

Artifact: A Hybrid Approach to Semi-Automated Rust Verification

Welcome to the README for the artifact accompanying the PLDI25 paper, “A Hybrid Approach to Semi-Automated Rust Verification.”

This document:

- Summarizes the main claims of the paper, supported by this artifact.
- Provides instructions for loading the Docker image, running experiments, and navigating the code.
- Highlights key code snippets from the OCaml and Rust codebases.
- Offers a brief tutorial on writing and running proofs with Gillian-Rust using the EvenInt example.

Table of Contents

1. Claims	1
2. Getting Started	1
3. Running the Case Studies	2
3.1. Gillian-Rust	2
3.1.1. Running all cases studies: <code>test_docker_gil.sh</code>	2
3.1.2. Correspondance with the table page 17 of the paper	3
3.1.3. Running individual case studies: <code>gr.sh</code>	4
3.2. Creusot	4
4. Rebuilding the code	4
5. Edit code in the docker container	5
6. Code Structure	5
7. Understanding the OCaml State Model: Prophecy Context	7
7.1. Structure Definition	7
7.2. Value Observer Producer	7
8. Reading the Gilogic Library	9
8.1. Ownership Predicate for <code>Option<T></code>	9
8.2. MutRef-Resolve Rule	9
9. Tutorial: EvenInt	10
9.1. Specifying the ownership predicate	10
9.2. Constructors	11
9.3. <code>add_two</code>	12
9.4. Exercise	13

1. Claims

The paper makes the following claims:

- We developed Gillian-Rust, an instantiation of the Gillian compositional symbolic execution platform for verifying type safety and functional correctness of Rust code.
- Several case studies have been conducted; their results appear both in the paper.
- The specifications used to verify unsafe code with Gillian-Rust can be efficiently reused by Creusot. In our hybrid approach, safely encapsulated unsafe Rust code is verified by Gillian-Rust and then reused in Creusot.

2. Getting Started

The artifact archive includes:

- The complete code required for the experiments.
- A Docker image that bundles the code and all dependencies.

We recommend using the Docker image directly. (Note: the image is built for x86_64; on Apple Silicon it is run on Rosetta and may be slower.)

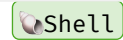
To run the experiments, execute the following commands:

```
# Unzip the image to obtain a Docker tarball
gunzip gillian-rust-docker.tar.gz

# Load the Docker image
docker load -i gillian-rust-docker.tar

# Run the container:
# For x86_64 (Intel MacOS, most Windows/Linux):
docker run -it gillian-rust
# For Apple Silicon (Apple M1-4 series):
docker run --platform=linux/amd64 -it gillian-rust

# Inside the container, run:
./test_docker_gil.sh
```



If no failure is reported, everything is running correctly!

3. Running the Case Studies

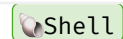
3.1. Gillian-Rust

3.1.1. Running all cases studies: `test_docker_gil.sh`

The script `test_docker_gil.sh` runs all Gillian-Rust case studies on compiled GIL files in `tests/{noproph,proph}/*.stdout`.

For instance, consider `tests/noproph/list_std.rs`, which corresponds to the **LinkedList – TS** case study (see page 18 of the paper):

```
Noproph verify: ../tests/noproph/list_std.rs
Parsing and compiling...
Preprocessing...
Obtaining specs to verify...
Obtaining lemmas to verify...
Obtained 11 symbolic tests in total
Running symbolic tests: 0.011728
Verifying lemma dll_seg_append_right::<T> ... Success
Verifying lemma dll_seg_l_to_r::<T> ... Success
Verifying lemma dll_seg_r_append_left::<T> ... Success
Verifying lemma dll_seg_r_to_l::<T> ... Success
Verifying lemma extract_head___proof::<T> ... Success
Verifying one spec of procedure LinkedList::<T>::pop_back ... s s s s s s s s s
Success
Verifying one spec of procedure LinkedList::<T>::front_mut ... s s s s Success
Verifying one spec of procedure LinkedList::<T>::push_back ... s s s s Success
Verifying one spec of procedure LinkedList::<T>::pop_front ... s s s s s s s s s
Success
```



```
Verifying one spec of procedure LinkedList::<T>::push_front ... s s s s Success
Verifying one spec of procedure LinkedList::<T>::new ... s Success
All specs succeeded: 0.126690
```

Gillian-Rust:

- Parses and preprocesses the GIL file to gather all specifications and lemmas.
- Verifies both hand-written lemmas (with explanations in the code) as well as the automatically derived one.
- Symbolically executes each Rust function. Each “s” indicates a successful branch which unified against the function’s post-condition.
- Reports the overall verification time.

The file `tests/noproph/list_std.rs` is heavily commented, making it easier to connect these concepts with the paper. We encourage a close look.

3.1.2. Correspondance with the table page 17 of the paper

The table on page 17 of the paper lists the case studies and their respective execution times. Below, we provide a table with the path to the corresponding files in the `tests/noproph` (for Type Safety, without prophecy reasoning) and `tests/proph` (for Functional Correctness, with prophecy reasoning) folders.

Name	Path
EvenInt	proph/even_int.rs
LP (TS)	noproph/lp.rs
LP (FC)	proph/lp_proph.rs
LinkedList (TS)	noproph/list_std.rs
LinkedList (FC)	proph/list_std_proph.rs
MiniVec	proph/minivec.rs
Vec (TS)	noproph/vec_std.rs
Vec (FC)	proph/vec_std_proph.rs

Counting Lines of Code: The table includes counts for executable lines of code (ELoC) and specification lines of code (SLoC). Since our annotations are written in Rust syntax, we lack an automated method for counting these lines. The numbers in the table are manually counted, aiming for accuracy while disregarding formatting whenever possible.

For example, in the annotations below, lines containing only punctuation are excluded. Thus, we count 6 lines of specifications:

```
#[specification(
    forall data.
    requires {
        dll_seg(head, tail_next, tail, head_prev, data)
    }
    ensures {
        dll_seg_r(head, tail_next, tail, head_prev, data)
    }
)]
```



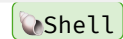
Comparison with RefinedRust: The paper reports preliminary timing comparison with RefinedRust for the EvenInt and MiniVec case studies. These results can be replicated by running these proofs using [the PLDI'24 artifact of RefinedRust](#).

3.1.3. Running individual case studies: `gr.sh`

To run a single case study, another script is provided. It is provided should users like to modify a case study and re-run it. It can be executed as follows:

```
# For Type Safety case studies
./gr.sh noproph/list_std -l disabled

# For Functional Correctness case studies
./gr.sh proph/list_std_proph --proph -l disabled
```



Note:

- This script first re-compiles the file and its dependencies, and then executes the proof backend. The first time it is executed, it may take more time as it recompiles also the gillogic library.
- `-l disabled` removes the slow and extensive logging performed by Gillian. Without this flag, proofs should execute about 3-4 times slower.
- The `--proph` flag is necessary to run case studies from the `proph` folder.

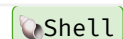
3.2. Creusot

The paper claims that the hybrid specifications verified by Gillian-Rust can be efficiently reused by Creusot.

To demonstrate this, we have proven sorting algorithms using Creusot (specifically, functions called `right_pad`, `gnome_sort`, and `merge_sort` in `test_crates/sort/src/main.rs`), reusing specifications that were proven in our case studies and copy-pasted in the `test_crates/proven` crate.

To run these Creusot proofs:

```
# In the test_crates/sort directory:
cargo creusot prove
```



Troubleshooting:

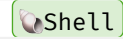
- If proofs fail, it is probably due to timeouts on older hardware or on Apple Silicon with Docker via Rosetta. adjust the "fast" and "time" fields in the `why3find.json` file.
- We are aware of an issue on some machines where running `cargo creusot prove` may yield the error `Error: can not calibrate prover cvc5@1.0.5`. This error is due to a potential bug in `why3find` which is developed independently of Gillian-Rust. Please try updating your version of docker and re-installing the docker image.

Note: The timings reported in the paper are obtained by commenting out all the code that is not relevant to each algorithm (respectively, Gnome Sort / Merge Sort / Right Pad), and running `cargo creusot && time cargo creusot prove` in the `test_crates/sort` folder.

4. Rebuilding the code

Should the user want to modify the source code and run an updated version, we provide a script to rebuild the entirety of the code (frontend, backend and case study GIL files):

```
./build_all_docker.sh
```

**Note:**

- The commands based on the `gr.sh` script will rebuild both frontend and backend, so it is not necessary to run the `./build_all_docker.sh` script before running `gr.sh`.
- The `cargo gillian` command, on the other hand, uses the *installed* version of the frontend and backend. Therefore, it is necessary to run `./build_all_docker.sh` before running `cargo gillian` if you have modified the code and want to use the new version.
- The artifact should be self contained: the `build_all_docker.sh` script makes use of the `--offline` flag of `cargo` to avoid downloading new dependencies. Similarly, `Opam` should not fetch anything from the internet as the Docker image already contains all the dependencies. It is also possible to inspect the source of all the OCaml dependencies in the `/root/.opam/gillian-rust/lib/` folder.

5. Edit code in the docker container

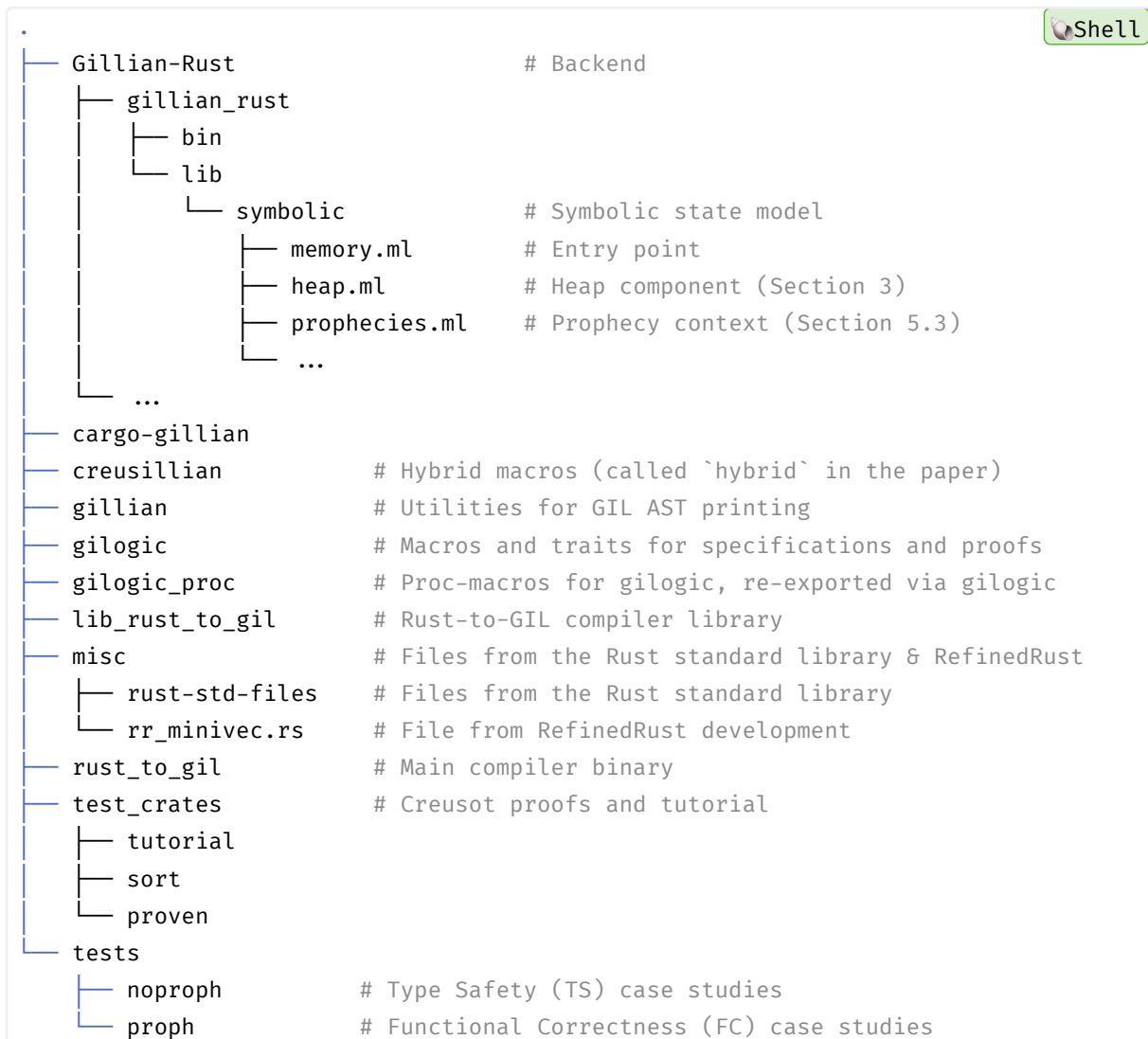
For an enhanced coding experience, we recommend using VSCode:

- [Install VSCode](#) along with the [Docker](#) and [Dev Containers](#) extensions.
- Open the Docker extension (in the left-hand sidebar), find your running container from your previous execution of `docker run`, right-click to “Start” (if needed) and “Attach Visual Studio Code.”
- Then, within the newly opened VSCode window, open the `/app` folder via the command palette (`Ctrl+Shift+P` → File: Open Folder).

If you prefer terminal editors, `vim` comes installed in the container.

6. Code Structure

Listing 1 provides an overview of the repository structure:



Listing 1: Code structure

- **Gillian-Rust:** Contains the backend symbolic execution engine based on Gillian. The symbolic folder houses the symbolic state model, as described in the paper. Key files include:
 - `memory.ml`: Entry point for the state model.
 - `heap.ml`: Contains the implementation of the heap component (Section 3).
 - `prophecies.ml`: Contains the implementation of the prophecy context (Section 5.3).
- **creusillian & gilologic_proc:** Contain macros for writing specifications and proofs. In particular, `creusillian` includes hybrid macros that desugar into either Gillian-Rust or Creusot macros, depending on the context.
- **lib_rust_to_gil & rust_to_gil:** `lib_rust_to_gil` is the main crate for the Rust-to-GIL compiler, while `rust_to_gil` is a simple binary that calls `lib_rust_to_gil`.
- **gillian:** Provides utilities for constructing and pretty-printing GIL ASTs in Rust, used by the compiler.
- **cargo-gillian:** A utility for running `cargo gillian`, which compiles Rust code to GIL and runs the proof engine on a given crate.
- **misc:** Contains files from the Rust standard library and the RefinedRust development repository. For Rust standard library files, the exact commit hash from which they were extracted is recorded at the top of each file. These files can be compared with `list_std[_proph].rs`, `vec_std[_proph].rs`, and `minivec.rs` in `tests/{noproph,proph}` to verify our modifications.

- `test_crates`: Includes sort and proven, the crates used in Creusot proofs, as well as `tutorial`, see Section 9.
- `tests`: Contains all Gillian-Rust case studies, split into:
 - `noproph`: Type Safety (TS) case studies.
 - `proph`: Functional Correctness (FC) case studies.

7. Understanding the OCaml State Model: Prophecy Context

For the interested reader, we explain the prophecy context—a simple component of the state model.

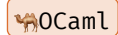
7.1. Structure Definition

In the paper, the prophecy context (χ) maps prophecy variables ($x \in PcyVar$) to a triple $(Expr, \mathbb{B}, \mathbb{B})$, where $Expr$ is a symbolic expression and the booleans indicate respectively whether the observer and the controller are present.

In OCaml (see `prophecies.ml`), the structure is defined as:

```
type prophecy = {
  value : Expr.t;      (* Future value (see below) *)
  obs_ctl: Expr.t;
  observer : bool;
  controller : bool;
}

type t = prophecy MemMap.t
```



`MemMap.t` is an alias for `Map.Make(String).t`, mapping strings to prophecy records. The paper's triple corresponds to `(obs_ctl, observer, controller)` in the code, and the extra field `value` represents the “future” of the prophecy variable. This field is used to store the value corresponding to the notation $\uparrow x$ in the paper and `RustHornBelt`. Because Gillian does not support defining pure functions, the future value is stored in the prophecy context.

7.2. Value Observer Producer

The paper describes two rules for producing a value observer—one with and one without a controller:

$$\frac{\chi(x) = (a', \perp, \top) \quad \chi' = \chi[x \mapsto (a', \top, \top)]}{(\chi, \pi).prod_{VO}(x, a) \rightsquigarrow (\chi', \pi \wedge (a = a'))} \text{VOBS-PRODUCE-WITH-CONTROLLER}$$

$$\frac{x \neg \in \text{dom}(\chi) \quad \chi' = \chi[x \mapsto (a, \top, \perp)]}{(\chi, \pi).prod_{VO}(x, a) \rightsquigarrow (\chi', \pi)} \text{VOBS-PRODUCE-WITHOUT-CONTROLLER}$$

In `prophecies.ml` the producer for the value observer is implemented as in Listing 2

```

let prod_value_obs pcy_env pcy_id obs_value =
  let+ new_block =
    match MemMap.find_opt pcy_id pcy_env with
    | Some { observer = true; _ } → Delayed.vanish () (* Duplicated resource *)
    | Some { value; obs_ctl; observer = false; controller = true } →
      (* We already have a controller, we learn equality *)
      let learned = [ Formula.Infix.( #= ) obs_value obs_ctl ] in
      Delayed.return ~learned
      { value; observer = true; controller = true; obs_ctl }
    | Some { value; observer = false; controller = false; obs_ctl = _ } →
      (* We have neither the controller nor the observer,
         the obs_ctl value is irrelevant *)
      Delayed.return
      { value; observer = true; controller = false; obs_ctl = obs_value }
    | None →
      Delayed.return
      {
        value = Expr.LVar (LVar.alloc ());
        obs_ctl = obs_value;
        observer = true;
        controller = false;
      }
  in
  MemMap.add pcy_id new_block pcy_env

```

Listing 2: Producer for the value observer

Gillian provides a monadic interface for symbolic execution (named `Delayed`), which allows us to define the rules without explicitly manipulating the path condition π , reducing the likelihood of errors.

The function computes a new prophecy quadruple, stored in `new_block`, for the given prophecy variable `pcy_id`. The lookup `MemMap.find_opt pcy_id pcy_env` retrieves the current prophecy block, leading to four possible cases:

- **Observer already exists:** If the prophecy variable already has an observer, execution vanishes (`assume(false)`). This is because producing the observer $VO_x(a)$ in a context where $VO_x(b)$ already exists leads to the separation logic assertion $VO_x(a) * VO_x(b)$, which implies `False` (since the observer is an exclusive resource).
- **Controller exists, but no observer:** We apply rule `VOBS-PRODUCE-WITH-CONTROLLER`. The new block is identical to the existing one, except that observer is now set to true. Additionally, we “learn” (i.e., add to the path condition) that the observer’s value must be equal to the controller’s existing value. This is done using the optional `learned` argument of `Delayed.return`, which takes a list of formulas to add to the path condition.
- **No observer, no controller:** We follow rule `VOBS-PRODUCE-WITHOUT-CONTROLLER`. The new block is identical to the existing one, except observer is set to true. The existing `obs_ctl` value is irrelevant in this case. This case does not exist in the paper, it is due to the future value of the prophecy variable which can be set with neither an observer nor a controller. However, it is analogous to the $x \notin \text{dom}(\chi)$ case.
- **Prophecy variable is new (not found in MemMap):** We apply rule `VOBS-PRODUCE-WITHOUT-CONTROLLER`. A new prophecy block is created with `observer = true` and the given observer value.

Since we need to define a future value for the prophecy variable, we allocate a fresh logical variable using `LVar.alloc`.

8. Reading the Gilogic Library

The gilogic crate (called Gilsonite in the paper) provides the macros and traits necessary to write specifications and proofs. We highlight two selected components: the ownership predicate for `Option<T>` and the lemma implementing the `MUTREF-RESOLVE` rule.

8.1. Ownership Predicate for `Option<T>`

The ownership predicate for `Option<T>` is defined directly in Rust, in the `gilogic/src/prophecies.rs` file as:

```
impl<T: Ownable> Ownable for Option<T> {
    type RepresentationTy = Option<T::RepresentationTy>;

    #[predicate]
    fn own(self, model: Option<T::RepresentationTy>) {
        assertion!((self == None) * (model == None));
        assertion!(|ax: T, repr: _| (self == Some(ax))
            * <T as Ownable>::own(ax, repr)
            * (model == Some(repr)))
    }
}
```

For `Option<T>` to have an ownership predicate, its inner type `T` must also have one—that is, `T` must implement the `Ownable` trait. Additionally, the pure mathematical representation of `Option<T>` is simply `Option<T::RepresentationTy>`, where `T::RepresentationTy` is the representation of `T`.

The `own` predicate takes two parameters:

1. The actual Rust value (`self`), an `Option<T>`.
2. Its abstract representation (`model`), an `Option<T::RepresentationTy>`.

The predicate defines ownership in two cases (separated by a semicolon, corresponding to a disjunction):

- **Case 1: None** If the value is `None`, then the representation must also be `None`.
- **Case 2: Some(ax)** If the value is `Some(ax)`, then:
 - The ownership predicate of `T` (`<T as Ownable>::own(ax, repr)`) must hold, meaning `ax` must satisfy the ownership conditions for `T`.
 - The representation of the `Option<T>` value must be `Some(repr)`, where `repr` is the abstract representation of `ax`.

This formulation ensures that ownership tracking extends naturally from `T` to `Option<T>`.

8.2. MutRef-Resolve Rule

The paper details the `MUTREF-RESOLVE` rule for finalising mutable references in the presence of prophecy variables:

$$\frac{}{\llbracket \&_{\text{mut}}^{\kappa} T \rrbracket(p, (a, a')) \Rightarrow^* \langle a = a' \rangle} \text{MutRef-Resolve}$$

The code contains a slight generalisation, also located in `gilogic/src/prophecies.rs`, which works should any existentials be frozen:

```
#[specification(forall m, frozen.
    requires { T::mut_ref_own_frozen(p, m, frozen) }
    ensures { (m.0 == m.1) }
)]
#[gillian::timeless]
pub fn prophecy_resolve<A: Tuple, T: FrozenOwn<A> + Ownable>(p: &mut T) { ... }
```

In Gilsonite, dollar signs $\$...\$$ are used instead of angular brackets $\langle...\rangle$ to denote an observation. In addition, the `#[gillian::timeless]` attribute indicates that the specification does not require the lifetime of the mutable reference `p` to be alive in order to be valid (i.e. the compiler will not automatically add the corresponding lifetime token to the pre- and post-condition).

Furthermore, the `FrozenOwn<A>` trait indicates that variables of the type described by the tuple `A` can be frozen in the ownership predicate of a mutable reference of type `T` (see Appendix or comments of the `list_std.rs` file).

9. Tutorial: EvenInt

⚠ Warning

Gillian-Rust is a prototype and may produce cryptic error messages.

We now provide a tutorial for the reader to follow in order to learn the basics of using Gillian-Rust (inspired by a `RefinedRust` tutorial). The reader is encouraged to open the `test_crates/tutorial` folder in the docker image (we recommend doing so in VSCode, following the instructions in Section 5).

9.1. Specifying the ownership predicate

In the `tutorial/src/lib.rs` file, you will find some boilerplate code together with the definition of a simple `EvenInt` structure:

```
struct EvenInt {
    num: i32,
}
```

The structure comes with a test function which dereferences a null pointer if the field `num` is not even:

```
#[creusillian::show_safety]
pub fn test(&mut self) {
    if self.num % 2 != 0 {
        unsafe { *std::ptr::null::<i32>() };
    }
}
```

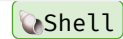
This function is only safe if the `num` field can **never** be odd. However, the ownership predicate currently does not enforce this, as it only asserts that the pure representation of the `EvenInt` be equal to its `num` field.

```
#[predicate]
fn own(self, model: i32) {
    // TODO
    assertion!((self == EvenInt { num: model }));
}
```

}

Try running Gillian-Rust on the crate:

```
$ cargo gillian --prophecies
Parsing and compiling...
Preprocessing...
Obtaining specs to verify...
Obtaining lemmas to verify...
Obtained 1 symbolic tests in total
Running symbolic tests: 0.004097
Verifying one spec of procedure EvenInt::test ...
FAILURE: Action load_value failed with: [MissingBlock: $l_null]
s f Failure
There were failures: 0.006235
```



There are two things to note here. First, at some point, loading a value from the heap fails, because Gillian-Rust correctly detects that the null location (denoted `$l_null`) should not be dereferenced. Second, the output indicates that two branches were taken, the first one was successful, but the second was not (“s f”). This is exactly what we expect, as the symbolic execution engine has explored both the branch where `self.num % 2 == 0` and where `self.num % 2 != 0`.

Info

To perform verification, Gillian-Rust first compiles the Rust code into an intermediate language called GIL. The compiled code can be found in the `tutorial/target/debug/tutorial-rlib.gil` file, should the reader be curious. Unfortunately, GIL is not the easiest language to read.

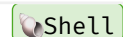
To fix this, one must correctly specify the ownership invariant of the `EvenInt` structure:

```
#[predicate]
fn own(self, model: i32) {
    assertion!((self == EvenInt { num: model }) * (model % 2 == 0));
}
```



After updating the predicate as above, running Gillian-Rust again should yield the following output:

```
$ cargo gillian --prophecies
...
Verifying one spec of procedure EvenInt::test ... s Success
...
```

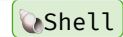


Note that, this time, only a single branch is explored. That is because our symbolic execution engine detects that, given that `model % 2 == 0`, the branch where `self.num % 2 != 0` is not feasible, and the corresponding branch is not explored.

9.2. Constructors

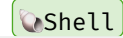
The `EvenInt` library defines two constructors: `new_unchecked` and `new`. The former is an unsafe function, as it transforms any value of type `i32` into an `EvenInt` without checking for the evenness of the value. You can check that Gillian is unable to prove the safety of `new_unchecked` by adding `#[creusillian::show_safety]` right above and running `cargo gillian --prophecies`:

```
Verifying one spec of procedure EvenInt::new_unchecked ... f Failure
```



On the other hand, adding `#[creusillian::show_safety]` on top of the new function, removing it from the `new_unchecked` function, and running Gillian-Rust yields:

```
Verifying one spec of procedure EvenInt::new ... s s Success
```



As expected, Gillian-Rust explores both branches: if `x` is even, then `new_unchecked` is called to create an `EvenInt` from `x`, and the `Some` variant of the option is returned, containing a **valid** value of type `EvenInt`; and otherwise `None` is returned, which is always a safe value of the `Option` type. Note that, here, since `new_unchecked` does not have a specification, it is symbolically executed by Gillian-Rust at the call site during the execution of `new`. Try, for instance, to return `EvenInt{ num: 1 }` in `new_unchecked` and re-running Gillian-Rust: the verification of `new` should now fail.

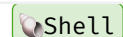
9.3. add_two

The `add_two` function is implemented by performing two successive calls to the `add` function, which unsafely increments the `num` field of the `EvenInt`. Our goal is to:

- Check that, even though the invariant of the `EvenInt` is broken in between the two calls to `add`, it is eventually re-established, ensuring the overall safety of `add_two`
- Check that `add_two` indeed adds 2 to the value.

First, try adding `#[creusillian::show_safety]` on top of the `add_two` function, and running Gillian-Rust using the `--log` flag:

```
$ cargo gillian --prophecies --log
```



```
Verifying one spec of procedure EvenInt::add_two ... f Failure
```

There are several issues with the current specification. First, `show_safety` checks that the function correctly executes without panics for any valid value of type `EvenInt`, and ends with the mutable reference pointing to a valid value of type `EvenInt`. Unfortunately, that is not the case, because `add_two` may overflow and panic. When run with `--log`, Gillian-Rust produces a (quite verbose) `file.log` file where the command was run. While it is difficult to navigate, an experienced user will eventually find the following mention:

```
Errors:
```



```
EFailReached(ASSERT, "overflow check failed")
```

To circumvent this, the user may specify as a pre-condition that the *representation* of the *current value* of the input mutable reference must be at most the representation of `i32::MAX` minus 2:

```
#[creusillian::requires((*self).@ < i32::MAX@ - 2)]
```



```
#[creusillian::ensures(true)]
```

```
pub fn add_two(&mut self) {
```

Unfortunately, that is still insufficient. Navigating through the log file will reveal that the post-condition could not be successfully unified. That is because the representation of the value has changed and, as described in the paper, “closing the borrow” requires updating the observer and controller accordingly, using the `MUT-AUTO-UPDATE` rule (Section 5.3).

The Gillian-Rust API provides a useful macro which performs all the “mutable reference cleanup” tactics that are useful when finalising a mutable reference: `mutref_auto_resolve!`:

```
#[creusillian::requires((*self).@ < i32::MAX@ - 2)]
```



```
#[creusillian::ensures(true)]
pub fn add_two(&mut self) {
    self.num;

    unsafe {
        self.add();
        self.add();
    }
    mutref_auto_resolve!(self);
}
```

When running Gillian-Rust, this should successfully verify! We can now verify that `add_two` indeed increments the value twice. In Pearlite (Creusot’s specification language, which we use here), one must specify that the value of the mutable reference *at the time it expires* (i.e. at the end of the function) is the *initial value* of the mutable reference incremented twice. To do so, we use the prophecy access operator “^”.

```
#[creusillian::requires((*self)@ < i32::MAX@ - 2)]
#[creusillian::ensures(^self)@ == (*self)@ + 2)]
pub fn add_two(&mut self) {
```



The reader may also try to write an erroneous post-condition (such as `(^self)@ = (*self)@ + 3`) and check that Gillian-Rust correctly fails to verify.

9.4. Exercise

The reader is invited to verify the `add_even` function as an exercise, by adding a pre- and post-condition, and required tactics in the body of the function.