PLC: Homework 5 [105 points out of 100]

Due date: Wednesday, April 17th
3 extra-credit points if you turn in by Tuesday, April 16th

About This Homework

For this homework, you will explore internal verification in Agda. You will also make some choices about the final project (though this will be posted a little after the posting date of April 3rd).

How to Turn In Your Solution

You should create a `hw5` subdirectory in your github.uiowa.edu repo. You will copy files from subdirectories of the `hw5` directory in the course repo.

Partners Allowed

You may work by yourself or with one partner (no more). See the instructions from `hw1` for details on how to submit your assignment if you work with a partner.

How To Get Help

You can post questions in the `hw5` section on Piazza.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the Resources tab of the Resources page on Piazza, for the locations and times for office hours.

1 Reading

Read Chapter 5 of Verified Functional Programming in Agda, available for free (on campus or VPN) here:

`https://dl.acm.org/citation.cfm?id=2841316`

2 Basics of internal verification [21 points]

Fill in the holes in `vector-todo.agda`. The functions there should behave just like Haskell's `init`, `last`, and `take` functions, respectively. These are 7 points each.

3 Type-preserving compilation [70 points]

In several steps you will perform some compiler transformation and prove in Agda that they are type-preserving (so an expression of type `T` is transformed to one of the same type).

The first step is to convert untyped expressions (`RawExpr` in `expr.agda`) into typed ones. These typed expressions are the ones we looked at in class April 4th. `RawExpr`s that are actually not typable cannot be converted to typed `Expr`s. So writing this conversion from untyped to typed is effectively writing a type checker for `RawExpr`s. A real type checker needs to produce error messages when typing fails, but here we will not attempt to do that.

1. In order to write such a type checker, it turns out a simple lemma about testing equality on values of (Agda) type `Tp` is needed. So fill in the first hole in `compiler.agda`. [10 points]
2. The type checker itself is `tpCheck` in `compiler.agda`. It takes a `RawExpr` and produces a Σ-type (see Section 5.3 of the book) containing a `Tp T` and an `Expr T` (that is, an expression whose object-language type is `T`). Fill in the code for this function. I have started one case for you using `with`, to give an idea of one way of writing the code which I found worked out well.

   When your code is working the `confirm` example in `compiler.agda` will type-check (in Agda) without any yellow highlighting.

   [25 points]
3. Next, write a function called `toIteExpr` that will transform an `Expr T` to a semantically equivalent `IteExpr T`. The idea is that the `IteExpr` type imposes some constraints on the form of the expression:
   - Additions are forced to be right associative by the `AddNum` constructor, which requires the first argument to be a number (and hence the first argument cannot be an addition expression itself).
   - All if-then-else expressions must appear at the top of the expression, not under an addition or a less-than expression.

These constraints are enforced through the types of the constructors of the `IteExpr T` type, the `LtExpr T` type, and the `AddExpr` type.

I found that to write this function I had to define several interesting helper functions. For example, I wrote a helper `addIte` with typing:

```
addIte : IteExpr TpNat → IteExpr TpNat → IteExpr TpNat
```

This function takes two `IteExpr TpNat` expression and forms the `IteExpr TpNat` expression representing their addition. This requires pulling any if-then-else's in the first or second summand out to the top. Please implement your solution to the overall `toIteExpr` function (and likely this helper) so that you pull if-then-else's from the left summand to the very top, above those in the right summand. (So you should resolve the ambiguity of whether to pull if-then-else's out of the first or second arguments of additions and less-than expressions in favor of the first argument.)

You can (and should!) test your `toIteExpr` function using some of the testcases in `expr.agda`, and possibly others. The problem is a bit tricky, but made much easier by the use of well-typed expressions, as the typing of the expressions helps prevent many mistakes that would be hard to catch in testing.

[35 points]

4 Choices for final project [14 points]

Clone the `https://github.uiowa.edu/astump/GraMA` for the GraMA final whole-class project, which we discussed in class April 11th. We will code in Agda except for the parser. Use the same process to clone the GraMA repo as we used at the start of the class for cloning the class repo. The difference is that now the whole class has write access to this GraMA repo!

Please add a file to the `teams` subdirectory for your team (either just yourself or you and one partner). The file should be named either just your hawkid if you will work by yourself, or in the format "hawkid-hawkid" where the two Hawkids are for the two partners (like "zcook-aboss", for example).

In this file, please list:

- Your team name (it cannot be rude!)
- The names of the team members (just so we don't have to look up from Hawkid)
- Your preferences, in order from most to least preferred (you can omit some choices, too), among the following parts of the project to work on:
  - GraMA parser (to be written in Haskell and interfaced to Agda) and printer
  - GraMA type checker
  - GraMA evaluator
  - printing a sequence of graphs to Graphviz format

    – sample graph-manipulating programs written in GraMA

As discussed in class, the only sensible way for us to try to tackle this as an entire-class project is for me to provide some critical interfaces. The plan is that I will craft datatypes in Agda for raw abstract syntax trees (ASTs) for GraMA programs, as well as typed ASTs. I will probably also provide a definition in Agda for graphs. Then the parser will target raw ASTs, the type checker will convert raw ASTs to typed ones, the printer and evaluator can both use typed ASTs, and the evaluator and printer to Graphviz will both use the graph datatype. The sample GraMA programs will initially be written as typed ASTs.

On Wednesday, April 17, I will release these interfaces and give further guidance for the above parts, along with assignments of teams (multiple teams will likely be assigned to each part).