UF

Computational category-theoretic rewriting

Kris Brown, Tyler Hanks, Evan Patterson, James Fairbanks

08.07.22

Hi, so we've written a quite applied paper which I want to talk about, but I also want to take this chance to share our broader vision of applied category theory.

So in fact I'll start with that overview to give some context into how we view this graph transformation project.

Status quo: Model as opaque code

```
"""
2 H2 + O2 → 2 H2O. Mass-action kinetics. Compare to experimental data and plot.
"""
def main():
    # experimental data
    real_data = [0.0101, 0.012, 0.023, 0.037, 0.045, 0.053, 0.061, 0.069,
    0.076, 0.083, 0.089, 0.096, 0.102, 0.108, 0.114, 0.119, 0.125, 0.130, 0.135,
    0.140, 0.145, 0.150, 0.154, 0.159, 0.163, 0.167, 0.171, 0.175, 0.179, 0.183,
    0.186, 0.190, 0.193, 0.197, 0.200, 0.203, 0.206, 0.209, 0.212, 0.215, 0.218,
    0.221, 0.224, 0.227, 0.229, 0.232, 0.234, 0.237, 0.239]
    # Initial concentrations
    H2,O2, H2O = 1.0, 2.0, 0.0
    dt = 0.01
    results = []
    for step in range(1,50):
        print("Step ", step)
        rate = 0.5 * H2**2 * O2
        H2 -= 2*rate*dt
        O2 -= rate*dt
        H2O += rate*dt
        results.append(H2O)

    plot(results, real_data) # Figure 4 in the paper
```

Many tasks we'd like to do cannot be done with arbitrary code (nor mathematical expressions).

- Update/repair code when assumptions change
- Explore alternate reaction networks to fit the data
- Generate the entire code from just declaring the reaction
- Check if another model is the same / a submodel
- Easily alter semantics (e.g. stochastic-based simulation)

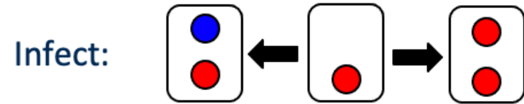
So we are trying to reform various practices in the real world, where a lot of scientific and engineering tasks are being represented in very opaque formats that make automatic reasoning and analysis nearly impossible. An example is the random scripts that a scientist might string together to perform a simulation of a chemical reaction network, as shown here. Although it might seem like a rigorous model in the language of mathematics or formal logic would be an improvement, we actually view these on par with each other, as they are all perfectly formal languages, perfectly powerful syntaxes, and therefore very hard to reason about.

Some brave people work on that, but, rather, we want to design software that allows people to work in restricted syntaxes that can be reasoned about, so that we can do things like update assumptions in a meaningful way, explore spaces of models, generate code automatically, check for equality and substructure relations, and easily alter the semantics.

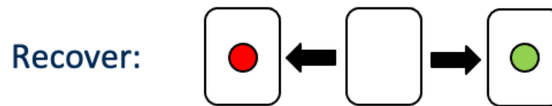
Our paradigm example for this kind of thing is the representation of chemical reaction networks as Petri nets.

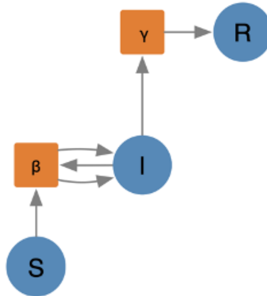
Model as functor from syntax category

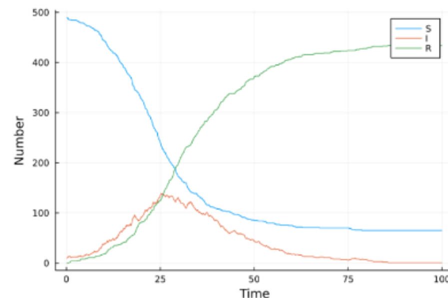
Infect:



Recover:







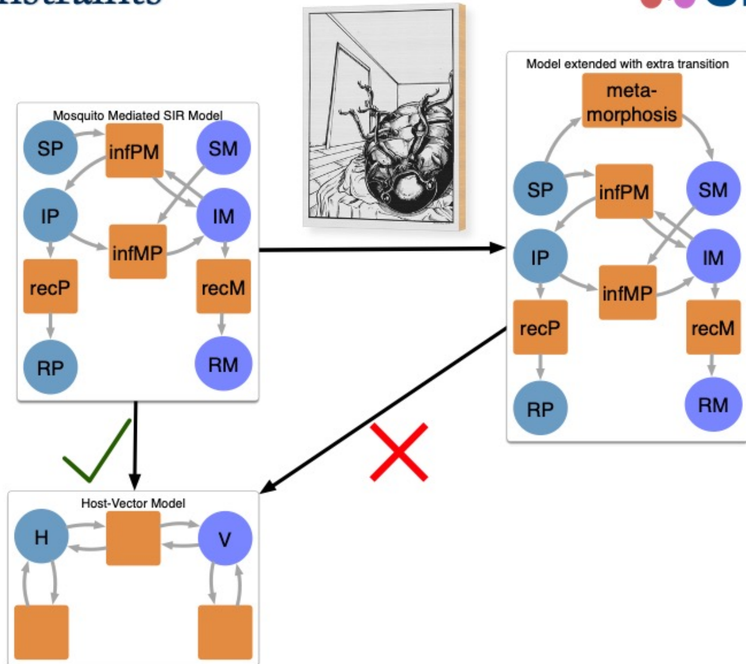
Specifying a reaction network as a Petri Net allows for automatically generating a simulator program.

Here's an example with the SIR epidemiology model, where the blue dots say that there exist susceptible, infected, and recovered people. The left box is a transition that says a susceptible person can combine with an infected person to make two infected people, and the other box says infected people recover at some rate.

We can actually use a trivial version of DPO that operates on sets rather than graphs and generate rules which can perform a discrete time simulation. Looking at each transition gives us such a rewrite rule.

Slice categories capture constraints

- Homomorphisms effectively assign "types" to elements of the domain model.
- Homomorphism search is type inference



Demanding that a C-Set morphism exists into some fixed model is a constraint.

Libkind et al. "An algebraic framework for structured epidemic modeling" (2022)

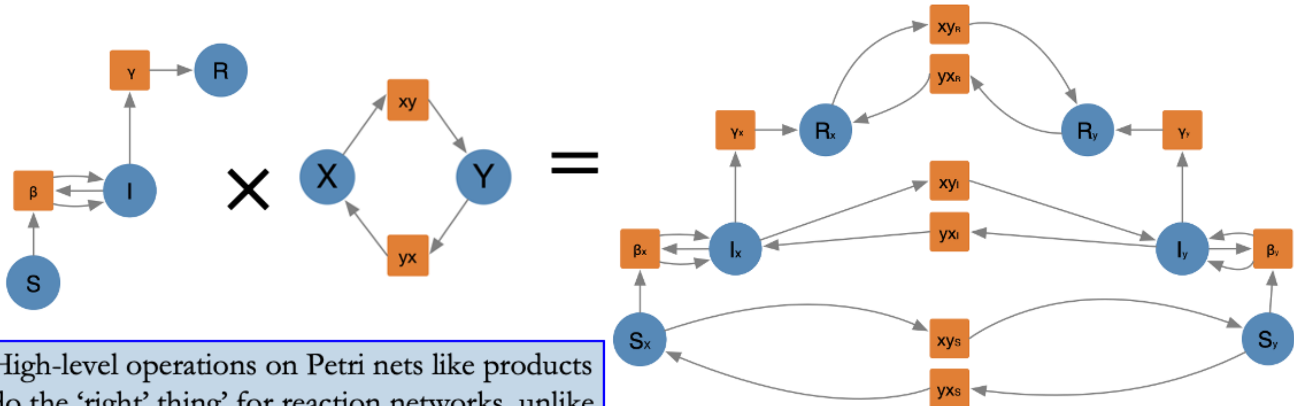
4

We can also have a syntactical notion of constrained Petri nets by considering slice categories. The set of Petri nets with a morphism into this particular model here are those which have labeled their states as either host or vector, and the only allowable transition types are ones that have one host in, one host out, one vector in one vector out, or a host and vector interacting. If you said "this is my class of models" and handed this off to your coworker who then added a transition which converts hosts into vectors, you'd have an automatic procedure for checking whether or not it's a valid model, which is something we lose when we say the model is the code that performs the simulation, or if we work in a completely unconstrained mathematical language like ODEs.

JAMESHAS BETTER PIC

Pullbacks characterize stratification

- How to combine basic ideas for models into complex models?

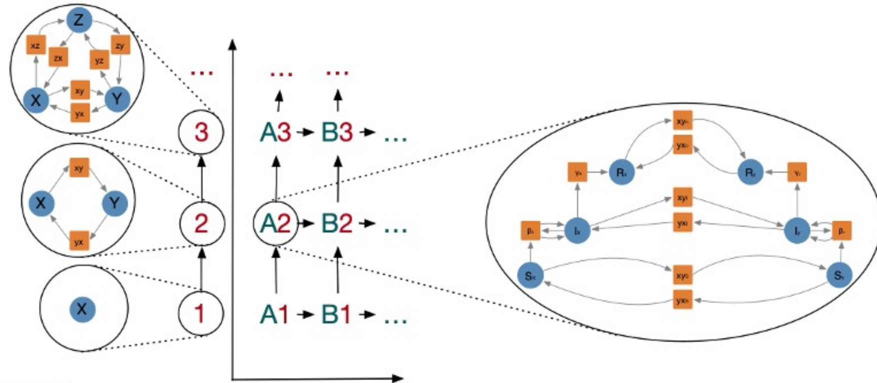


High-level operations on Petri nets like products do the 'right' thing for reaction networks, unlike for symbolic syntax or raw ODEs.

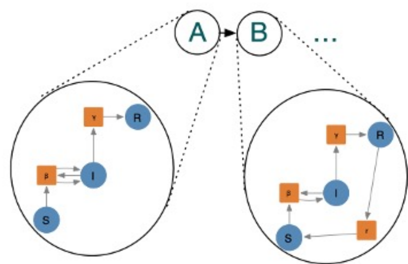
It also turns out that basic notions of limits and colimits correspond to useful concepts when we've modeled our domain as a category.

Functors into model categories are model spaces

“City dimension”



Products of diagrams take different ‘dimensions’ and yield a product model for each combination of elements from each dimension.



“Disease dimension”

6

Here I show not a product of individual models but a product of diagrams, where the category of diagrams in Petri have as objects functors into Petri.

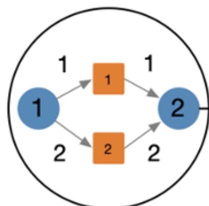
We have an entire ‘dimension’ so to speak of transportation models and a whole dimension of disease models.

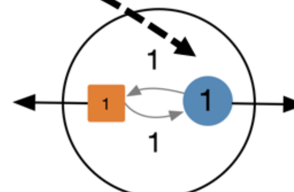
If this process were coded up in a script, it would look like a nested for-loop for each dimension. But now we’ve represented that process algebraically and can compose it with other kinds of model space constructions (which are usually limits or colimits in the category of diagrams, but could also come from graph transformation specialized to this category.

Hierarchical operadic composition

Database queries can be built hierarchically using a wiring diagram syntax. Raw SQL queries are not composable this way.

"Find all catalysts (*that are the product of two reactions*) and the reactions they catalyze"






```
SELECT state2.id
FROM   S AS state1, S AS state2,
       T AS tran1,  T AS tran2,
       I AS in1,    I AS in2,
       O AS out1,   O AS out2
WHERE  in1.is   = state1.id,
       in1.it   = tran1.id
       out1.os = state2.id
       out1.ot = tran1.id
       …
```

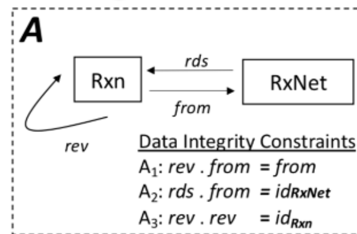
```
SELECT tran1.id, state1.id
FROM   S AS state1, T AS tran1,
       I AS in1,    O AS out1
WHERE  in1.is  = state1.id,
       in1.it  = tran1.id
       out1.os = state1.id
       out1.ot = tran1.id
```

The picture form has a special advantage in that it is is compositional – you could take the query on the left and substitute it into the query on the right.

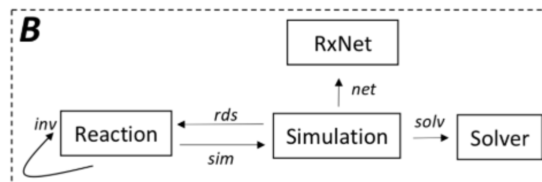
SQL code in contrast, doesn't let you do this kind of substitution (you'd have to write a special purpose algorithm to handle all the edge cases)

Transferability via functorial data migration

Migrate / merge data in different schemas



Data can be automatically moved between different structures when our models change. A tedious and error-prone process otherwise.



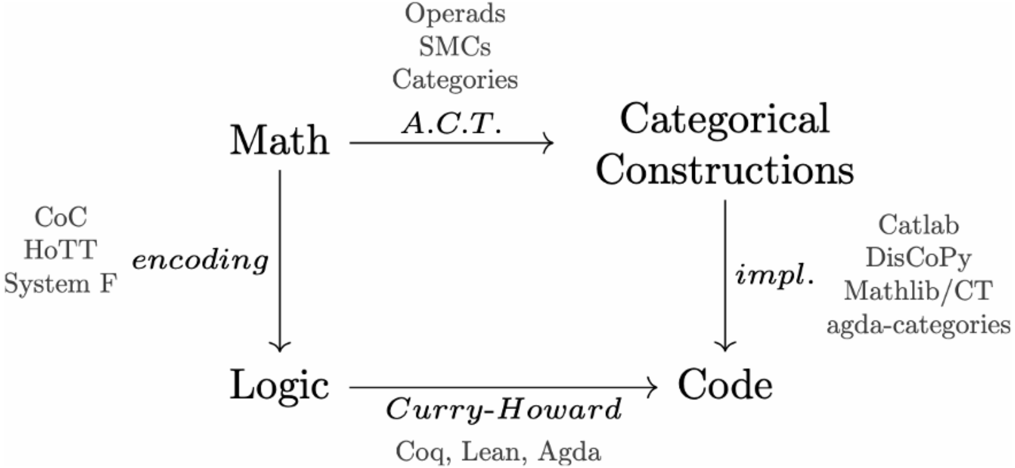
Brown, Spivak, Wisnesky. "Computational Data integration for Computational Science" (2019).

Generality is also related to transferability. If one's model of the world updates and you want to migrate your infrastructure from the old to the new, it's possible to do this knowing just from declaring the relationship of the structure of the old data to the new data.

In particular we can migrate a set of rewrite rules from one model to another if this can be done functorially.

This can't be done when the model is a uniform block of arbitrary programming language code.

Two broad strategies for computational category theory



Outline

Introduction

- A paradigm of computational category theory

Background

- What are C-Sets?
- Relation between C-Sets and typed graphs

Results

- Performance / high-level comparisons
- Extensions
- Applications

Takeaways

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

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 category theoretic definition of a C-set is a functor from C (which is a category) to **Set**. But all that means is that you assign a set for each vertex in C, a function for each edge in C, and you also satisfy equational laws that C has. Let me show some examples.

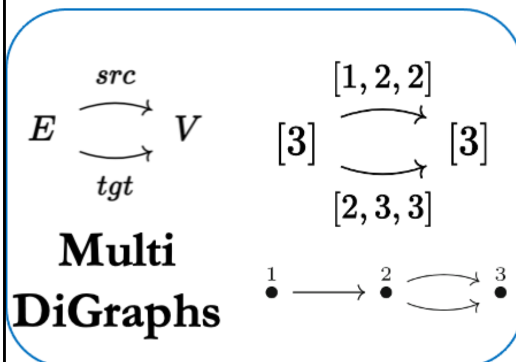
Background: What are C-Sets?

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



Here on the top 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.

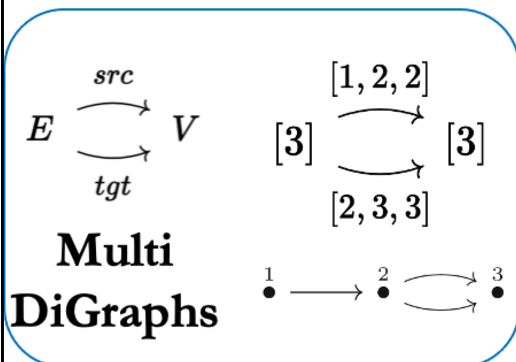
Background: What are C-Sets?

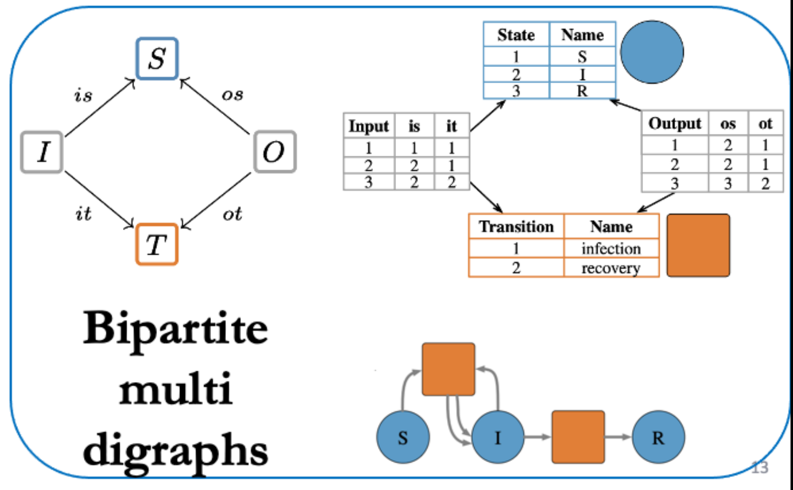
Generalizes a broad class of data structures,

- 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





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

13

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.

Background: What are C-Sets?

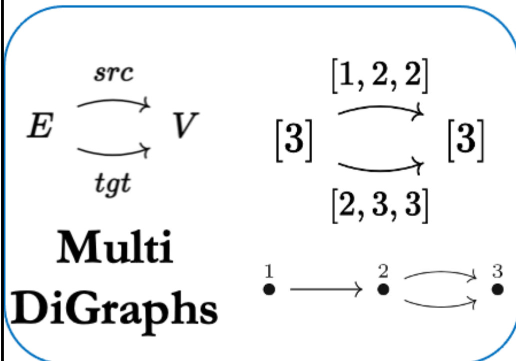
Generalizes a broad class of data structures,

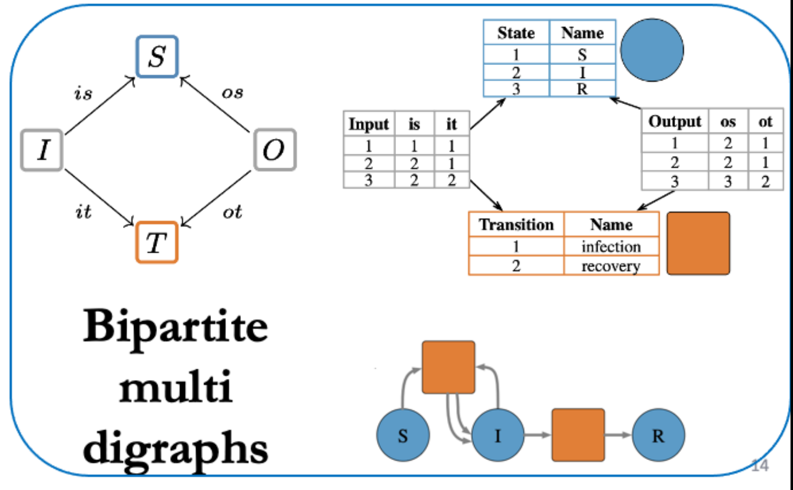
- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Efficient due to Julia's macro system + JIT compilation

Parameterized by "C", which is a schema.

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





14

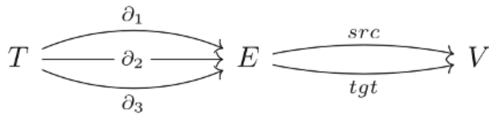
As a side note, despite the variety of things that can be expressed, Julia allows for these data types to be implemented efficiently.

For example, consider the C-set which encodes sparse graphs

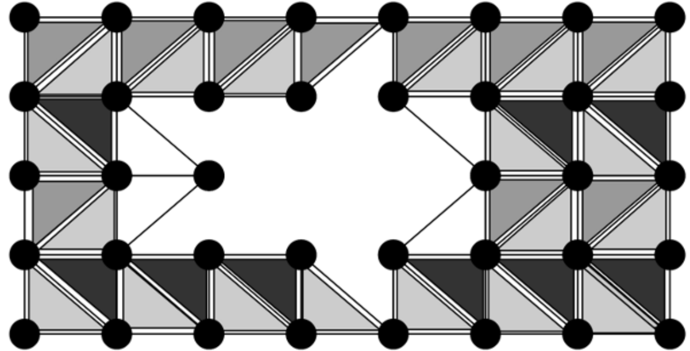
This was benchmarked against Julia's native graph library, Lightgraphs, and performed competitively.

2D Semi-simplicial set C-set

Indexing Schema: Δ_2



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

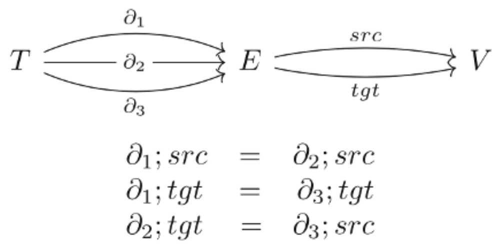


15

Just two more examples. This first one would be a C-set to define two-dimensional semisimplicial sets. This allows us to talk about graphs where SOME triples of edges can form triangles.

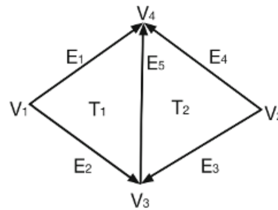
2D Semi-simplicial set C-set

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

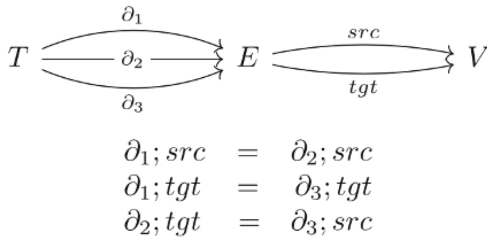


16

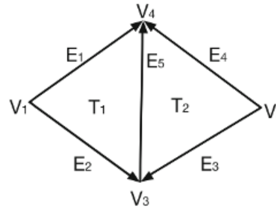
Here's a smaller example of something we can represent. Two triangles which share one of their edges.

2D Semi-simplicial set C-set

Indexing Schema: Δ_2



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

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

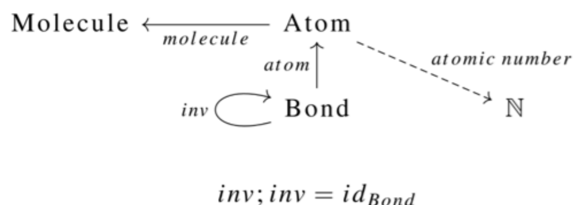
Chemical Species C-Set

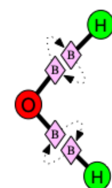
```julia
using Catlab, Catlab.Theories, Catlab.CategoricalAlgebra

@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}
```





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

18

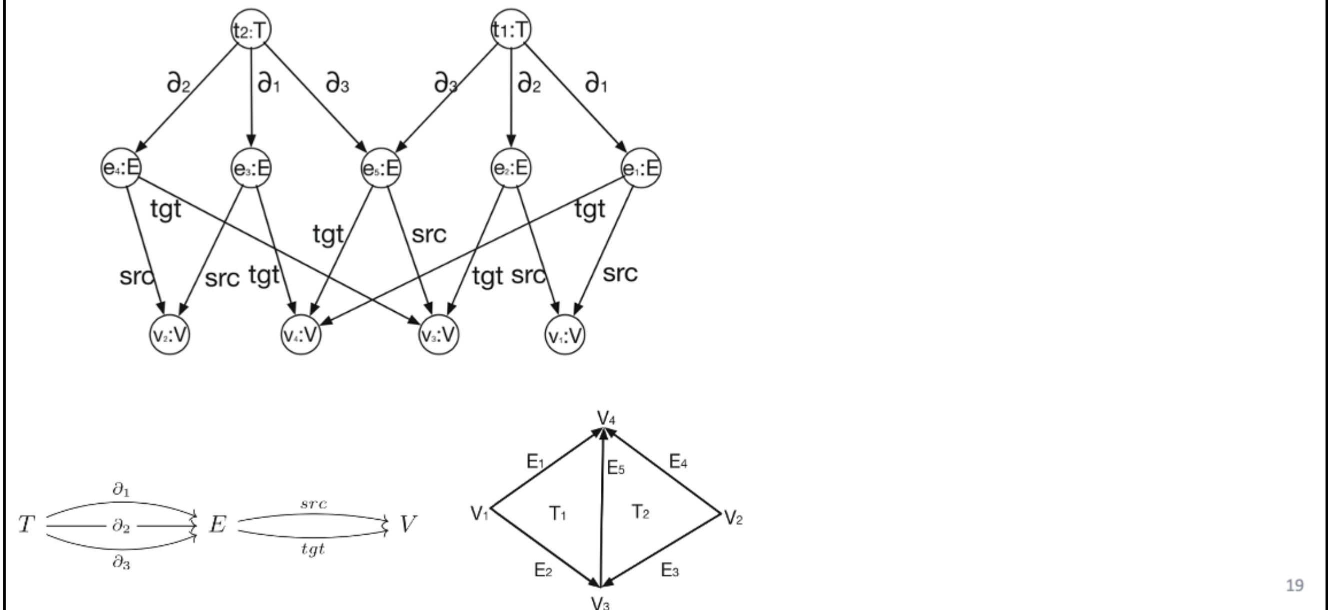
So now let’s return to our chemistry problem. The trickiest issue here is representing chemical bonds as undirected. Although directed graphs were very easy to represent, with two sets and a src+tgt function, undirected graphs can be encoded by having a Bond table where two different rows of the table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

Background: Relation between C-Sets and typed graphs

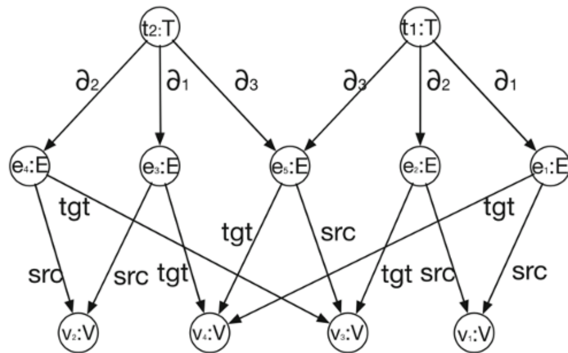


19

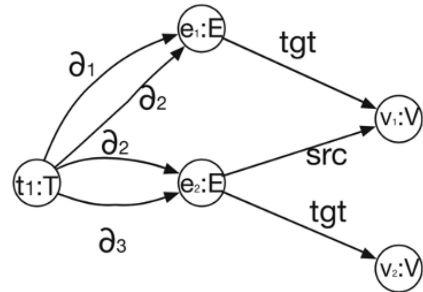
C-sets are a generalization of typed graphs, although they are closely related. We can faithfully convert a C-Set like the one on the bottom into the typed graph above. Because this conversion is faithful in a way that category theory can make precise, many algorithms that operate on typed graphs will compute the correct thing for C-sets.

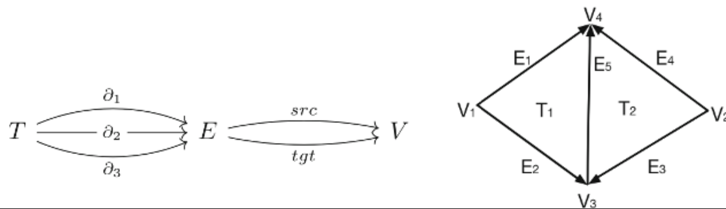
On the right, there is a valid typed graph but it's not a valid C-set for a variety of reasons, for example there are multiple edges assigned as delta-2 of t1, and e1 has zero vertices assigned as its source.

Background: Relation between C-Sets and typed graphs



Valid typed graph, invalid C-Set for our schema

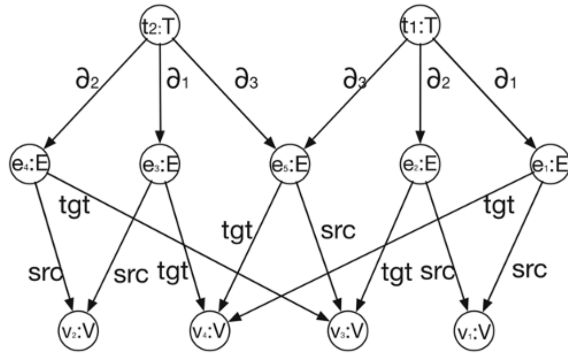




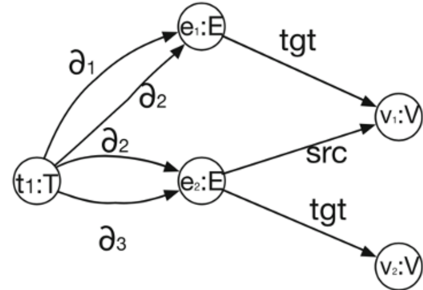
20

On the right, there is a valid typed graph but it's not a valid C-set for a variety of reasons, for example there are multiple edges assigned as delta-2 of t1, and e1 has zero vertices assigned as its source.

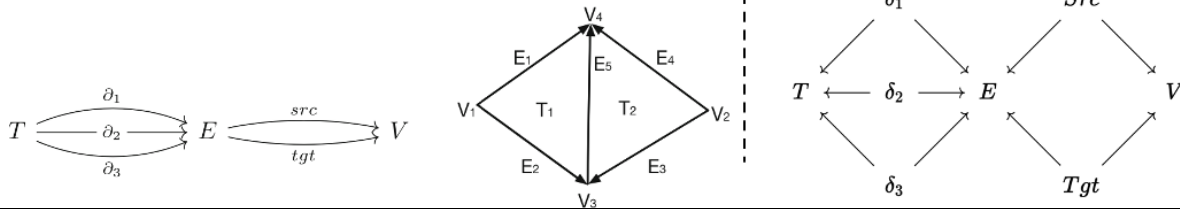
Background: Relation between C-Sets and typed graphs



Valid typed graph, invalid C-Set for our schema



But, representable as a C-Set on this schema:

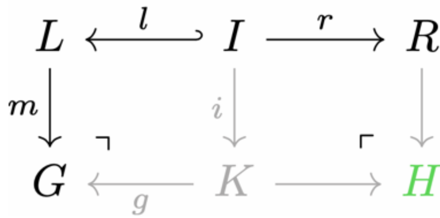


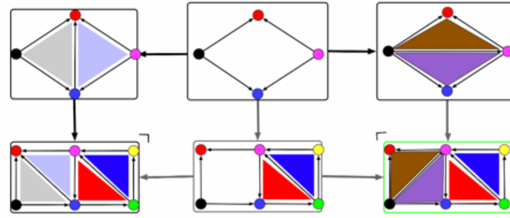
21

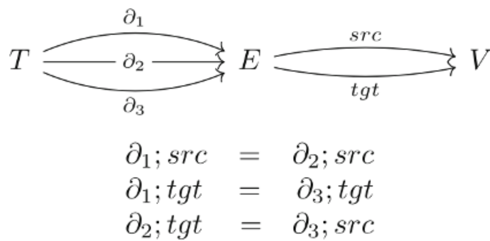
It is a valid C-set on this schema, where we’ve replaced all the arrows with a span of arrows, i.e. replaced functions with relations. So while C-sets can handle the looseness of relations by choice, it is more expressive because it can encode the constraint of functions.

WHY USE GROTHENDIECK TO COMPUTE SECOND SCHEMA BOTTOM RIGHT?

Double pushout rewriting








```
l = homomorphism(mesh_I, mesh_L)
r = homomorphism(mesh_I, mesh_R)
rule = Rule(l, r)

# find a match to rewrite with
mesh_H = rewrite(rule, mesh_G)

# use a specific match
match_map = homomorphism(mesh_L, mesh_G)
mesh_H = rewrite_match(rule, match_map)
```

22

A quick note on performance is that some of the data of typed graphs is encoded STRUCTURALLY in C-Sets, and that allows them to be linearly more memory efficient, and their memory is also layed out efficiently. Something we didn't take advantage of here was the fact that the foreign keys of a database can be indexed, leading to some algorithms having different computational complexity. In fact, finding all homomorphisms is such an algorithm, so we will want to make a mature implementation of that that uses joins.

Extension: SPO

$$
\begin{array}{ccc}
L & \xrightarrow{rule} & R \\
{\scriptstyle match}\downarrow & & \downarrow \\
G & \longrightarrow & H
\end{array}
$$

$T \rightrightarrows E \rightrightarrows V$ (maps $\partial_1, \partial_2, \partial_3 : T \to E$; $src, tgt : E \to V$)

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

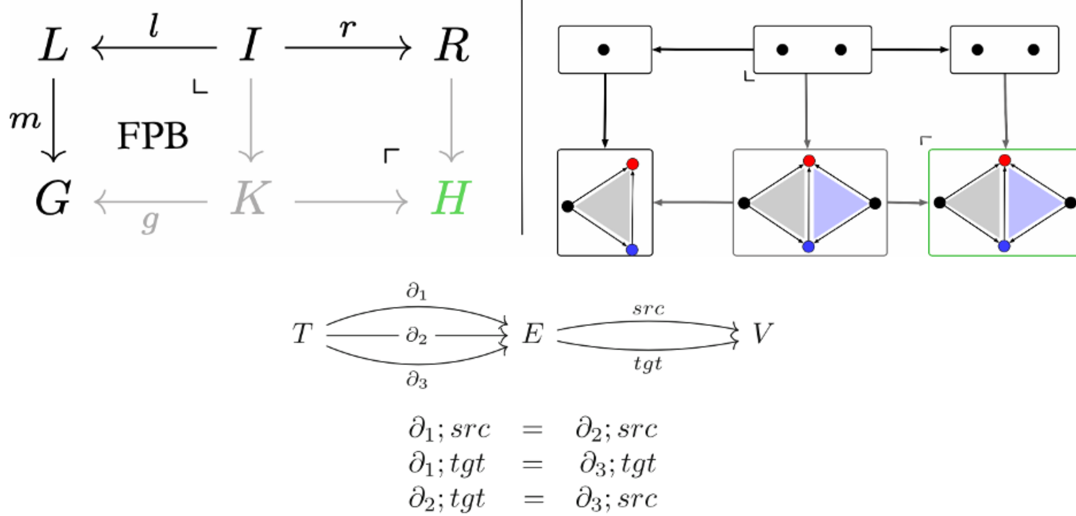
```
rw = Rule(l, r) # DPO by default
rw_spo = Rule{:SPO}(l,r)
rw_sqpo = Rule{:SqPO}(l,r)
```

23

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.

Extension: SqPO



24

The most interesting thing about the sesqui pushout for me was that the equations of the schema

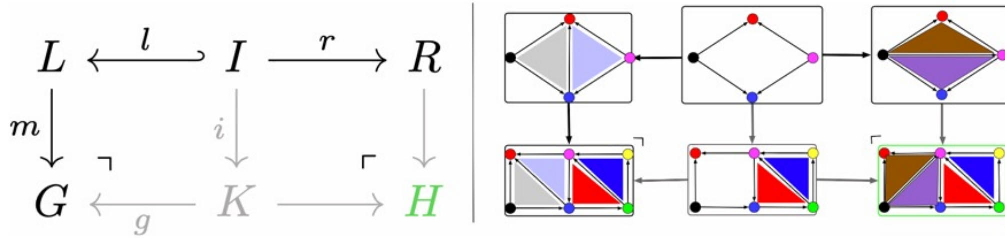
Comparisons

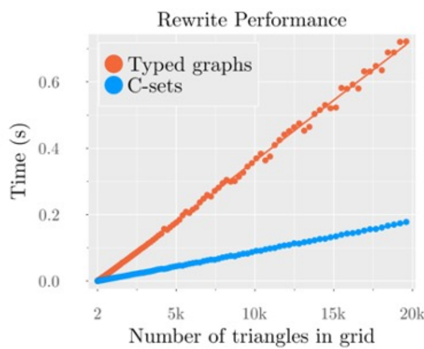
Software	Typed Graphs	$\mathcal{C}$-sets	Rewrite type	CT Env	Last update	GUI	Scripting Env	Library vs. App
AGG[29]	Y	N	S	N	2017	Y	N	Both
Groove[23]	Y	N	S	N	2021	Y	N	App
Kappa[13]	N	N		N	2021	Y	Y	App
VeriGraph[1]	Y	N	D	Y	2017	N	Y	Lib
ReGraph[12]	Y	N	Sq	N	2018	N	Y	Lib
Catlab[11]	Y	Y	D,S,Sq	Y	2022	N	Y	Lib

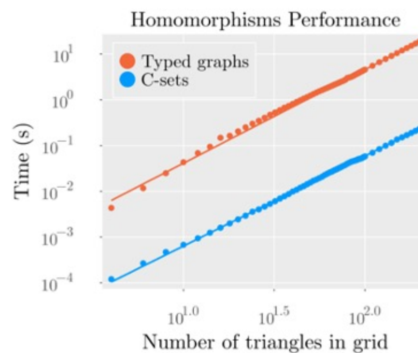
25

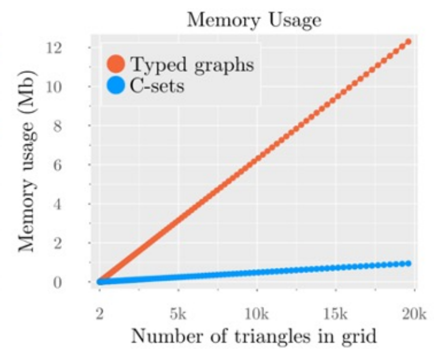
WIKI would be nice.

Catlab C-sets vs Catlab typed graphs





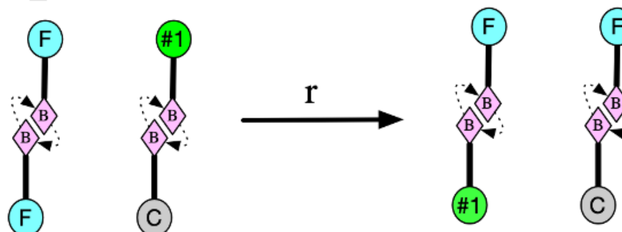




A quick note on performance is that some of the data of typed graphs is encoded STRUCTURALLY in C-Sets, and that allows them to be linearly more memory efficient, and their memory is also layed out efficiently. Something we didn't take advantage of here was the fact that the foreign keys of a database can be indexed, leading to some algorithms having different computational complexity. In fact, finding all homomorphisms is such an algorithm, so we will want to make a mature implementation of that that uses joins.

Variables and Negative Application Conditions

Finer-grained control over possible rewrites

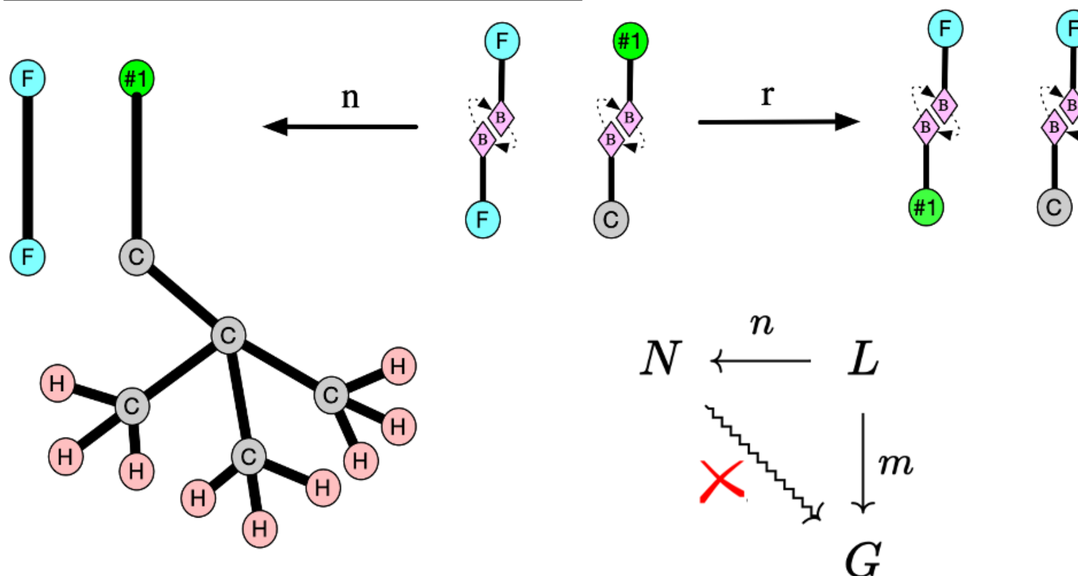


You can get a lot of mileage out of the language I've just described, but I've found that more features are helpful for using rewriting to accomplish practical computations.

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

Variables and Negative Application Conditions

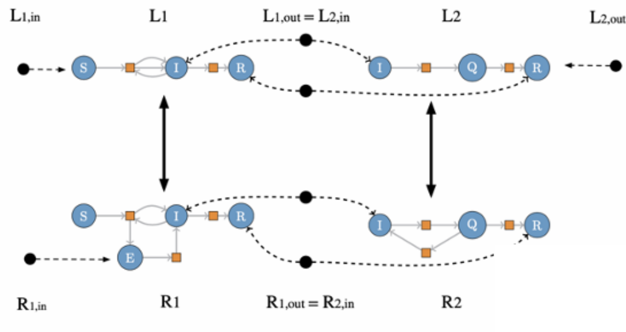
Finer-grained control over possible rewrites

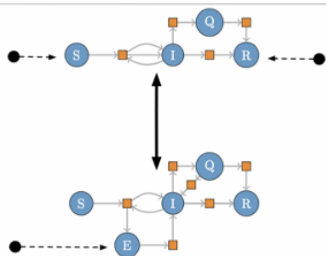


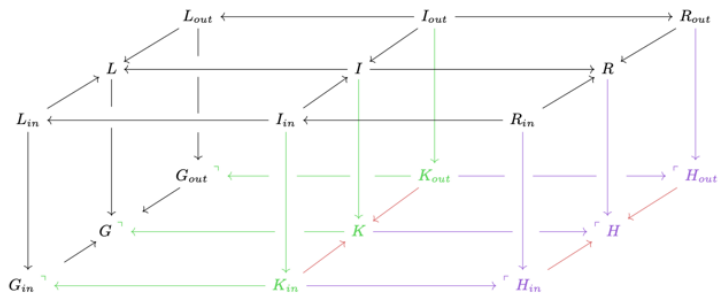
Negative application conditions add an extra map to the rewrite rule with the following meaning: it embeds the pattern L in some larger context that specifies when NOT to apply the rule. Basically, when we find a match to G, we need to check if there exists a map from N to to G such that this triangle 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.

Extension: Structured Cospan Rewriting



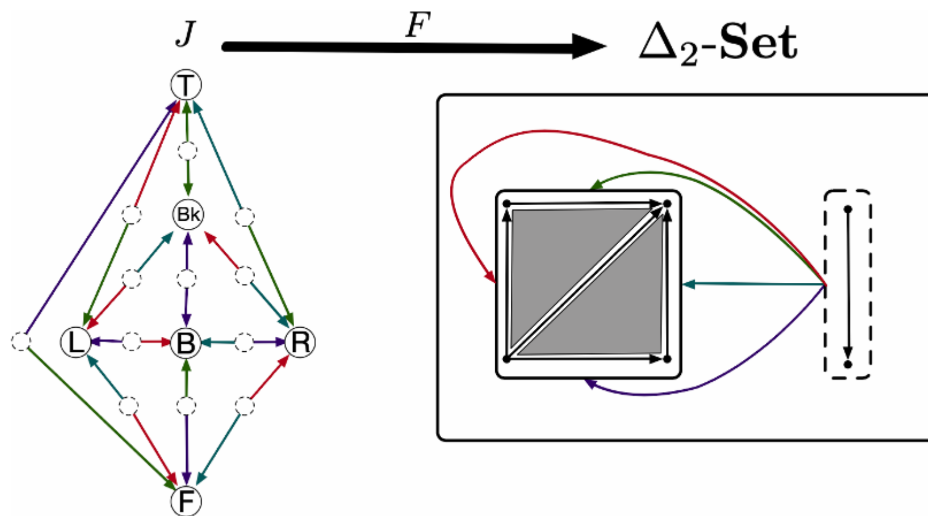




29

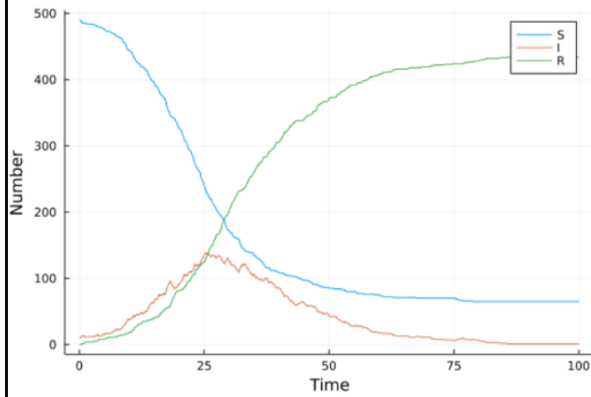
We can perform rewriting in other categories in Catlab, such as the category of structured cospans. These represent things with an interface.

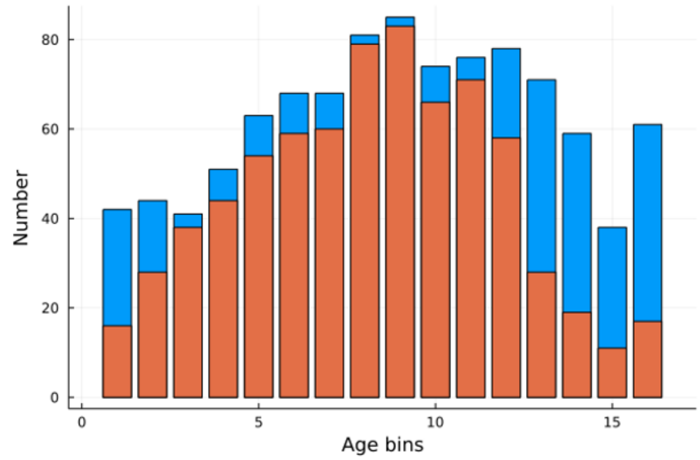
Extension: Distributed graphs



We have support for limits and colimits of diagrams, of which distributed graphs are a special case. So if we can come up with an algorithm for pushout complement of these objects, we will be able to perform rewriting on them.

Application: Agent based modeling

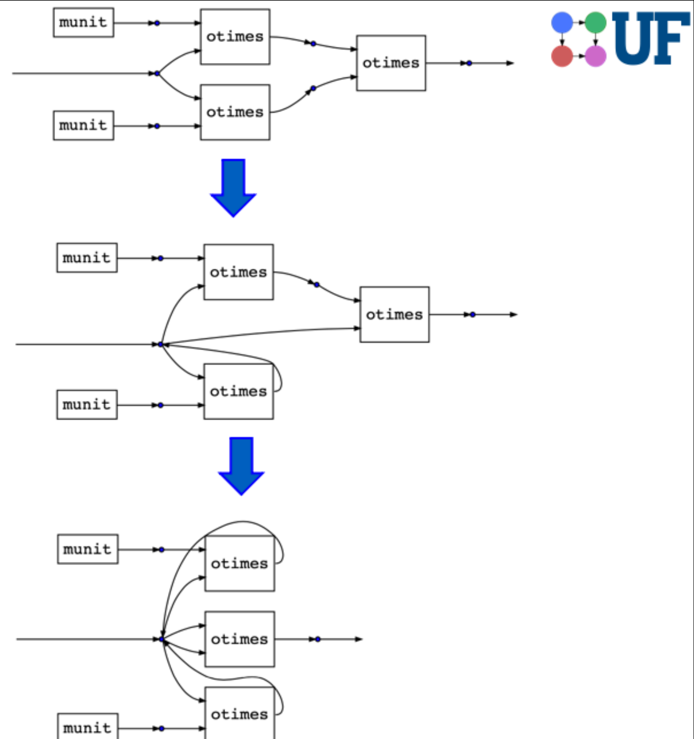




Requires scheduling events to happen in the future after a match.
So we must compose match morphism with derived rules.

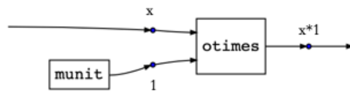
Equational reasoning

Use rewriting to prove
(1*x) * (x*1) = (x*x)



Left identity rule

When you see this pattern…



You can add this pattern as a parallel path.

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.

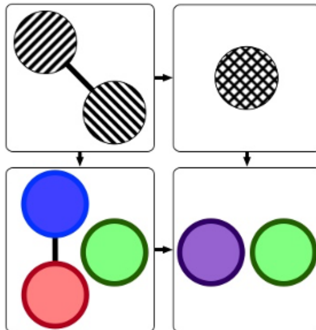
CLICK applying a right identity rule allows us to see that our result is computable merely as x*x which is an optimized program relative to the starting point.

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.

Future work

Library is still in its infancy, many planned improvements:

- More advanced scheduling (what is the right formalism for this?)
- Support for debugging rewrite rules (see *why* a match was rejected that you expect to take place)
- Performance (incremental search for matches, how do we do this?)
- Search up to isomorphism (remove symmetries in results to avoid double counting)
- More robust parallelization of rewrite execution



AlgebraicRewriting.jl

Thanks!

Evan Patterson

Sophie Libkind

David Spivak

T. Hanks

James Fairbanks

Sean Wu

Andrew Baas

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