Bex Golovanov
Project Proposal

For my project, I would like to build a library which rigorously defines and constructs basic ring/group theory. There is a much more expansive and likely better library already in existence called Numeric.Prelude, which includes almost all mathematical structures and provides a type hierarchy. My goal is not to replicate this package, but rather to provide a basis to demonstrate/prove some interesting algebraic results, primarily focused around deriving Galois theory/correspondence and using the mobius function to count irreducible polynomials in a finite field. Time permitting, I would want to do some work with gaussian integers, but I think this rigorous construction may already be pretty ambitious. I don't plan on using/importing any aspects of Numeric.Prelude, though I'm sure some type structure will overlap by definition of mathematics. Moreso just wanted to acknowledge that something similar to this already exists (unsurprisingly I was not the first person ever to see the potential for coding algebra into Haskell).

The "easy" part will be the basic foundational definitions and corresponding type hierarchy: things like groups, rings, fields, etc. As far as defining homomorphisms and isomorphisms, I expect to be creating lots of Functor instances since these morphisms map structures to structures. Technically it's not the mathematical definition of functor but go figure, people have already coded up some algebraic topology as well. Using Functor, Applicative, and Monad will definitely be useful for my goals given the inherently abstract nature of what I want to do. I don't want the group definition to be a member of Num because groups don't have to correspond to "numbers". When creating my new types, I'll be defining them as both instances of Num and whatever container I select for group—I may have to look into defining my own type classes as well. Numeric.Prelude will definitely help with guidance in this area, should I need it. It is also possible this level of abstraction may require some GADTs, but I won't know that for sure until I get started.

The "medium/hard" part will be the math of it all. Defining Galois groups and field extensions is already pretty nontrivial without trying to program it, so it will require a fair amount of thought and reorganizing theorems into language Haskell can understand (some functor f "wrapping" math in Haskell). Having taken Algebra at this school and learned Galois theory, I will just need to refresh my knowledge, but this exterior component will be fairly manageable/within the scope/time constraints of this project. More ambitious is the goal of

presenting the application of something like the Mobius function, so I will need to study that separately to make sure I understand it enough to code it within the type hierarchy I create.

Another avenue I can take if the mobius function proves impractical is to use the foundation I create to reconstruct the real numbers from an algebraic perspective. Truthfully, the field of abstract algebra is incredibly vast, so I could flip to a random page of my old textbook and try to program that, I just happen to like the mobius function. In an entirely different direction, I can use the algebra foundation to define stabilizer codes from quantum error correction, since those are constructed from group theory. I could program an elementary instance of the [[7,1,3]] or [[17,1,5]] codes, and demonstrate how they work in encoding and decoding qubits.

My inspiration for this project is the algebra class I took in the spring, and one of the first classes this quarter where we learned that Haskell has theoretical constructions of data along with physical/bit-defined data types. I appreciate how the way Haskell goes about type definitions fits very neatly with mathematical definitions and I want to take advantage of that.