Universal Property of Quotient Set Generator

Welcome to my Universal Property of Quotients Generator Program! The purpose of this program is purely educational and is meant to generate all of the possible commutative diagrams (up to isomorphism) you can create with a set endowed with an equivalence class, an equivalence invariant function from our set with an equivalence class to an arbitrary set, and the quotient set generated by the set with an equivalence relation. What does any of that mean? I will explain in this README file! The purpose of this README is to give a general explanation of how this program works and how you can use it at home. Without any further ado, let's get started!

Background

Before I explain how to run the program, I will give some background on the concepts that this program explores. However, for some of these concepts, I will talk about them as if you know everything about them; this mainly regards basic details on sets and mathematical logic. If you do not understand these, I would suggest reading an introductory Set Theory textbook (I personally recommend *Introduction to Set Theory, 3rd Edition* by Karel Hrbacek and Thomas Jech) or taking a Discrete Mathematics course at a college. With that out of the way, let's begin!

Equivalence Relations and Quotient Sets

Suppose we have a set A consisting of an arbitrary number of elements. While the elements themselves could be interesting, what is more intriguing is how the elements relate to each other. Hence, we can define a relation R on A as a subset of the Cartesian product A x A where a pair (x,y) is an element of R if and only if "x R y" where x and y are elements of A. This sounds entirely abstract, so I will provide an example. Consider the equality relation, where two elements relate if they are equal to each other; this relation is known under its recognized symbol "=". There are plenty of relations to enumerate, but there is a special kind of relation that many mathematicians love to focus on. For any relation R, we will define a few properties. If every element of a set A relates to itself (i.e. for all elements a of A, we have that "a R a"), we say that R is reflexive. If for all elements a and b of A, we have that "a R b" implies "b R a", then we say that R is symmetric. Lastly, iff for all elements a,b, and c of A, we have that "a R b" and "b R c" implies "a R c", then we say that R is transitive. If a relation R has all of the previously listed properties, then we call R an equivalence relation. Why do mathematicians love these types of relations so much? The reason is that with an equivalence relation on a set, we can define equivalence classes. Given an element a of A and an equivalence relation R on A, we define the equivalence class [a] to consist of the elements b of A such that "a R b". Essentially,

the importance of these classes come from the fact that we can take every element of A and put it into a class consisting of other elements that it relates to; in other words, we can represent A as a union of distinct larger collections of elements rather than just each individual element [**NOTE**: I am assuming that you agree that the union of all the equivalence classes is the set A itself and that the classes are pairwise disjoint. If you do not see why this is, consider reading that Set Theory book I mentioned]. With this in mind, we might want to consider how our set A relates to its equivalence classes. To do so, we define a new set called the quotient set, which we'll denote by A / R. The quotient set under an equivalence relation R is the set of all equivalence classes on A using R. Now how does this quotient set relate to our original set? Since every element of A belongs to some equivalence class, then we can try mapping from our set A to the quotient set A / R. We will call this mapping the quotient function (denoted by q), which we note must be surjective by how equivalence classes are defined. This function in essence sends every element of A to the equivalence class containing it. The last thing to discuss in this section is what is known as an invariant function relating to equivalence relations. Suppose we have a function f from A to B for sets A and B and A is endowed with an equivalence relation R. We say that f is R-invariant if and only if for all elements a and b of A, "a R b" implies that f(a) = f(b). This statement may sound bizarre, but it is extremely important for proving the main theorem of this program.

Universal Property of Quotient Set

Suppose we have sets A and B, where A is endowed with an equivalence relation R. Now suppose we create an R-invariant function f from A to B. Cool, now you have an R-invariant function! However, don't you wish you could make a direct one-to-one relation between the two sets A and B? Sadly, given how you constructed f, this may not be the case; in fact, it may be that you can't create a one-to-one between the two due to their inherent structures. Some more bad news is that we cannot create a bijection here either. Rather, we can decompose our function f into a form where we see how the relation R affects the mapping. Let me explain. Let's take a look at our function f. For each element a in A, we note that a will map to some element in B and that a will relate to some elements of A (or just itself). We also know that since f is R-invariant, then when two elements in the domain relate, then they map to the same element in the codomain. Therefore, instead of considering each map from each individual element, we can just consider the map from the equivalence classes of A. We can then map each element of A to its equivalence class (the quotient map q). From there, we can then define a map (let's call it f') from the quotient set to the codomain of f where for each equivalence class [x], f'([x]) = f(x). Since f is R-invariant, then it is clear that f' is well-defined. The existence of f' such that f' o q = f is called the Universal Property of the Quotient Set. Why is this property important? As the name implies, it is universal to all R-invariant functions, meaning we can find a way to break down a relatively boring R-invariant function into an interesting function from the quotient to the codomain of f.This universal property,while not as potent for quotient sets, is very potent for most general algebraic structures like groups, rings, topologies, etc; in fact, the universal property can be generalized for algebraic structures as a whole (although that is far beyond the

scope of what I could explain). With such a magnificent property in hand, wouldn't it be wonderful to be able to explore all of the possible formations we can make with arbitrary R-invariant functions between sets? Well, you have come to the right place, as that is what this program intends to do!

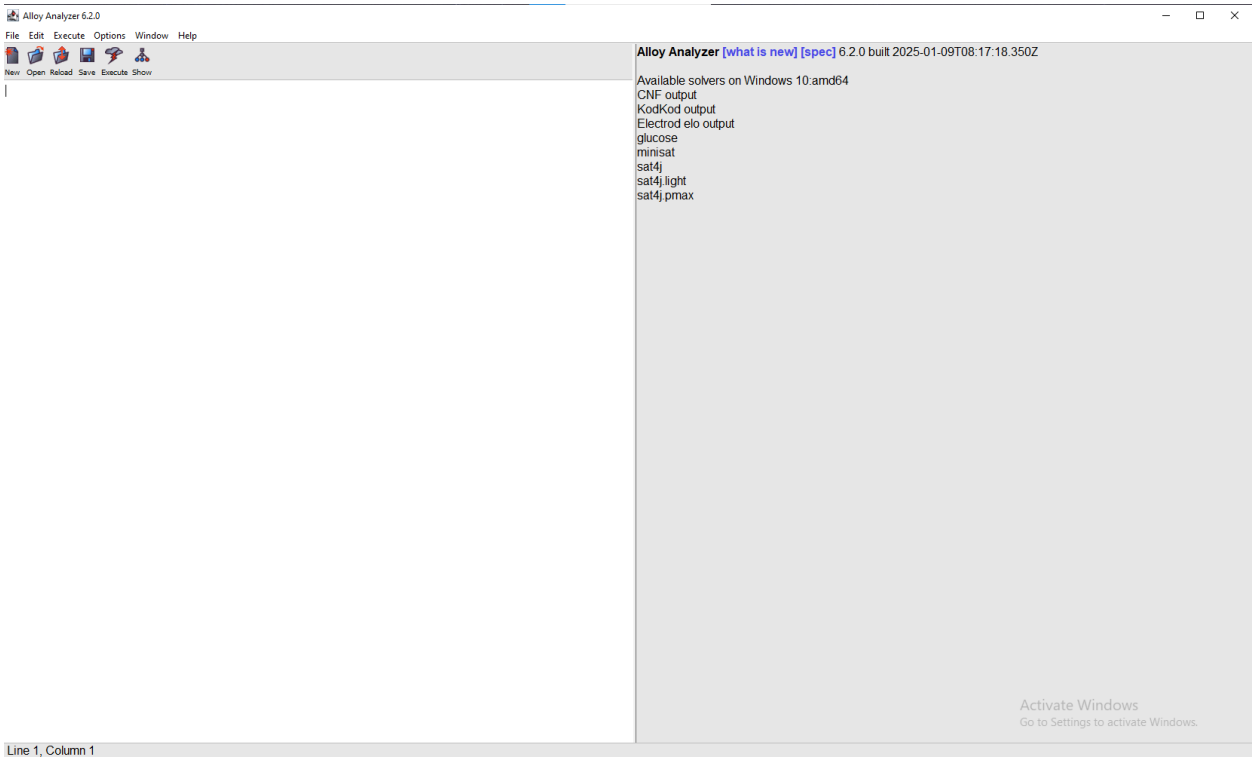
Alloy and the Program

Your first question when looking at this program is likely "What is Alloy?" Alloy is an open source language and software analyzer created by the Software Design Group at MIT. For more information on Alloy, visit alloytools.org, which contains information about the background of the language, how to download it, and how to learn the language. The two main reasons I used the language was for its visualizer and its unique syntax and semantics. The best way to describe this language is that it is akin to pure first order logic. What does that mean? We only work with atoms and the relationship between atoms. Hence, all the information we are given in the visualizer is purely circles representing atoms connected by arrows representing a relationship. Why is this important for our universal property? A huge component of the property is the commutative diagram it creates between the set A, its quotient set, and the set B. Since sets are just collections of objects, we can represent the elements of a set as just atoms and the mappings between the sets can be the arrows connecting one atom to another. Therefore, the visualizer helps to visualize how the individual elements of each set are mapped or mapped to. Now, what separates this language from another language is that its syntax is defined by set relations or logic. What does this mean? This means that we can define our predicates and functions using the common words of logic (like "if and only if", "implies", etc), which help to display theorems without confusing the reader about what certain components mean. It also means that, as mentioned, we can create mappings between sets with our predicates, which help us to define the functions we need to create our universal property. With that clarified, we will now move onto how to make the program work and what each part of the program means[**NOTE:** I will not explain how to download the program, as alloytools.org/download.html explains exactly that; it should be noted that as of the writing of this README, I used Alloy 6.2.0 to create the program].

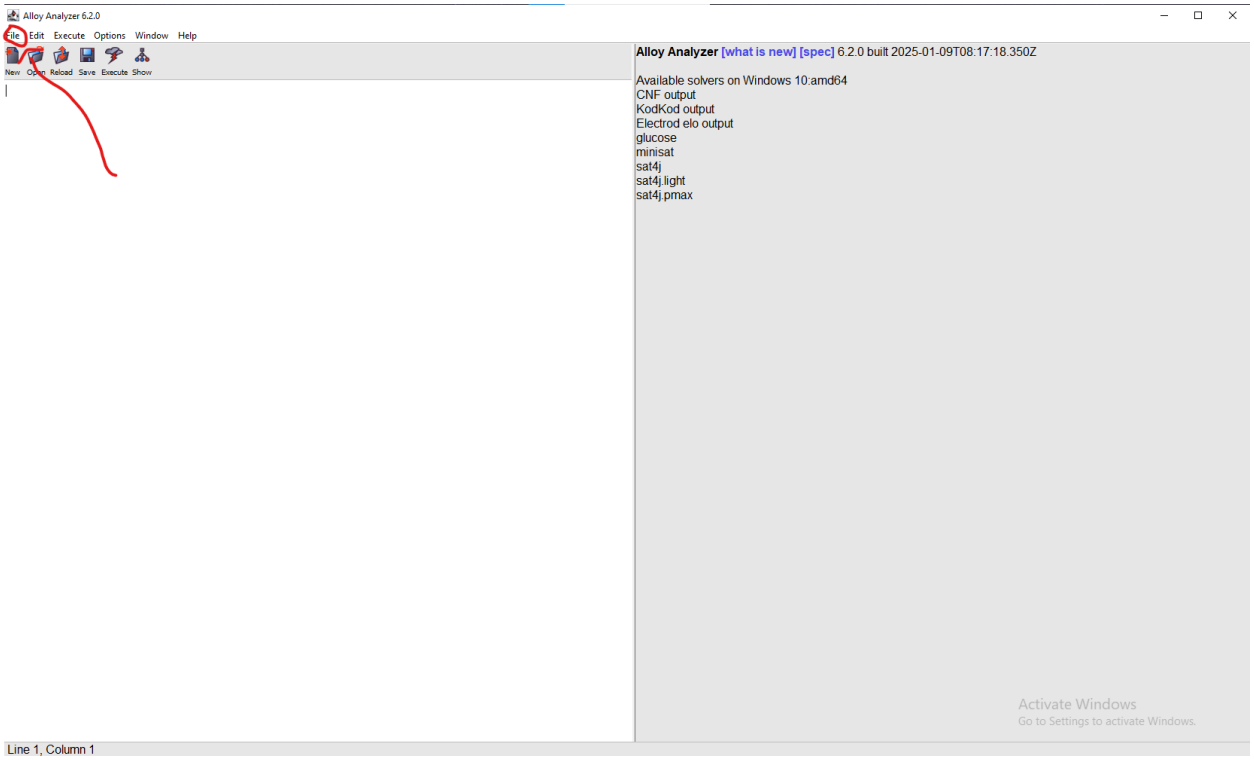
Breakdown of the Program

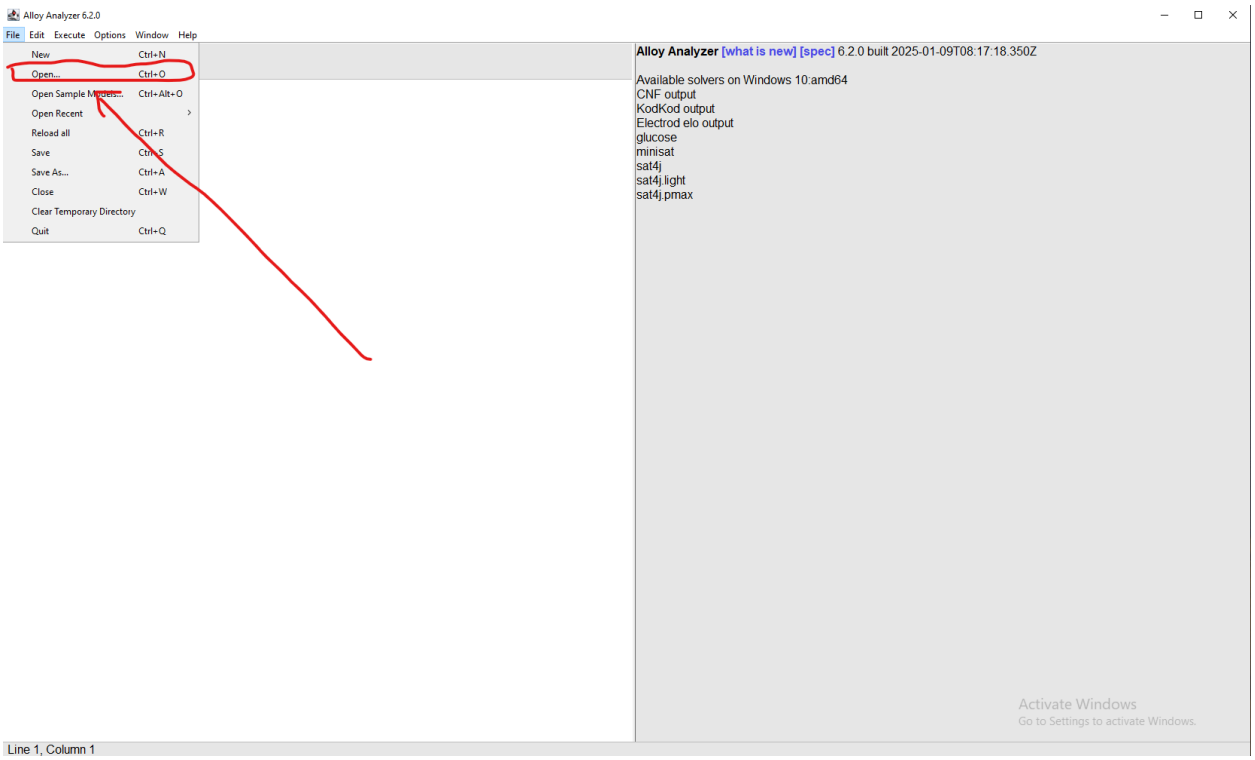
Starting up Alloy

Once you download Alloy and my program, open up Alloy. You should see a screen that looks like this:

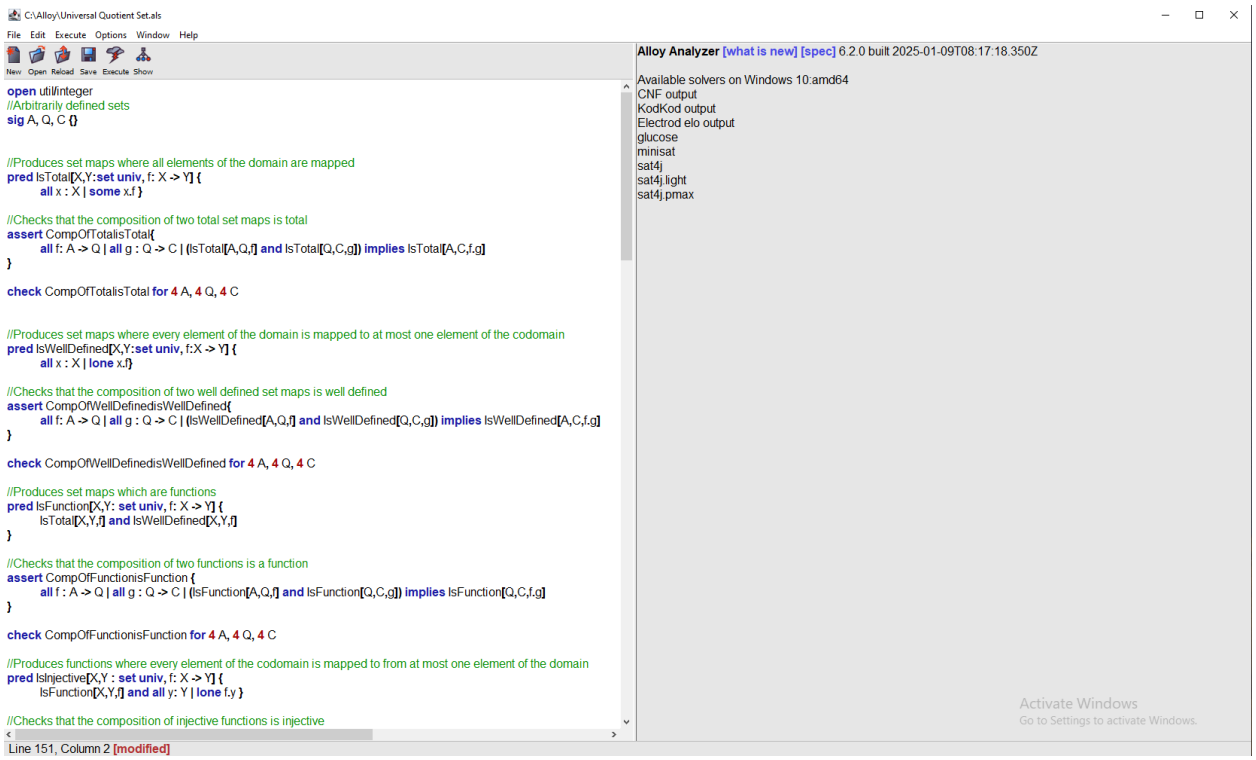


In order to open my program, you will click on the File tab at the top left and click Open, as shown below:





After you click Open, you will find the file “Universal Quotient Set.als” and double click on it to open it. Once you do so, you should have a screen that looks like this:



Congratulations! You have opened the program! But how do you run it? What does all of this mean? Don't worry, dear reader, as I will go through each step of the program and explain what each process means.

Remark on Alloy

Before we begin, I should note that I will NOT be explaining most of the actual syntax of Alloy itself, as doing so would take way too much time. Therefore, if you are confused about what "pred" or "assert" means, I highly suggest reading the Alloy documentation, which is available on alloytools.org.

Code Analysis

Let's get started! We first start of with the basics

```
open util/integer
//Arbitrarily defined sets
sig A, Q, C {}
```

The "open" command imports the integer module, which will allow us to write some functions later down the line. We then define three signatures: A, Q, and C. All of these signatures

represent arbitrary sets with elements delineated by the name of the set with a number suffix (which we will see in the visualizer). While the sets themselves are entirely abstracted, in this program they have specific places in the quotient set diagram. For the domain and codomain of the R-invariant function, we use A and C, respectively, to represent them . For the quotient set A / R, we use Q to represent it (Q for quotient). Moving on, we define a few predicates which will produce functions between sets:

```
//Produces set maps where all elements of the domain are mapped
pred IsTotal[X,Y:set univ, f: X -> Y] {
        all x : X | some x.f }

//Checks that the composition of two total set maps is total
assert CompOfTotalisTotal{
        all f: A -> Q | all g : Q -> C | (IsTotal[A,Q,f] and IsTotal[Q,C,g]) implies IsTotal[A,C,f.g]
}

check CompOfTotalisTotal for 4 A, 4 Q, 4 C


//Produces set maps where every element of the domain is mapped to at most one element of the codomain
pred IsWellDefined[X,Y:set univ, f:X -> Y] {
        all x : X | lone x.f}

//Checks that the composition of two well defined set maps is well defined
assert CompOfWellDefinedisWellDefined{
        all f: A -> Q | all g : Q -> C | (IsWellDefined[A,Q,f] and IsWellDefined[Q,C,g]) implies IsWellDefined[A,C,f.g]
}

check CompOfWellDefinedisWellDefined for 4 A, 4 Q, 4 C

//Produces set maps which are functions
pred IsFunction[X,Y: set univ, f: X -> Y] {
        IsTotal[X,Y,f] and IsWellDefined[X,Y,f]
}

//Checks that the composition of two functions is a function
assert CompOfFunctionisFunction {
        all f : A -> Q | all g : Q -> C | (IsFunction[A,Q,f] and IsFunction[Q,C,g]) implies IsFunction[Q,C,f.g]
}

check CompOfFunctionisFunction for 4 A, 4 Q, 4 C
```

The first predicate, “IsTotal”, will produce mappings between two sets such that for every x in the domain, there exists at least an element y in the codomain such that f(x) = y. The assertion that follows, “CompOfTotalisTotal”, just checks that if you compose two total mappings, then that mapping will itself be total. The second predicate, “IsWellDefined”, will produce mappings between two sets such that for every element x in the domain there exists at most one element y in the codomain such that f(x) = y (or in other words, if x = x’, then f(x) = f(x’)).The following assertion “CompOfWellDefinedisWellDefined” just checks that if you compose two well-defined mappings, then the mapping itself will be well-defined. The final predicate, “IsFunction”, will produce mappings between two sets such that the mappings are both total and well-defined (in other words, the predicate will produce functions). The following assertion “CompOfFunctionisFunction” just checks that if you compose two functions, then the

composition will itself be a function. Now we move onto some more important building blocks for maps:

```
//Produces functions where every element of the codomain is mapped to from at most one element of the domain
pred IsInjective[X,Y : set univ, f: X -> Y] {
        IsFunction[X,Y,f] and all y: Y | lone f.y }

//Checks that the composition of injective functions is injective
assert CompOfInjectiveisInjective {
        all f : A -> Q | all g : Q -> C | (IsInjective[A,Q,f] and IsInjective[Q,C,g]) implies IsInjective[A,C,f.g]
}

check CompOfInjectiveisInjective for 4 A, 4 Q, 4 C

//Produces functions where every element of the codomain is mapped to from at least one element of the domain
pred IsSurjective[X,Y:set univ, f: X -> Y] {
        IsFunction[X,Y,f] and all y : Y | some f.y
}

//Checks that the composition of surjective functions is surjective
assert CompOfSurjectiveisSurjective {
        all f: A -> Q | all g : Q -> C | (IsSurjective[A,Q,f] and IsSurjective[Q,C,g]) implies IsSurjective[A,C,f.g]
}

check CompOfSurjectiveisSurjective for 4 A, 4 Q, 4 C

//Produces functions which are both injective and surjective
pred IsBijective[X,Y : set univ, f : X -> Y] {
        IsInjective[X,Y,f] and IsSurjective[X,Y,f]
}

//Checks that the composition of bijective functions is bijective
assert CompOfBijectiveisBijective {
        all f: A -> Q | all g : Q -> C | (IsBijective[A,Q,f] and IsBijective[Q,C,g]) implies IsBijective[A,C,f.g]
}

check CompOfBijectiveisBijective for 4 A, 4 Q, 4 C
```

The first predicate, “IsInjective”, will produce functions between two sets such that for every element y in the codomain, there exists at most one element x in the codomain which maps to y (in other words, if f(x) = f(x’), then x = x’). The following assertion “CompOfInjectiveisInjective” will check that if you compose two injective functions, then the composition will itself be an injective function. The second predicate, “IsSurjective”, will produce functions between two sets such that for every y in the codomain, there exists at least one element x in the domain such that f(x) = y. The following assertion “CompOfSurjectiveisSurjective” will check that if you compose two surjective functions, then the composition will itself be a surjective function. The final predicate, “IsBijective”, will produce functions between two sets which are both injective and surjective (i.e. will produce bijective functions). The following assertion “CompOfBijectiveisBijective” will check that if you compose two bijective functions, then the composition itself will be bijective. Now we move on to defining equivalence relations:

```
//Produces relations on a set which are reflexive
pred IsReflexive[X: set univ, f: X -> X] {
	all x : X | (x -> x) in f
}

//Produces relations on a set which are symmetric
pred IsSymmetric[X : set univ, f: X -> X] {
	all x : X | all y : X | (x-> y) in f implies (y -> x) in f
}


//Produces relations on a set which are transitive
pred IsTransitive [X: set univ, f: X -> X] {
	all x : X | all y : X | all z : X | (x -> y) in f and (y -> z) in f implies (x -> z) in f
}

//Produces equivalence relations on a set
pred IsEquivalence[X: set univ, f : X -> X]{
	IsReflexive[X,f] and IsSymmetric[X,f] and IsTransitive[X,f]
}
```

The first predicate, “IsReflexive”, will produce mappings from a set to itself where every element x must map to itself (the mapping can include other mappings, but must fix every element of the domain). The second predicate, “IsSymmetric”, will produce mappings from a set to itself where if an element x maps to an element y, then the element y must map to the element x as well. The third predicate, “IsTransitive”, will produce mappings from a set to itself where if an element x maps to an element y and an element y maps to an element z, then the element x must map to the element z. The final predicate, “IsEquivalence”, will produce mappings from a set to itself which are reflexive, symmetric, and transitive (i.e. the mapping represents an equivalence relation on a set). The next part of the code is arguably the most difficult to understand and will thus need a proper explanation:

```
//Defines a mapping between the distinct equivalence classes of a set such that each equivalence class maps to
//another equivalence class exactly once; the purpose of this is to calculate the number of distinct equivalence
//classes, where the cardinality of g is exactly that number.
pred EquivClassMap[X : set univ, f : X -> X, g : X -> X] {
	(all x : X | one x2 : x.*f | one x3 : X - x.*f | x2.g = x3 and #(x.*f <: g) = 1) or ((some x : X | #X = #x.*f) implies one x : X | g = x -> x)
}

//Maps elements from different equivalence classes to each other. Used to calculate the number of
//equivalence classes for a given equivalence relation
pred IsNotEquivalence[X,Y: set univ, f: X -> X, g : X -> X] {
	IsEquivalence[X,f] and EquivClassMap[X,f,g]
}
```

The first predicate, “EquivClassMap”, was the hardest part of this code to write. The purpose of this predicate is to produce a mapping between each distinct equivalence class so that we can calculate the number of distinct equivalence classes in an instance (I will explain why we need this in the predicate defining the diagram). Why was this code so hard to write? In any normal language with iteration, we can easily calculate the number of equivalence classes by looking at each element and putting them into their respective equivalence classes and just calculate the length of the list of equivalence classes. However, Alloy does not have typical iteration; all predicates and functions are defined with set relational language(there are some functions in

the integer module which allow some basic operations between numbers, but it is limited). Therefore, we have to come up with another way to calculate this. The idea I came up with was to map each equivalence class to another equivalence class exactly once; this will produce a mapping whose cardinality is exactly the number of distinct equivalence classes. However, how will we know when an element of an equivalence class has been mapped to another equivalence class? In other words, how can we make sure that the equivalence classes are only mapped exactly once? We will use a little friend called “transitive closure”. In the predicate, you may notice that we use “x.*f”. The asterisk represents something called the transitive closure. The transitive closure allows us to take an element x and produce the set of all elements which x is connected to by the mapping f (including x itself). Since f is an equivalence relation, then this is essentially producing the set of elements that exist in the same equivalence class as x. We then have a way to define the set of elements in an equivalence class. With this knowledge in mind, we can then break down the predicate. The predicate is constructed as an or statement with two statements. The first statement says that for every element x, we will take exactly ONE element from its equivalence class and map it to exactly ONE element from another equivalence class which is not the one our element x is an element of. However, it may be the case that g has other mappings which evade this rule through trickery; therefore, we then say that the cardinality of each mapping from an equivalence class to another equivalence class is exactly one (<: represents a domain restriction). Hence, we have successfully defined a method to extract the number of distinct equivalence classes; however, there is one edge case we need to consider: what if there is only one equivalence class? Our previous construction will fail to produce anything, as there are no other equivalence classes to map to. Therefore, we reach the second statement of our or statement, which states that if we have an element of x where the cardinality of the set X is equal to the cardinality of the equivalence class with x, then we will just the equivalence class to itself. Now we are done! With this predicate in tow, we can then define the next predicate “IsNotEquivalence”, which will take two mappings from a set to itself and make one of them an equivalence relation and the other the distinct equivalence class mapping. With these predicates defined, we can then define a few functions:

```
//Calculates the size of the equivalence class with representative x
fun EquivalenceClassSize[X : set univ, f: X -> X, x : X] : Int {
        #(x.*f) }

//Calculates the number of different equivalence classes of an equivalence relation
fun EquivalenceClassPartitions[X: set univ, f: X -> X, g : X-> X] : Int {
        #g }
```

The first function “EquivalenceClassSize” takes a set, an equivalence relation on the set, and an element of the set and returns an integer which is the size of the equivalence class which x represents. The second function “EquivalenceClassPartitions” takes a set, an equivalence relation on the set, and the distinct equivalence class mapping and returns an integer representing the number of distinct equivalence classes. With the basics of equivalence relations and functions defined, we can now move onto defining the important aspects of the universal property diagram:

```
//Produces functions which are invariant to the equivalence defined on the domain
pred IsInvariant[X,Y : set univ, f : X -> X, g : X -> Y] {
	IsEquivalence[X,f] and IsFunction[X,Y,g] and all x1, x2 : X | (x1 -> x2) in f implies (x1.g = x2.g)
}

//Produces surjective function which maps each element of the set endowed with an equivalence
//relation to its respective equivalence class in the quotient set
pred IsQuotientMap [X, Y : set univ, f : X -> X, g : X -> Y, h : X -> X] {
	IsNotEquivalence[X,Y,f,h] and IsFunction[X,Y,g] and IsSurjective[X,Y,g] and all x1, x2 : X | (x1 -> x2) in f iff (x1.g = x2.g)
}


//Produces as many elements of the quotient set as there are equivalence classes for a sharper image
pred QuotientSetEqualsEquivalenceClasses [X,Y:set univ, f : X -> X, g : X -> X] {
	(IsNotEquivalence[X,Y,f,g] and some x : X | lt[EquivalenceClassSize[X,f,x],#X]) implies #Y = EquivalenceClassPartitions[X,f,g] else #Y = 1
}

//Produces the quotient map without excess elements of the quotient set
pred TrueQuotientMapRepresentation [X,Y : set univ, f : X -> X, g : X -> Y, h : X -> X] {
	IsQuotientMap[X,Y,f,g,h] and QuotientSetEqualsEquivalenceClasses[X,Y,f,h]
}
```

The first predicate “IsInvariant” produces functions which are invariant under the equivalence relation f (i.e. for all elements x1 and x2 in the domain X, then if x1 relates to x2 by f, then for the function g, g(x1) = g(x2)). The second predicate “IsQuotientMap” produces functions which represent the quotient map from a set X to its quotient set under the equivalence relation f. The third predicate “QuotientSetEqualsEquivalenceClasses” is made to fix a small issue with the previous predicate, which is that we may have elements in Q which are not mapped to (even though the function defined is surjective, there are some weird rules with Alloy’s language). Hence, this predicate is made to make sure that the number of distinct equivalence classes equals the number of elements of the quotient set (obviously). The code essentially says that if we have our distinct equivalence class mapping and an equivalence relation, then we split into two cases. The first case is that if there is an element of our domain X whose equivalence class is less than the size of X itself, then the number of elements of the quotient set will equal the number of distinct equivalence classes. The second case says that if every element of our domain X is in one equivalence class, then the quotient set will only have one element (NOTE: you may note that this is unnecessary, as the predicate “EquivClassMap” accounts for instances of a singular equivalence class; this is indeed true and I only kept this predicate in for fear of breaking something). The final predicate “TrueQuotientMapRepresentation” combines both of the previous predicates and produces quotient functions between a set X and its quotient set under the equivalence relation f. With our invariant function and quotient function defined, we can finally present the main predicate of this program:

```
//Produces the commutative diagram representing the universal property of the quotient set
pred UniversalPropertyGenerator {
	some f, i: A -> A | some g : A -> Q |  some h : A -> C | some j : Q -> C | IsInvariant[A,C,f,h] and TrueQuotientMapRepresentation[A,Q,f,g, i] and all a : A | a.h = a.g.j
}
```

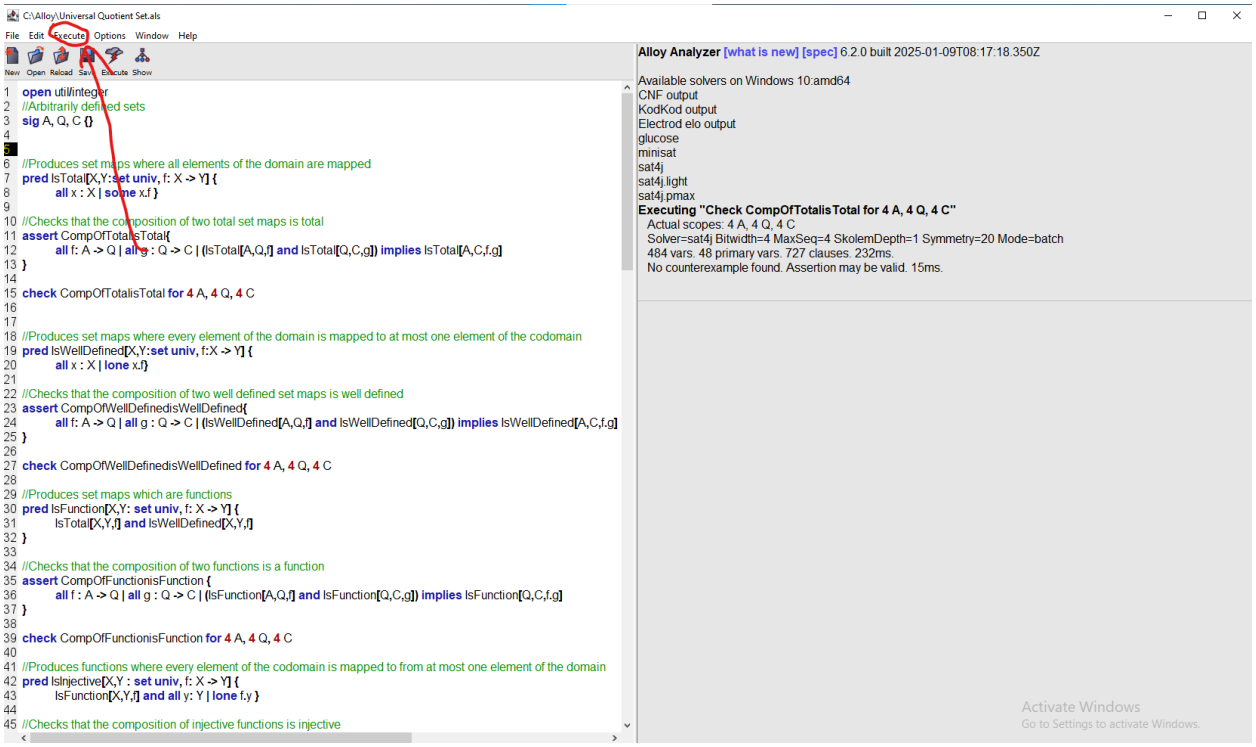
The main jewel of this program, “UniversalPropertyGenerator”, takes our sets A, Q, and C and creates the commutative diagram as mentioned before. Now that we have all of our code explained, I will explain how to run the program itself.

Running the Code

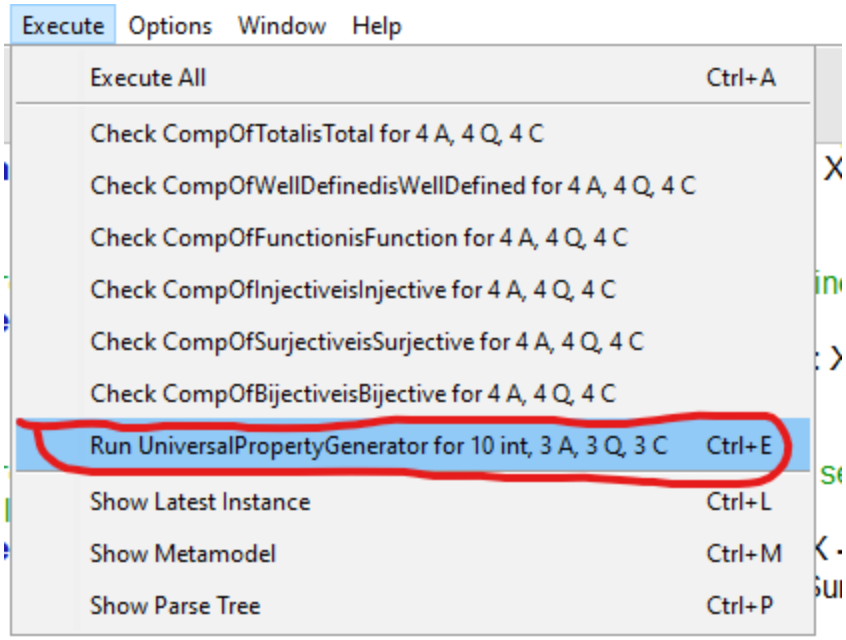
So how do we run the code? The first thing to notice is this snippet of code:

```
run UniversalPropertyGenerator for 3 A, 3 Q, 3 C, 10 Int
```

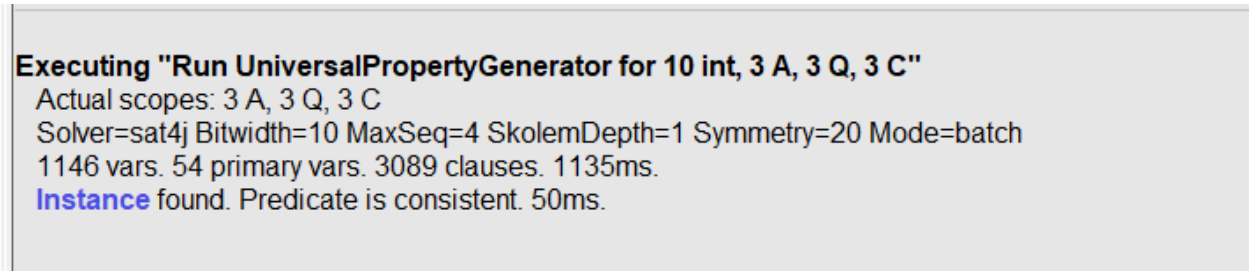
This code essentially says that we will run our predicate “UniversalPropertyGenerator” using up to 3 atoms for A, 3 atoms for Q, 3 atoms for C, and 10 bits for integer representation. This will then produce the commutative diagrams for sets of cardinality 0 to 3. If you want to change this, you can adjust the value for A and adjust the values for Q and C to match A (the program might fail to produce all of the diagrams if you don’t make all of the atoms for A,Q, and C equal). You do not need to worry about changing the value for Int unless you choose to do very large values for A,Q, and C. Something important to note about Alloy is that it is a FINITE modeling language, meaning it will only produce our diagrams up to a certain value; therefore, we cannot represent infinite sets in this program. If you want to only show the diagrams for a specific cardinality of A, you can add in the syntax “exactly” before the value for A. For example, if you wanted to represent the diagrams for an A of cardinality 4, you would write **run UniversalPropertyGenerator for exactly 4 A, 4 Q, 4 C, 10 Int**. With this in mind, how do we actually make this snippet of code execute? To do so, click on the “Execute” tab to the left of “Options” and to the right of “Edit” as depicted below:



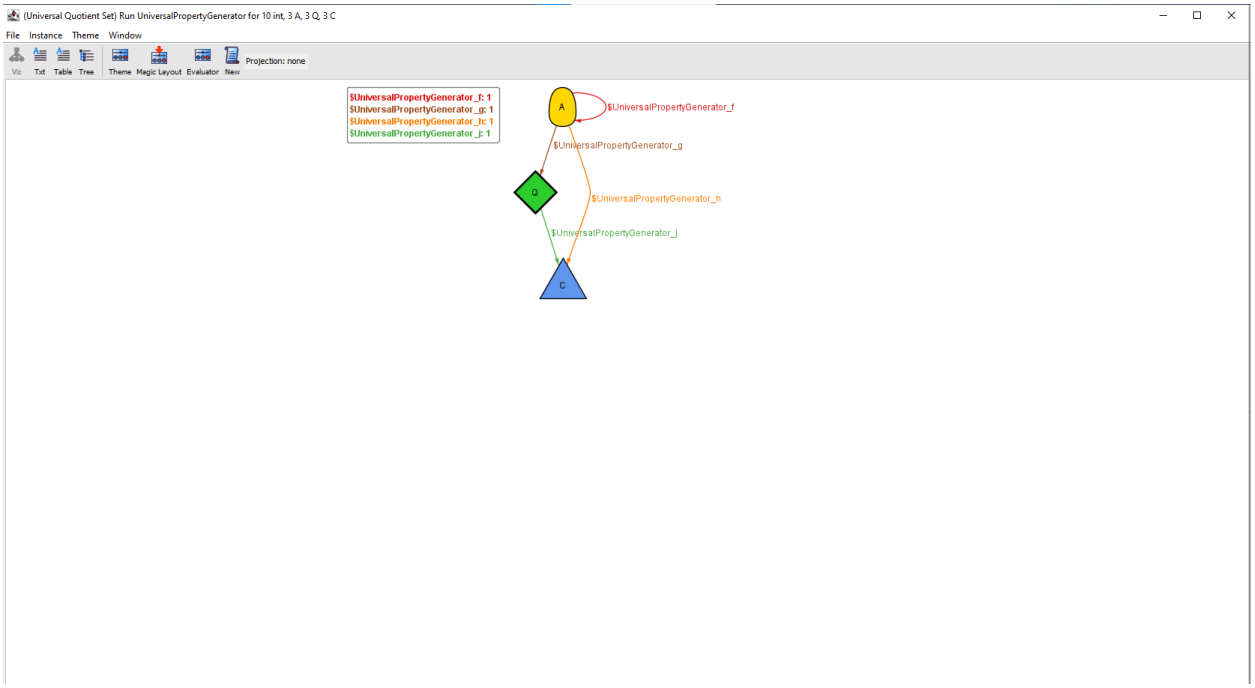
When you click “Execute”, you will see a pulldown menu with a bunch of functions you can run. For our purposes, you will go down to the bottom and click on "Run UniversalPropertyGenerator for 10 int, 3 A, 3 Q, 3 C" as shown below:



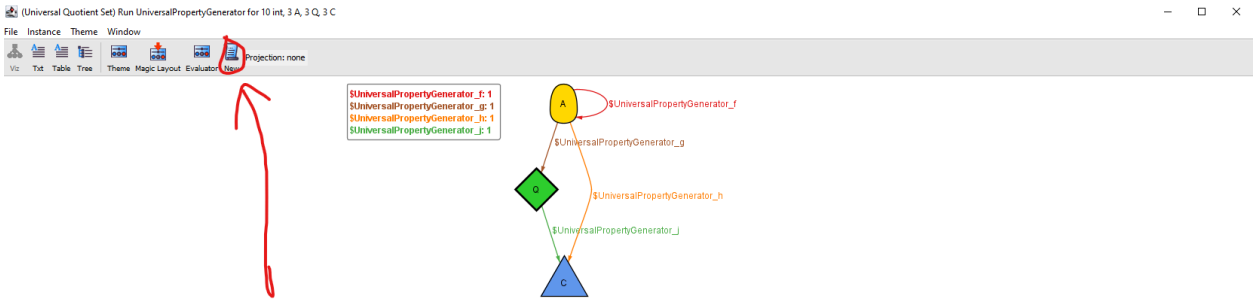
You can also run the other functions if you want, which are the assertions mentioned in the code breakdown section. All that an assertion does is that when you run it, it will check to see if what you wrote in the assertion is true by checking for counterexamples. If no counterexamples are found, it will produce nothing. If one is found, it will allow you to look at the instance of how your assertion fails. Once you run the bottom function, you will see the program run on the right window and after some time, it will produce the following message:



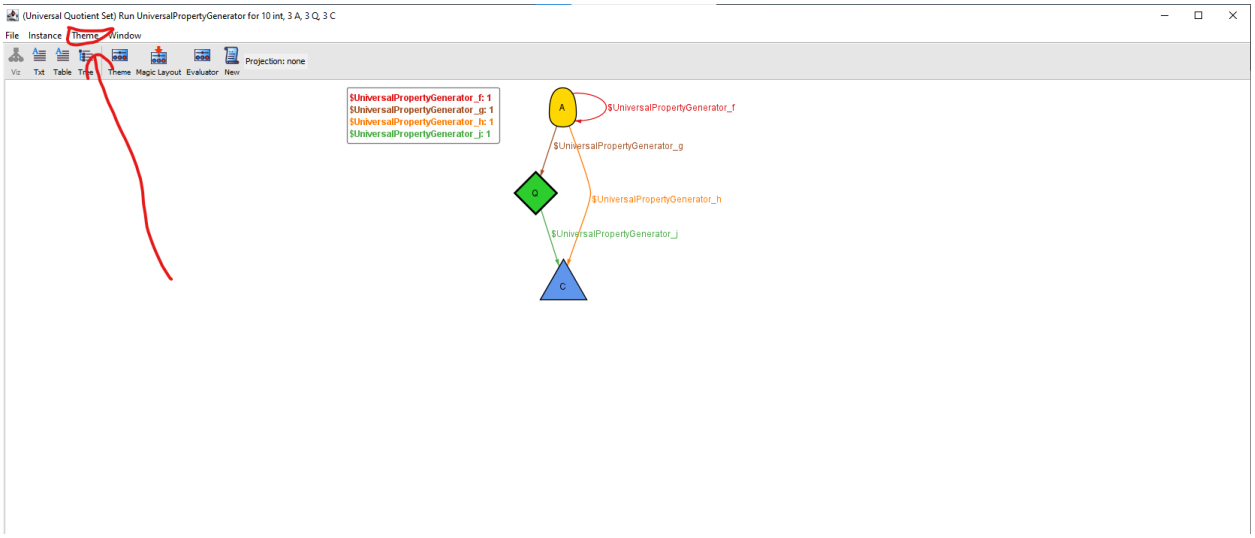
The important part of this code is the highlighted “Instance” button. Clicking on this will bring you to the visualizer, where you can see the commutative diagrams in action! After clicking on “Instance”, you will get the following window:



All of this hard work has paid off, as we now get to see all of the generated commutative diagrams created by the universal property. In this image, we note that each node represents an element of our sets A,Q, and C. Note that f represents the equivalence relation on A, g represents the quotient map from A to Q, h represents the invariant function from A to C, and j represents the function produced by the universal property. However, this is just one instance of it. If we want to produce more, we simply click on the “New” button at the top, as shown below:



When we click that button, we will produce another commutative diagram. You can keep on clicking “New” until you have exhausted all of the diagrams (up to symmetry), in which it will give a message and send you back to the code. One thing you might notice is that my diagram might look different from yours. To fix this, you can change the theme of the program by clicking on the “Theme” button at the top left, as shown below:



After clicking on it, you will see a drop down menu. Click on “Load Theme”, which will send you into your files. My program came packaged with a theme called “quotientset.thm”. Click on it and it will upload my theme to the visualizer, making your diagrams match mine. You can also make your own theme by clicking on the “Theme” button with the image, which will allow you to change the shapes of the atoms, the color of the atoms, and much more. For more information on how to change themes, visit the Alloy website.

Conclusion

With the code broken down and explained, it is in your hands to use this program for good. I hope you enjoy generating those diagrams like I did! If you have any more questions about the program or if you find a mistake, please contact me at my Github account. In the future, I might make another one of these programs, but for various algebraic structures like groups, rings, fields, etc.