MACAULAY2 - RTG SEMINAR

SEPTEMBER 18TH, 2023

Today, we will learn about `Type`s (basically `Class`es in many other programming languages) in Macaulay2.

We've seen packages recently that created DG algebras. A `DGAlgebra` is a subclass (in the parlance of C++) of `MutableHashTable`. This type was defined by the following code.

```
DGAlgebra = new Type of MutableHashTable
```

Today we'll learn how to interact with types (aka classes), make our own, and even customize how they are displayed to the user.

Step 1 (types and subtypes)

If the user passes you an object, you might want to know what type of object it is. You might also want to know if it is a subclass of a certain type. Try out the following code.

```
L = {0,1,A}
S = (5,6,7)
class L
class S
class L === List
class S === List
class class L
parent class L
parent parent class L
help class L
```

If you look at the bottom of the help, you will see that:

```
The object List is a type, with ancestor classes VisibleList < BasicList < Thing.
```

In Macaulay2, everything descends from `Thing`.

Sometimes you just want to know if an object is a class which is a subclass of something else. You can do this with the command `instance`, which returns `true` if the first argument is of Type the second argument (or the second argument is a parent/grandparent/etc). Try out the following.

```
M = new MutableList from L
instance(L, List)
instance(L, VisibleList)
instance(M, VisibleList)
instance(M, BasicList)
instance(L, HashTable)
```

Explore the classes and parents of some other objects you have looked at before.

Step 2 (make a class and constructors in a file):

If you try to define `ZZ/6[x]` you'll discover it is not implement it yet. Let's make a simple class for polynomials in one variable, modulo n. This is not the best way to add this functionality to Macaulay2! We are doing it for the purpose of learning about classes, and to practice more with `HashTable`s, `List`s, etc.

Create a new file, I called mine `MyClass.m2` and add the following code to it (or if you have a better name).

```
PolynomialModN = new Type of HashTable
```

The PolynomialModN will have three keys, `Modulus`, `VariableName`, `Coefficients`, the modulus, the name of the variable (stored as a string) and a `HashTable` storing the coefficients.

The `Coefficients` entry will have keys specifying the degree of each nonzero monomial. Each key will point/evaluate to the coefficient. For example, the HashTable corresponding to

```
{
    {0 => 3, 1=>2, 1000=>5}
}
```

would describe the polynomial $3 + 2x + 5x^{1000}$.

Most objects in Macaulay2 are subclasses of `HashTable` as we've discovered. Typically, you will want to your class to have several keys. This is enforced by creating a specialized *constructor* function (method) which take in user input and outputs an actual object.

Exercise 1. We will make a low level constructor which will take in `Modulus, VariableName, Coefficients` as arguments. This is the structure of mine, although I left out some key code.

```
polynomialModN = method();
polynomialModN(ZZ, String, HashTable) := (n, varName, coeffTable) -> (
    ...
    newCoeffTable := ...  --take all the values modulo n.
    new PolynomialModN from {Modulus => n, VariableName => varName, Coefficients => newCoe
);
```

Figure out how to replace the `...` with a different `HashTable` that has the same keys but takes the values of `coeffTable` modulo `n`.

Hint: Look up `applyValues` in the help. Note `5%3` should compute the remainders modulo 3. Optionally, you can have it delete keys which point to values of `0`.

That's sort of a low level constructor.

Exercise 2. Overload your constructor to instead pass it an element of `ZZ[x]` (or `ZZ[T]`, etc). Here's how I started.

```
polynomialModN(ZZ,RingElement) := (n,f) -> (
    R := ring f;
    genList := gens R;
    if (#genList > 1) then error "Expected a polynomial in one variable or fewer";
    varString := toString(genList#0);
    coeffList := coefficients f;
    ... --here I create a HasHTable
    polynomialModN(n, varString, ...)
);
```

You may need to look up the help on `coefficients`.

After you have completed this exercise, make sure try out your class and make sure your constructors work!

Step 3 (displaying your type):

You've made your type, and your code can create it, but the output doesn't look very pretty. Now you want to make it display nicely.

To do this, you overload functions which are called when your `Type` is going to be displayed. We're going to do this in two ways. First we overload the `toString` function. We're going to play around with string manipulation to accomplish this, remember, to concatenate strings you use the `|` operator. So `"ab"|"c"` creates the string `"abc"`. You can convert any number to a string by `toString`.

Exercise 3. Overload the `toString` function to make it input `PolynomialModN`s and output strings representing polynomials modulo n. For example, if I run something like:

```
toString polynomialModN(5, "x", new HashTable from {1=>7, 3=>2})
```

should output something like:

```
2*x^1 + 2*x^3
```

Hint: I used a loop. Note, in this function, but outside my loop I used the `curString := ""` assignment to make a blank string variable. Inside the loop, I had commands

```
curString = curString | ...
```

using the `=` sign (there are good reasons for using `:=` and `=` where I did – ask me).

What we just did was probably not the best way to accomplish this, and we want to take advantage of existing code. Let's try another way.

Exercise 4. Make a helper function which takes a `PolynomialModN` and outputs a the corresponding polynomial in `ZZ[x]` (changing `x` to whatever the appropriate variable is). I called my function `polynomialModNToPolynomial`.

Hint: Note, the command `S := ZZ[G#VariableName]` will create a ring over the integers with the variable whatever name you used before. You can then grab the variable for `S` with the command (gens S)#0.

Now that we have created our helper function, we can use Macaulay2's code to display things, simply by adding this to your file.

```
net PolynomialModN := G -> (
    net polynomialModNToPolynomial(G)
)
```

If all is working, you should be able to run a command like

```
polynomialModN(5, "x", new HashTable from {1=>3, 3=>1,18=>2})
```

and get something like:

```
         18    3
o17 = 2x   + x  + 3x

o17 : PolynomialModN
```

Step 4 (binary operations):

We want to be able to add or multiply such polynomials by each other, or with integers. Let's implement that.

To create a binary operation `+`, the basic syntax is the following.

```
PolynomialModN + PolynomialModN := PolynomialModN => (P1, P2) -> (
    ...
)
```

where `...` is where you add the two polynomials (and return the resulting `PolynomialModN`).

Exercise 5. Create binary operations where you can add, subtract, and multiply `PolynomialModN`s. Also create operations where you can multiply by integers (ie, `ZZ * PolynomialModN :=` and `PolynomialModN * ZZ :=`). This should throw an error if the variables are different.

Hint: A good way to do this is to convert the two objects to elements of `ZZ[x]` (or whatever common the variable is), then add them in that ring, and the convert the result back to a `PolynomialModN`.