EECS 662

Programming Languages

Project 1 - Predicting Failure

Mini Project 1 - Predicting Failure
EECS 662 - Programming Languages

The first objective of this miniproject is to develop your first type checker. You will start with the ABE language presented in class and develop techniques for predicting failure. The second objective is developing a trivial optimizer that removes constants in `+` and `if` expressions.

To aid in your quest, the file p1.hs implements the Haskell `ABE` and `TABE` data types and function signatures. If you would like to play with a parser, p1-parser.hs adds a parser from strings to the `ABE` data type. You do not need the parser for this project. It is included if you find the concrete syntax easier to read and to give you another example using Parsec.

Exercise 1

Write an interpreter for the ABE language discussed in class and presented in our text extended to include multiplication and division.

```
ABE ::= number | boolean
      ABE + ABE |
      ABE - ABE |
      ABE * ABE |
      ABE / ABE |
      ABE && ABE |
      ABE <= ABE |
      isZero ABE |
      if ABE then ABE else ABE
```

1. Write a function, `evalM :: ABE -> (Maybe ABE)`, that takes a ABE data structure and interprets it and uses `Maybe` to return an ABE value or `Nothing`. Your `eval` function should only check for divide-by-zero errors and negative numbers at run-time.
2. Write a function, `evalErr :: ABE -> (Maybe ABE)`, that takes a ABE data structure and interprets it and uses `Maybe` to return an ABE value or `Nothing`. Your `evalErr` function should do full run-time type checking in addition to divide-by-zero and negative number checking.
3. Write a function, `typeofM :: ABE -> (Maybe TABE)`, that returns either `Nothing` representing an error or an `TABE` value representing a type.
4. Write a function, `evalTypeM` that combines `typeM` and `evalM` into a single operation that type checks and evaluates an ABE expression. Take advantage of the `Maybe` monad to ensure `evalM` is not called when `typeofM` fails.

Exercise 2

And now, something completely different. Remembering that programs are just data structures, write a new function called `optimize :: ABE -> ABE` that does two things:

1. If the expression *x* `+ 0` appears in an expression, replace it with *x*.
2. If the expression `if true then` *x* `else` *y* appears in an expression, replace it with *x*. Similarly for `false` and *y*.
3. Write a new function, `interpOptM::ABE -> Maybe ABE`, that integrates your new `optimize` into your `ABE` interpreter by calling it right before `evalM`.

Do not make this harder than it is. The optimizer replaces literal `0` in addition and literal `true` and `false` in the `if`. That's all. Nothing else.

Notes

Most if not all the code for the ABE `evalM` and `typeofM` functions can be found in our text. Again, I would encourage you to try to write as much of them as possible without referring to the textbook examples. The `optimize` function is a chance for you to stretch a bit. There are resources in the text and online that will help you if you don't see a solution immediately. Recursion is your friend and the pattern we use for interpreters is a useful thing.

To give you an idea of the effort required for this mini-project, my code is about 150 lines long and took me roughly an hour to write and debug. I view this as a reasonably easy project at this point in the semester. Do not put it off as many of you are still becoming friends with Haskell. Hopefully the previous project shook out any difficulty with Haskell tools.