MACAULAY2 - RTG SEMINAR

SEPTEMBER 11TH, 2023

In the past we created functions. However, we didn't specify what kind of thing they can have as their input. Sometimes that's important, as you might want to treat an `Ideal` that is input differently from a `Module` which is treated differently from a `Ring`. Sometimes you also want optional arguments, or to let a function take different numbers of arguments. To do this, instead of creating a function you:

create a `Method`

Step 1 (create a new file):
Create a new file, perhaps `MethodTest.m2`. In that file enter the following code.

```
frobeniusPower = method();

frobeniusPower(ZZ, Ideal) := (e, J) -> (
    L := flatten entries gens J;
    p := char ring J;
    frobeniusL := apply(L, u -> u^(p^e));
    ideal(frobeniusL)
);

frobeniusPower(Ideal) := (J) -> (
    frobeniusPower(1, J)
);
```

`load` the file and make sure it loads.

Try it out in Macaulay2. That is, create a ring in characteristic $p > 0$, create an ideal I, and run:

```
frobeniusPower(I), frobeniusPower(1, I), frobeniusPower(5, I),
frobeniusPower(frobeniusPower(I)).
```

frobeniusPower is "overloaded", there is more than one way to execute it.

What this function does is it raises the generators of an ideal to the pth (or p^eth) power where p is the characteristic. It's frequently denoted by $I^{[p]}$. It can also be thought of as taking an ideal of R and expanding to the target of the Frobenius ($R \cong F_* R$), and viewing the result as an ideal $F_* R$. See if you can figure out how the function is doing it. You can also try it on other things. See what happens when you run `frobeniusPower(5)`.

Create an ideal, and see what it does.

Step 2 (an aside on modules).
Modules in Macaulay are given by presentation matrices, or as free modules. Run the following code in Macaulay2.

```
R = ZZ/7[x,y];
A = matrix{{x, 0}, {0, y^2}, {y^2, x^3}}
M = cokernel A
```

```
peek M
presentation M
F1 = R^3
peek F1
F2 = R^{0,-1,1}
peek F2
```

Note that Macaulay2 keeps track of the degrees a basis of a free module, or of other modules.

You can also do things like the following. Make sure you understand what is going on in below.

```
I = ideal(x+y^2, y^3)
M1 = I*R^1
M2 = I*R^2
(ambient M2)/M2
```

Step 3 (overload the function to do more):

Let's go back to the file you were using before.

Exercise 1: Add functionality to `frobeniusPower` so that you can run:

```
frobeniusPower(Matrix), frobeniusPower(Module), frobeniusPower(ZZ, Matrix),
                        frobeniusPower(ZZ, Module).
```

Where `frobeniusPower(Matrix)` will raise all the elements of the matrix to the pth power.

One way to turn a matrix into a list is to apply `entries Matrix`. Note, to get the entries of a matrix one at a time, you might want a nested loop or nested `apply` functions.

Also, `frobeniusPower(Module)` should give you the module obtained by tensoring with F_*R and viewing the result as an F_*R-module (this is obtained just by raising the presentation matrix entries to the pth power, since tensor product is right exact).

Alternate strategies. You can do this in other ways (there are ways to apply ring maps to ideals or matrices). However, I ask you use a strategy similar to what is done on ideals above as that will practice the syntax we've learned recently. We'll get to implement another strategy shortly, and we'll discover that applying ring maps to things doesn't quite do the same thing.

Step 4 (adding methods to options): Sometimes you want to have optional inputs that control how a function is run. This can also be used as an alternative to overloading the function in some cases.

Let's redefine our method `frobeniusPower` you created.

Change the line `frobeniusPower = method();` to

```
frobeniusPower = method(Options => {Relative => false});
```

You will also need to modify every function you created to take options. For example, I modified my functions for taking in `Ideal`s as follows.

```
frobeniusPower(ZZ, Ideal) := opts -> (e, J) ->  (
    L := flatten entries gens J;
    p := char ring J;
    frobeniusL := apply(L, u -> u^(p^e));
    ideal(frobeniusL)
);

frobeniusPower(Ideal) := opts -> (J) -> (
    frobeniusPower(1, J, Relative => opts#Relative)
);
```

`opts` becomes a variable in your function. It is a `OptionTable` (a subsclass of `HashTable`) which includes whatever options were set.

Note, I passed along the options from `frobeniusPower`.

Exercise 2. Modify your functions so they all can take in the option `Relative`. Make sure they all work. If you mess up the syntax, they will crash.

Right now, the option `Relative` doesn't do anything. We want to add some functionality. If the user sets `Relative` to `true`, it should work with the relative Frobenius instead of the absolute Frobenius.

Remark. The *relative Frobenius* on the polynomial ring `B[x,y]`, for instance, is the function that is the identity on the coefficient ring `B` and sends x, y to x^p, y^p.

The following code is one way to accomplish this. Let's create a new function that creates the relative Frobenius map.

```
relativeFrobenius = method();
relativeFrobenius(ZZ, Ring) := (e, V) -> (
    if (class V === PolynomialRing) then (
        map(V, V, frobeniusPower(e, vars V))
    )
    else (
        error "relativeFrobenius only implemented for PolynomialRing";
    )
);
relativeFrobenius(Ring) := (V) -> (
    relativeFrobenius(1, V)
);
```

Note, after adding this function, and reloading your file, you can apply a `RingMap phi` to an ideal `J` by `phi(J)`, or to a matrix `A` by `phi(A)`, or to a module `M` by `tensor(phi, M)`. Try it out!

Exercise 3. Modify all your functions so that they work with relative Frobenius or absolute Frobenius. My code looks something like:

```
frobeniusPower(ZZ, Ideal) := opts -> (e, J) ->  (
    if (opts#Relative === false) then (
        ...
        ideal(frobeniusL)
    )
    else if (opts#Relative === true) then (
        ...
    )
    else(
        error "Option Relative expected true or false.";
    )
);
```

Verify that the function works differently, and is behaving correctly. Note, over `ZZ/p[x,y]`, relative Frobenius is the same as the ordinary Frobenius. Verify this. Try it on rings like
`(ZZ/5[t]/ideal(t^2-3))[x,y]`
too. Or use `GF` to create finite fields of order p^e for $e > 1$.

Exercise 4 (Requires more background). For those who have thought about relative Frobenius in general. Think about how to make `relativeFrobenius` behave correctly on quotient rings.