

The logo for the University of Florida, consisting of the letters 'UF' in white on an orange square background.

Compositional exploration of combinatorial models

Kris Brown, Tyler Hanks, James Fairbanks

20.07.22



Hi, I'd like to discuss some work we did this year at the Generalized Algebraic Techniques for Applied Science research group led by James Fairbanks.

We're very used to searching through spaces of models where we have a parameter space of continuous values, but in contrast we're looking for a way to search through discrete spaces of models that have variations in structure, rather than parameters.

We're not going to make this process fully automatic, but we're going to give a language for domain experts to work with that will let them work declaratively, compositionally, and hierarchically.

Outline



Introduction

- Why represent scientific models combinatorially?

Model space exploration

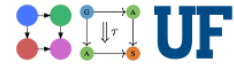
- The category of diagrams as a category of model spaces
- Example limits and colimits
- Composition recipes
- Limits and colimits: implementation

Model Selection

- Best fit chemical reaction network example

I'll start by motivating the idea of combinatorial representations of scientific models, then I'll go over some constructions in category theory, and then show you how we apply it.

Alternative: model as opaque code / math / logic



```
"""
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).

1. Generate the entire code from just declaring the reaction
2. Easily alter semantics (e.g. stochastic-based simulation)
3. Construct models compositionally (operad)
4. Construct models compositionally (limits).
5. Explore alternate reaction networks to fit the data

3

In the real world, a lot of scientific and engineering tasks are being represented in very opaque formats that make automatic reasoning and analysis nearly impossible.

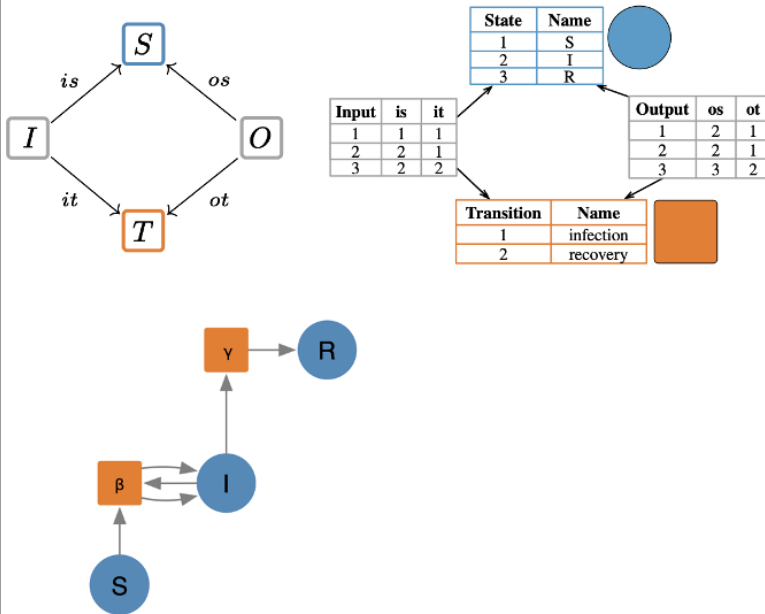
An example is the random scripts that a scientist might string together to perform a simulation of a chemical reaction network, as shown here.

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.

We want to work in restricted syntaxes which enables us to do lots of cool and useful things with our knowledge automatically.

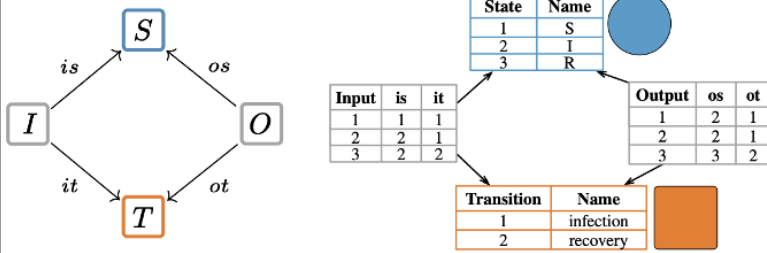
I'll give some quick examples of the first four of these example tasks that we are enabled to do by working with combinatorial models, and the bulk of the talk will focus on number 5.

1. Functorial generation of code



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. On top we show how this is captured by a C-set, which is a copresheaf or functor into Set from a small category C with four objects and four generating morphisms.

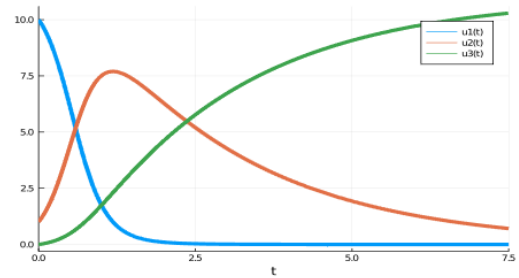
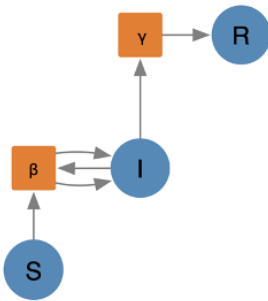
1. Functorial generation of code



Specifying a reaction network as a Petri Net allows for automatically generating an ODE simulation.

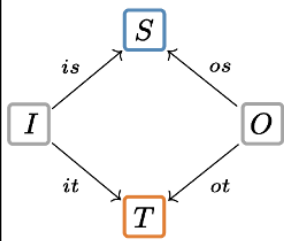
$$r_{\beta} = k_{\beta} \cdot [S] \cdot [I]$$

$$r_{\gamma} = k_{\gamma} \cdot [I]$$



We can map the components of this combinatorial representation into inputs for an ODE problem that can be solved to perform a simulation.

2. Easily change semantics with different functor



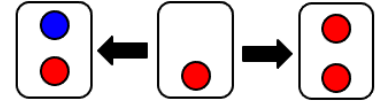
| State | Name |
|-------|------|
| 1 | S |
| 2 | I |
| 3 | R |

| Input | is | it |
|-------|----|----|
| 1 | 1 | 1 |
| 2 | 2 | 1 |
| 3 | 2 | 2 |

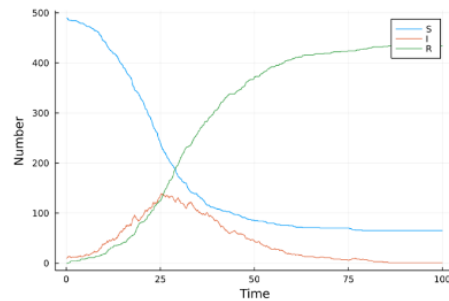
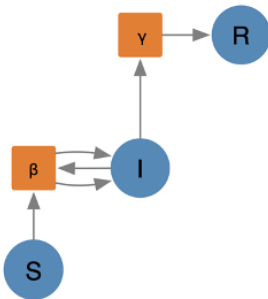
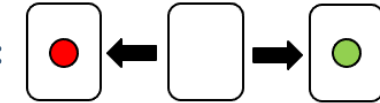
| Transition | Name |
|------------|-----------|
| 1 | infection |
| 2 | recovery |



Infect:



Recover:



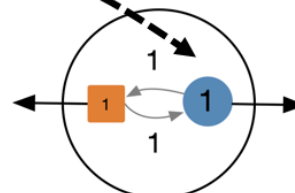
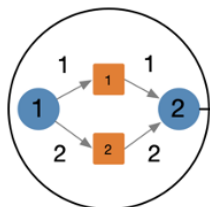
But separating the syntax and semantics of our model now makes it easy to map the transitions to a certain flavor of rewrite rules, and then perform a discrete time simulation to get a stochastic model semantics.

3. Build models compositionally (operad)



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
    
```

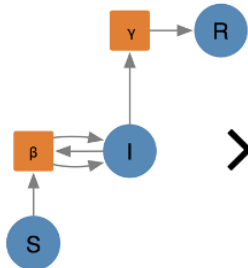
We can also view Petri nets being given another semantics, that of queries over a database of chemical reactions (i.e. we look for homomorphisms from the petri net into the database).

If we worked directly with the semantics as our model, we'd discover it's quite difficult to substitute queries inside each other, yet this is very straightforward thing to do working in a syntax category of undirected wiring diagrams.

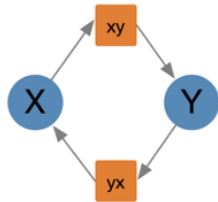
4. Build models compositionally (limits)



“SIR model”

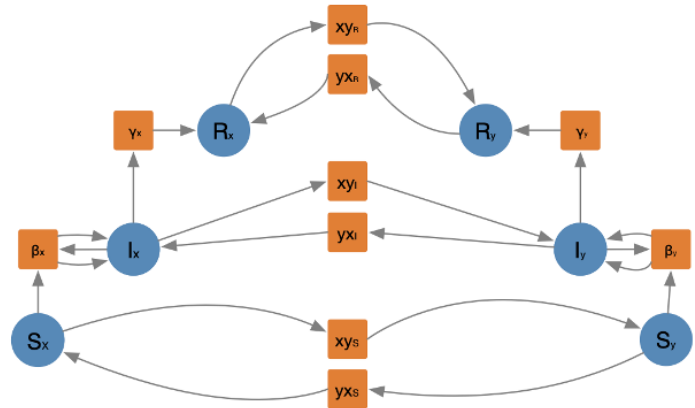


“Two-city model”



=

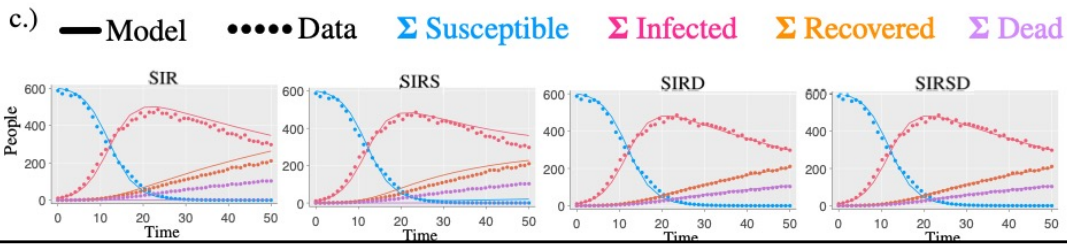
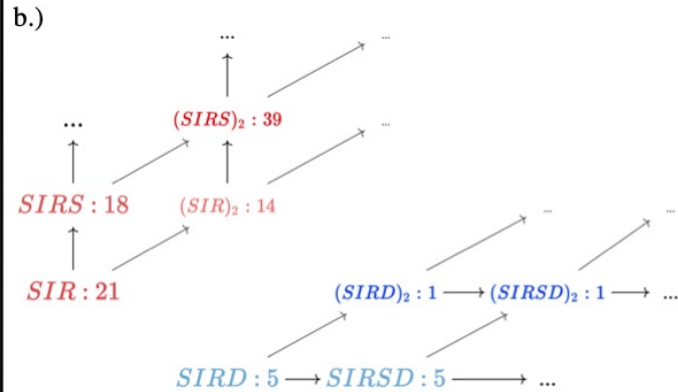
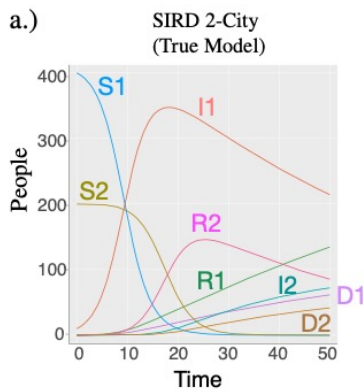
“Two-city SIR model”



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.
Sophie talked about this yesterday in her demo.

5. Automated Model Selection



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).

- a) the ground truth model
- b) the space of possible models exploring different assumptions about disease dynamics and geography
- c) the resulting quality of fit plots. The SIRD 2-city model is the best, which recovers the ground truth.

The color coding in b) is the loss function value for the optimal parameters. This shows why a "set of models" is insufficient you really need the "space". All the

models in red are bad, we could prune that half of the space as a heuristic. All of the models above and to the right of SIRD_2 are basically going to have the same loss value because you can always set parameters to 0 and recover SIRD_2. This means we can stop once we hit SIRD_2.

This approach lets you take a category of models and “slap a geometry on there” then you can use that geometry to design more efficient algorithms. The more structure in the space, the more efficient the algorithms. A **set** has no geometry so you just have to check them all. A graph lets you do some reasoning about neighborhoods.

Outline



Introduction

- Why represent scientific models combinatorially?

Model space exploration

- The category of diagrams as a category of model spaces
- Example limits and colimits
- Composition recipes
- Limits and colimits: implementation

Model Selection

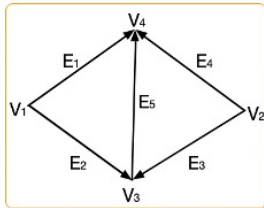
- Best fit chemical reaction network example

I'll now introduce the category of diagrams

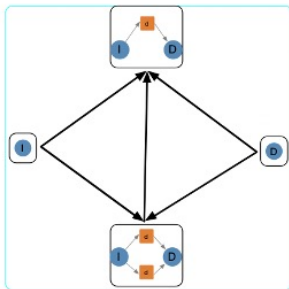
The category of diagrams



A particular diagram in Petri



Shape



Diagram

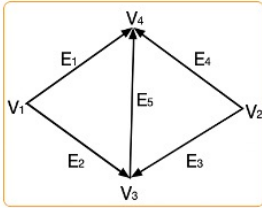
Here's an example diagram in Petri, which is a functor from a small category into the category of Petri nets.

We have a shape category, which is presented by a finite graph. For each object we give a Petri net, for every generating arrow we give a petri net morphism. Later we will give this the interpretation of a space of models, when our notion of model happens to be a Petri net.

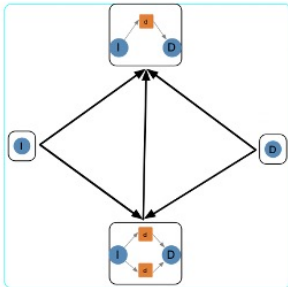
The category of diagrams: a lax slice category



A particular diagram in Petri



Shape

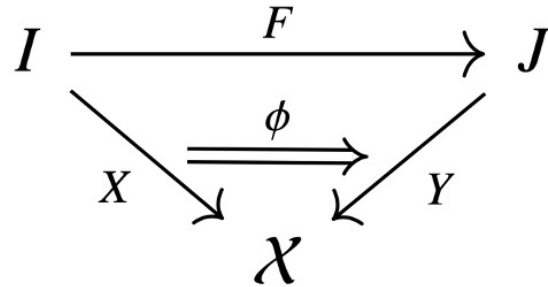


Diagram

$$(F, \phi): (I, X) \rightarrow (J, Y)$$

Shape map: F

Diagram map: ϕ



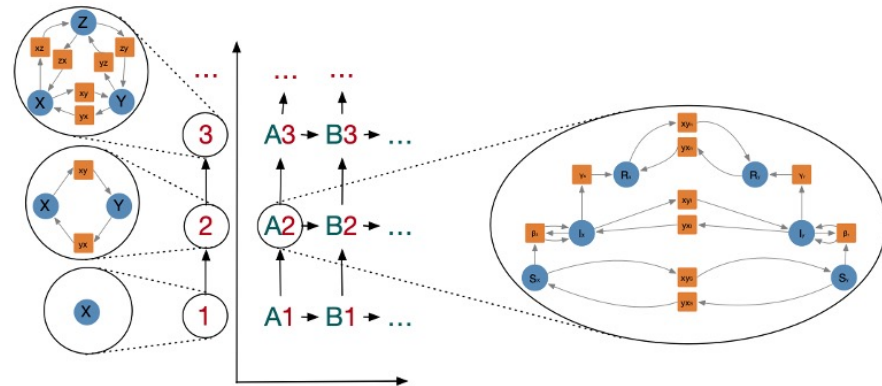
We want pushouts and pullbacks of diagrams

Lax version of slice category in Cat gives us bicompleteness.

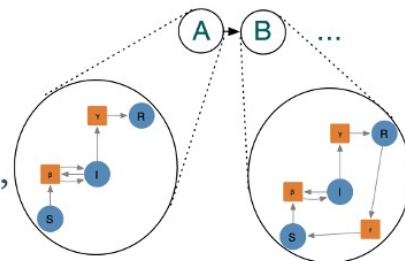
Model space product



“City dimension”



“Disease dimension”

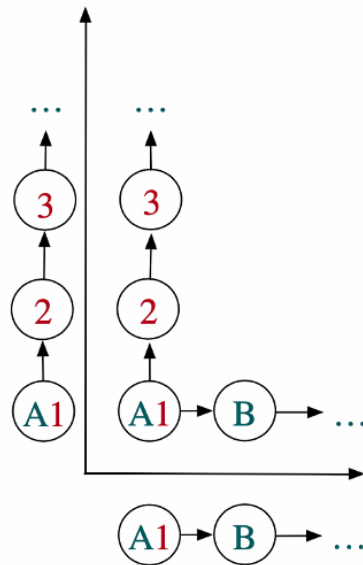


13

We've seen stratified petri nets as the product of simpler petri nets. But we can take that one more level meta, the simpler petri nets are often living in their own dimensions, and we are interested in all combinations from the different dimensions. So it turns out this multidimensional model exploration is precisely the product of model spaces

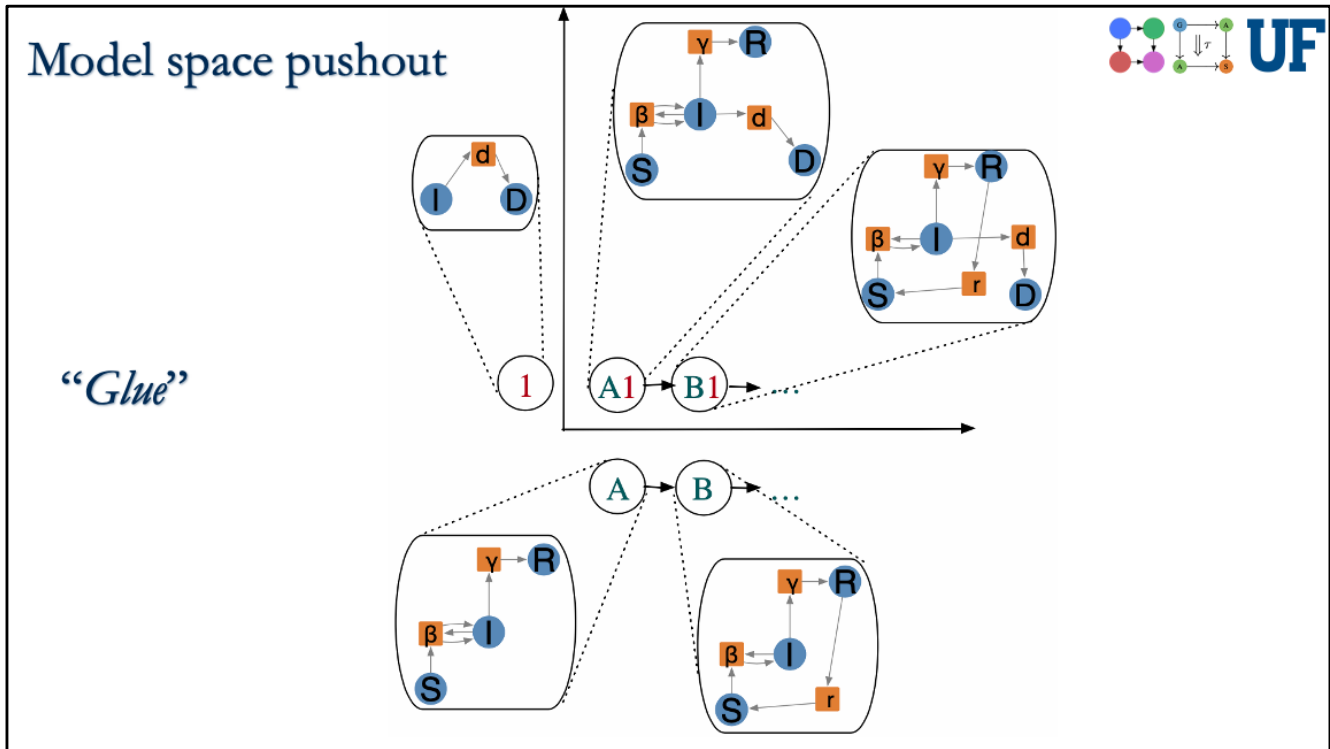
This is something people obviously already do, but they're doing this with a double for loop in a throwaway python script rather than working declaratively and compositionally.

Model space pushout



Of course we don't always want just grid-like model spaces. Sometimes there are either-or decisions to be made, and this is captured with colimits. Here, a pushout of two dimensions that share the same first model leads to a branching model space (with no changes to the Petri nets underneath).

But then you may ask what happens if they're not equal. That's actually an interesting and practical situation.

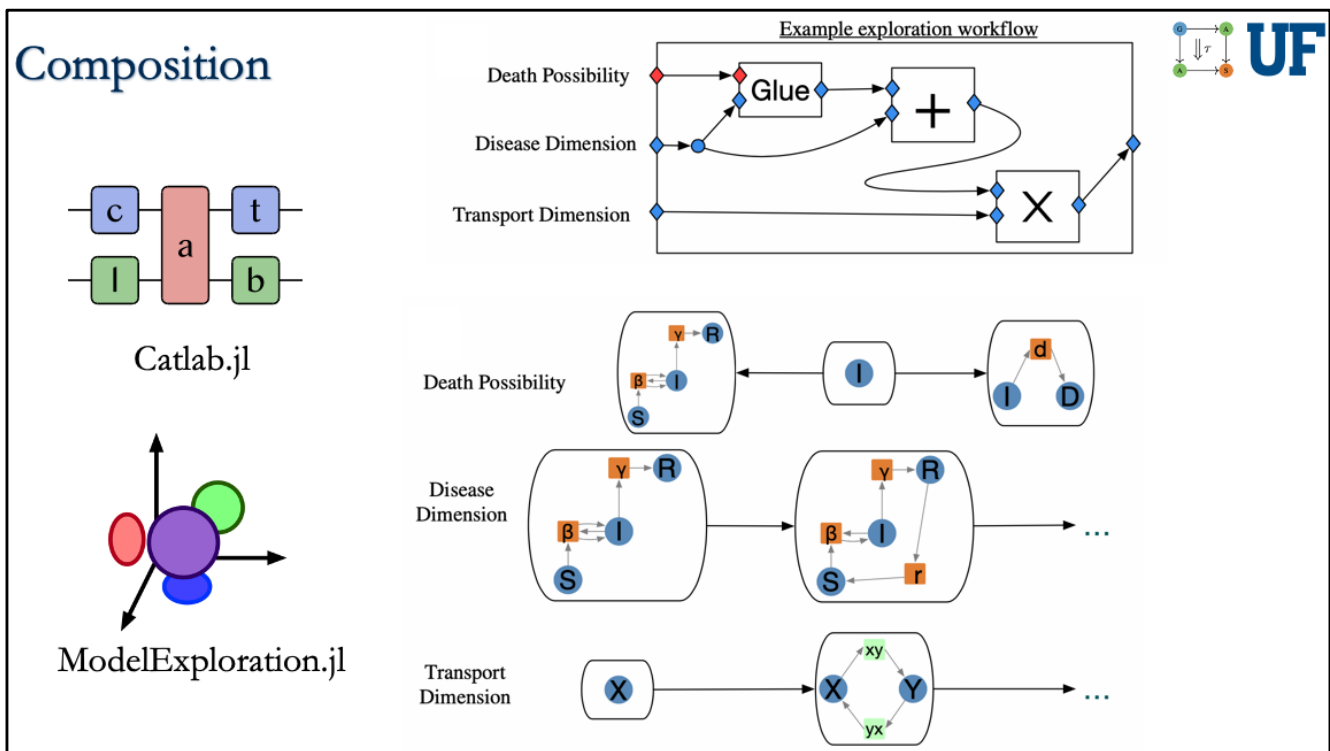


Suppose my apex of a pushout of diagrams has one object, and that object is relating the 1 model visualized on the left to the A model visualized on the bottom, which is the SIR model.

By taking a pushout of Petri nets, this would glue the infected to death transition to the SIR model and give you the model that would live under the A1 object in the colimit diagram. However, there is a morphism from A1 to B1, and we need to send that death state somewhere in B.

But our A to B data says nothing about where death should go, so in fact we need to freely add it (this is the left kan extension at work). So in fact, we've glued the death transition to the entire disease dimension implicitly by gluing it to the SIR model explicitly.

So this was just a small taste of the kind of powerful language you have by working with limits and colimits



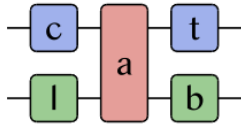
So those examples can be thought of as building blocks, and in reality complex model spaces would be built by assembling those in a coherent way. Wiring diagrams offer a nice visual syntax for these recipes of combining these operations, and they nicely separate the composition strategy from the actual inputs that will ultimately get plugged in.

I've shown here an example of combining that combines the three examples we just saw into one recipe for a composite model space.

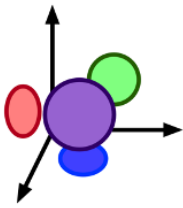
This isn't a code talk, but we have in all this implemented in ModelExploration.jl, which depends on Catlab.jl. So you really just need to define these three diagrams, provide the data of the pushout and call product or coproduct on the diagrams and you'll actually compute the result.

Let's see what that is.

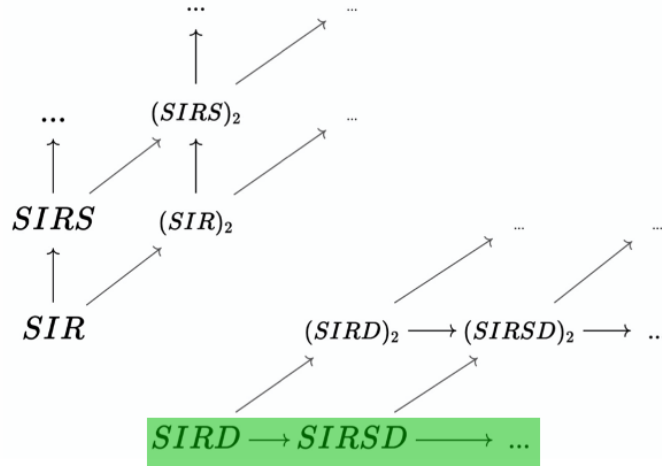
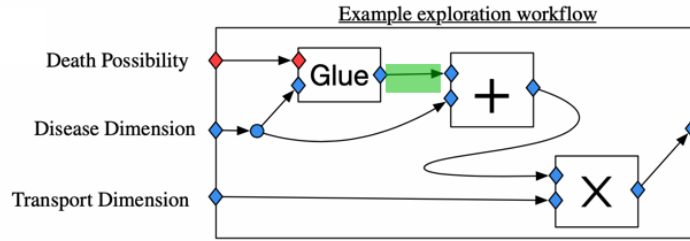
Composition result



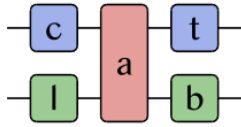
Catlab.jl



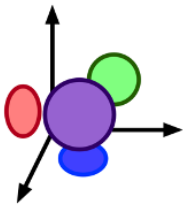
ModelExploration.jl



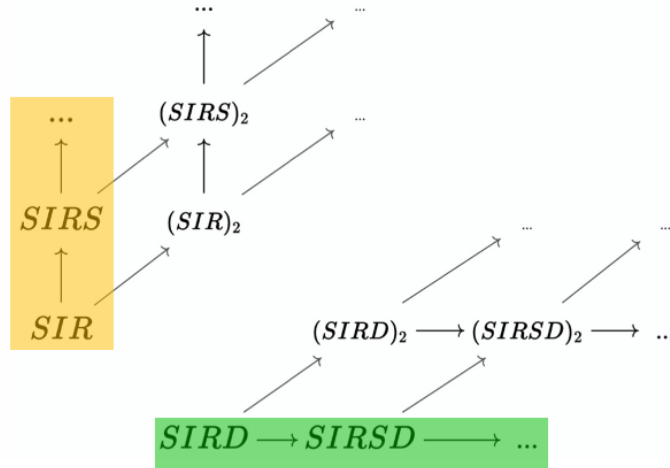
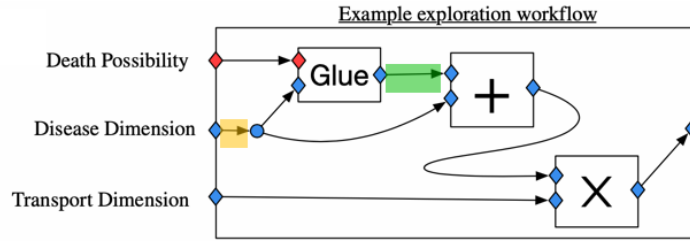
Composition result



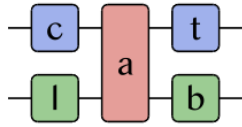
Catlab.jl



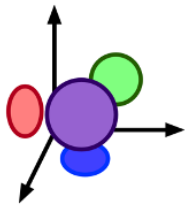
ModelExploration.jl



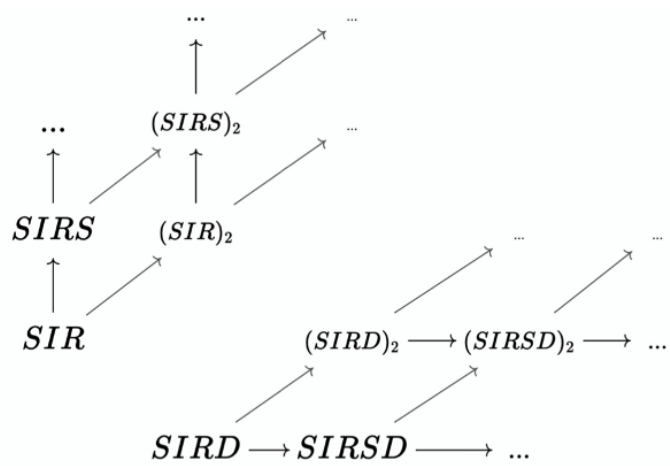
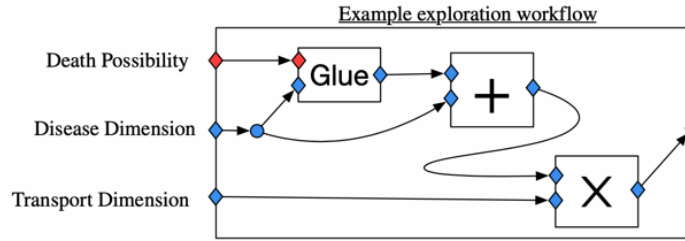
Composition result



Catlab.jl



ModelExploration.jl



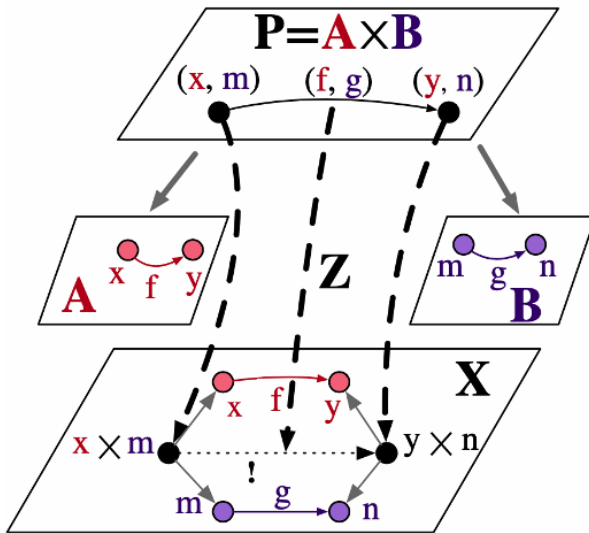
So, moving forward, we glue the death transition to the disease dimension, and that gives us this bottom line of models. We coproduct that with the disease dimension that does not have death glued to it, because suppose we're not sure whether we want that in our model - which gives us the first vertical sequence of models. So this front face, in the plane of the page, is the model space that lives on this wire going between the sum and the product. When we product it with an n-city model, you get the dimension that goes into the page.

This could be an infinite model space in all of these dimensions, but for now let's just work with the 8 models we see here.

Limits - implementation



Product



Peschke and Tholen (2020)

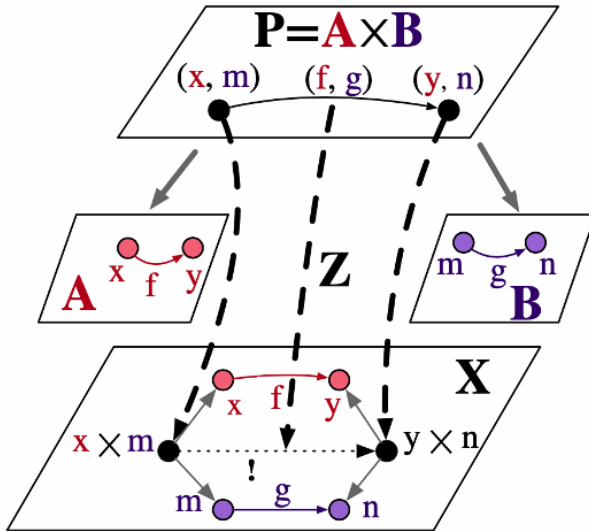
So I will present some work by Peschke and Tholen (2020) who worked out what the limits and colimits are in this category. Their description was quite high level and concise, and in my struggle to implement these limits and colimits on a computer I had to really unpack the description, so let me share some of their results in a more concrete fashion.

Here's a generic picture of a morphism in the product of two diagrams, to give some intuition. The shape is just the product of the shapes of the two diagrams. We need a functor from this shape into the base. You simply do the natural thing on objects, which takes a product of objects to the product their corresponding objects in the base category. We need a morphism in the base between the two product objects, but we can obtain this using the universal property of products as shown below, since we can map into each of the components of y times n .

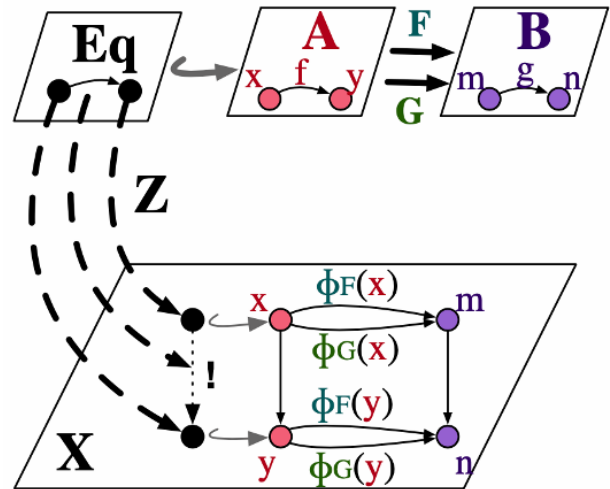
Limits - implementation



Product



Equalizer



Peschke and Tholen (2020)

Just as the product of diagrams translated neatly into computing a product shape-wise and using products in the underlying base category, an equalizer diagram has as its shape an equalizer of the functors that constitute the shape components of the diagram morphisms.

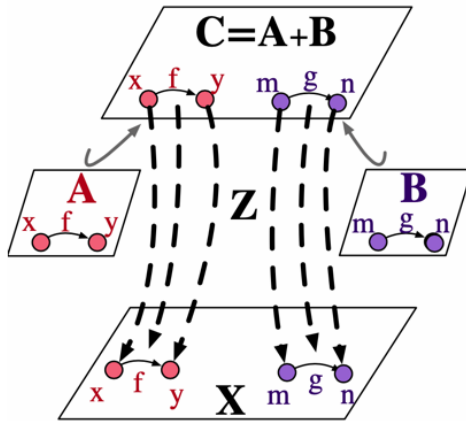
It turns out we can compute where to send the objects of an equalizer to by using equalizers of data in the diagram maps of the morphisms. Essentially, the constraint these diagram maps are natural means that whenever we have big F and big G equalizer a morphism in A, they will have parallel morphisms in their diagram map which can be equalized. Again we can use a universal property to determine where the morphism in the equalizer must map to.


So overall, limits of diagrams are a very intuitive extension of limits in the base category.

Colimits - implementation



Coproduct



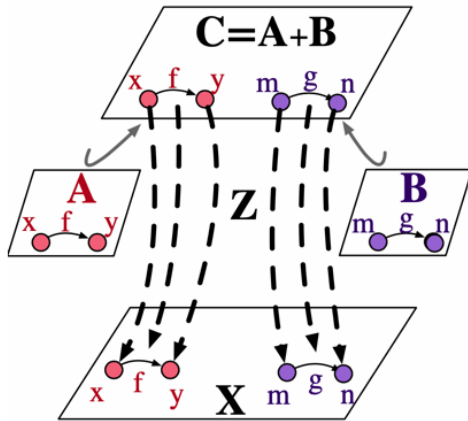
 Peschke and Tholen (2020)

Coproducts are also simple – again we’re going to assume the right thing to do is to take a coproduct of the shape categories and then do something coproducty underneath.

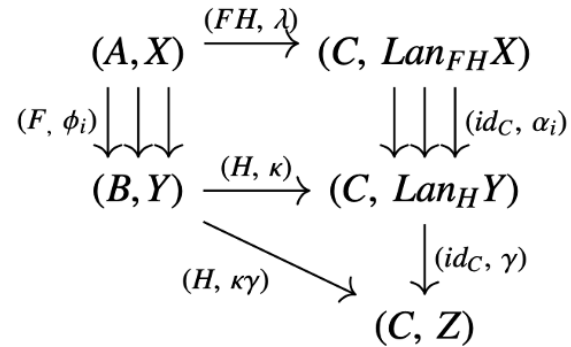
Colimits - implementation



Coproduct



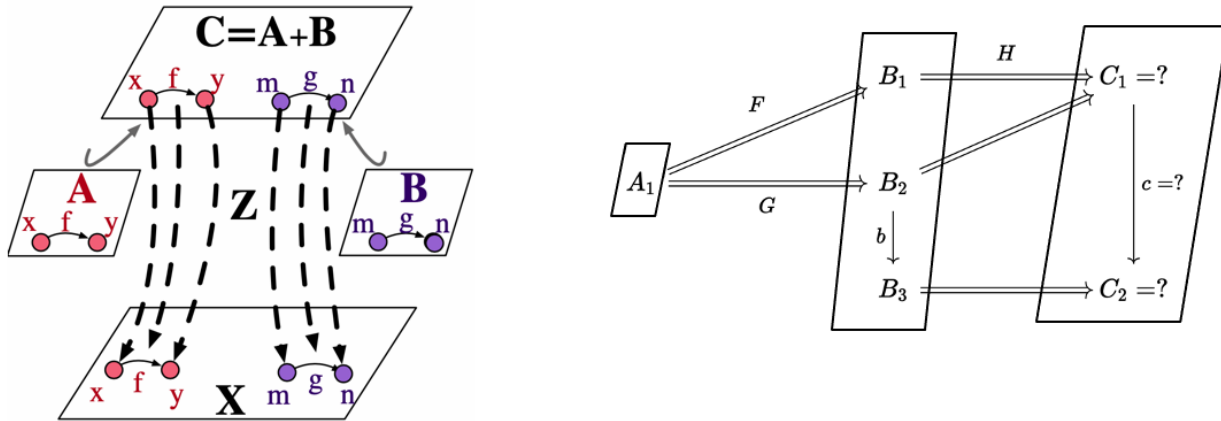
Coequalizer



Peschke and Tholen (2020)

Lastly, then, we want to take a coequalizer of the shape maps and do something coequalizer-y in the base category. But you quickly find there is no obvious such thing to do. You don't have parallel maps in the base category to coequalize, so we need to perform this construction here, which I've shown as a commutative diagram in the category of diagrams. If you want intuition for this or an example, I have some hidden slides and can discuss that if we have extra time.

Colimits in the category of diagrams

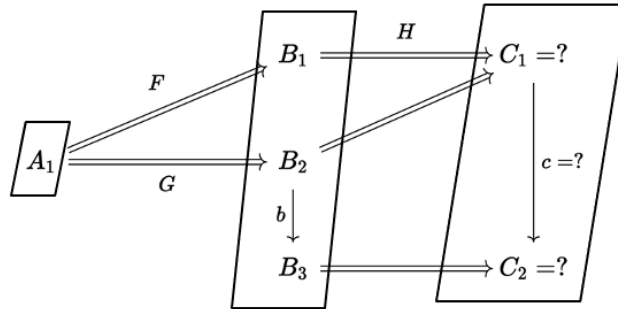


Peschke and Tholen (2020)

Coproducts are also simple – again we’re going to assume the right thing to do is to take a coproduct of the shape categories and then do something coproducty underneath.

Lastly, then, we want to take a coequalizer of the shape maps and do something coequalizer-y in the base category. But you quickly find there is no obvious such thing to do. For example, consider diagram morphisms F, G with the following shape maps. We have a morphism for F and G in the underlying base category, but that can’t be coequalized b/c they may not share the same target. Moreover it’s not clear what to assign to C_2 or the morphism.

Coequalizers in the category of diagrams



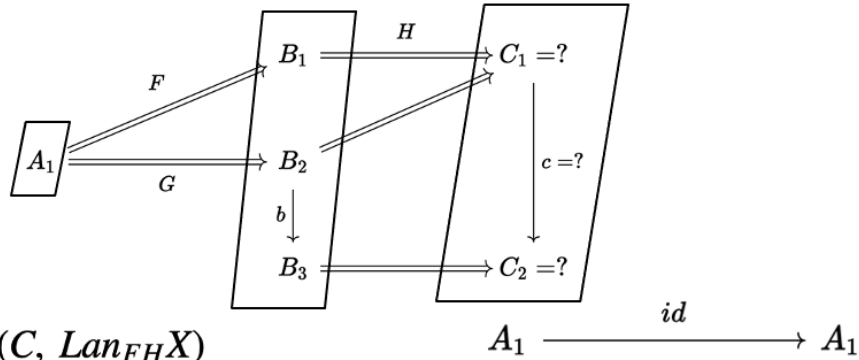
$$\begin{array}{ccc}
 (A, X) & \xrightarrow{(FH, \lambda)} & (C, Lan_{FH}X) \\
 \downarrow (F, \phi_i) & & \downarrow (id_C, \alpha_i) \\
 (B, Y) & \xrightarrow{(H, \kappa)} & (C, Lan_H Y) \\
 \searrow (H, \kappa\gamma) & & \downarrow (id_C, \gamma) \\
 & & (C, Z)
 \end{array}$$

Peschke and Tholen (2020)

Here is the commutative diagram that describes the algorithm for constructing the coequalizer. The morphism data must all be coerced into the form of diagrams of shape C, and only then will the data of F and G be coequalizable.

<https://q.uiver.app/?q=WzAsMTIsWzAsMSwiQV8xIl0sWzIsMCwiQl8xIl0sWzIsMSwiQl8yIl0sWzIsMiwilQl8zIl0sWzQsMCwiQ18xPT8iXSxbNCwyLCJDXzI9PyJdLFswLDMslkFfMSJdLFsyLDMslkFfMSJdLFswLDQsIkjfmStCXzliXSxbMiw0LCJcXzErQl8zIl0sWzAsNSwiQ18xIl0sWzIsNSwiQ18yIl0sWzAsMSwiRilsmCx7ImxldmVsljoyfV0sWzAsMiwRylsMix7ImxldmVsljoyfV0sWzIsMywiYilsMI0sWzQsNSwiYz0/Il0sWzEsNCwiSCIsMCx7ImxldmVsljoyfV0sWzIsNCwiliwxLHsibGV2ZWwiOjJ9XSxbMyw1LCliLDIseyJsZXZlbcI6Mn1dLFs2LDcslmlkIl0sWzgsOSwiaWRfe0JfMX0rYiJdLFs2LDgslIxcGhpX0YiLDIseyJjdXJ2ZSI6MX1dLFs2LDgslIxcGhpX0ciLDAseyJjdXJ2ZSI6LTF9XSxbNyw5LCJcXHBoaV9GliwyLHsiY3VydmUiOjF9XSxbNyw5LCJcXHBoaV9HO2iLDIseyJjdXJ2ZSI6LTF9XSxbMTAsMTEsIiEiLDAseyJzdHlsZSI6eyJib2R5Ijp7Im5hbWUiOiJkYXNoZWQifX19XSxbOSwxMSwiXfX0aW55IGNvZXEiLDAseyJzdHlsZSI6eyJoZWFljp7Im5hbWUiOiJlcGkifX19XSxbOCwxMCwiXfX0aW55IGNvZXEiLDAseyJzdHlsZSI6eyJoZWFljp7Im5hbWUiOiJlcGkifX19XV0=>

Coequalizers in the category of diagrams

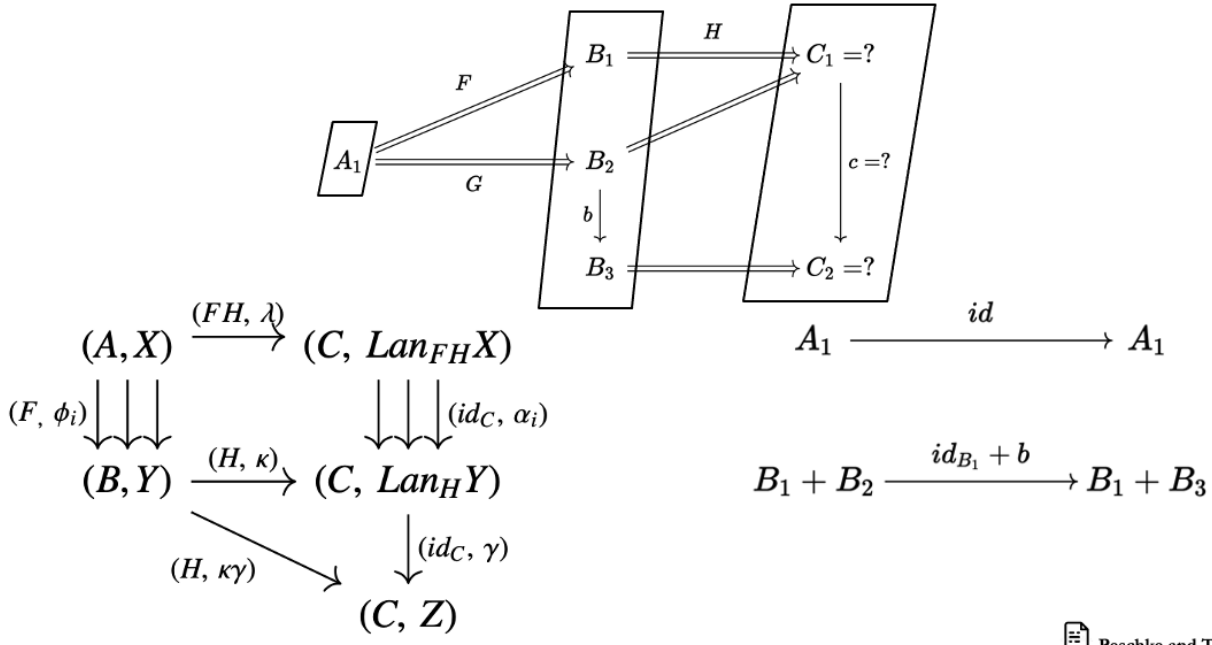


$$\begin{array}{ccc}
 (A, X) & \xrightarrow{(FH, \lambda)} & (C, Lan_{FH}X) \\
 \downarrow (F, \phi_i) & & \downarrow (id_C, \alpha_i) \\
 (B, Y) & \xrightarrow{(H, \kappa)} & (C, Lan_H Y) \\
 \searrow (H, \kappa\gamma) & & \downarrow (id_C, \gamma) \\
 & & (C, Z)
 \end{array}$$

Peschke and Tholen (2020)

For this specific example, if we push “A” into the diagram of shape of C with a left kan extension, we're sticking the A1 into a C1 shaped hole. The map lowercase c needs to map that to something, so we need to add another A1 into the C2 shaped hole and give an identity map.

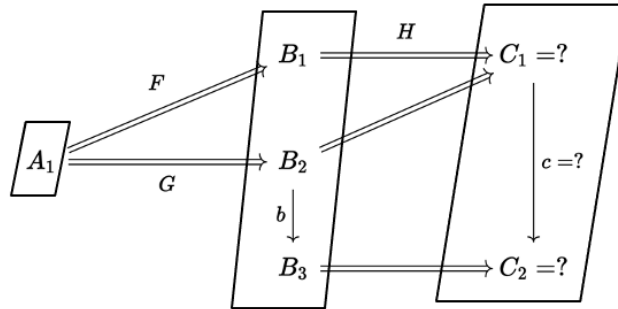
Coequalizers in the category of diagrams



Peschke and Tholen (2020)

If we're pushing B into C, B1 and B2 are both stuck into the C1 shaped hole, and B3 gets stuck into the C2 shaped hole. However, we know how the B2 components gets mapped to B3, whereas we have nothing to say about how B1 gets mapped there, so in fact we need again to create another copy of B1 and map to it via the identity.

Coequalizers in the category of diagrams

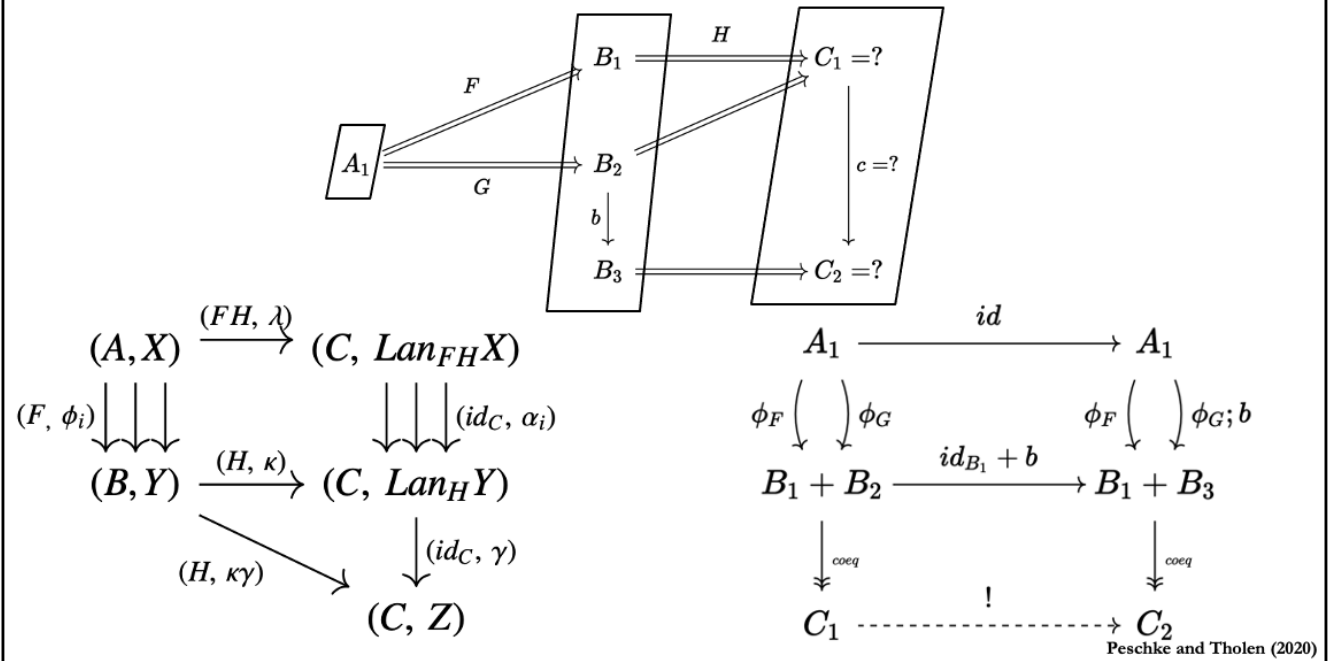


$$\begin{array}{ccc}
 (A, X) & \xrightarrow{(FH, \lambda)} & (C, Lan_{FH}X) \\
 \downarrow (F, \phi_i) & & \downarrow (id_C, \alpha_i) \\
 (B, Y) & \xrightarrow{(H, \kappa)} & (C, Lan_H Y) \\
 & \searrow (H, \kappa\gamma) & \downarrow (id_C, \gamma) \\
 & & (C, Z)
 \end{array}
 \qquad
 \begin{array}{ccc}
 A_1 & \xrightarrow{id} & A_1 \\
 \downarrow \phi_F & & \downarrow \phi_F \\
 B_1 + B_2 & \xrightarrow{id_{B_1} + b} & B_1 + B_3
 \end{array}$$

Peschke and Tholen (2020)

It turns out the universal property of left kan extensions gives us maps between these objects and in fact they are parallel arrows, meaning we can coequalize them and get one object in the base category for every object in C!

Coequalizers in the category of diagrams



When we do that, we can use the universal property of coequalizers to get what morphism lowercase c gets mapped to.

Hopefully this gives some intuition for the motivation of this rather complex construction for coequalizers, at least relative to limits and coproducts. But with these constructions implemented in Catlab, we can now construct diagrams via limits and colimits, with the caveat that the left kan extension of finite diagrams can sometimes be infinite.

Outline



Introduction

- Why represent scientific models combinatorially?

Model space exploration

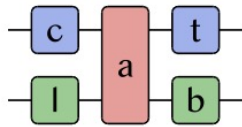
- The category of diagrams as a category of model spaces
- Example limits and colimits
- Composition recipes
- Limits and colimits: implementation

Model Selection

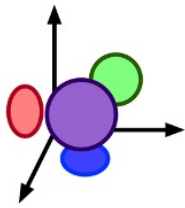
- Best fit chemical reaction network example

So now we're going to apply the model space construction to a problem.

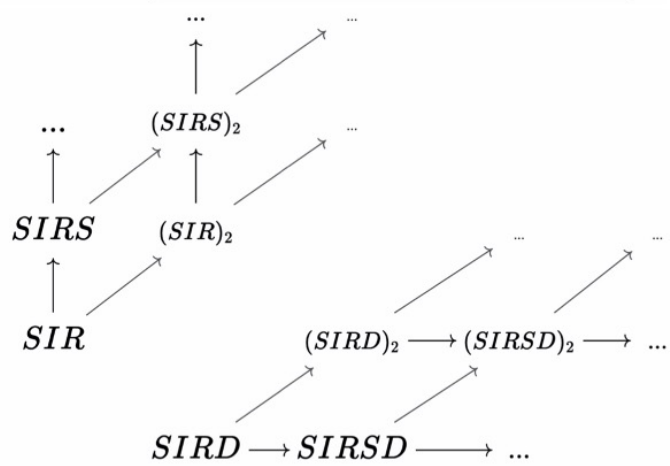
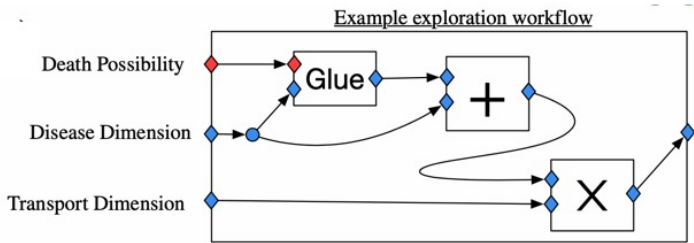
Composition result



Catlab.jl



ModelExploration.jl

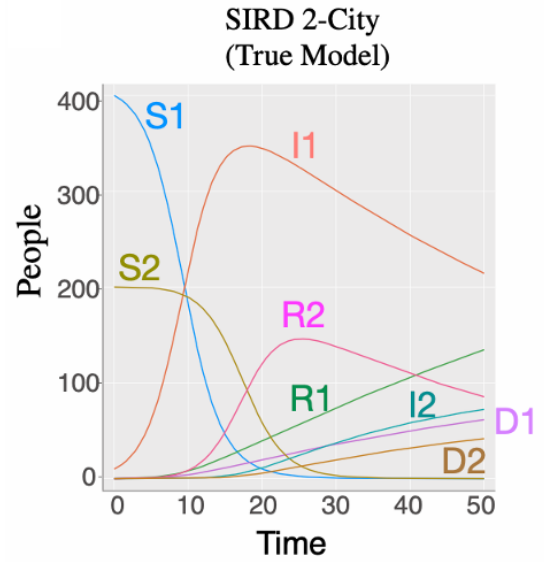


So remember, this was our resulting model space. we're going to explore these 8 models to find which one best fits some real world data we have. This is how we gain mechanistic understanding of natural phenomon - generating hypothesis and testing them.

One particular model selection strategy

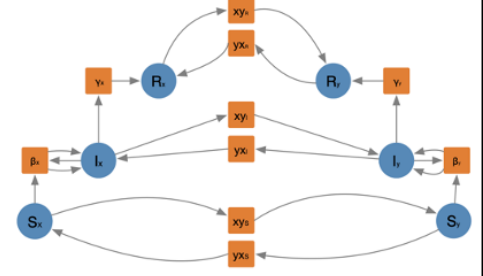
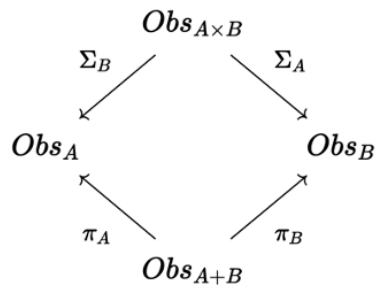
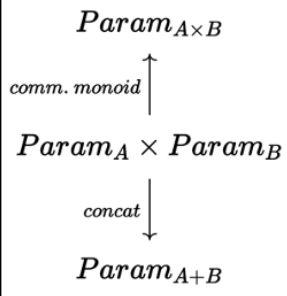


$$Loss(y, \hat{y}, \vec{p}) = \sum_{t \in T} \sum_{s \in \{S, I, R\}} (y_s(t) - \hat{y}_s(\vec{p}, t))^2$$



Suppose we have real data that looks like the right (with some added noise).

Model selection results



S: 90%, I: 10% and City X: 33%, City Y: 67% For $(SIRD)_2$, we have S_X : 10%, S_Y : 25%
 For $(SIRD)_2$, we have S_X : 30%, D_X : 0% S: 35%

Projection maps of (co)limits allow us to move data to/from the primitive model spaces to the composite model space.

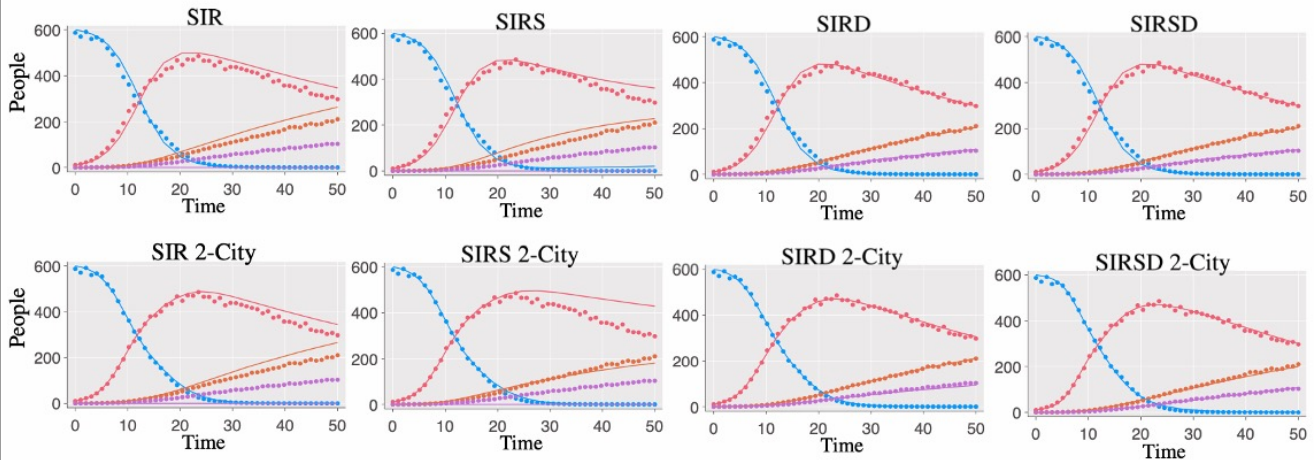
MORPHISMS IN THE CATEGORY:

- push forward initial concentrations & parameters
- Have metric of complexity (if mono) breadth first search
- <https://q.uiver.app/?q=WzAsNixbMSwwLCJQYXJhbV97QVxcdGltZXMGn0iXSxbMCMwLCJQYXJhbV9BI0sWzlsMSwiUGFyYW1fQjJdLFs0LDAslk9ic197QVxcdGltZXMGn0iXSxbMywxLCJQYnNfQSJdLFs1LDEslk9ic19CII0sWzEsMF0sWzlsMF0sWzMsNCwiXFXtaWdtYV9ClwiyXSxbMyw1LCJcXFNpZ21hX0EiXV0=>

Model selection results



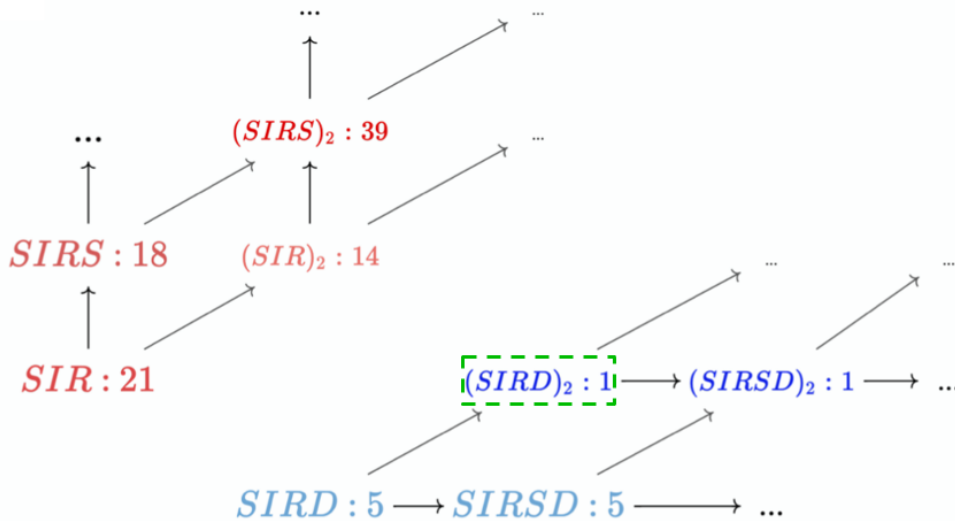
— Model
 ●●●● Data
 Σ Susceptible
 Σ Infected
 Σ Recovered
 Σ Dead



MORPHISMS IN THE CATEGORY:

- push forward initial concentrations & parameters
- Have metric of complexity (if mono) breadth first search
- <https://q.uiver.app/?q=WzAsNyxbMCwwLCJQYXJhbV97QVxcdGltZXMgQn0iXSxbMCwxLCJQYXJhbV9BIFxcdGltZXMgUGFyYW1fQiJdLFszLDAsIk9ic197QVxcdGltZXMgQn0iXSxbMiwxcjYnNfQsJdLFs0LDEsIk9ic19CII0sWzAsMiwuUGFyYW1fe0ErQn0iXSxbMywyLCJYnNfe0ErQn0iXSxbMSwwLCJcXGZvb3Rub3Rlc2l6ZSBjb21tLlxcIG1vbm9pZCJdLFsyLDMsIlxcU2lnbWFFQilsMI0sWzIsNCwiXFXtaWdtYV9BI0sWzEsNSwiXFXmb290bm90ZXNpemUgY29uY2F0liwyXSxbNiwzLCJcXHBpX0EiXSxbNiw0LCJcXHBpX0liLDJdXQ==>

Model selection results



By our loss function we can evaluate the best fit models for each of these petri nets and make some observations, such as two of the models here having extremely low error relative to the rest. However, because we restricted our model category to have monic morphisms, they also serve as a proxy for complexity. So we can take the least element among the collection of “accurate enough” models as our overall model selection function.

Important to note that the code we wrote does this pipeline automatically given a finite composite diagram.

Future work



- More interesting “primitive” model space constructors
- Wiring diagram visualizer / GUI
- Lazy state space exploration
- Alternative applications as demonstrations (Boolean functions, circuits, NN)
- Hierarchical loss functions (optimizing overall goal + subgoals, together)

Thanks!



T. Hanks



Sean Wu



James Fairbanks



IHME

Institute for Health Metrics
and Evaluation



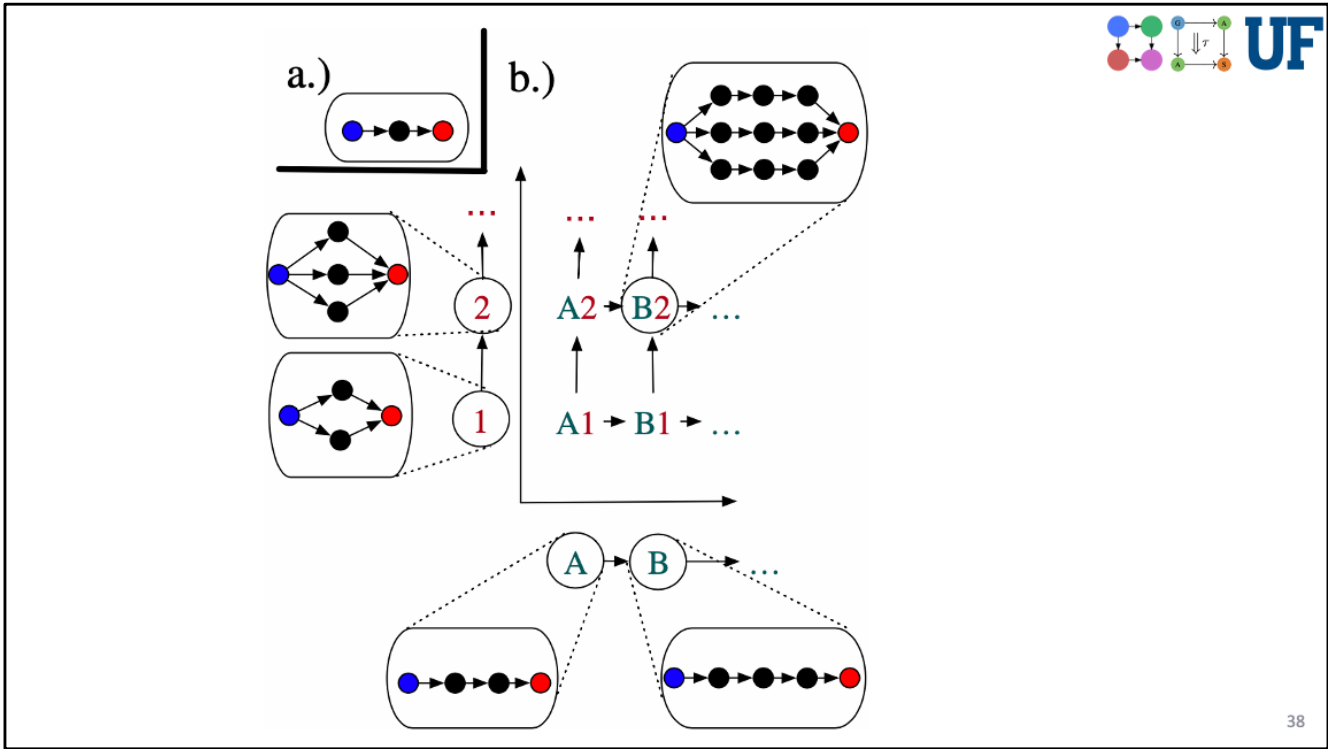
Andrew Baas

TOPOS



INSTITUTE





Hidden slide: neural network architectures with reflexive graphs