Finite Fields in Haskell

Rebecca Golovanov

December 15, 2023

Contents

1 Categories and Types

This section corresponds to the `Fields.hs` file.

The highest level definition of algebraic types we need is as follows:

Definition 1.1. A **group** G, endowed with some operation $\circ$, has the following:

1. an identity e, such that $g \circ e = e \circ g = g$ for any $g \in G$
2. an inverse for each group element, such that $g \circ g^{-1} = g^{-1} \circ g = e$

Note: commutativity is not required

Definition 1.2. A **ring** R is endowed with $+$ and $\times$, with the following properties:

1. R is a group under $+$
2. R is commutative under $+$
3. Distributive property: $a(b + c) = a * b + a * c$ for $a, b, c \in R$

Definition 1.3. A **field** F is a ring such that

1. F is an abelian group under $*$, in addition to $+$
2. F has $0, 1$ and $0 \neq 1$, where 0 and 1 are the additive and multiplicative identities respectively

Note: when a set is a multiplicative group, this means division is well defined over that set. For example, $\mathbb{Z}$ is not a multiplicative group because you can't divide any integer by any integer to get an integer.

1.1 Groups

The **group** type should carry with it the following information : operation, operation inverse, operation identity. The issue is that one can't encode this into a typeclass construction, at least not without dependent typing, which I really did not want to get into.

Work around: create a data type called `GrpElem`, which contains information about operation, identity, and element. The inverse operations is either written as a definition of `<->` or `inverse`, depending on the context.

1.2 Fields; Combining Additive and Multiplicative Groups

A **field** is essentially the merging of an additive and multiplicative group, where given additive A, we can define multiplicative $M = A \setminus \{0\}$. When we have these two groups together, this means that addition,subtraction,multiplication, and most importantly division, are all well defined. This dual nature of each element is difficult to encode.

This information needs to be analyzed as early as the `<+>` operation, to understand how addition looks on elements of the multiplicative group. The opposite is needed (defining additive A given M) when defining `<**>` for the ring component of a field.

Work around: The strategy I settled on was based on a

```
case (lookup (sum/prod) (ref)) of
Just val  → val
Nothing  →  defaultVal
```

format. The field itself, some sort of list, would be composed of tuples (a, m) where a and m had different generators/group operations, but evaluated to the same value. Basically, an association between the additive and multiplicative groups.

Another tricky component is being able to define the multiplicative group, which comes down to being able to identify a generator for the group. (ex, 2 is a generator for the multiplicative group of $\mathbb{Z}/5\mathbb{Z}$: $\{2^1, 2^2, 2^3, 2^4\} \to \{2, 4, 3, 1\}$ (0 is not in the multiplicative group). This issue is addressed in Section 2.3.

Evolution: In my first iteration of code, I realized a lot of my functions were sort of "towing around" the base field which accompanied the values I wanted to manipulate. When I wanted to add some `(a1,m1),(a2,m2)` tuples, I would either have to pass in a base field every time, or generate it within the operation every time, neither of which were appealing options.

This issue of having to always pass in some data that I wasn't ever directly acting on lent itself very nicely toward a stateful function definition. This led to the construction of the `FieldElem {[a]→(a,[a])}` data type. I originally wanted to specify that this value a has to be an instance of **Field,** but this wasn't going to work for two main reasons.

1. In cases like tuples, where I needed a reference list to make the proper association, I was defining operations with respect to the state/base itself. The **Field** properties depended on the `FieldElem` type, and not the a I passed in. Of course in situations like rationals or reals I could define those operations in isolation, but in that case I would need to tow around this list in the first place.

2. As I learned from my first attempt to define the `Group` type class, you can't utilize any of the functor/applicative/monad functions Haskell has to offer if you need to specify types within your larger class. There is a path towards dependent types in that direction but I did not have time to go down it.

Not specifying types allowed me to take advantage of the monad framework and make `FieldElem` an instance of monad, applicative, and functor. I have to admit I definitely didn't take full advantage of everything this had to offer; there is definitely some going into the data itself instead of staying higher order, but it's still prettier code than it was originally. I think my prettiest two lines are the function for finding the multiplicative inverse of a field element:

```
minv ::  FieldElem [Tuple] Tuple  →  FieldElem [Tuple]
minv = (= << ) (fieldOp 2 inverse)
```

`fieldOp` is a `a → m a` function which, given an index specifying the fst or snd tuple element, some function `f::a → a`, and some field element, neatly performs $f(a)$ without touching the state of the field element. Even prettier here is that I was also able to eta-reduce the field element.

2 Finite Fields as Vector Spaces

This section corresponds to the `Vectors.hs` file.
Here is where we get into both meatier code and much, much meatier math.

2.1 Formalizing Finite Fields (Crash Course Attempt)

Definition 2.1. Let H be a subgroup of G. The **left cosets** (left/right doesn't really matter here, it's more important in noncommutative groups) of H in G is the set $\{gH : g \in G\}$. For example, the left cosets of $3\mathbb{Z} \subset \mathbb{Z}$ are $\{0+3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}\}$, where all elements in $3\mathbb{Z}$ "equal" 0.

Definition 2.2. Let S and R be rings. The **quotient ring** R/S, is the set of left cosets of S in R. This is why we write integers modulo some p as $\mathbb{Z}/p\mathbb{Z}$. Everything divisible by p is in the 0 coset, and the remainders are the left cosets.

We can also think of cosets as *equivalence classes,* where $g, f \in G$ satisfy some equivalence relation $\sim_H$ if they are in the same coset of G/H. Like how $4 = 7 = 1 \pmod 3$.

The *order* of some finite field is some integer p such that $x^p = x$ for all x in the finite field.

Definition 2.3. A **field extension** L/S, is a quotient ring except L is a field, and is sometimes referred to as the base field.

One example is the complex numbers, which are isomorphic to a field extension of the reals :

$$\mathbb{C} \simeq \frac{\mathbb{R}}{x^2+1} = \mathbb{R}(i)$$

In this example, this means there is some $\alpha \in \mathbb{C}$ such that $\alpha^2 + 1 = 0$, since we should have $x^2 + 1 = 0$ as it is in the 0 coset. The notation $\mathbb{R}(i)$ means we "added" i to the reals, so numbers in $\mathbb{R}(i)$ can be written as $a + bi$ with $a, b \in \mathbb{R}$.

Quotient-ing out by polynomials is the backbone of field extensions, since we can think of these field extensions as $L(\alpha, \beta, \ldots)$ where we just "add" the roots $(\alpha, \beta \ldots)$ of the polynomial to our base field.

The fundamental theorem of algebra states "every degree n polynomial with complex coefficients has n complex roots". This is applied in the finite field space by saying, essentially, given some degree n polynomial over some field $\mathbb{F}$, there is some field extension of $\mathbb{F}$ which contains all n roots.

So, the big theorem/classification formalizing finite fields:

Theorem 2.4. *The order of a finite field is a prime power. For every prime p there are fields of order $q = p^k$, and all fields of the same order are isomorphic. For every $x \in \mathbb{F}_q$, we have $x^q = x$, and $(x^q - x) = \prod_{a \in \mathbb{F}_q}(x - a)$.*

The last part is saying the field $\mathbb{F}_q$ has exactly q elements, and every $a \in \mathbb{F}_q$ is a root to the polynomial $x^q - x$. From here, there's a lot of interesting stuff to do with Galois theory and automorphisms/homomorphisms/pretty diagrams, but first there is the task of somehow, coding the construction/classification of finite fields into Haskell.

2.2 Vector Spaces in Haskell

Remember how I said we could write $z = a + bi$ for any $z \in \mathbb{R}(i)$? We can also think of z as a $2d$ vector $[a, b]$ over the reals. This approach is how I tried to formalize and generalize the construction of finite fields.

It was simple enough to turn a base field into some $k-$dimensional vector space (here's where `sequenceA` came in). I just needed to compute the $k-$dimensional basis and scale and sum across all combinations of coefficients of the base field. Defining addition/subtraction was easy since the addition/subtraction of vectors happens element-wise, so you don't have to worry (yet) about what the other dimensions *mean* besides being a symbolic extension.

How do we define these other dimensions? These come from the roots of the irreducible polynomial p we used to get to our extension. The irreducible part means that we can't factor p into anything while maintaining our given coefficients, which also means p should have no factors in our base field. For example, $x^2 + 1$ doesn't have any roots in $\mathbb{F}_3$ (where $\mathbb{F}_3$ is the finite field with 3 elements). Then we have a nice quadratic (by which I mean degree 2) extension, where the elements of $\mathbb{F}_{3^2} = \mathbb{F}_9$ are defined by $a + bi$, where $a, b \in \mathbb{F}_3$.

Another important point about irreducibility/polynomial extensions is that every element in the field extension which has a "new" root must have order exactly q, in this case 9. This means that for any $k < 9$, given some $a + bi$ where $b \neq 0$, we have $(a + bi)^k \neq a + bi$. The nonzero b part is important because the elements of $\mathbb{F}_9$ with $b = 0$ are in the base field $\mathbb{F}_3$, so they have order 3.

Once we've identified the proper roots/how they interact, we basically get division for free because the multiplicative inverse of any $a + bi \in \mathbb{F}_9$ is just $(a + bi)^8$. We have to define multiplication though, which is where knowing how the roots act comes in. For example,

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

However, we can't use i as the default quadratic root for any prime.

Consider $\mathbb{F}_5$. In this field, we have $-1 = 4$, so $\sqrt{-1} = 2$, meaning we can factor $x^2 + 1$ in $\mathbb{F}_5$, so this polynomial is not irreducible. For $\mathbb{F}_5$, we could use $x^2 - 2$, since there is no $\sqrt{2}$ that can be written as some element in $\mathbb{F}_5$.

I did not want to have to compute irreducible polynomials for each field to find my roots. The following is how I generalized quadratic field extensions, with framework in place for generalizing to higher degrees:

2.3 The Meat

Formally, $x^2 + 1$ factors in $\mathbb{F}_5$ because 2 is not a *quadratic residue*. As an example, the quadratic residues of 5 and 7 can be computed as follows:

$$1^2, 2^2, 3^2, 4^2 \to 1, 4, 1, 1$$
$$1^2, 2^2, 3^2, 4^2, 5^2, 6^2 \to 1, 4, 2, 2, 4, 1.$$

This means that in $\mathbb{F}_5$, 1 and -1 have a defined square root, and in $\mathbb{F}_7$, this is the case for $1, 2$, and 4. That means $2, 3$ and $3, 5, 6$ are fair game for $\mathbb{F}_5$ and $\mathbb{F}_7$, respectively.

A fun fact is that numbers which are not quadratic residues make perfect generators for multiplicative groups. I generalized my construction of the base field (given a prime p) by making a list of the residues, filtering them out from the original list, and picking the first non-residue to be the generator.

Another fun fact is that whatever number you pick to be your generator g gives you a free irreducible quadratic: $x^2 - g$. So, theoretically, given any prime p and a generator g, the quadratic extension looks like

$$a + b\sqrt{g}, a, b \in \mathbb{F}_p.$$

I didn't want to compute or write square roots in my code. In fact, that's the whole thing we try to avoid when we think of these field extensions as vector spaces. I needed a symbolic, yet functional notation for these new variables, so I created a new data type: `Var,` and more importantly, a subset of my `Fin` type, `Fin Var`.

The idea is as follows: we want `Fin Var` to carry information about the order of the field, with tools in place to specify how coefficients interact (like how i has order 4, but $i^2 = -1$, so it affects the base coefficient and the i component goes to 0). The goal with my method is to make it easier to generalize extensions past quadratic, which is why my `FVectors` data type is a list of (var,coefficient) tuples, to allow for as many vars as necessary. For example, a cubic extension requires two adjoined roots α, β, in its vector representation. When it gets down to the nitty gritty though, I haven't figured out how to generalize the behavior of n roots. I could probably just figure out which numbers aren't cubic residues, quartic residues, etc. and go from there but I'd rather find a higher-order solution, so to speak.

Notice that when we write vectors in this space $[a, b, c]$ in terms of their roots $a(\alpha) + b(\beta) + c(1)$, we get a polynomial in α and β. We can then think of $a + bi$ as a polynomial in i, and observe that the formula for multiplying complex numbers is really a simplification of

$$(a + bi)(c + di) = ac + (bc + ad)i + (bd)i^2,$$

where we've defined $i^2 := -1$.

So, defining multiplication for these higher order fields should look something like polynomial multiplication, with coefficients in the base field and variables as the new roots. Let's say we haven't come up with i as the new dimension in $\mathbb{F}_9$, so all we know is we have some new variable α which satisfies some irreducible quadratic in $\mathbb{F}_3$. Then computing, say, that m^4 for some $m \in \mathbb{F}_9$ looks like

$$\begin{aligned}(x + y\alpha)^4 &= x^4 + 4(x^3y\alpha + x(y\alpha)^3) + \binom{4}{2}(x^2(y\alpha^2)) + (y\alpha)^4 && (1)\\ &= x^2 + (xy\alpha + xy\alpha^3) + 6(x^2(y\alpha^2)) + y^2\alpha^4 && (2)\\ &= x^2 + xy(\alpha + \alpha^3) + y^2\alpha^4 && (3)\end{aligned}$$

The simplification in step (2) comes from $x, y \in \mathbb{F}_3$, so they have order 3, and the simplification in step (3) comes from coefficients in $\mathbb{F}_3$, so $6 \equiv 0 \in \mathbb{F}_9$. But this didn't simplify very nicely at all, because all we know about α is that we need it to be a quadratic root. Namely, we need $\alpha^2 \in \mathbb{F}_3$. We also need α to have order 9, so $\alpha^k \neq \alpha$ for any $k < 9$. We can't pick square roots randomly, since we saw in the case of $\mathbb{F}_5$ that we can't set $\alpha := i$ because $\sqrt{-1}$ already exists in $\mathbb{F}_5$.

When enacting polynomial multiplication over haskell, the computer doesn't know ahead of time how these new values simplify, so in an instance like this we would get a result of $[1, xy, 0, xy, 1]$, even though this is only a degree two extension, so the final result should be of the form $[a, b]$. If we use i, we can easily compute

$$1 + xyi + xyi^3 + 1 = 2 + xyi - xyi = 2 \to [2, 0]$$

But we don't know a priori whether we can use i or not, or really anything about our new variable (assuming we want to generalize to all degrees of extensions).

2.4 Simplifying Polynomials

Here is where we run into issues. How do we define this new `Fin Var` to act how we want, without computing irreducible polynomials or passing in a hardcoded value for each prime?

We've established that we would like our α to have order p^k, given some k vector space. In my code, we compute the whole polynomial without any simplification of α, and in a function aptly titled `simplify`, we utilize our definition of `Fin Var`.

I designed a reduction method, given some scalar s, some degree k, and tuple (`Fin Var`, `FieldElem`), as follows:

1. let (`Fin ord gen pow,f`) := (`Fin Var`, `FieldElem`)
2. $(q, r) \leftarrow$ `(pow 'div' k, pow 'mod' k)`
3. `return (Fin ord gen r,f <**> (s ^ q))`

The first part of the tuple just reduces us down to the degree of extension we want. The value r will be at most $k-1$, which guarantees a k vector (once we add like terms). The tricky part is the scalar.

Let's apply $\mathbb{F}_9$ to this to make sure it makes sense, because it doesn't explicitly define α to be i. The scalar we pass in is 2 (I'll explain why in a second), so whenever q is odd, we get $2^q = 2 \equiv -1 \pmod 3$. Whenever q is even, we get $2^q = 1 \pmod 3$. This scalar is acting like i^2, and determines the sign of the new value. The value of r lets us know whether or not i has a nonzero coefficient.

Extending this to any quadratic extension, we don't need to worry about signs, as long as we have our scalar defined. As I said earlier, this turns out to be pretty simple. None of the field generators are quadratic residues, so the polynomial $x^2 - g$ is irreducible in the base field, so we can just set our scalar to g. This is not fast. In fact, it gets very slow around $\mathbb{F}_{7^2} = \mathbb{F}_{49}$. However, this method means we are able to generate the quadratic extension of any prime field. In fact, and I haven't tested this, I think it means we can generate any power-of-2 extension (with a little fiddling). This is because my polynomial datatype is built to become totally general, so it can also take vectors as its values, and then we could multiply polynomials with vectors as the variable, and isn't that exciting!

3 Other Paths and Further Work

3.1 Generalized Degrees

As I said, I haven't yet come up with a way to generate new roots for degree-k extensions that doesn't rely on finding the cubic/quartic/etc non-residues. However, the time I spent defining what `Fin Var` means is essential for exploration into this area.

A simplified version of this project which I alluded to would be to only focus on quadratic extensions with no eye toward higher degrees. As I said, any prime $p \not\equiv 1 \pmod 4$ should be able to use i as its adjoining root. I'll briefly justify this. First I'll show why we can't use i for $p \equiv 1 \pmod 4$ Let g be a generator for $\mathbb{F}_p$. Then g has order p, meaning $g^{p-1} = 1$.

$$\begin{aligned} g^{p-1} &= 1 \\ g^{4k+1-1} = g^{4k} &= 1 \\ g^{4k} &\neq g^{2k} \neq g^k \\ \sqrt{g^{4k}} = g^{2k} &= -1 \\ \sqrt{g^2 k} = g^k &= \sqrt{-1} \end{aligned}$$

Thus, there is some element in $\mathbb{F}_p$ that, when squared, is equal to -1.Now let $p \equiv 3 \pmod 4$ (these cases

cover all primes except for 2)

$$\begin{aligned} g^{p-1} &= 1 \\ g^{4k+2} &= 1 \\ g^{4k+2} &\neq g^{2k+1} \\ \sqrt{g^{4k+2}} = g^{2k+1} &= -1 \end{aligned}$$

By definition, $2k+1$ is odd, so there is no square root of -1. For the primes that do have a square root of negative 1, we can go back to the generator trick. In fact, we can just use the generator trick for any quadratic (as we ended up doing), but it's nice that we already have a defined system for multiplying complex numbers, where we wouldn't have to deal with any of that simplifying stuff ourselves. I actually began with/computed this approach early on, but was unsatisfied with the lack of flexibility for all other cases.

3.2 Quotient Types

I spent a little bit of time trying to figure out how to encode quotient types, and what it means to be a coset or equivalence class in type theory. Apparently this is big in homotopy type theory, and I did not have time to learn all of that. However, I think that would definitely be a different promising approach to generalizing this area, since it would provide a more robust abstract framework and rely less on direct computation.

3.3 Galois Theory and Algebra

A big motivation to generalize degree extensions is to be able to talk about more interesting relations between different finite fields and their automorphism groups. I was able to do a little bit implicitly, since in my $\mathbb{F}_3, \mathbb{F}_5$, and $\mathbb{F}_7$ demos, I check that raising each element by p fixes the base field elements and not the rest (Frobenius automorphism), and confirm that raising each element by p^2 fixes the entire field. The Galois group of any quadratic finite field is $(1 = p^2, p) \sim \mathbb{Z}/2\mathbb{Z}$. The groups get bigger and more interesting when you can go beyond quadratic, and when you can generalize how many roots you add.

As far as general algebra goes, I made such a big fuss of wanting group operations to be implicit in a group definition and having functors at the same time for a reason. This was to set a good base for defining homomorphisms and isomorphisms, since homomorphisms effectively map generator to generator, and operation to operation. The classic definition is, given groups G, H, and a homomorphism $\phi : G \to H$, we have

$$\begin{aligned} \phi(x +_G +y) &= \phi(x) +_H \phi(y) \\ \phi(e_G) &= \phi(e)_H \end{aligned}$$

where $+_H, +_G$ denote the group operations of H and G, respectively. Mapping generators to generators is typically most relevant for finite groups, since it gets complicated when groups aren't finitely generated.

3.4 Total Generality

While I defined specific finite field types (like `FVector` and `FPolynomial`), that was for my personal ease of not wanting to deal with making sure all the type checking was up to par. I made sure to make the actual framework of operations and types (ex. the fieldelem type) not depend on finite integral characteristics at all, because I want to be able to use this to do things with rational, real, and complex numbers, since they all play important roles in class field theory, which is my lofty and ambitious eventual goal.

Finally, I chose not to make my Group typeclass a subclass of the Num type, because I wanted to be able to define the free group, and the operation for that is defined as concatenating strings. Rings and fields would be good candidates for subclasses of Num, but I didn't think of that until right now. The point is, I wanted to make space for non-commutative / less "nice" groups, should I ever want to use them in my code.

4 Showing and Testing

The `Showing.hs` and `Testing.hs` files were written to demonstrate/be able to print the work done in `Vectors.hs` and `Fields.hs`.