Gillian Student Lab 2023

Introduction

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 :)

General instructions

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 annotations for every function to succesfully verify. Note that for this lab, we use Gillian's "manual" mode; in practice, Gillian can infer a lot of these annotations on its own. For every while loop, the invariant is already provided.

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!

Logical commands

- `unfold pred_name(param1, ... paramN)`
- `fold pred_name(param1, ... paramN)`
- `apply lemma_name(param1, ... paramN)`
- `if (bool_exp) { .. } else { .. }`
- `assert {bind: #x, #y, #z} P(#x, #y, #z)`
- invariants: they are written for you.

Some proofs will require you to bind variables with the `assert` statement. 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) ]]
```

(I'm double-sided)

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, []) ]]
```

Feedback

We'd hugely appreciate it if you'd complete a quick feedback form at the end of the lab. Find it by following the link or scanning the QR code below:

https://forms.gle/ZcHpM1MCqXogStPs9

(I'm double-sided)