Final Project

Part 1: A Prime Sieve Formulated as a Lazy Stream

Due: 11:59pm Wednesday, Dec 16, 2020

Overview

Write a (functional) Java program that constructs `primes`, the lazy inifinite stream of prime numbers 2, 3, 5, 7, ... where numbers are represented using Java type `int`. Ignore the fact that the `int` type only supports integer values less than 2^31. We will only compute the first few thousand primes, so no one will notice. Obviously such a program could easily be generalized to use type `long` or the unbounded integer type `java.math.BigInteger`. The support file `primeSieve.java` provides the interface `IntStream` which is the root of a composite class hierarchy supporting lazy streams of `int`, including methods to print finite ones in conventional Lisp-like notation. The formulation of streams supported in Java starting with Java 8 is *not* functional. Simple operations like extracting the first element of a stream or the length of a finite stream destroy the stream. Consequently, we must develop our own `IntStream` library from scratch. Fortunately, it is not very difficult and requires comparatively little code. The equivalent program in Haskell (generalized to unbounded integers) is simply

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Of course, Haskell has built-in support for lazy streams and the recursive definition of functions (like `filterprime`) using pattern matching all supported by an aggressive optimizing compiler. Your task is to extend the provided code to support a static final field called primes in the top-level interface IntStream bound to the lazy infinite stream of primes (ignoring the fact that `int` arithmetic will overflow when numbers get too large. You also need to augment the JUnit test file (compatible with JUnit 4 as provided by DrJava) `IntStreamTest.java` to further test your code. You may assume that all of the provided code is correct. [Please tell us if you discover any bugs!] You do not need to test any of the methods in the `IntStream` interface provided by the original version of `primeSieve.java`.

If the course staff makes any enhancements to the support code files `primeSieve.java` and `PrimeSieveTest.java`, we will post messages to that effect to Piazza.

Complications

Java is strictly a call-by-value language and lacks macros, so we are going to have to wrap the stream argument in a "cons" construction in a suspension (an object with a method to evaluate it and a cell to store that value). This will make translating the simple Haskell code above more cumbersome. There is a price to writing functional code in Java, a language not intended to support lazy evaluation. In Racket/Scheme, we would make the stream cons operation a macro that automatically performs the wrapping.

Submisison

Inside the svn directory `https://svn.rice.edu/r/comp311/turnin/F20/<netid>/`, create a directory `JavaProject` and put your modified files `primeSieve.java` and PrimeSieveTest.java in that folder.

Part 2: A Boolean Simplifier Written In Haskell

Your task is to reimplement the Boolean Simplifier in Haskell which is very easy if you are comfortable with the core constructs of Haskell. We are providing support code to handle all IO. You only have to use the API provided by the support code. All of the support code including test files for this project is located in the Rice SVN repository at the URL https://svn.rice.edu/r/comp311/course/HaskellProject/.

First you need to review the description of Homework 5 for to recall the high-level steps in the simplifier. Your Haskell code will use types very similar to the corresponding Java program, but they are slightly more detailed and their explication and usage is simpler than the corresponding Java class hierarchies.

Type Definitions and Operations

All of the critical type definitions that you will use are defined in the support file `Types.hs` in the SVN repository. In `Types.hs`, the type `SimpleExpr` is either just a Boolean Constant, *i.e*, `True` or `False` or an identifier (a string). The type `BoolExpr` consists of more complex boolean expressions which are trees with leaves that are of type `SimpleExpr` with intermediate nodes (constructors) `BAnd`, `BOr`, `BImplies`, `BNot` and `BIf`. The first three are binary (two children), `BNot` is unary, and `BIf` is ternary as you would expect. You must wrap a `SimpleExpr` in the constructor `BLeaf` to make it a value of type `BoolExpr`. Recall that in Haskell every value has a unique basic type (akin to a Java class).

The type `IfExpr`, which is disjoint from `BoolExpr` (just as `IfForm` is disjoint from `Form` in the corresponding Java program) is either a `SimpleExpr` (wrapped in the constructor `ILeaf`) or an `IIf` which is an if-expression whose subexpressions are if expressions (akin to `IfIf` in the corresponding Java program). There is a third type `NExpr` corresponding to normalized if-expressions. The Java program does not introduce this third type, but it could have done so at the cost of introducing more class definitions. Haskell is so much lighter weight that the extra cost is negligible. Recall that a normalized if-expression can only have literals in the test position.

The simplifier itself consists of same parts as the coresponding Java program. A skeleton for the simplifier appears in `Simplifier.hs`. You only need to fill in the missing parts of this file. These are the main steps of the simplification:

- toIf :: BoolExpr -> IfExpr. It is called convert-to-If/convertToIf in HW05/HW08.
- normalize :: IfExpr -> NExpr.
- eval :: Env -> NExpr -> NExpr. The type Env = [(String, Bool)]. You may want to use the function lookup in the Haskell Prelude.
- toBool :: NExpr -> BoolExpr. It is called convert-to-bool/convertToBool in HW05/HW08.
- reduce :: BoolExpr -> BoolExpr. This function is simply the composition of the preceding four functions. You also need to implement the auxiliary (help) function headNormalize :: NExpr -> NExpr -> NExpr -> NExpr.

Testing

Tests are defined in the file `Tests.hs`. Add some simple tests for `eval` and `toBool`. Add your tests to the unitTests definition as well. Consider adding extra tests for the other functions as you see fit. We are providing a parser `String -> BoolExpr` called `parseBoolExpr` defined in the file `Parser.hs`. There is also an unparser in the file `UnParser.hs`. We have setup some machinery in `Main.hs` so that you can use the parser to feed input to the simplifier directly. To run this, simply run the main action, either using ghci, or compile it with ghc and then run it. Ask Agnishom if you need help with this.

Development

The assignment is self-contained in the sense that we do not require you to install any libraries beyond what is already available in the base package. So, it should be possible to run everything on https://repl.it/. Corky is going to do the assignment on his laptop using Haskell Platform. He will post messages on Piazza stating what tools he is using.

In the process of your development, you may, by mistake, include an infinite loop which builds a very large expression in memory. Haskell programs can be very fast and this could very quickly consume a lot of memory freezing your computer. This is sometimes a frustrating part of the development experience. If everything is done right, however, the simplifier should be able to simplify the expression in bigData1 in under a second.

General Instructions

Annotate all your top level definitions with types. Do not import any libraries. Ask the course staff if you need exemption from this policy. Do not rename or change the type of any function that has already been provided to you.

Submission Instructions

Inside the svn directory `https://svn.rice.edu/r/comp311/turnin/F20/netid/`, create a directory `HaskellProject` and put the modified files for `Simplifier.hs` and `Tests.hs`. For example, since Agnishom's netid is ac132, he would create `https://svn.rice.edu/r/comp311/turnin/F20/ac132/HaskellProject/` and put his files there. You should not need to upload any other files, since these are the only two files you will need to modify. If you do need to include anything else; please ask course staff if you have any questions in this regard. Do not create any subfolders. Please use the exact path given here.

Honor Code

The final project is conducted under the same honor code as our programming assignments. You may ask the staff questions and/or post questions on Piazza about the conceptual issues in the projects but no sharing of code with other people (students and non-students) or copying of code from any source, notably reference books and the internet, is permitted. In general, we prefer that you ask questions on Piazza so that all students in the class can see the answer.