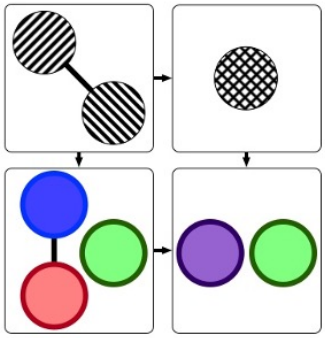
Hi, I'm excited to share some recent work performed at UF with James Fairbanks and Tyler Hanks in collaboration with Evan Patterson at the Topos Institute. Together we've released this package, AlgebraicRewriting, which implements an algebraic technique called graph transformation. This is part of a software ecosystem called AlgebraicJulia.

Outline



AlgebraicRewriting.jl

Declare and execute rewriting systems

What problems can it be used for?

Controlled data transformation

How do you use it?

Draw pictures!

What advantages does it have?

Transparency
Generality
Transferability

At a high level, the library allows you to declare and execute rewriting systems.

*click*So the important questions I want to answer here are what problems can be solved with rewriting systems, how do you actually use this software, and some of the unique advantages of this approach.

*click*In short, the technique is a declarative language for transforming data in a very controlled way.
The language of specifying how to transform data of type X is to draw patterns and substitutions which are also instances of type X, whatever that may be.
And this offers virtues of transparency, generality as it is applicable to wide varieties of data structures, and transferability
(meaning you can do you work with some data structure, and then realize you need to radically change that structure in some way, but there is a way to migrate your rules into the new formalism in a way that's not possible to do when expressing data transformation as code).
CLICK So let's start getting into more detail about the first point, which was: what this is used for?

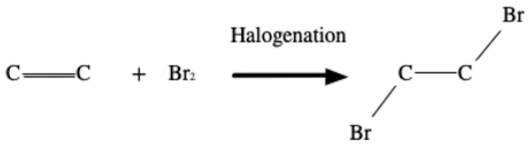
What kind of data transformations?

Think of as a special kind of SQL query:

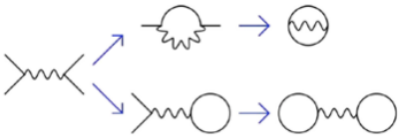
- Identifies a region of your data (a la SELECT FROM WHERE)
- Modifies that region of the data in a specified manner

Changing the state of some (complex) data structure

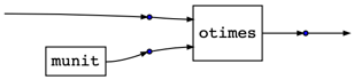
Performing a simulation



Generating structures via a grammar



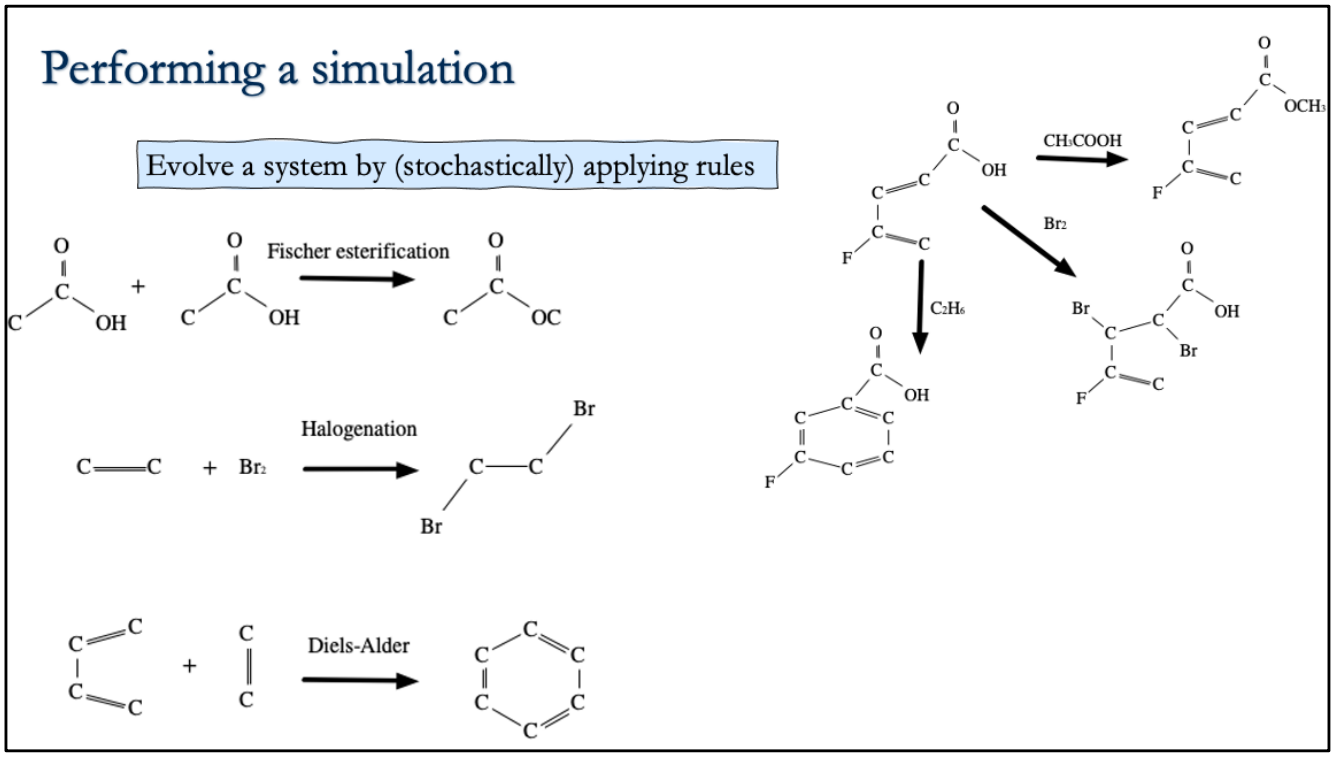
Equational reasoning



In general, we have some possibly complex data structure and we want to modify it. Of course, you can write imperative code to do such a thing, but the closest analogue which is declarative would be SQL.

There you can imagine writing a query that identifies part of your data structure to change, and then performs an update based on what the results of the query were.

click I'll try to highlight three kinds of things you could do with it before picking one to dive deeper into. These three things are performing simulations, generating structures via a grammar, and equational reasoning.



So our first example of something you can do with graph transformation is to set up a rewriting system by defining a set of so-called rewrite rules.

For example, you could encode your knowledge of organic chemistry into a set of reactions that look like this.

Once you've defined the data structure for your molecules, AlgebraicRewriting lets you define rewrites in terms of those molecules like here.

CLICK Now with these defined, you can initialize a state of some system (a bath of molecules) and apply the rewrites at predetermined frequencies

You can then chart the trajectories of the various concentrations of molecules. Here's An example showing how all three rewrites rules could be instantiated to this particular

Molecule and be applied to produce different results.

Generating structures via a grammar

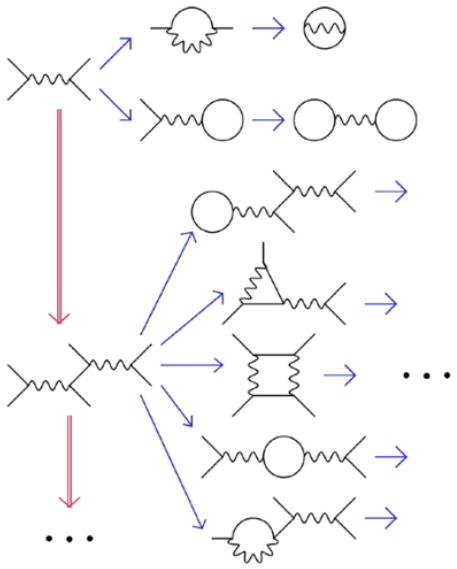
Explore/sample a space of instances of some data structure

Scientific model exploration:
- Explore chemical reaction networks to find one that best fits the data

Model-driven development:
- Check that a robot cannot enter a bad state via transition sequence

Property based testing / verification:
- Check that code which processes an X does not hit edge cases in diverse sampling of X's.

Generating Feynman diagrams



Marcolli and Port. "Graph grammars, insertion lie algebras, and quantum field theory." *Mathematics in Computer Science.* (2015)

The next application is generation of structures via a grammar. Here, we could have a variety of intentions that are associated with exploring some possible space of structures.

For example, if I define a datatype for my scientific model of interest (perhaps a chemical reaction network, because I have chemical concentration time series data),
I want to generate hypotheses to test against that data, but I don't want to generate things completely randomly. Defining rewrite rules that smartly explore the space
Of chemical reaction networks is very helpful here.

Or we can represent the state of a robot and define rules such as "if you see this, do that" and explore the possible states the robot can get into.
This is high-level planning that can root out some nasty conceptual flaws at the best possible time, which is before any code needs to be written.
This paradigm is sometimes called model-driven development.

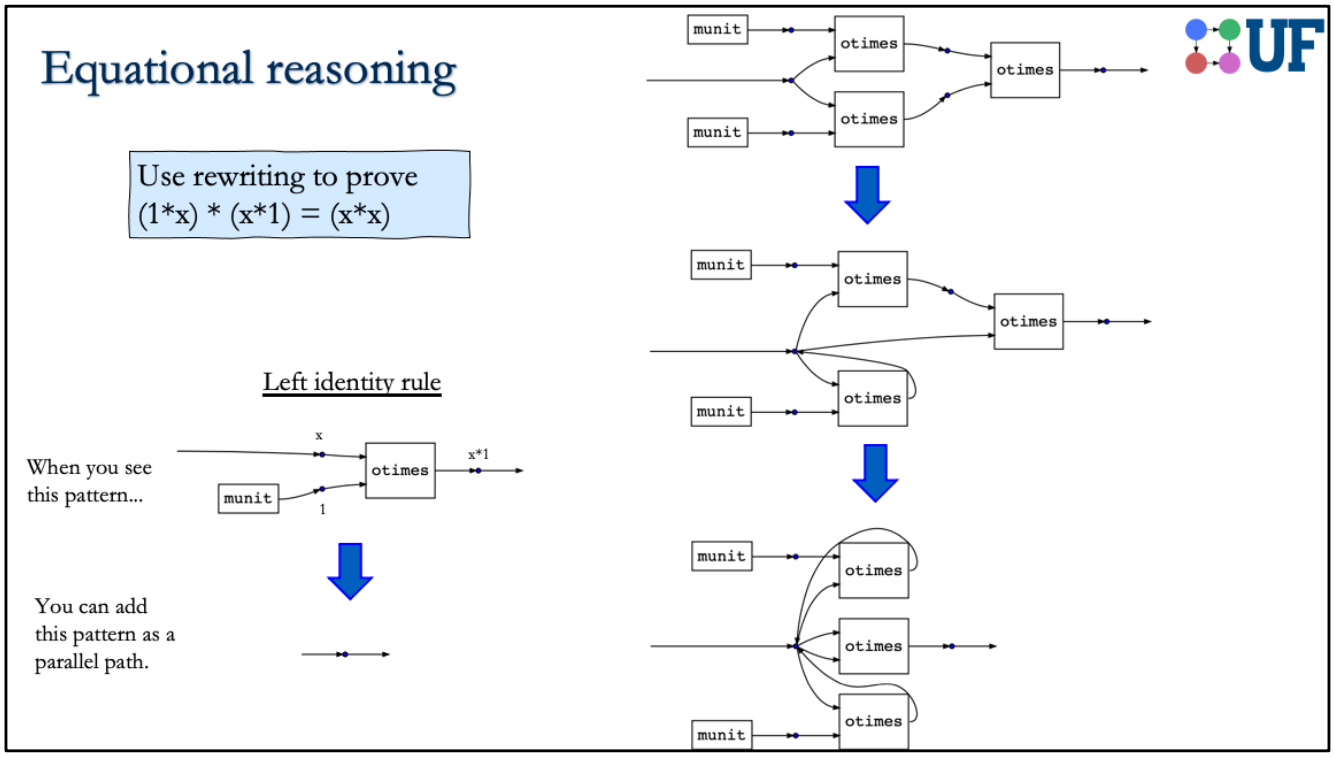
And relatedly, you might want to have a controlled exploration of some rich datatype if you're writing code which is supposed to handle
Instances of such a datatype. This is generally more robust than simple unit tests

where you pick a particular input to test your function

On and then hope for the best.

As an example, I have on the right a figure from a paper that shows a deep connection between Feynman diagrams and graph grammars,

Showing how you can generate all Feynman diagrams by applying some simple rules.



The last example I want to highlight is equational reasoning. This is 1*x*x*1

*click*Here the semantics of a rewrite rule is to say: X is equivalent to Y means I can replace X with Y whenever I see it.

However we can also do this in a non-destructive way when we represent expressions as wiring diagrams, where you view information flowing from left to right, and the boxes as operations.

For example, here we have the expression X times 1, and if we were to assert an equivalence between this and a bare wire which simply passes along its input, this would be

Equivalent to adding a wire which connects “x” to “x*1”, which tantamount to asserting their equivalence in this wiring diagram language.

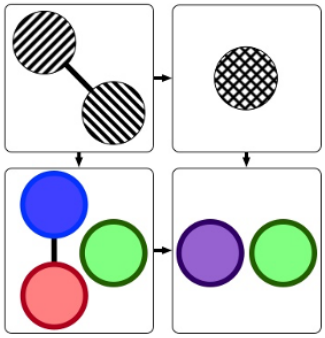
CLICK So now let’s apply this to a real term which is 1*x times x*1

CLICK we see that applying the rewrite rule has added the information of the left identity rule to our graph by collapsing certain nodes together.

CLICK applying a right identity rule allows us to see that our result is computable merely as x*x which is an optimized program relative to the starting point.

This process is also called “equality saturation” in the language of e-graphs, which is a great technique for maintaining lots of information about equivalent expressions in a compact way.

Outline



AlgebraicRewriting.jl

Declare and execute rewriting systems

What problems can it be used for?

Controlled data transformation

How do you use it?

Draw pictures!

What advantages does it have?

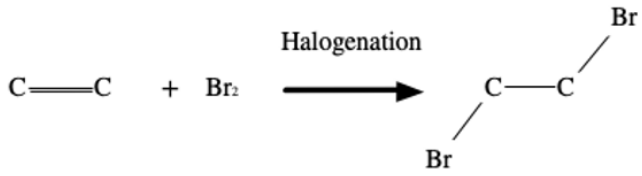
Transparency
Generality
Transferrability

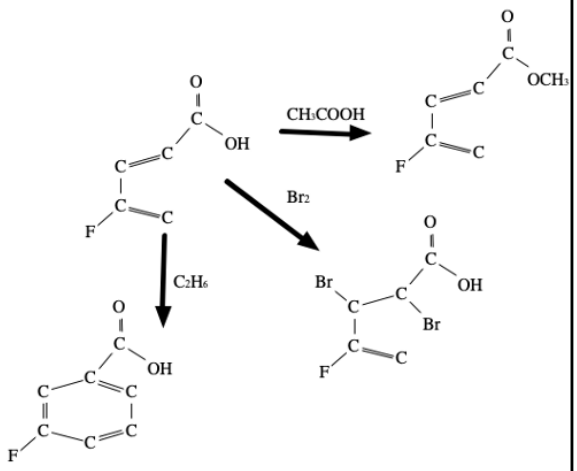
Hopefully that showed that these kinds of methods can be applied in diverse settings.

Now I want to talk more about a particular example to show what it's like to code up such a rewriting system.

How to use it - chemistry example

1. Pick a C-set to represent one's data structure
2. Express the patterns and replacements as a set of rewrite rules
3. Construct a schedule to perform the rewrites





So I'll walk through the process of using AlgebraicRewriting to perform a chemistry simulation.

The three major steps are presented here. We'll start with defining the data structure which we will perform rewrites on.

Background: What are C-Sets?

Generalizes a broad class of data structures,
- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Parameterized by "C", which is a schema.
- Assign a set to each vertex in C
- Assign a function to each edge in C

So I need to introduce C-sets, which are an important kind of datatype that we widely use in AlgebraicJulia. We're not limited to rewriting on C-Sets but as we'll see, they are very general and suffice for most practical applications

They generalize many flavors of graphs, tabular data, as well as combinations of the two. Because they are at least as expressive as relational databases, that's why I say they are expressive enough for practical purposes.

A C-Set isn't a specific data type, but rather once you provide a "C", you get a data type. This "C" plays the role of a schema for a relational database.

The "C" comes from "Category" because the things we do with our data structure are based on the math of category theory, but don't worry you don't need to know any category theory to fluently use these data structures

Background: What are C-Sets?

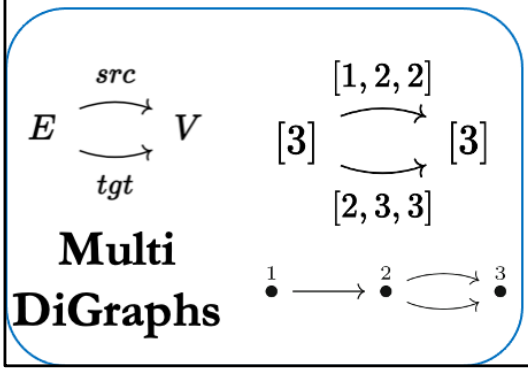
A data structure like an in-memory database

Generalizes a broad class of data structures,

- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Parameterized by "C", which is a schema.

- Assign a set to each vertex in C
- Assign a function to each edge in C



Here on the bottom left is the schema for graphs. If you plug this in as "C", a C-set contains the data of a graph. Note we are using that strategy of representing sets as just vectors [1,2,3,...] and functions as vectors. If we want to represent this graph here at the bottom, we say that the edge and vertex sets have three elements, and these functions here define source and target.

Background: What are C-Sets?

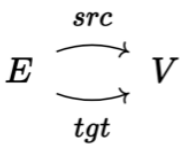
A data structure like an in-memory database

Generalizes a broad class of data structures,

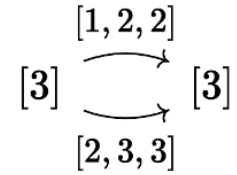
- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

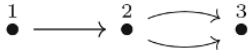
Parameterized by "C", which is a schema.

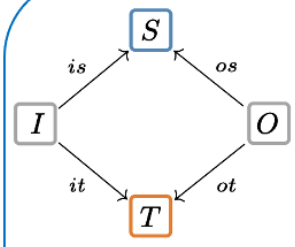
- Assign a set to each vertex in C
- Assign a function to each edge in C



Multi DiGraphs







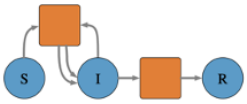
State	Name
1	S
2	I
3	R

Input	is	it
1	1	1
2	2	1
3	2	2

Output	os	ot
1	2	1
2	2	1
3	3	2

Transition	Name
1	infection
2	recovery

Bipartite multi digraphs

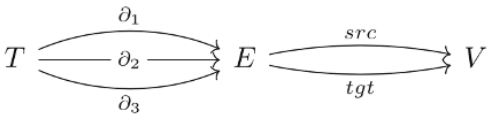


11

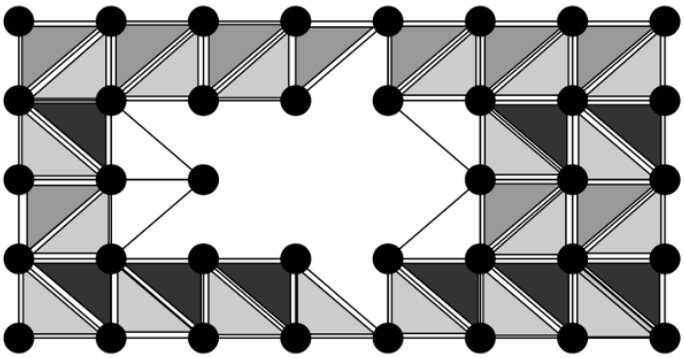
Slightly more complicated is this schema for bipartite graphs. We visualize these by making one vertex type blue and the other orange, and we have two types of arrows each with their own source and target functions.

2D Semi-simplicial set C-set

Indexing Schema: Δ_2



$$\begin{aligned} \partial_1; src &= \partial_2; src \\ \partial_1; tgt &= \partial_3; tgt \\ \partial_2; tgt &= \partial_3; src \end{aligned}$$



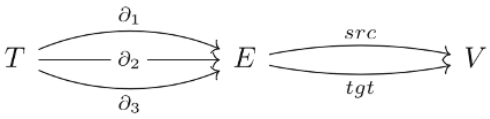
A data structure like an in-memory database, with path equations

12

This C-Set allows us to talk about graphs where SOME triples of edges can form triangles. Note we can have path equations in our schema, which is expressibility not present in databases. These equations make sure our triples of edges actually form triangles.

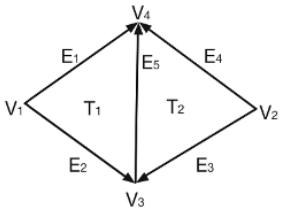
2D Semi-simplicial set C-set

Indexing Schema: Δ_2



$$\begin{aligned} \partial_1; src &= \partial_2; src \\ \partial_1; tgt &= \partial_3; tgt \\ \partial_2; tgt &= \partial_3; src \end{aligned}$$

Example instance



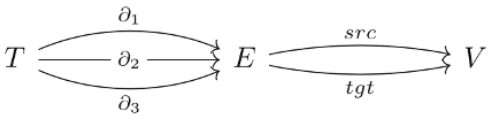
A data structure like an in-memory database, with path equations

13

Here's a smaller example of something we can represent. Two triangles which share one of their edges.

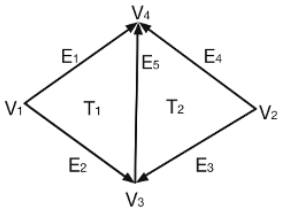
2D Semi-simplicial set C-set

Indexing Schema: Δ_2

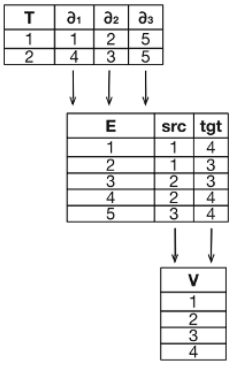


$$\begin{aligned} \partial_1; src &= \partial_2; src \\ \partial_1; tgt &= \partial_3; tgt \\ \partial_2; tgt &= \partial_3; src \end{aligned}$$

Example instance



Database representation



T	∂1	∂2	∂3
1	1	2	5
2	4	3	5

E	src	tgt
1	1	4
2	1	3
3	2	3
4	2	4
5	3	4

V
1
2
3
4

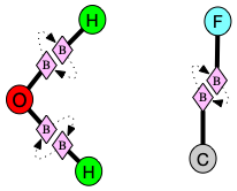
A data structure like an in-memory database, with path equations

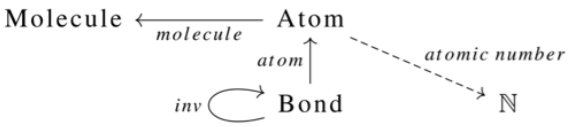
14

The database style representation looks like this. Note we have two triangles, five edges, and four vertices.

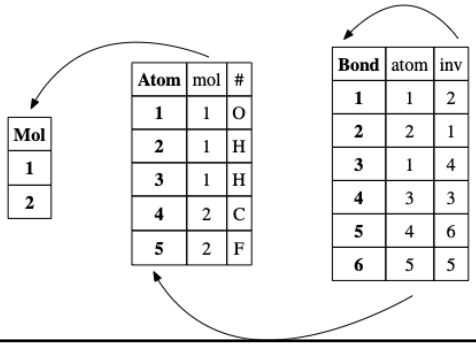
Chemical Species C-Set

```
using Catlab, Catlab.Theories, Catlab.CategoricalAlgebra

@present TheoryChem(FreeSchema) begin
  (Molecule,Atom,Bond)::Ob
  inv::Hom(Bond, Bond)
  atom::Hom(Bond, Atom)
  mol::Hom(Atom, Molecule)
  Num::AttrType
  atomic_num::Attr(Atom, Num)

  inv ⋅ inv = id(Bond)
end

@acset_type Chem_Generic(TheoryChem)
const Chem = Chem_Generic{Int}
```





$$inv; inv = id_{Bond}$$



Mol
1
2

Atom	mol	#
1	1	O
2	1	H
3	1	H
4	2	C
5	2	F

Bond	atom	inv
1	1	2
2	2	1
3	1	4
4	3	3
5	4	6
6	5	5

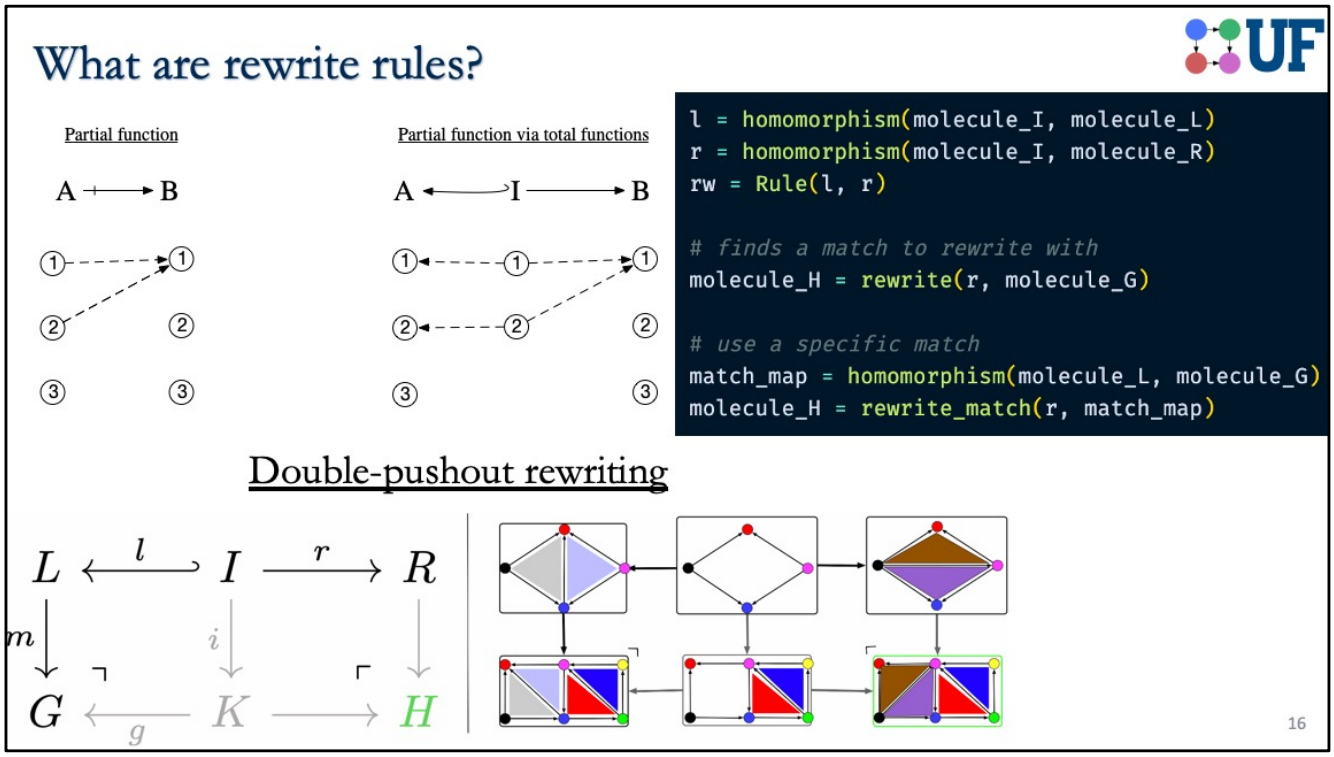
15

So now let's return to our chemistry problem. The trickiest issue here is representing chemical bonds as undirected. Although directed graphs were very easy to represent, with two sets and a src+tgt function, undirected graphs can be encoded by having a Bond table where two different rows of the table get used to represent a single bond.

We require an involution function on the set of Bonds to make sure that they are paired up.

The code on the screen shows how Catlab lets you declare a custom data type like this.

CLICK You see here we've represented a pattern with a water molecule and a C-F bond. Constructing instances of this datatype can be done in many ways, ranging from low-level imperative operations to high level constructions. t In any case, you just need to construct this database instance here. Now we want to create the rewrite rules.



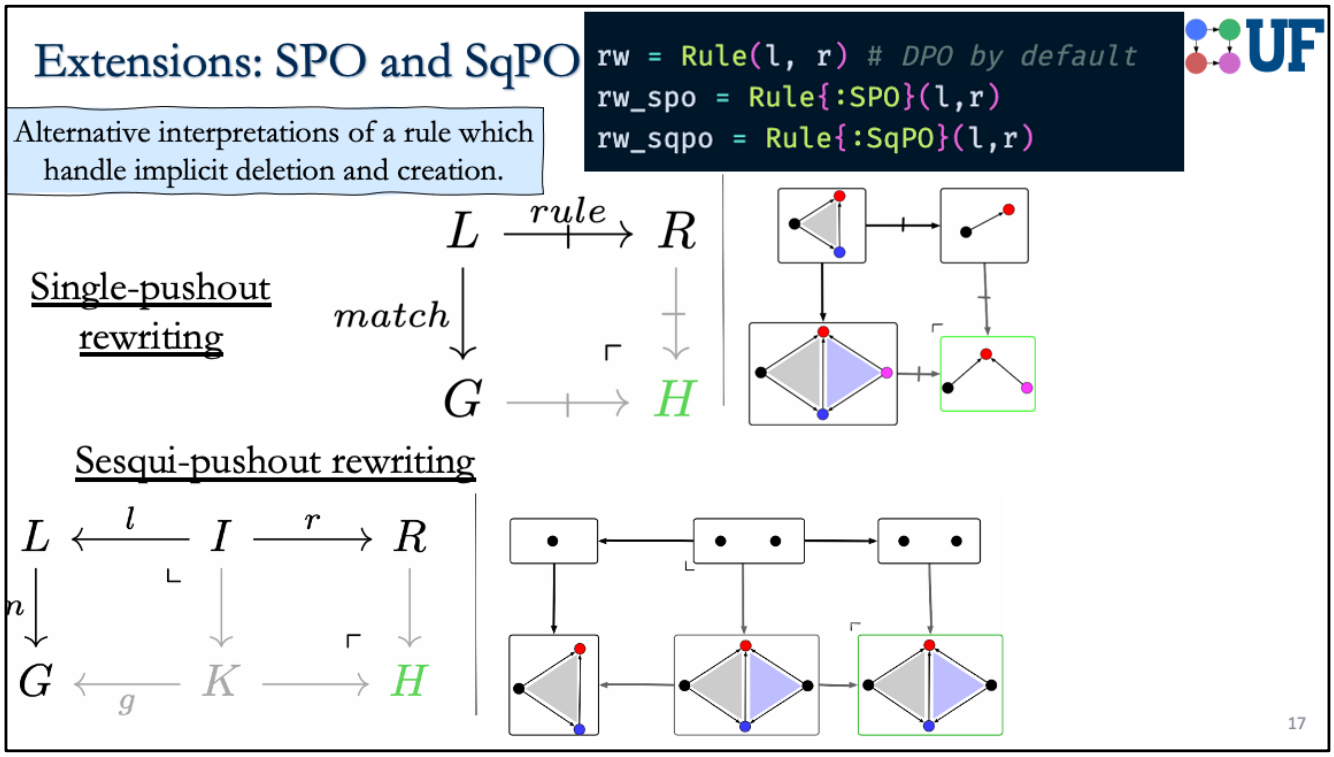
We need a brief detour to explain the mathematical theory of graph rewriting. This requires us to have a notion of a partial mapping from one thing to another. Let's start with the most familiar case of this, which is partial maps between sets. It's just a function that does not need to be defined on all inputs. Interestingly we can represent such a partial function using two total functions, as shown on the right. Note we also require the leftward facing function to be injective for this to make sense.

CLICK With that, we can talk about a C-Set rewrite rule as being defined by simply a partial map from L to R. We have a left hand C-Set, which is the pattern we want to match to something within the instance we are rewriting. We have the intermediate C-Set "I", which identifies the things we are NOT deleting, and then we are mapping the things that we don't delete into R, which is the thing we are replacing the pattern with.

Given a match "m" into our data that we are trying to rewrite, the pushouts of the name are very general ideas from category theory that allow us to write one implementation for rewriting that can be specialized in a very broad variety of contexts. No deep understanding of how to define this generically is needed though. You can see in this particular example we are matching a pair of triangles and flipping the internal edge, and in our actual system of interest we are applying that transformation to the leftmost pair of triangles.

click you can see the library exposes this high level rewrite interface. You get a

Rule by declaring a partial function as a pair of maps. Note that Catlab can find the map for you, and you can give it extra information to narrow down the options if it's ambiguous. You then can either explicitly provide a match or let it be found automatically. In any case, it's easy to perform an individual rewrite!



Before moving on to the scheduling of some set of rewrites as a rewrite system, I want to draw attention to some alternatives to double pushout rewriting. DPO is usually sufficient, but there are some alternative semantics we can give our rewrite rules. First we discuss single pushout rewriting, where I'm omitting the "I" column of the partial maps for brevity. The difference between SPO and DPO is how you handle the following edge case: what happens when you delete something that is connected to other things. In the case of this picture, I'm deleting the blue vertex, yet the blue vertex is connected to part of the mesh that is not matched by our L pattern. We aren't told explicitly what to do! DPO would simply forbid this as a rewrite – it only lets you delete things that you're implicitly deleting. SPO is more liberal about things and will CASCADE delete anything that is incident to something that gets deleted.

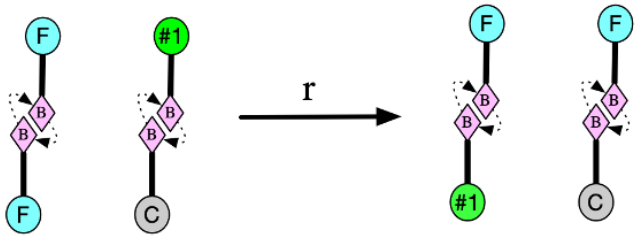
click Sesqui pushout rewriting is the third major paradigm of rewriting, and the practical difference with the previous two is that it let's you copy things implicitly. So if I take this triangle and I duplicate the point on the left, imagine me dragging it over to the right hand side – SqPO will impliclty copy everything incident to that point, meaning we get a whole other triangle generated by this rewrite.

*click*There are situations in which different rules call for different kinds of rewriting, even in the same simulation. This is why the semantics of the rewrite rule is data attached to the rule itself, and the interface for using the rule is

identical.

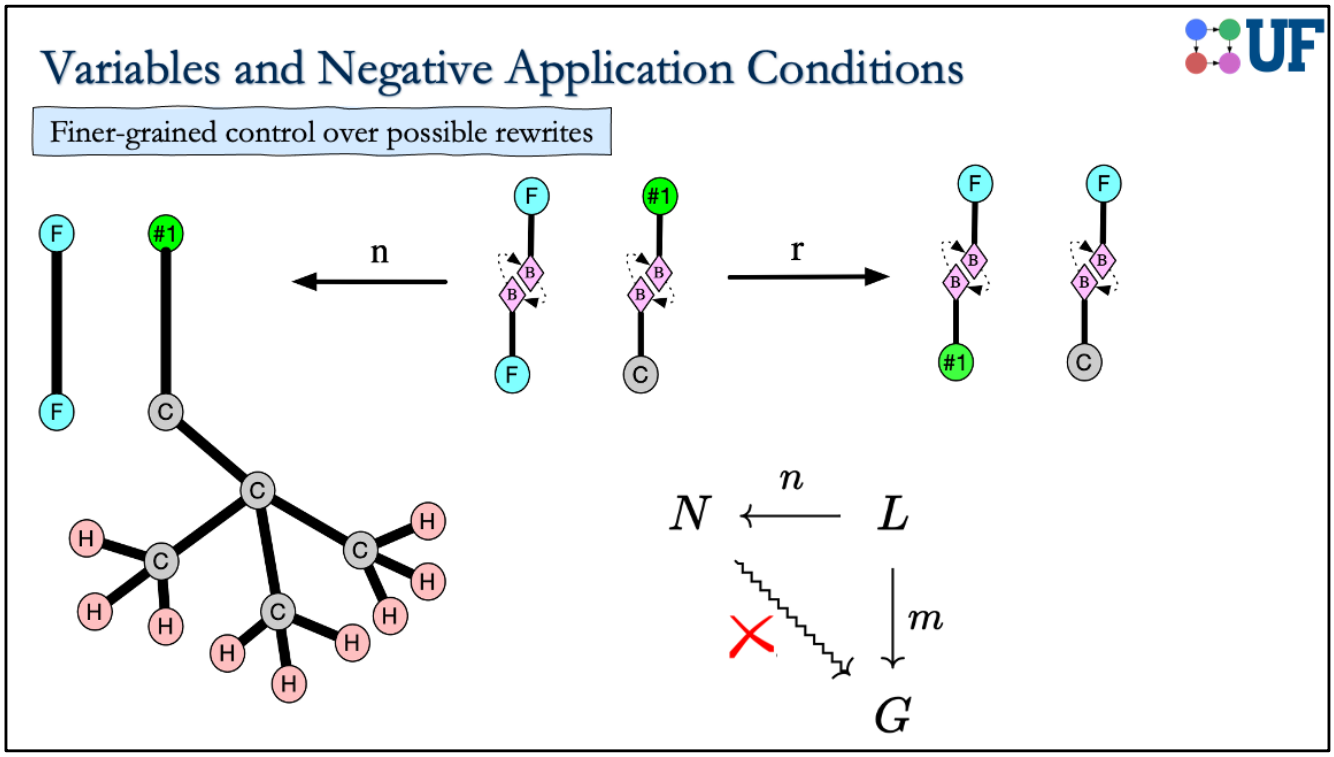
Variables and Negative Application Conditions

Finer-grained control over possible rewrites



You can get a lot of mileage out of the language I've just described, but I've found that more features are helpful for using rewriting to accomplish practical computations.

Firstly, you may want to match certain attributes without knowing exactly what they are ahead of time. In the case of our chemistry example, the only attribute in the schema is the atomic number (in a database, we think of attributes as plain old columns with real-world data, rather than foreign keys pointing to some other table). Technically speaking, maps between C-Sets are required to agree exactly on attributes, so if you label an atom as carbon in your pattern, it can only match with carbon atoms. But what if we want to



Negative application conditions add an extra map to the rewrite rule with the following meaning: it embeds the pattern L in some larger context that specifies when NOT to apply the rule. Basically, when we find a match to G, we need to check if there exists a map from N to to G such that this triangle commutes, i.e. does there exist a squiggly map such that taking the 'n' morphism and then the squiggly morphism is identical to taking the match morphism 'm'.

A chemistry-relevant instance of this is the "tert butyl" group, which chemists use to prevent reactions from happening at certain parts of a molecule they wish to prevent. It's a big bulky thing that blocks even reactive molecules like fluorine. By adding this NAC, we can better represent the underlying chemistry.

Scheduling

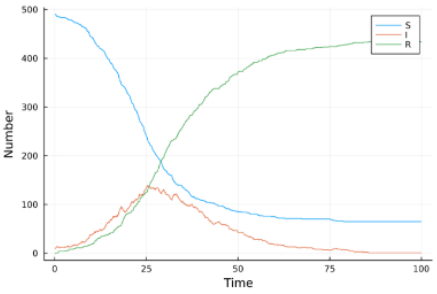
```
rules = [rule_1,rule_2,rule_3]
LinearSchedule(rules)
RandomSchedule([rule_1,rule_1,rule_2,rule_3])
WhileSchedule(rand)
LinearSchedule([WhileSchedule(rand), rule_4, RandomSchedule(rules)])
```

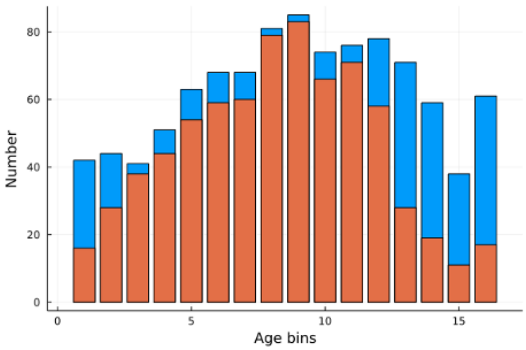
[1,2,3]

[3,1,2] [2,3,1]

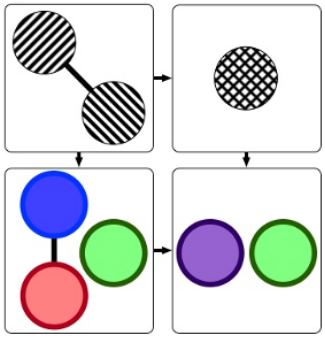
[3,1,2|1,2,3|3,2,1|1,2]

[3,1,2,1,2,3,3|4|1,3,2]





Outline



AlgebraicRewriting.jl

Declare and execute rewriting systems

What problems can it be used for?

Controlled data transformation

How do you use it?

Draw pictures!

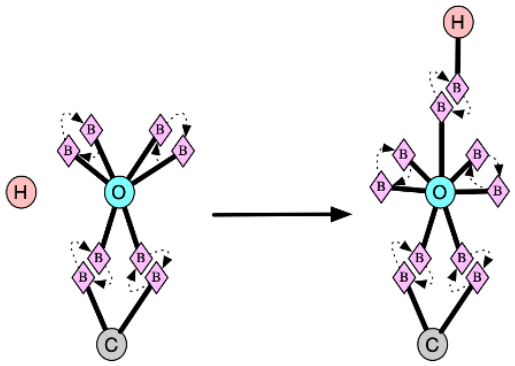
What advantages does it have?

Transparency
Generality
Transferrability

Advantages of the algebraic approach

Primary alternatives:

- Rewrites as standalone functions + manual update of state
- Domain-specific simulation software
- OOP-style agent-based modeling

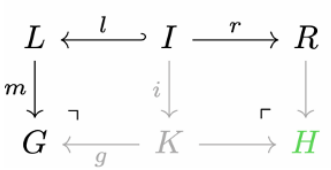


Generality:

- DPO just requires your datatype to implement notions of 'pattern match', 'pushout', and 'pushout complement'

Transparency:

- We can verify independently whether each of our rewite rules preserves the "octet rule"
- Rewriting via DPO/SPO/SqPO induces a partial map from starting instance to the rewritten instance



If we noticed after doing our simulation for 100 steps that the octet rule was not being satisfied by some of our molecules (suppose we expect this to be the case), it could be an ordeal to search through source code to try to identify the root cause of the bug.

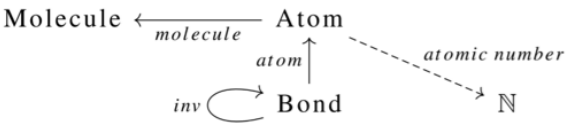
However, with our dynamics specified by the rewrite rules expressed as concrete structures, we can write a simple function to go through our rewrite rules and identify which one is causing the problem – data is much easier to analyze with automation than code.

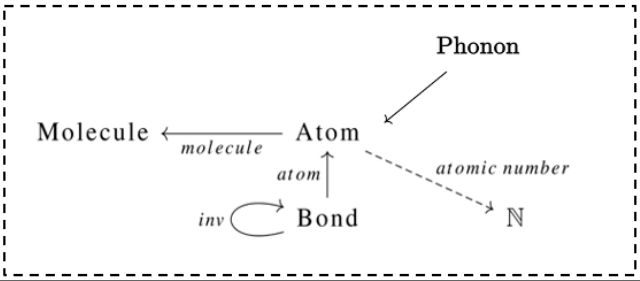
click On the subject of transferability, I want to highlight a few variations we could make on our schema.

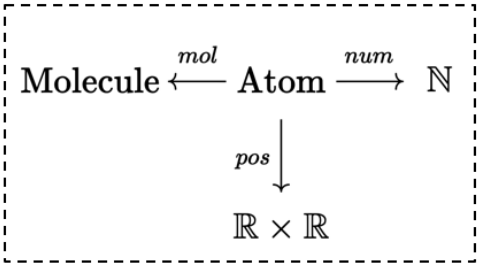
Advantages of the algebraic approach

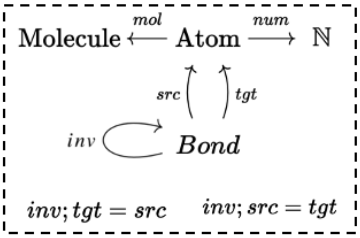
Transferability

- Various modifications we can make to schema
- For 'functorial' data migrations, we have mathematical guarantee that dynamics are unchanged, i.e. it is a faithful representation of original
- Data migration done declaratively at the schema level, computation is automated by Catlab.jl





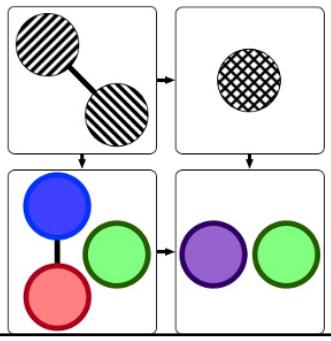




Future work

Library is still in its infancy, many planned improvements:

- More advanced scheduling
- Support for debugging rewrite rules (see *why* a match was rejected that you expect to take place)
- Performance (incremental search for matches, rather than restarting after each rewrite)
- Search up to isomorphism (remove symmetries in results to avoid double counting)
- More robust parallelization of rewrite execution



AlgebraicRewriting.jl

Sounds interesting? Let us know!

I'd like to thank all these people for welcoming me and being such fun collaborators!