. In this example, we see that StatWhy prevents the p-value hacking by calculating correct p-values with the results of conducted hypothesis tests.

The following code, `ex_hack` in `doc/doc_examples/example5.ml`, is an example of p-value hacking:

doc/doc_examples/example5.ml

```
open CameleerBHL

module Example5 = struct
  open Ttest

  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = Atom (Pred ("<", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))
  let fmlA_u = Atom (Pred (">", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))
  let fmlA = Atom (Pred ("=!", [ RealT (mean t_n); RealT (Real (Const 1.0)) ]))

  (* Example of p-value hacking *)
  let ex_hack d1 d2 =
    let p1 = exec_ttest_1samp t_n 1.0 d1 Two in
    let p2 = exec_ttest_1samp t_n 1.0 d2 Two in
    let p = min p1 p2 in
    (p1, p2, p)
  (*@
    (p1, p2, p) = ex_hack d1 d2
    requires
      is_empty (!st) /\
      sampled d1 t_n /\ sampled d2 t_n /\
```

```
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      (Eq p = compose_pvs fmlA !st (* This is incorrect *)
        && (World !st interp) |= StatB (Eq p) fmlA) /\
      (Leq (p1 +. p2) = compose_pvs fmlA !st (* This is correct *)
        && (World !st interp) |= StatB (Leq (p1 +. p2)) fmlA)
  *)
end

end
```

`ex_hack` performs the t-test for the mean of `t_n` twice, using different datasets `d1` and `d2`. Then it obtains the p-values for each test, `p1` and `p2`, and reports the lower p-value `p` (defined by `min p1 p2`).

The postcondition of this function consists of two main formulas:

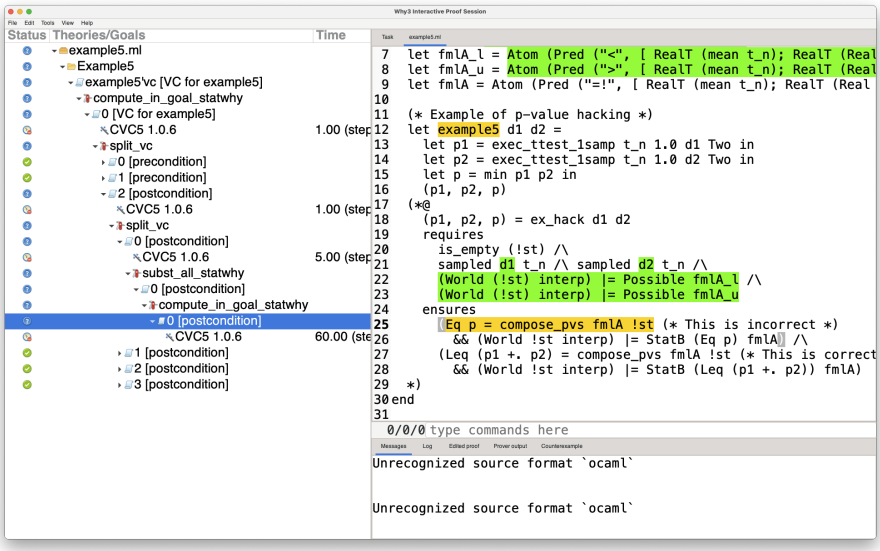
```
Eq p = compose_pvs fmlA !st && (World !st interp) |= StatB (Eq p) fmlA
```

and

```
Leq (p1 +. p2) = compose_pvs fmlA !st
&& (World !st interp) |= StatB (Leq (p1 +. p2)) fmlA.
```

In the former assertion, `Eq p = compose_pvs fmlA !st` is a wrong interpretation of the result. In contrast, the latter states that the sum of `p1` and `p2` is the p-value of `fmlA`, which is correct.

StatWhy does validate the latter, but not the former; `Eq p = compose_pvs fmlA !st` fails to be validated:



6.4 Hypothesis Tests for More Than Two Population Means

This example illustrates hypothesis tests to analyze the difference between more than two population means.

StatWhy provides the specification of the *one-way ANOVA (analysis of variance)* to test for overall differences among the groups. It also includes the non-parametric *Kruskal-Wallis test* and the *Alexander-Govern test* as alternatives when the assumptions of ANOVA are not satisfied.

These methods test whether all the given population means are identical or not. To test the differences of each pairing of the given means, we should use a multiple comparison method such as Tukey's HSD test (see Section 6.5.)

One-Way ANOVA We show the specification of the one-way ANOVA in StatWhy as follows:

```
val exec_oneway (ps : list population) (ys : dataset (list (list real))) : real
writes { st }
requires {
  length ps = length ys &&
  length ps >= 2 /\ length ys >= 2 &&
  for_all2 (fun p y -> match p with | NormalD _ _ -> sampled y p | _ -> false
       end) ps ys /\
  (forall p q : population. mem p ps /\ mem q ps ->
    (World !st interp) |= eq_variance p q) /\
  (forall s t : population. mem s ps /\ mem t ps /\ not eq_population s t ->
    ((World !st interp) |= Possible (mean s <' mean t) /\
     (World !st interp) |= Possible (mean s >' mean t)))
} ensures {
  let pv = result in
  pvalue result /\
  let h = oneway_hypothesis ps in
  !st = !(old st) ++ Cons ("oneway", h, Eq pv) Nil
}
```

It is worth noting that the input arguments for terms and datasets used in the one-way ANOVA are lists, which allows for the comparisons of arbitrary finite groups of data.

The one-way ANOVA assumes the following conditions:

1. Each population of the samples is normally distributed.
2. All the populations have the same variance.

The former assumption is formalized as:

```
for_all2
  (fun p y -> match p with | NormalD _ _ -> sampled y p | _ -> false end)
  ps ys
```

The latter is specified by:

```
(forall p q : population. mem p ps /\ mem q ps ->
  (World !st interp) |= eq_variance p q)
```

For the sake of brevity, we introduce an abbreviation that is not included in the WhyML syntax. In the above specification, the alternative hypothesis is abbreviated as `oneway_hypothesis terms`, which represents all the possible combinations of comparisons. For example, `oneway_hypothesis` applied to `[termA; termB; termC]`[11] represents the formula (`termA` $\neq$ `termB`) $\wedge$ (`termA` $\neq$ `termC`) $\wedge$ (`termB` $\neq$ `termC`).

The following code conducts the one-way ANOVA for three population means.

doc/doc_examples/mlw/ex_oneway.mlw

```
module Oneway_example
  use cameleerBHL.CameleerBHL
  use oneway.Oneway

  let function p1 = NormalD (Param "m1") (Param "v")
  let function p2 = NormalD (Param "m2") (Param "v")
  let function p3 = NormalD (Param "m3") (Param "v")

  let function t_m1 = mean p1
  let function t_m2 = mean p2
  let function t_m3 = mean p3

  let ex_oneway (d1 d2 d3 : (list real))
  (* Executes one-way ANOVA for 3 population means *)
    requires { is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
                 (Cons p1 (Cons p2 (Cons p3 Nil)))
                 (Cons d1 (Cons d2 (Cons d3 Nil))) /\
               ((World !st interp) |= Possible (t_m1 <' t_m2)) /\
               ((World !st interp) |= Possible (t_m1 >' t_m2)) /\
               ((World !st interp) |= Possible (t_m1 <' t_m3)) /\
               ((World !st interp) |= Possible (t_m1 >' t_m3)) /\
               ((World !st interp) |= Possible (t_m2 <' t_m3)) /\
               ((World !st interp) |= Possible (t_m2 >' t_m3))
             }
    ensures {
      let p = result in
      let h = (t_m1 =!' t_m2) /\* (t_m1 =!' t_m3) /\* (t_m2 =!' t_m3) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_oneway (Cons p1 (Cons p2 (Cons p3 Nil))) (Cons d1 (Cons d2 (Cons d3
        Nil)))
end
```

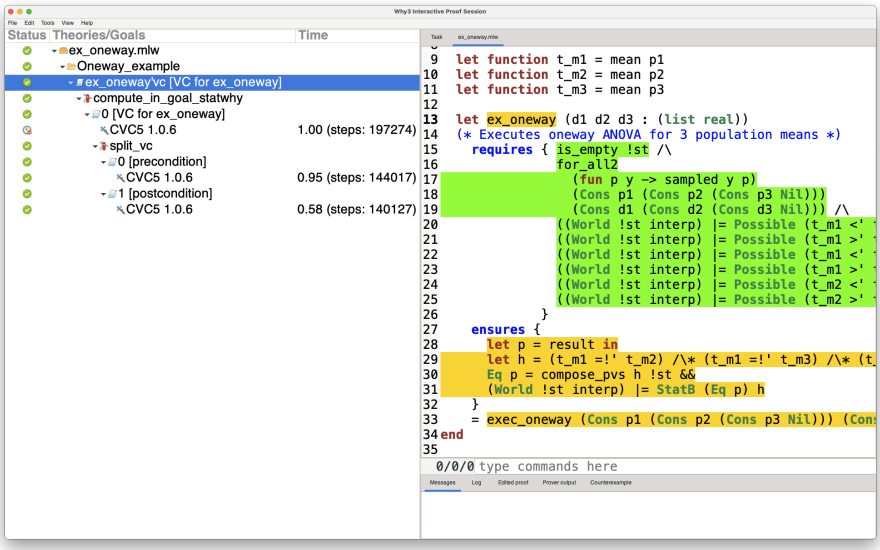
In the postcondition, `h = fmlA /\* fmlB /\* fmlC` confirms the concrete form of the alternative hypothesis `h` [12].

[11] In WhyML programs, this list notation is not available and you need to construct lists with bare algebraic data types, e.g., `Cons termA (Cons termB (Cons termC Nil))`.

[12] `/\*` is the syntax sugar for the conjunction of two hypotheses. Similarly, `\/+` is for the disjunction of two hypotheses.

To verify the code above, execute the following command:

```
$ statwhy doc/doc_examples/mlw/ex_oneway.mlw
```



Alexander-Govern Test The *Alexander-Govern test* requires that each sample comes from a normally distributed population, but relaxes the assumption of equal variance on ANOVA. We show an example of the Alexander-Govern test as follows:

doc/doc_examples/mlw/ex_alexandergovern.mlw

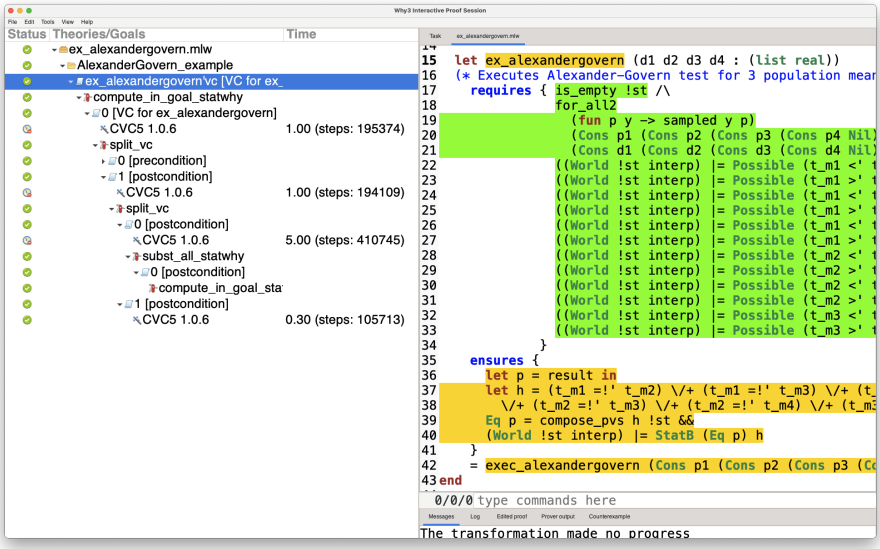
```
module AlexanderGovern_example
  use cameleerBHL.CameleerBHL
  use alexandergovern.AlexanderGovern

  let function p1 : population = NormalD (Param "mean1") (Param "var1")
  let function p2 : population = NormalD (Param "mean2") (Param "var2")
  let function p3 : population = NormalD (Param "mean3") (Param "var3")
  let function p4 : population = NormalD (Param "mean4") (Param "var4")

  let function t_m1 = mean p1
  let function t_m2 = mean p2
  let function t_m3 = mean p3
  let function t_m4 = mean p4

  let ex_alexandergovern (d1 d2 d3 d4 : (list real))
  (* Executes Alexander-Govern test for 3 population means *)
    requires { is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
```

```
                (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil))))
                (Cons d1 (Cons d2 (Cons d3 (Cons d4 Nil)))) /\
              ((World !st interp) |= Possible (t_m1 <' t_m2)) /\
              ((World !st interp) |= Possible (t_m1 >' t_m2)) /\
              ((World !st interp) |= Possible (t_m1 <' t_m3)) /\
              ((World !st interp) |= Possible (t_m1 >' t_m3)) /\
              ((World !st interp) |= Possible (t_m1 <' t_m4)) /\
              ((World !st interp) |= Possible (t_m1 >' t_m4)) /\
              ((World !st interp) |= Possible (t_m2 <' t_m3)) /\
              ((World !st interp) |= Possible (t_m2 >' t_m3)) /\
              ((World !st interp) |= Possible (t_m2 <' t_m4)) /\
              ((World !st interp) |= Possible (t_m2 >' t_m4)) /\
              ((World !st interp) |= Possible (t_m3 <' t_m4)) /\
              ((World !st interp) |= Possible (t_m3 >' t_m4))
          }
    ensures {
      let p = result in
      let h = (t_m1 =!' t_m2) \/+ (t_m1 =!' t_m3) \/+ (t_m1 =!' t_m4)
        \/+ (t_m2 =!' t_m3) \/+ (t_m2 =!' t_m4) \/+ (t_m3 =!' t_m4) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_alexandergovern (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil)))) (Cons d1 (
        Cons d2 (Cons d3 (Cons d4 Nil))))
end
```

Note that the assumption of equal variance is no longer necessary in the precondition.

To verify the code above, execute the following command:

```
$ statwhy doc/doc_examples/mlw/ex_alexandergovern.mlw
```



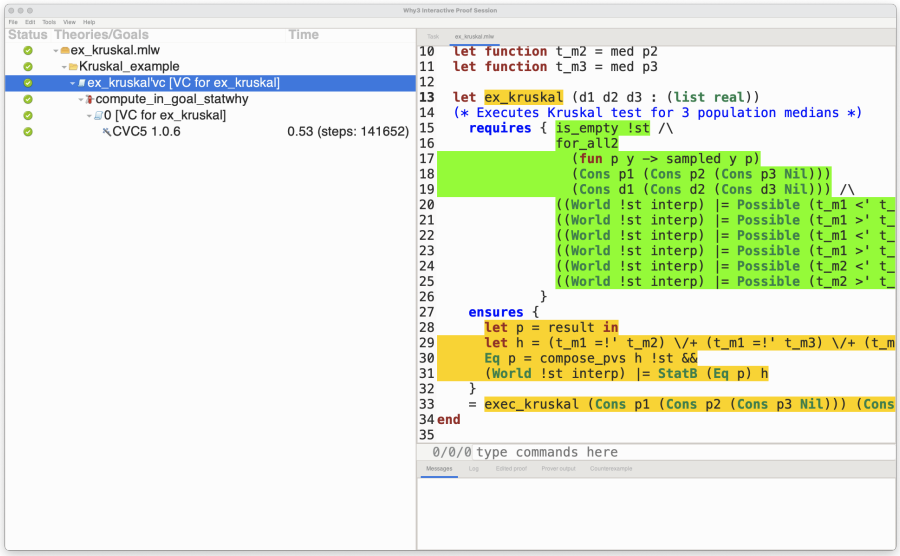
Kruskal-Wallis Test The *Kruskal-Wallis H-test* is a non-parametric variant of ANOVA; it does not assume that each sample is from normally distributed populations with equal variance. Here is an example of the Kruskal-Wallis H-test whose null hypothesis is that all medians of three populations are equal:

doc/doc_examples/mlw/ex_kruskal.mlw

```
module Kruskal_example
  use cameleerBHL.CameleerBHL
  use kruskal.Kruskal

  let function p1 = UnknownD "p1"
  let function p2 = UnknownD "p2"
  let function p3 = UnknownD "p3"

  let function t_m1 = med p1
  let function t_m2 = med p2
  let function t_m3 = med p3

  let ex_kruskal (d1 d2 d3 : (list real))
  (* Executes Kruskal test for 3 population medians *)
    requires { is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
                 (Cons p1 (Cons p2 (Cons p3 Nil)))
                 (Cons d1 (Cons d2 (Cons d3 Nil))) /\
               ((World !st interp) |= Possible (t_m1 <' t_m2)) /\
               ((World !st interp) |= Possible (t_m1 >' t_m2)) /\
               ((World !st interp) |= Possible (t_m1 <' t_m3)) /\
               ((World !st interp) |= Possible (t_m1 >' t_m3)) /\
               ((World !st interp) |= Possible (t_m2 <' t_m3)) /\
               ((World !st interp) |= Possible (t_m2 >' t_m3))
             }
    ensures {
      let p = result in
      let h = (t_m1 =!' t_m2) \/+ (t_m1 =!' t_m3) \/+ (t_m2 =!' t_m3) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_kruskal (Cons p1 (Cons p2 (Cons p3 Nil))) (Cons d1 (Cons d2 (Cons d3
        Nil)))
end
```

You can verify the code above by running:

```
$ statwhy doc/doc_examples/mlw/ex_kruskal.mlw
```



6.5 Tukey's HSD Test

In this section, we show the verification of a program for a popular multiple comparison test called the *Tukey's HSD test*.

Given datasets from multiple groups, Tukey's HSD test examines the differences in means for each pair of groups. In StatWhy, the specification of the test is as follows:

```
val exec_tukey_hsd (ppls : list population) (ys : dataset (list (list real))) :
      array real
writes { st }
requires {
  Length.length ppls = Length.length ys /\
  for_all2 (fun p y -> match p with | NormalD _ _ -> sampled y p | _ -> false
       end) ppls ys /\
  for_all
    (fun t -> let (x, y) = t in
      (World !st interp) |= eq_variance x y)
    (combinations_poly ppls) /\
  for_all
    (fun t -> let (s, t) = t in
      ((World !st interp) |= Possible (mean s <' mean t) /\
       (World !st interp) |= Possible (mean s >' mean t)))
    (cmb_term ppls)
}
ensures {
  let ps = result in
  length ps = div2 ((Length.length ppls) * (Length.length ppls - 1)) /\
  let b = length ps in
  (forall i : int. 0 <= i < b -> pvalue (ps[i])) /\
  let cmbs = combinations (map (fun p -> RealT (mean p)) ppls) "=!" in
```

```
  !st = (cmb_store cmbs ps 0) ++ !(old st)
}
```

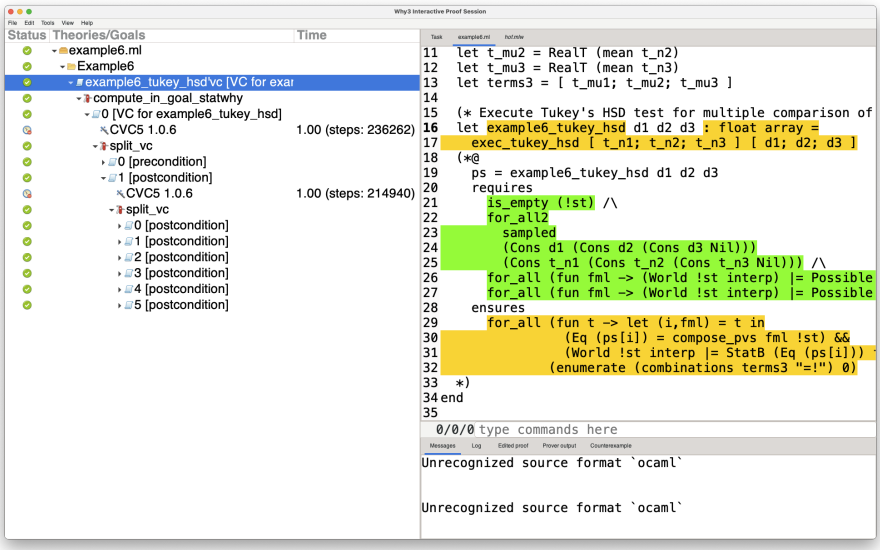
`ppls` denote the list of populations being tested, and `ys` is the datasets sampled from the populations. `exec_tukey_hsd` returns an array of p-values. These p-values are sorted in lexicographic order, such as $p_{12}, p_{13}, p_{14}, p_{23}, p_{24}, p_{34}$, where p_{ij} is the p-value for the comparison of groups i and j.

`exec_tukey_hsd` requires that all the populations have the same variance, which is specified by

```
for_all
    (fun t -> let (x, y) = t in
      (World !st interp) |= eq_variance x y)
    (combinations_poly ppls)
```

`example6_tukey_hsd` in `doc/doc_examples/example6.ml` conducts Tukey's HSD test for three groups. To verify the code, execute the following command:

```
$ statwhy doc/doc_examples/example6.ml
```



6.6 Scalability of StatWhy

We conducted experiments to evaluate the performance of the program verification using StatWhy. We performed the experiments on a MacBook Pro with Apple M2 Max CPU and 96 GB memory using the external SMT solver CVC5 1.0.6.

Table 1 summarizes the execution times for StatWhy to verify programs for popular single-comparison hypothesis testing methods. The verification of these

testing methods is very efficient even when precondition formulas are relatively large (e.g., in Alexander-Govern test and χ^2 test).

Table 1: The execution times (sec) of programs for popular single-comparison hypothesis testing.

File names	Test methods	Times (sec)
ex_alexandergovern.mlw	Alexander-Govern test	8.75
ex_bartlett.mlw	Bartlett's test	0.87
ex_binom.mlw	Binomial test	0.46
ex_chisquare.mlw	χ^2 test	5.50
ex_ftest.mlw	F-test	0.45
ex_kruskal.mlw	Kruskal-Wallis test	0.88
ex_oneway.mlw	One-way ANOVA	2.55
ex_ttest.mlw	One-sample t-test	0.46
	Student's t-test	0.48
	Welch's t-test	0.43
	Paired t-test	0.44

Table 2 shows the execution times for StatWhy to verify hypothesis testing programs for practical numbers of disjunctive/conjunctive hypotheses. The verification of these programs is very efficient, since our proof strategy explained in [3] can accelerate the proof search for programs with folding operations and test histories.

Table 3 presents the execution times (sec) for various multiple comparison methods. For a practical number of groups, StatWhy verifies the multiple comparison programs efficiently.

References

1. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - providing ocaml with a formal specification language. In: Proc. of the 24th International Symposium on Formal Methods (FM 2019). Lecture Notes in Computer Science, vol. 11800, pp. 484–501. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_29
2. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Proc. of the 22nd European Symposium on Programming (ESOP 2013). Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
3. Kawamoto, Y., Kobayashi, K., Suenaga, K.: Statwhy: Formal verification tool for statistical hypothesis testing programs. CoRR **abs/2405.17492** (2024). https://doi.org/10.48550/ARXIV.2405.17492
4. Kawamoto, Y., Sato, T., Suenaga, K.: Formalizing statistical beliefs in hypothesis testing using program logic. In: Proc. KR'21. pp. 411–421 (2021). https://doi.org/10.24963/kr.2021/39

Table 2: The execution times (sec) for verifying hypothesis testing programs with different numbers of disjunctive (OR) and conjunctive (AND) hypotheses. In practice, the number of hypotheses in hypothesis testing is less than 10.

#hypotheses	2	3	4	5	6	7	8	9	10
OR	8.77	8.89	8.84	8.94	9.01	9.01	9.04	9.16	9.23
AND	8.82	8.72	8.86	8.98	8.95	9.03	9.11	9.17	9.46

Table 3: The execution times (sec) for various multiple comparison methods. #groups (resp. #comparisons) represents the number of groups (resp. combinations of groups) compared in the hypothesis testing. In practice, #groups in multiple comparison is at most 7.

		#groups					
Test methods	Metric	2	3	4	5	6	7
Tukey's HSD test	Times (sec)	0.37	9.09	9.33	9.81	15.27	16.39
	#comparisons	1	3	6	10	15	21
Dunnett's test	Times (sec)	0.48	8.98	9.17	9.61	9.62	9.77
	#comparisons	1	2	3	4	5	6
Williams' test	Times (sec)	0.48	8.90	9.04	9.16	9.23	9.58
	#comparisons	1	2	3	4	5	6
Steel-Dwass' test	Times (sec)	0.44	9.05	9.43	9.76	15.10	16.24
	#comparisons	1	3	6	10	15	21
Steel's test	Times (sec)	0.49	8.79	8.92	9.11	9.43	9.74
	#comparisons	1	2	3	4	5	6

5. Kawamoto, Y., Sato, T., Suenaga, K.: Sound and relatively complete belief hoare logic for statistical hypothesis testing programs. Artif. Intell. **326**, 104045 (2024). https://doi.org/10.1016/J.ARTINT.2023.104045
6. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for ocaml. In: Proc. of the 33rd International Conference on Computer Aided Verification (CAV 2021), Part II. Lecture Notes in Computer Science, vol. 12760, pp. 677–689. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_31