Gillian Student Lab 2024

Welcome to the Gillian student lab! In this lab, you'll be using Gillian and its debugger to add missing proof tactics to WISL programs. Gillian is a parametric platform for program verification that can be instantiated to different languages; WISL is one such instantiation for a simple while language, like you've been using in the course.

Getting started

This lab has been designed to run on the lab machines. If you haven't already, copy the lab files to your home directory (or clone from https://github.com/GillianPlatform/gillian-lab) and open it in VSCode:

```
$ cp -r /vol/rr/gillian-lab ~/
$ code ~/gillian-lab
```

VSCode should suggest an extension to install; please install it. If it doesn't, search for the "Gillian Debugger" extension on the marketplace. The configuration in this repo should be enough for it to work out of the box on the lab machines. If it doesn't, please tell us 🙂

You are provided with 2 files: `SLL.wisl` and `DLL.wisl`, which contain lemmas and functions. The goal is to use WISL and its debugger to insert the right proof tactics for every function to successfully verify. Note that in practice, Gillian can infer a lot of these annotations on its own; for this lab, we disable this by using Gillian's "manual" mode. For every while loop, the invariant is already provided.

Note that you can quickly return to a point in the execution after modifying the program by setting a breakpoint on the relevant line, restarting the debugger (↻), and clicking the Continue button (⏵).

Please tell us if you encounter an error and you do not know what to do with it, or if the debugger crashes – you're some of the first people to use this. We had fun organising this lab, and hope you'll have fun too! We'd also hugely appreciate it if you could give us some feedback on the form provided.

Proof tactics

Folding & unfolding predicates

As you saw in the example, you can fold a predicate with:

```
[[ fold pred_name(param1, ... paramN) ]]
```

and similarly unfold a predicate with:

```
[[ unfold pred_name(param1, ... paramN) ]]
```

Applying lemmas

A number of lemmas are provided for you, and you'll need to apply them similarly to how you have in proof sketches; you can apply them like so:

```
[[ apply lemma_name(param1, ... paramN) ]]
```

(I'm double-sided)

Proof tactics (continued)

Logical assertions

You can assert a logical condition with:

```
 [[ assert someCondition ]]
```

Some proofs will require you use `assert` to bind variables. For example, let's imagine that you need to apply a lemma, and one of the parameters is the value contained in a cell at the address in variable `t`, i.e. your state contains `t -> ?`, and you want to apply `some_lemma(t, ?)`. The problem is, you do not have any program or logical variable available that contains the right value to use as second parameter. One solution would be to use a program variable:

```
 v := [t];
 [[ apply some_lemma(t, v) ]]
```

However, modifying the program for the sake of the proof is against the spirit of things! That's when `assert {bind: ..}` comes in:

```
 [[ assert {bind: #v} (t -> #v) ]];
 [[ apply some_lemma(t, #v) ]]
```

Conditionally applying tactics

You can use if/else in a logical context, like so:

```
 [[ if (condition) { .. } { .. } ]]
```

Loop invariants

Loop invariants are provided for you where necessary. They are declared like so:

```
 [[ invariant {bind: #x, #y} P(#x, #y) ]]
```

Common issues

Syntax

The syntax of WISL can be a bit tricky:

- Put everything in parentheses! Operator precedence may be unpredictable.
- There is a semi-colon BETWEEN each command inside a block, but not at the end (e.g. the last statement in an if-else block)
- Logical commands are surrounded by `[[ .. ]]`.

Automatic unfolding in preconditions

Even in manual mode, Gillian will automatically unfold any predicate in the precondition of a function if it is not recursive.

In particular, the `dlist` predicate gets unfolded into its `dlseg` definition automatically.

Folding a list with one element

You may have trouble trying to fold `SLL` with a single value, i.e. `SLL(x, [v])`. This can go wrong because Gillian can't find the base case, `SLL(null, [])` in your predicate state. Since the base case doesn't require any resources from your state, you're free to fold it from nothing, like so:

```
[[ fold SLL(null, []) ]]
```

(I'm double-sided)