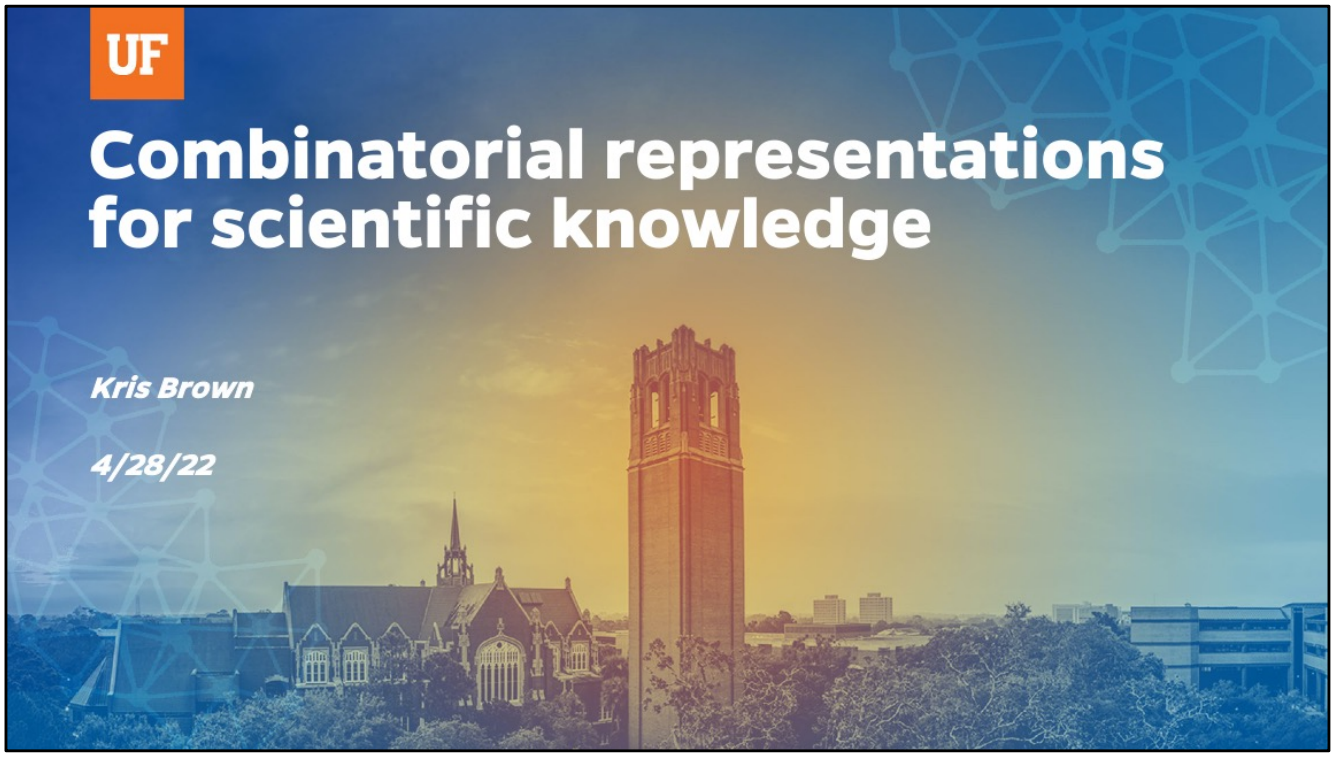
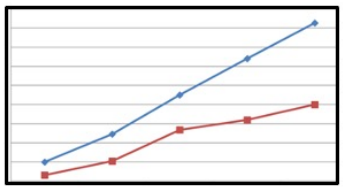



Hi, I'm a postdoc working with James Fairbanks for the past 8 months. Prior to that I was working as a computational chemist where I struggled a lot with issues that I think we're making great progress towards addressing. I hope if you're a computational scientist that this talk convinces you that these techniques are worth looking more into. This will be an applied category theory talk, so I'll focus on the applications rather than the theory.

# Status quo: Model as opaque code

```python
"""
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
```



- Knowledge informally represented in publications.
- Often the only formal representation is code.

2

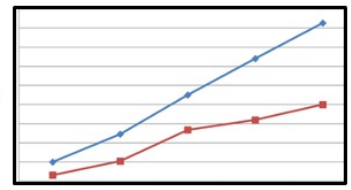
I want to start by providing contrast with a very pervasive form of scientific knowledge, which seemed almost necessary to me while doing computational research.

This is a small Python script which takes some data and compares it to a small simulation of a chemical reaction. This one is self-contained, but in general they live in a filesystem and can refer to each other. The big idea behind such a script is sometimes mentioned informally in a publication, and that is an important form of scientific knowledge but one we'll ignore for this talk because it is informal and can't be manipulated by tools (short of NLP).

# Status quo: Model as opaque code

```python
"""
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
```



- Knowledge informally represented in publications.
- Often the only formal representation is code.

3

If you point to a figure in the paper and ask the author ”What does this mean?” and demand a rigorously defined object, you’ll get pointed to this piece of code (though you’d also need to know lots of details about their computer environment: versions of libraries, environment variables, etc.).

You could ask what are the downsides of this being the reality for many scientists.

# Status quo: Model as opaque code

```python
"""
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 that cannot be done with arbitrary code (nor mathematical expressions).

The abstract critique of this style is that it mixes together data, structure, and semantics, which all should be kept apart.

The corresponding practical critique is lose the opportunity to do some cool things with our models if we make this mistake.

# Status quo: Model as opaque code

```python
"""
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
```

- Update code when assumptions change

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

We'd like to have a language for declaring our assumptions and how they've changed, and when we do so, it's as if our code gets updated into the new framework.

# Status quo: Model as opaque code

```python
"""
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
```

- Update code when assumptions change
- Explore alternate reaction networks to fit the data

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

There's a process that we all do of tweaking parameters and testing hypotheses until we get something that matches our experimental result – this process isn't made explicit or at all automated with this paradigm, even though there are certain mechanical and tedious aspects of it that beg to be automated.

# Status quo: Model as opaque code

```python
"""
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
```

- Update code when assumptions change
- Explore alternate reaction networks to fit the data
- Generate the entire code from just declaring the reaction

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

In some sense writing the code at all seems like an undesirable, error-prone task. If you look at the description at the top of the function, we ought to be able to just declare that information and get the whole simulator for free.

# Status quo: Model as opaque code

```python
"""
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
```

- Update 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

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

We have many informal notions of when models are equivalent or submodels of each other in some sense, and regardless of what we pick we cannot actually compute that without a better formal representation of the model.

# Status quo: Model as opaque code

```python
"""
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
```

- Update 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)

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

Lastly, once you make some syntax-semantics distinction, it becomes clear that we should be able to give the same syntax a different semantics, such as taking this chemical reaction and instead running a stochastic individual-based simulation rather than the analytical, aggregate ODE simulation that you see here.

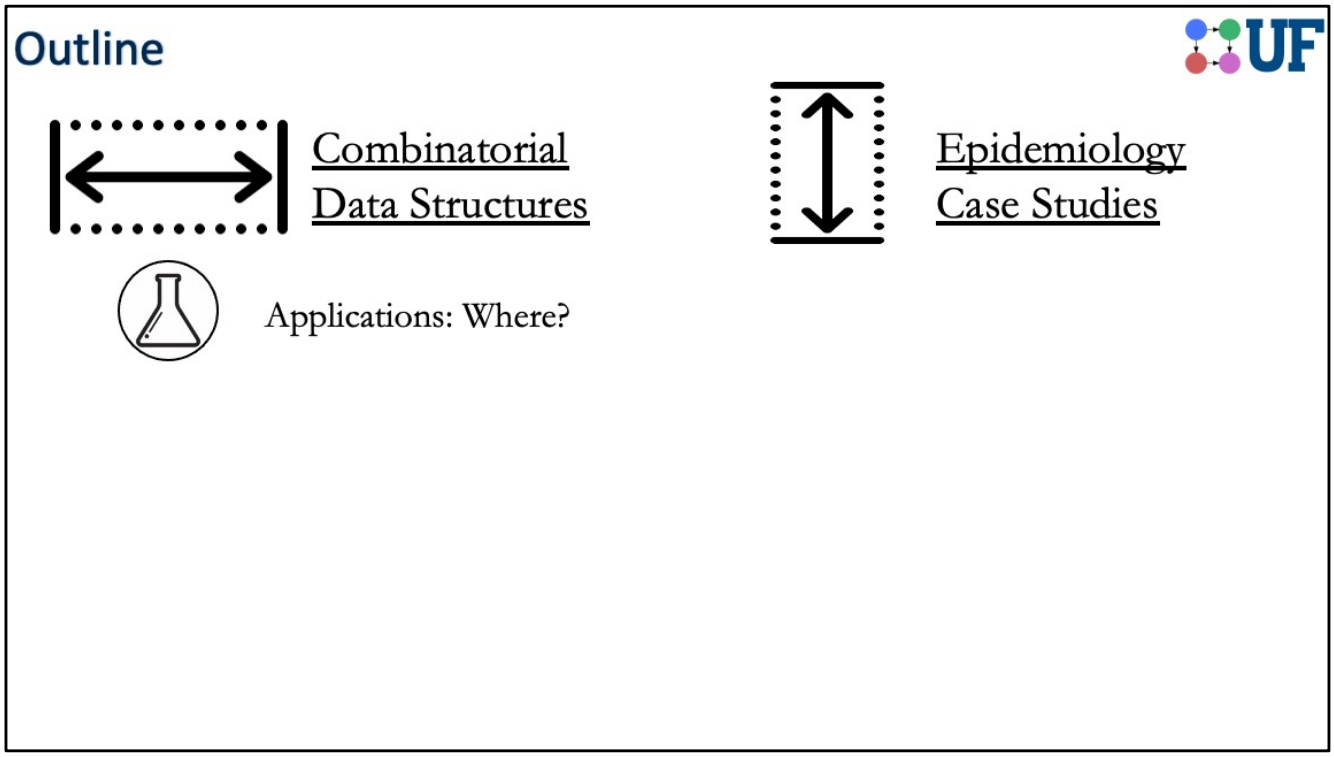
Overall, there are many tasks that we'd like to do that can't be done with arbitrary code. We'll see even giving a thorough specification of our system in the language of differential equations does not help us towards these goals.

## Outline

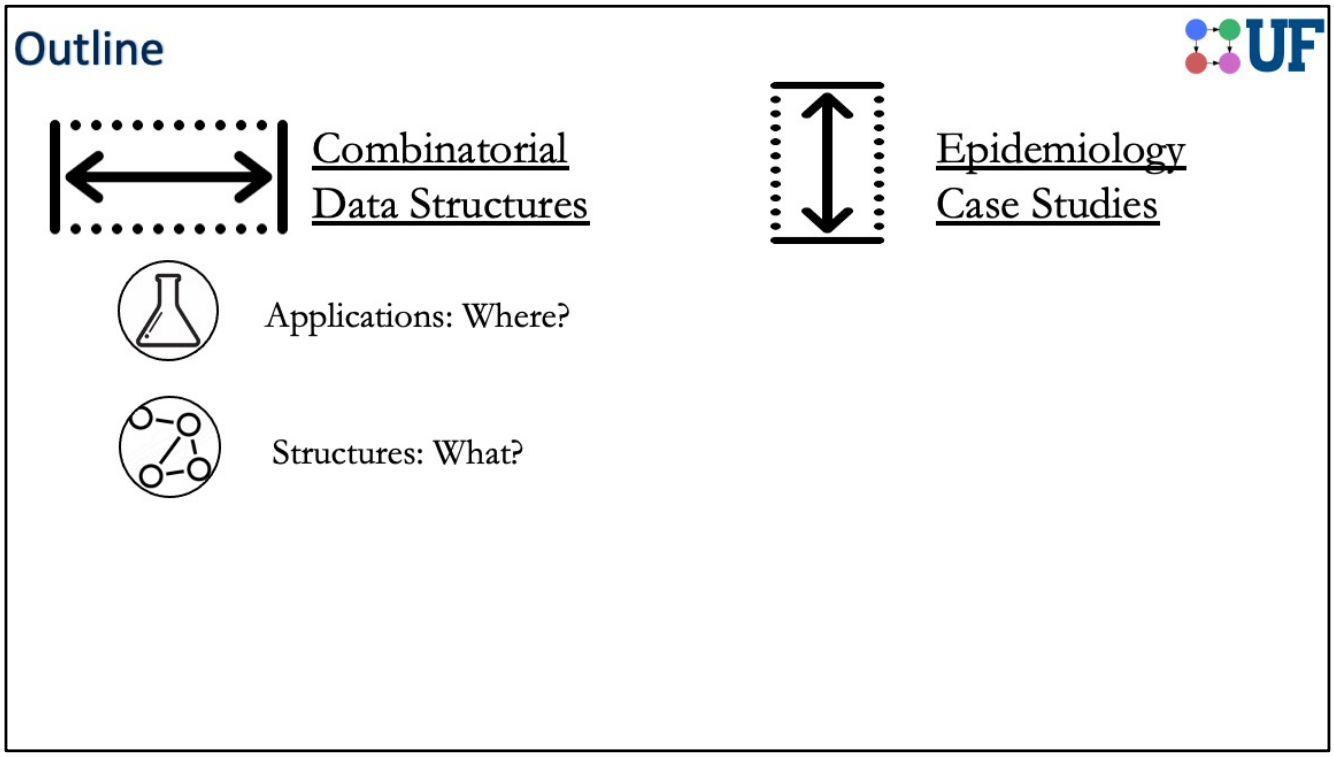
Combinatorial Data Structures

Epidemiology Case Studies

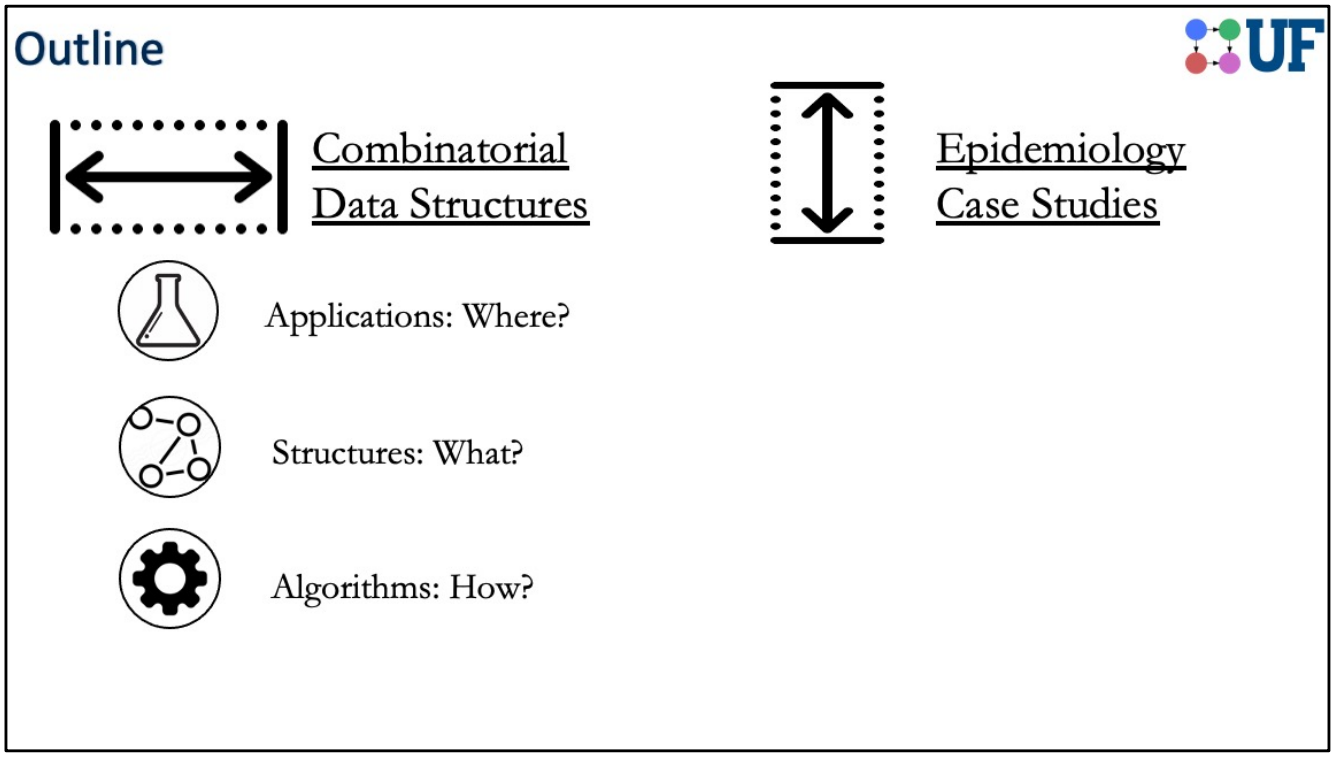
So I want to this talk in two parts, at first giving a bird's eye view with a lot of breadth, and then picking a particular type of scientific model and going in depth.



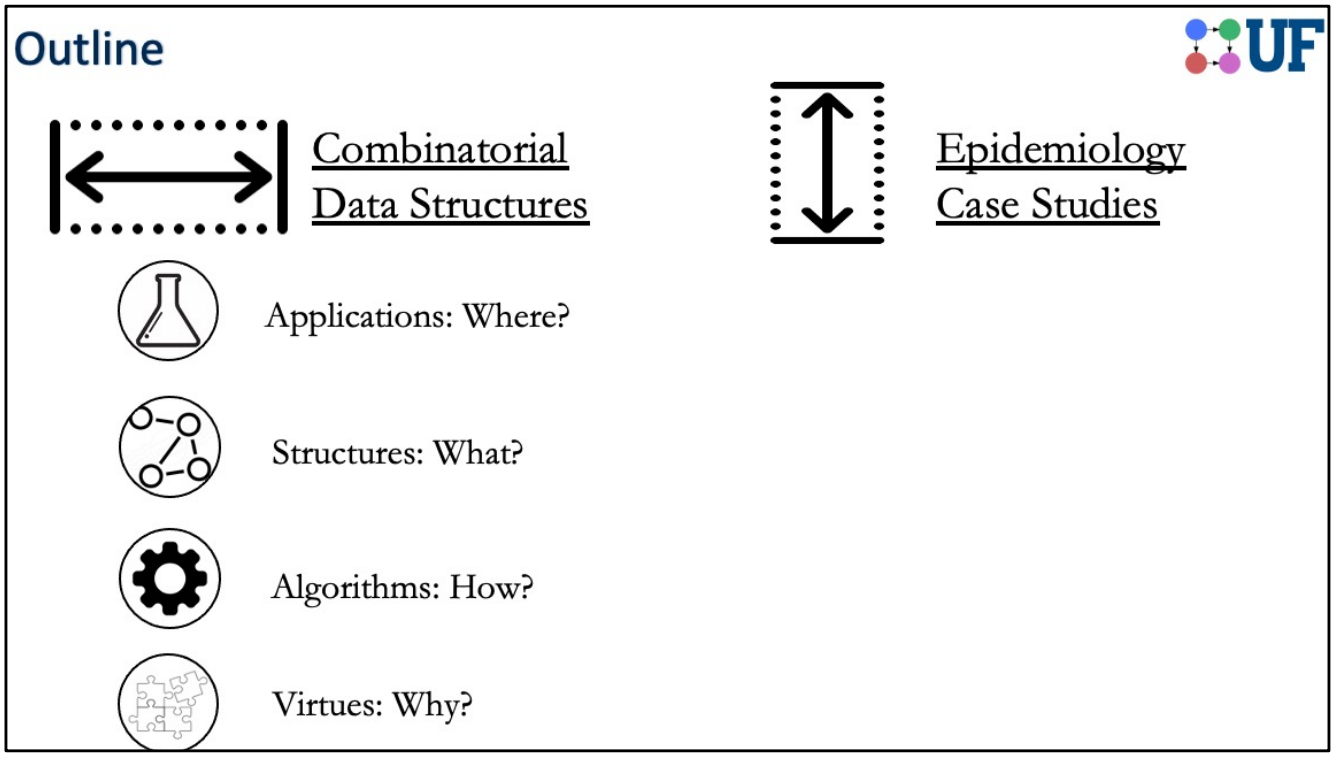

So I'll first say where are real world domains in which we'll apply combinatorial structures



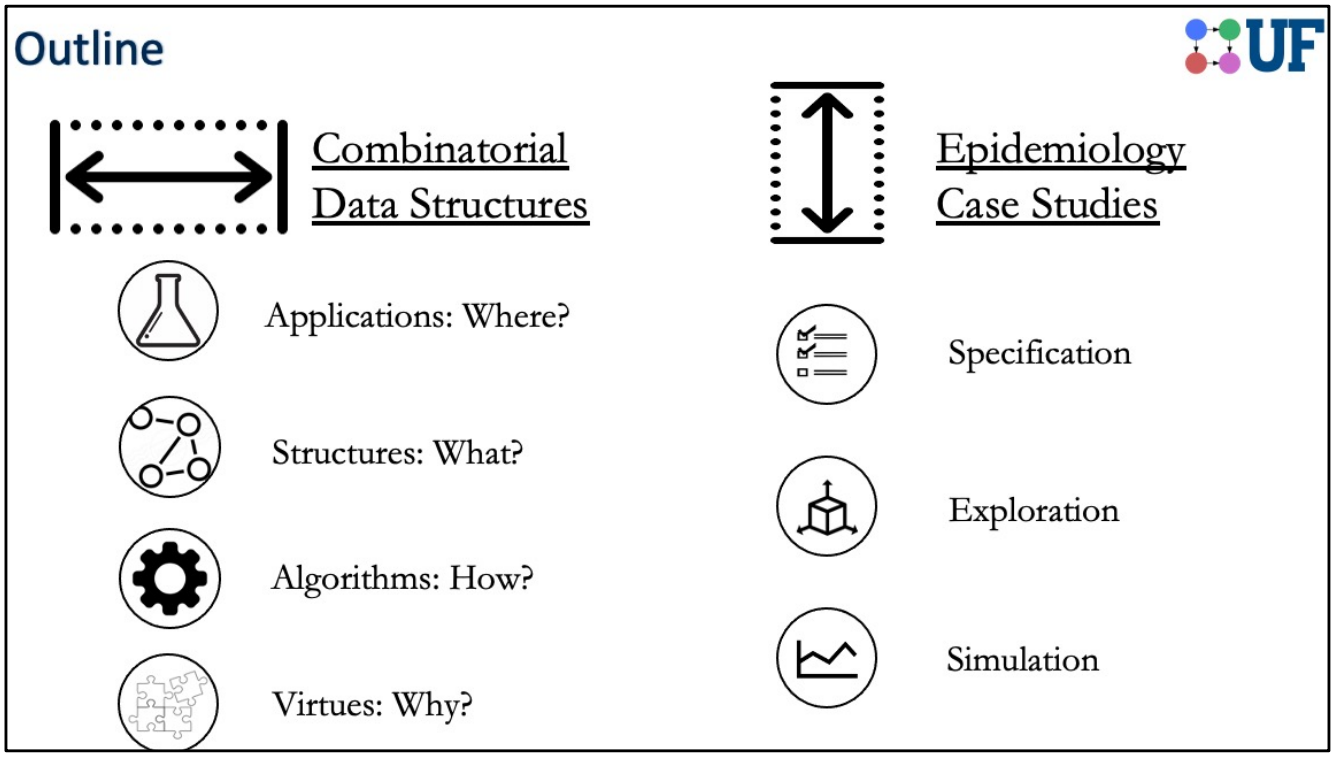

Then I'll get more into examples of what they are.




I'll highlight some algorithms that can operate on them.




And then try to summarize what are the virtues of using them over some standard alternatives.

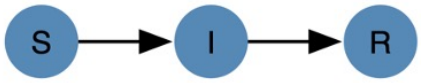



The second part will focus on Petri Net models in epidemiology and talk about declaring, exploring, and simulating them all in a transparent way.

First half I'll make a lot of claims and you'll have to take my word for it... But in the second half I'll give more evidence for the claims I make... though a rigorous understanding of why this works requires some category theory.

## Where: model specification

How to represent the data of a scientific model?




(informal diagram)

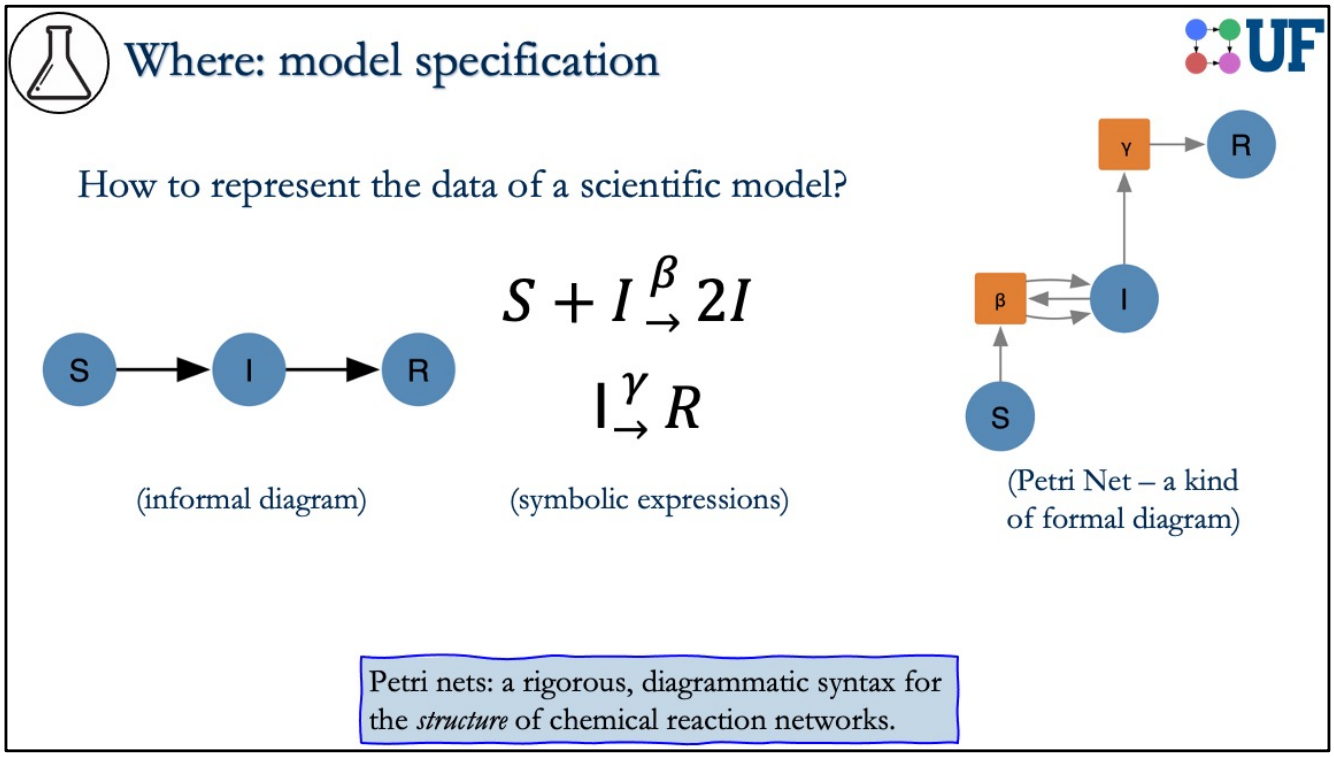
$$S + I \xrightarrow{\beta} 2I$$

$$I \xrightarrow{\gamma} R$$

(formal expression tree)

So to begin with this question, note that a standard solution is to use some sort of informal diagram along with natural language to convey intuition, and then provide some mathematical equations or code which are unambiguous and precise.

As an example which will be featured many times in this presentation, here is a very simple kind of model used in epidemiology called the SIR model. The intuition is that people start out susceptible to some disease, such as COVID. They then can become infected with some probability, and then they transition to recovered with some probability. The expressions that make this precise actually say that there exist two transition events: one susceptible person and one infected person combine to make two infected people, and one infected person becomes one recovered person.




An improvement that we would propose is that there exists a type of diagram which is well defined and intuitive like a graph, but it actually can be unambiguously converted into those formal reaction expressions. This is called a Petri net, and you can see how the multiple arrows are keeping track of the data in the formal expression. I will give a formal definition for it later in the talk.

That might seem like a superficial difference from the equation formalism, but consider now the addition of constraints to some class of reactions we want to consider.

## Where: model specification

How to represent constraints associated with chemical reaction networks?

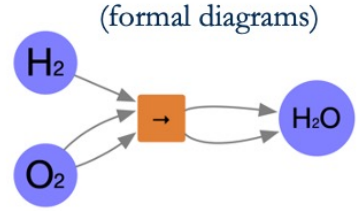
- No more than two reactants per reaction

(formal expression trees)

$2H_2 + NO \rightarrow N_2 + 2\,H_2O$

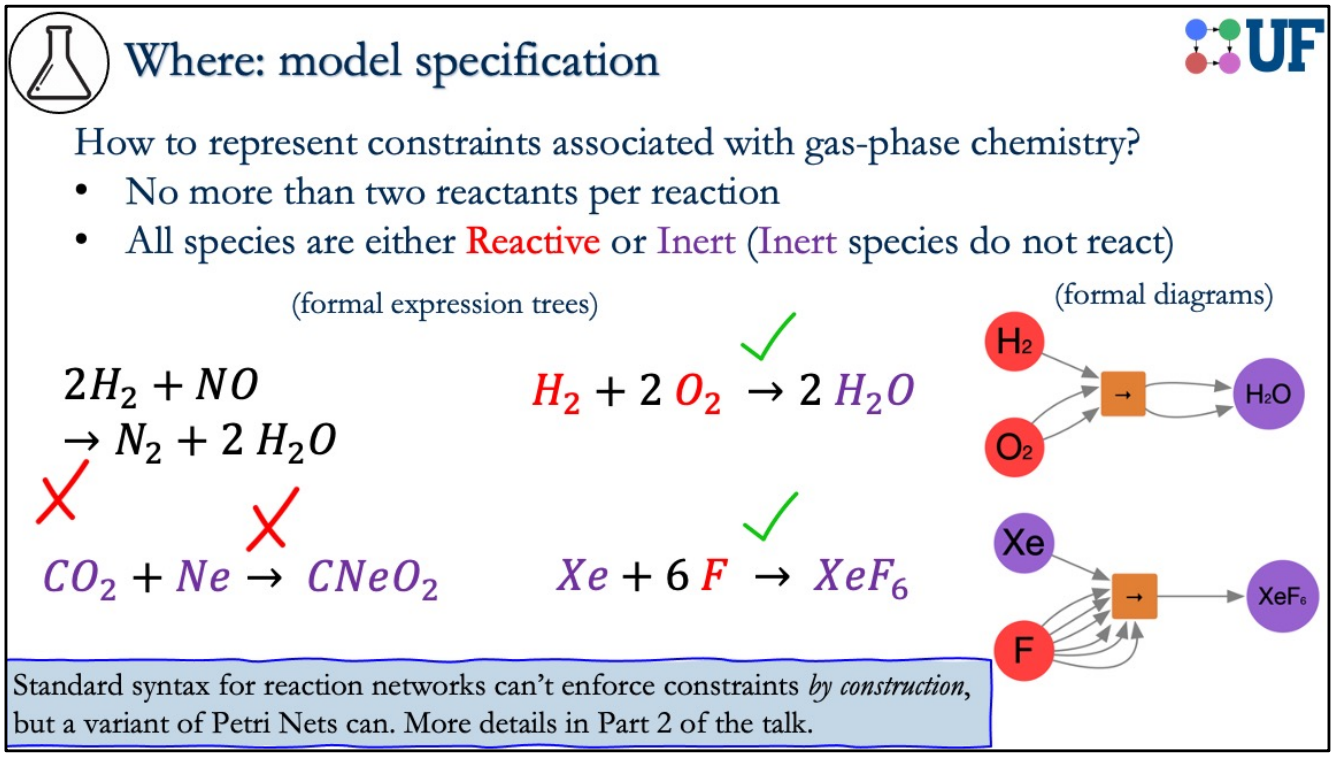
$H_2 + 2\,O_2 \rightarrow 2\,H_2O$

(formal diagrams)




Standard syntax for reaction networks can't enforce constraints *by construction*, but a variant of Petri Nets can. More details in Part 2 of the talk.

In the case of chemistry, we often want to exclude from consideration reactions that contain three things that react at the same time. And you can ask: What data structure do you represent your reactions in such that you automatically enforce this constraint?




I give another more sophisticated kind of constraint below, saying that all species must be tagged as either inert or reactive and furthermore you can't have a reaction between two inert species.

The punchline will come in the second half of the talk, where I describe something very closely related to the petri net you see here that IS able to represent these constraints purely by construction, meaning all possible diagrams you can draw correspond precisely to the subset of reactions which satisfy these logical constraints.

So this is something you can do working in this alternative syntax to the syntax of mathematical expression trees. It's less powerful in what it can represent, but more powerful in what we can do with it. (that's a recurring theme)

## Where: model comparison

- When are two models structurally the same, in some sense?
- When are two models behaviorally the same, in some sense?

$$\left(\begin{tikzcd} C:\Omega_t^0 \arrow[r, "\partial_t"] \arrow[d, "d"'] & \dot{C}:\Omega_t^0 & d\phi:\widetilde{\Omega}_t^3 \arrow[l, "\star^{-1}"'] \\ dC:\Omega_t^1 \arrow[rr, "k\star"'] & & \phi:\widetilde{\Omega}_t^2 \arrow[u, "d"'] \end{tikzcd}\right) \longrightarrow \left( C:\Omega_t^0 \underset{k\Delta}{\overset{\partial_t}{\rightrightarrows}} \dot{C}:\Omega_t^0 \right)$$

(Two formal graphical presentations of heat diffusion)

Diagrammatic representation of multiphysics allows us to check behavioral equivalence, unlike working with the raw PDEs

Patterson et al. "A diagrammatic view of differential equations in physics" (2022)

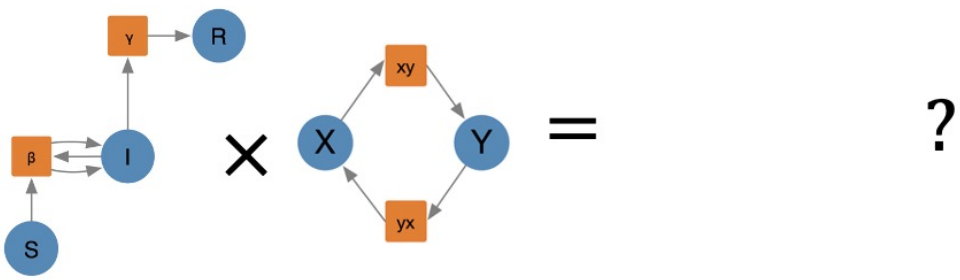
Another application of combinatorial data structures is to compare whether two models are the same, structurally or behaviorally.

In the example here I'm showing, a different kind of formal, graphical representations for differential equations is used to reperesent the heat equations in two different ways (things can be named differently and structured differently).

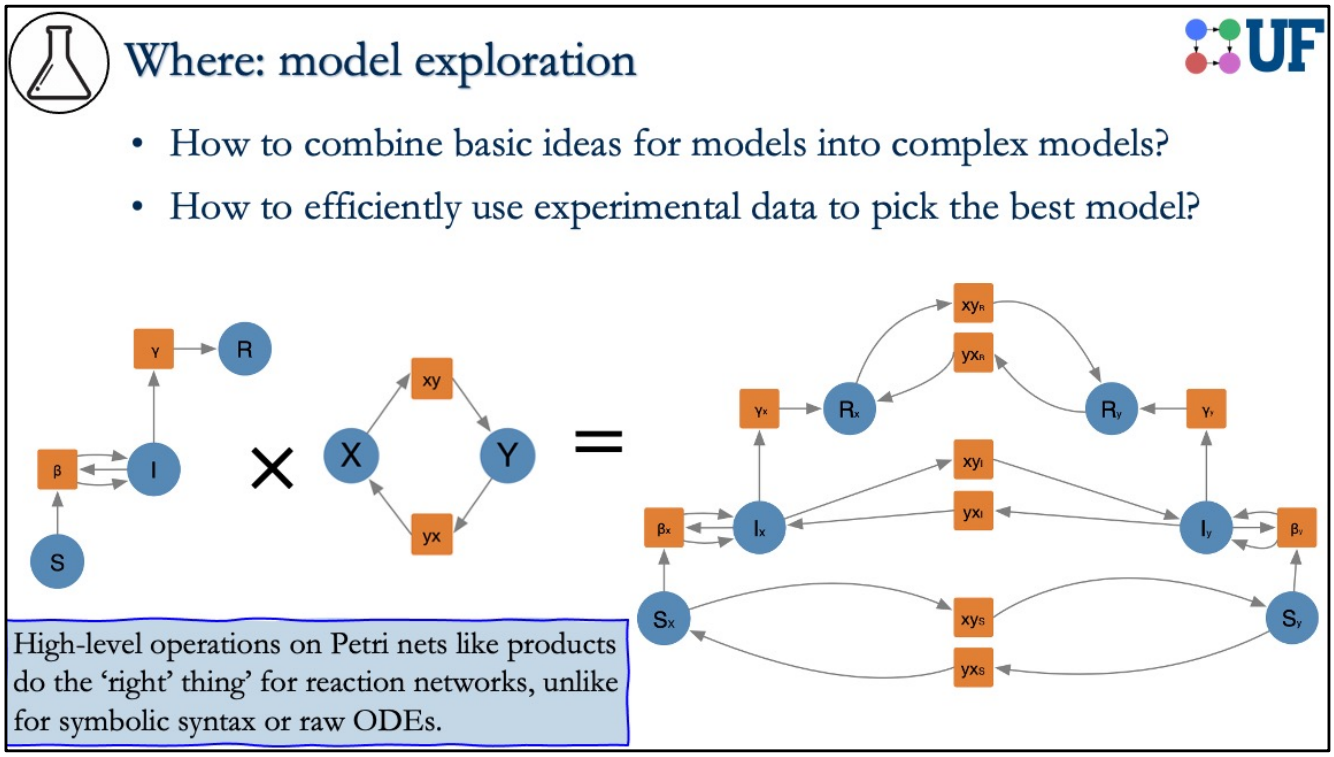
But in virtue of how they are represented, we have a decidable algorithm which computes behavioral similarity by determining whether or not the solutions of one system of equations are all solutions of the other. This is not something you can do if just handed a bunch of PDEs, because again raw mathematical expressions have way too much expressive power, so in general it's rare for decidable algorithms that do useful things with them.

## Where: model exploration

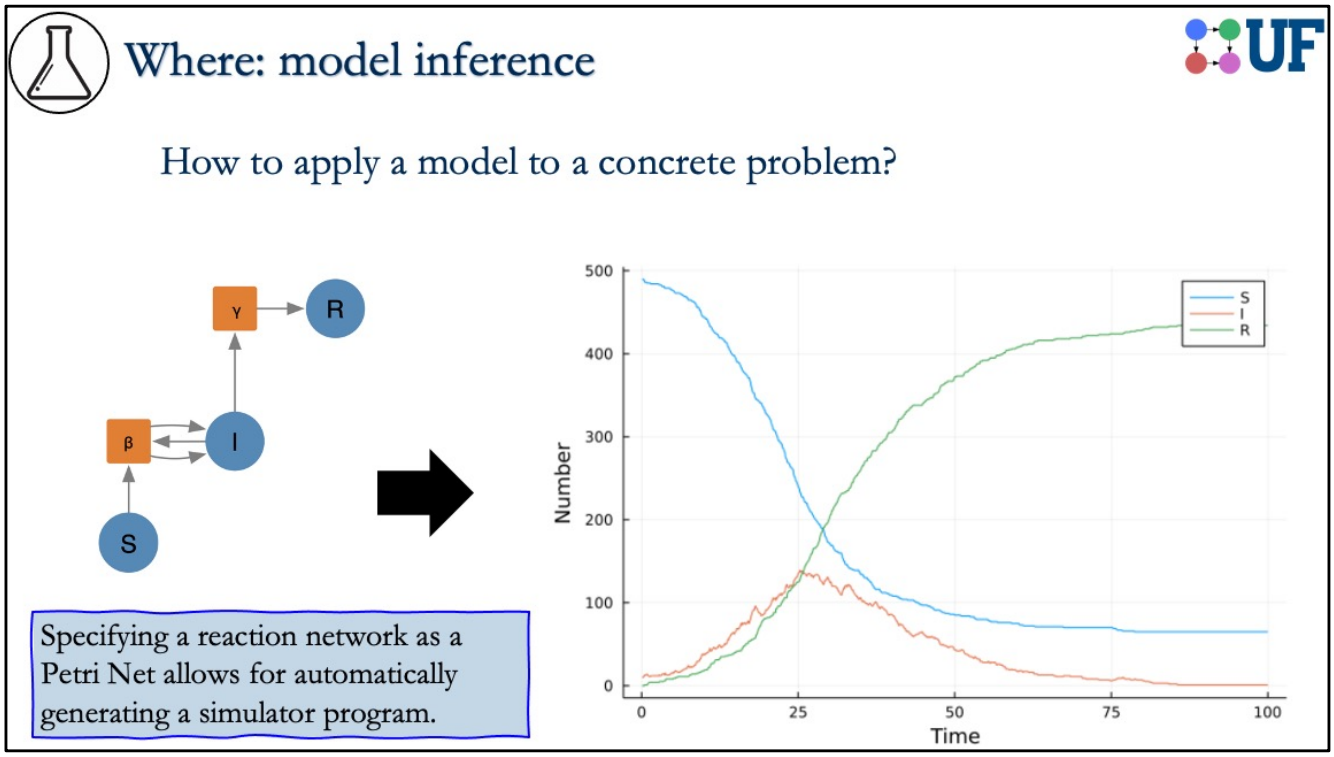
- How to combine basic ideas for models into complex models?
- How to efficiently use experimental data to pick the best model?



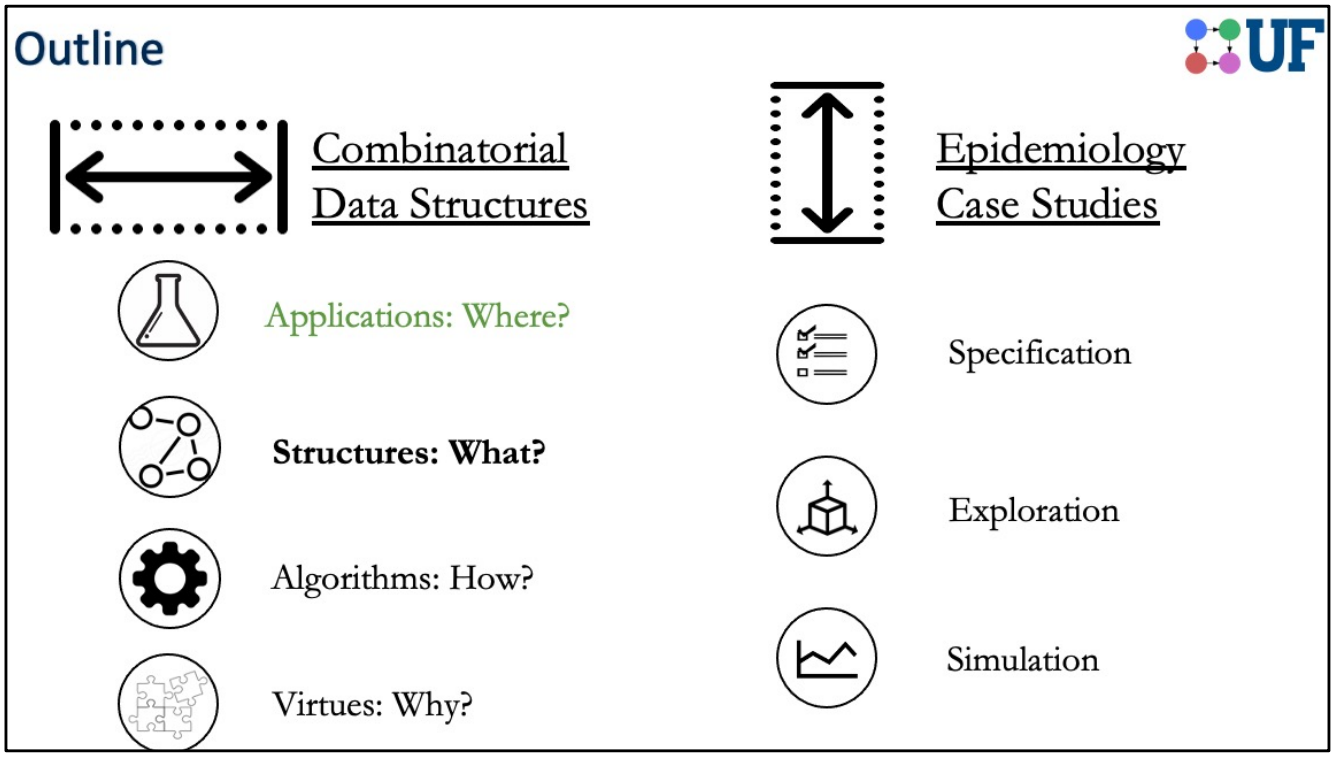

Another place we want to deploy these representations is the model exploration problem. I'll go more into detail about this later, but as a teaser, consider what it would mean to combine two Petri net models, one of which is the SIR model and another which we call a "two city model", where people live in United States and Europe and can move between them.




The intuitive answer can be elegantly using these Petri net representations (category theory tells us what the right notion of product is for them), whereas taking the product of the literal differential equations wouldn't yield anything useful.

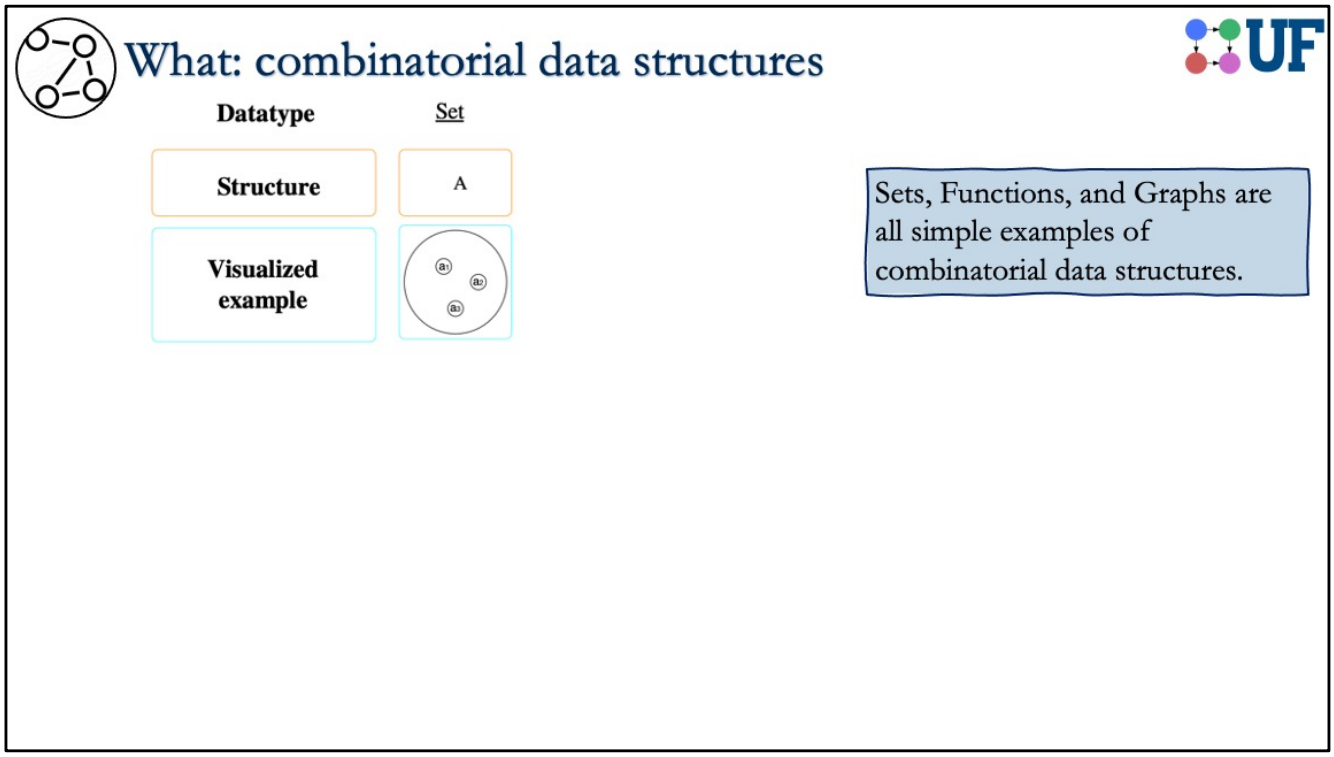



The last application area I want to highlight is the problem of model inference. If you were handed this petri net on the left, could you automatically generate a simulator which accepts initial conditions and reaction rates and produces a plot on the right? This is a useful feature because scientists really shouldn't be writing code – we envision a future where they can work purely in their true domain, and with the right representations of their models the solvers should be automatically generated.

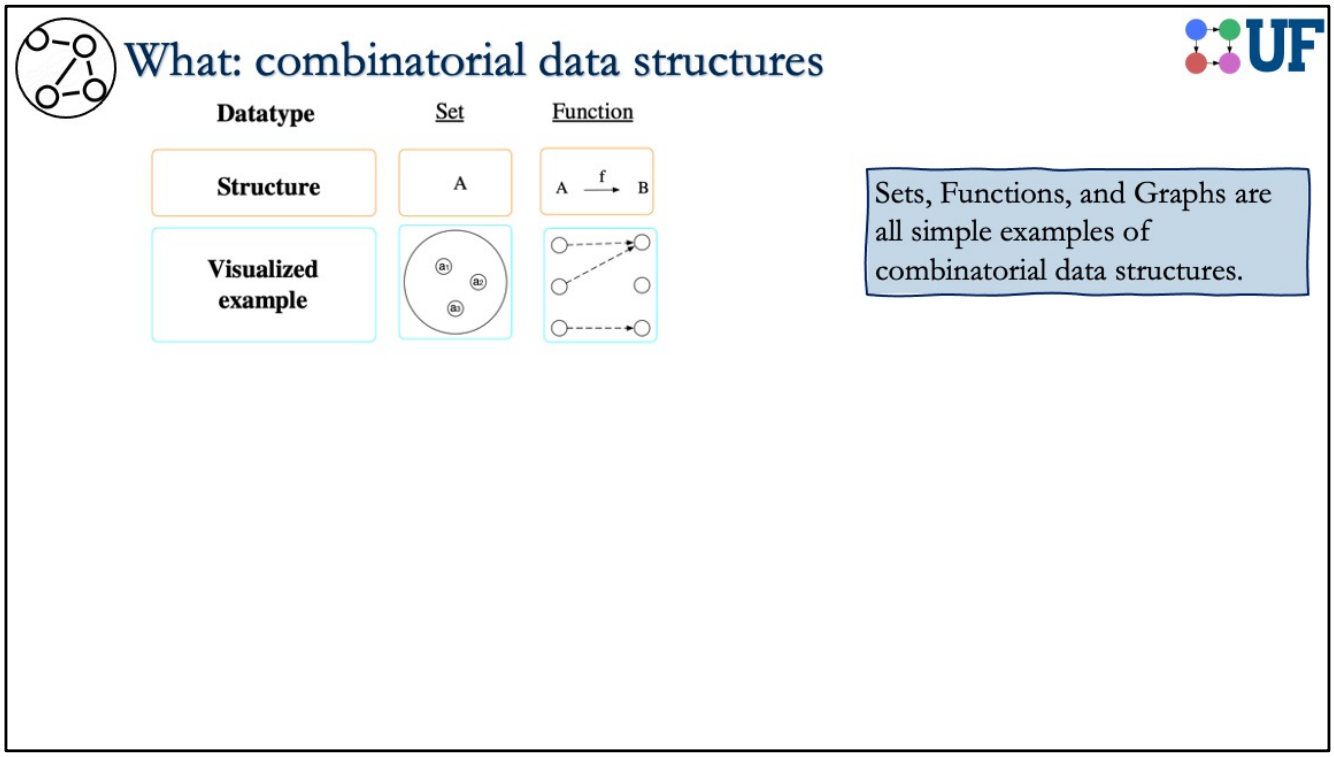



Alright that was whirlwind of applications, but it's good to know the motivation for why we're doing this. Now I'll say more about what these combinatorial structures are.

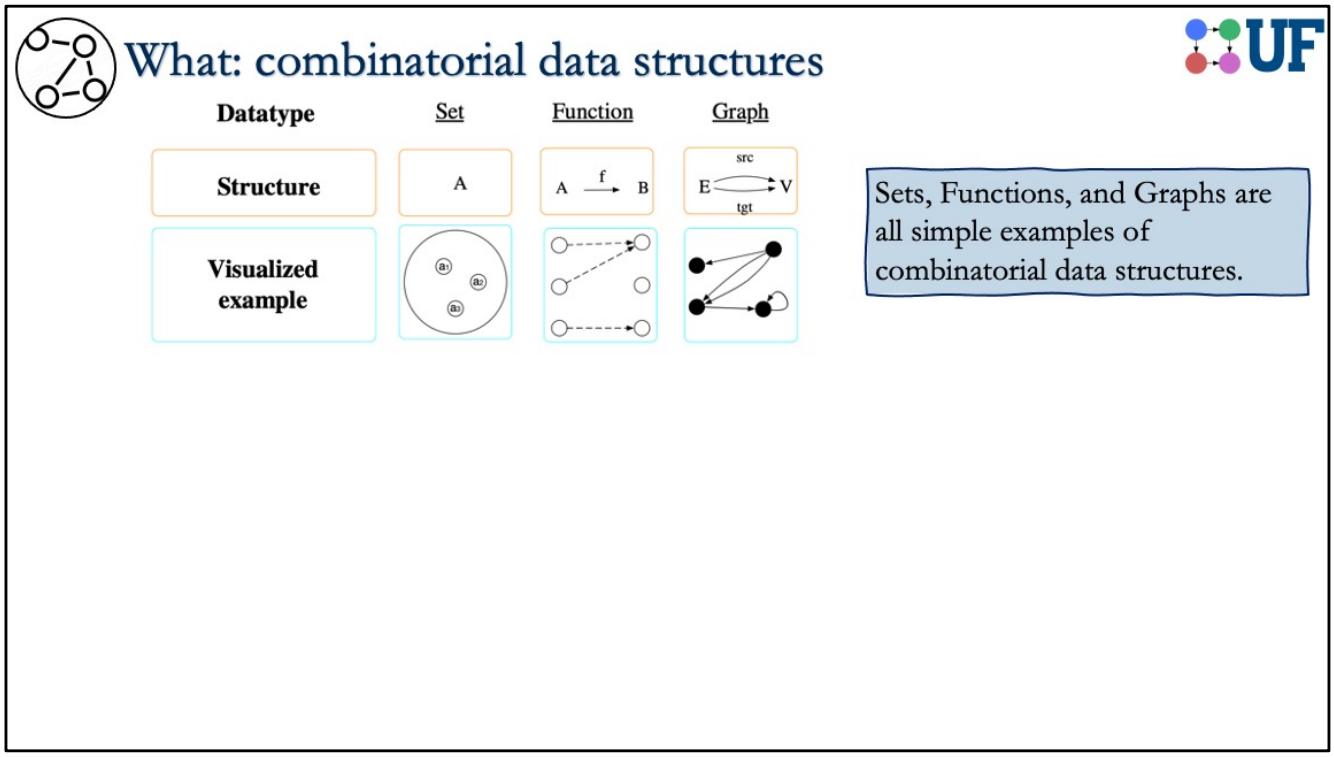
Now the next slide has a lot of examples of the data structures that make those applications possible. CAVEAT: it will be abstract and you don't actually need to fully grasp them in order to use them or understand the applications. What I'll show is what makes the math behind the picture




So combinatorial data structures I think of as elements that are connected. Here are some basic examples. Starting with a set which is just a bunch of unconnected elements, no structure

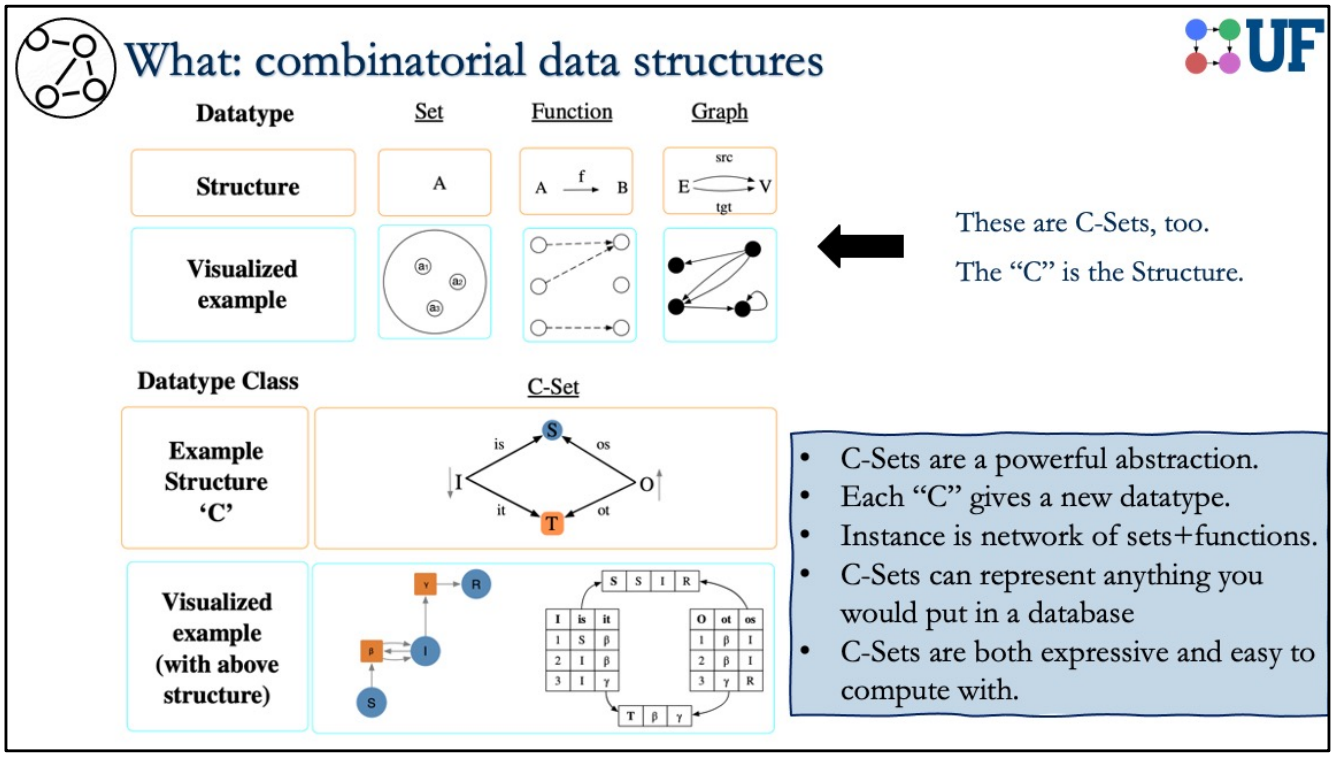



Functions then allow us to connect elements living in different sets.

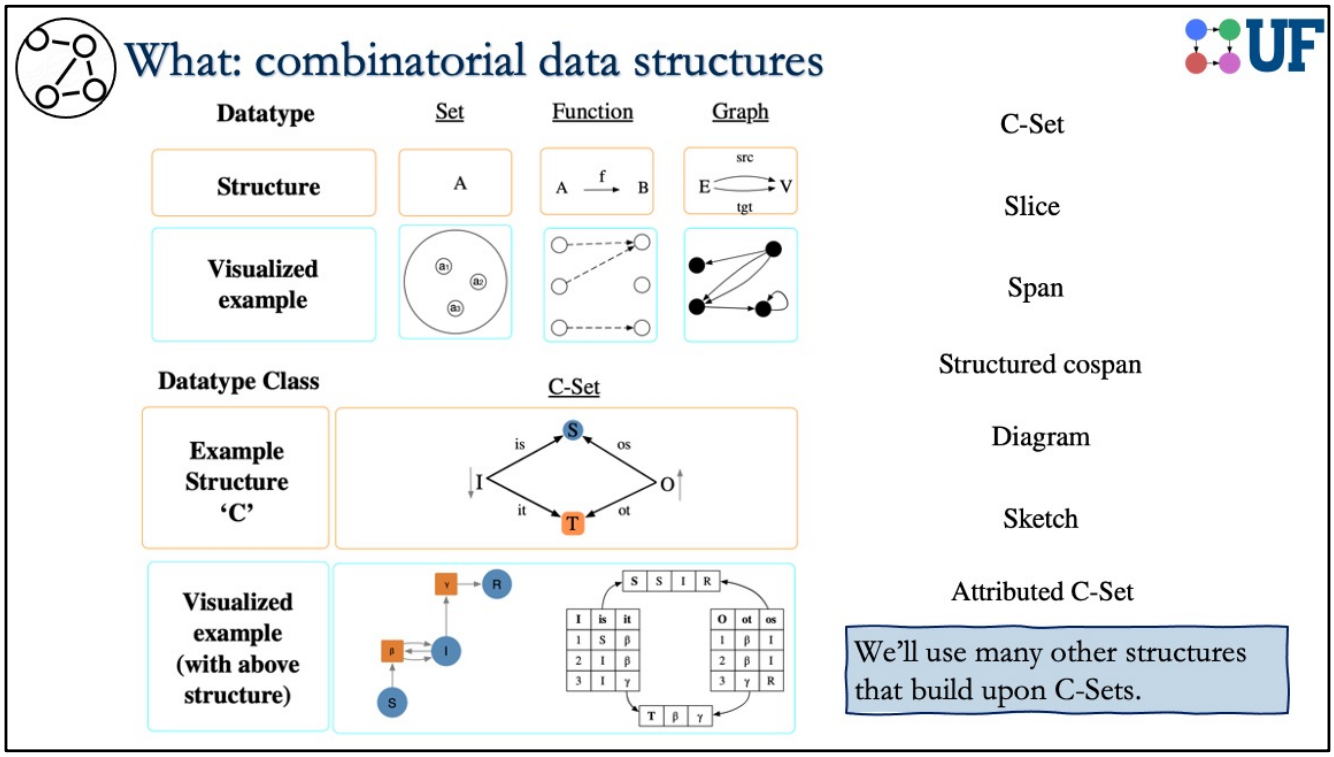



Directed graphs are really just a pair of functions, where the edges themselves are a set and the functions say where their source and target are.

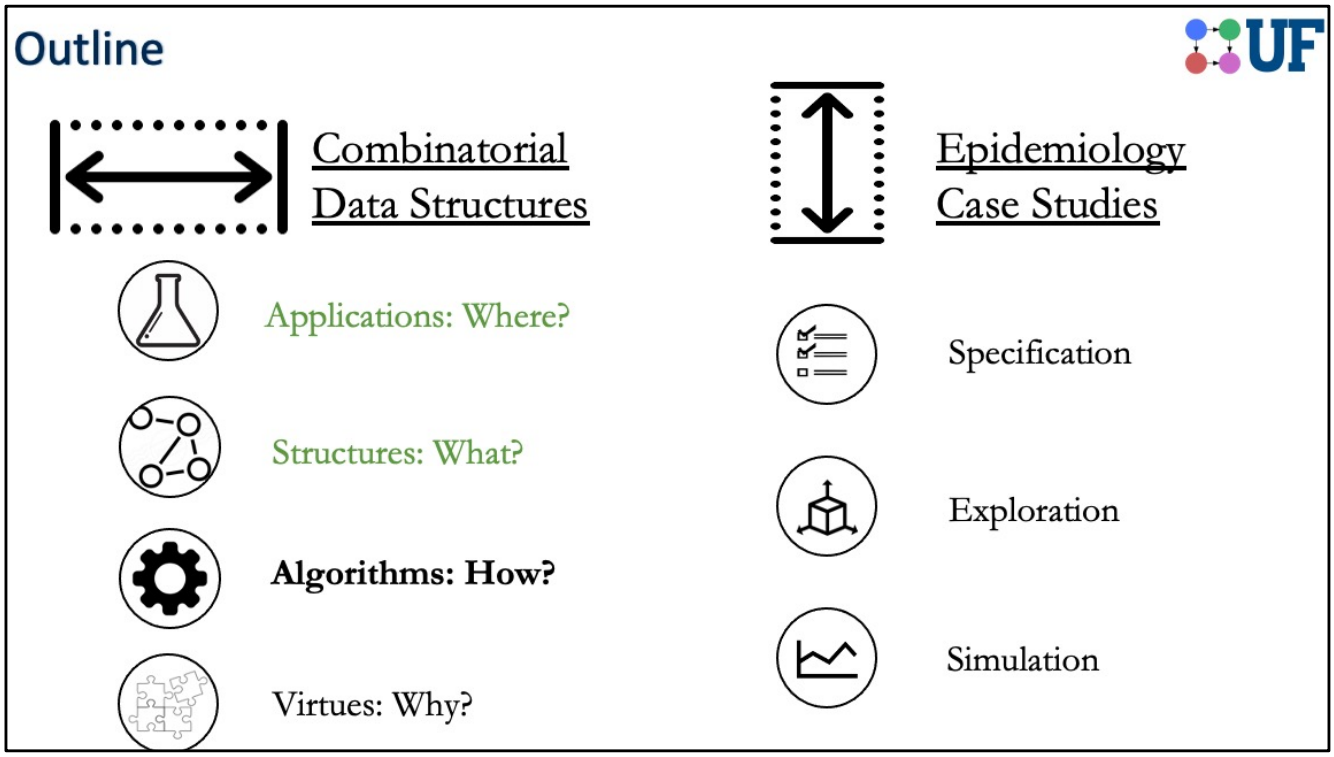
As you can see, each of these examples is going up a level of abstraction. The next level of abstraction requires us to think of the structure itself as a variable parameter.



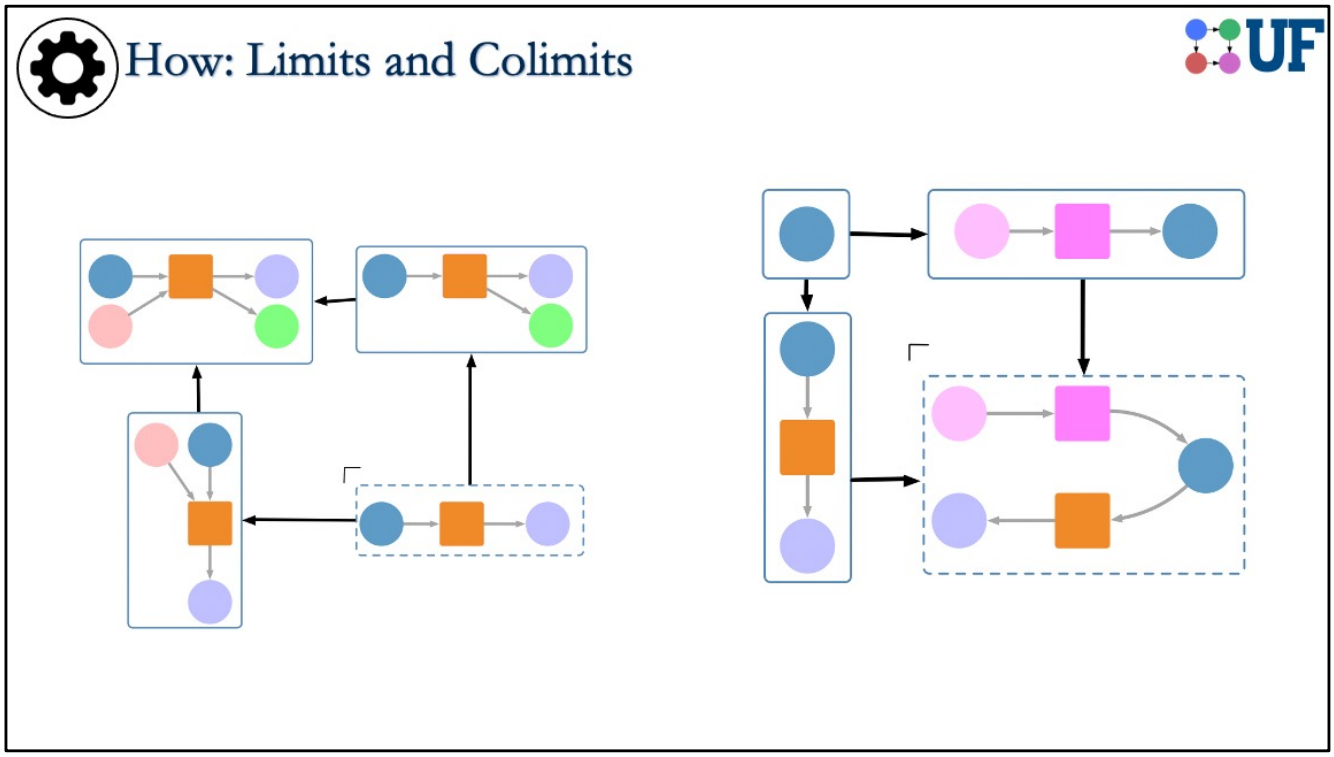

Call this parameter "C" (it turns out it's something called a category, but think of it as a graph).That's to say, we're now working with data structures, where the shape of the structure is itself a parameter. This is really powerful because it means if you write a C-set algorithm to work with arbitrary C, it's like you've implemented an algorithm for a massive variety of very different looking datatypes.



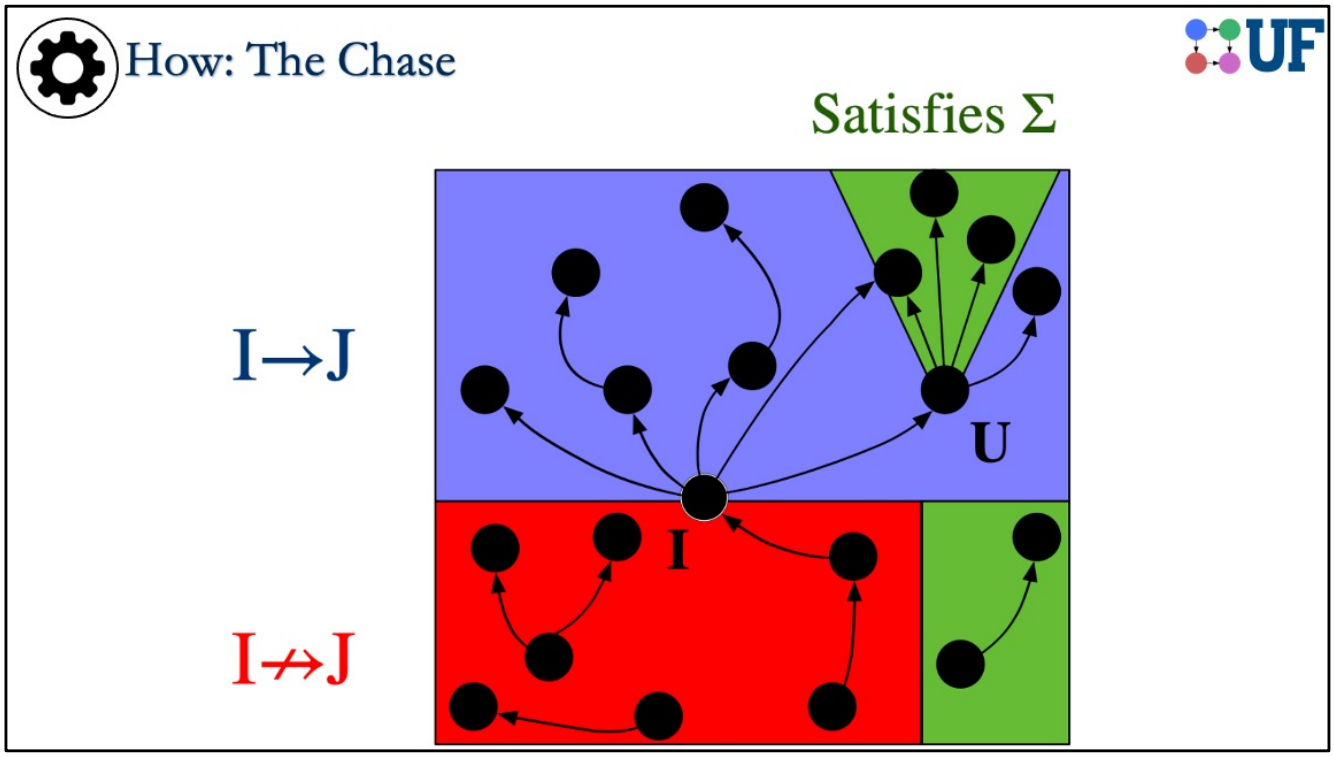

On my list of structures on the right, C-sets are just the starting point, and all the ones below it actually build more abstractly upon C-sets. But we'll introduce those as needed once we start talking about the key projects I'd like to showcase.




I wanted to give a quick overview of some algorithms that operate on these combinatorial structures but that gets a bit technical and it's probably best to show applications of those algorithms, so I'll leave skip those slides for now.



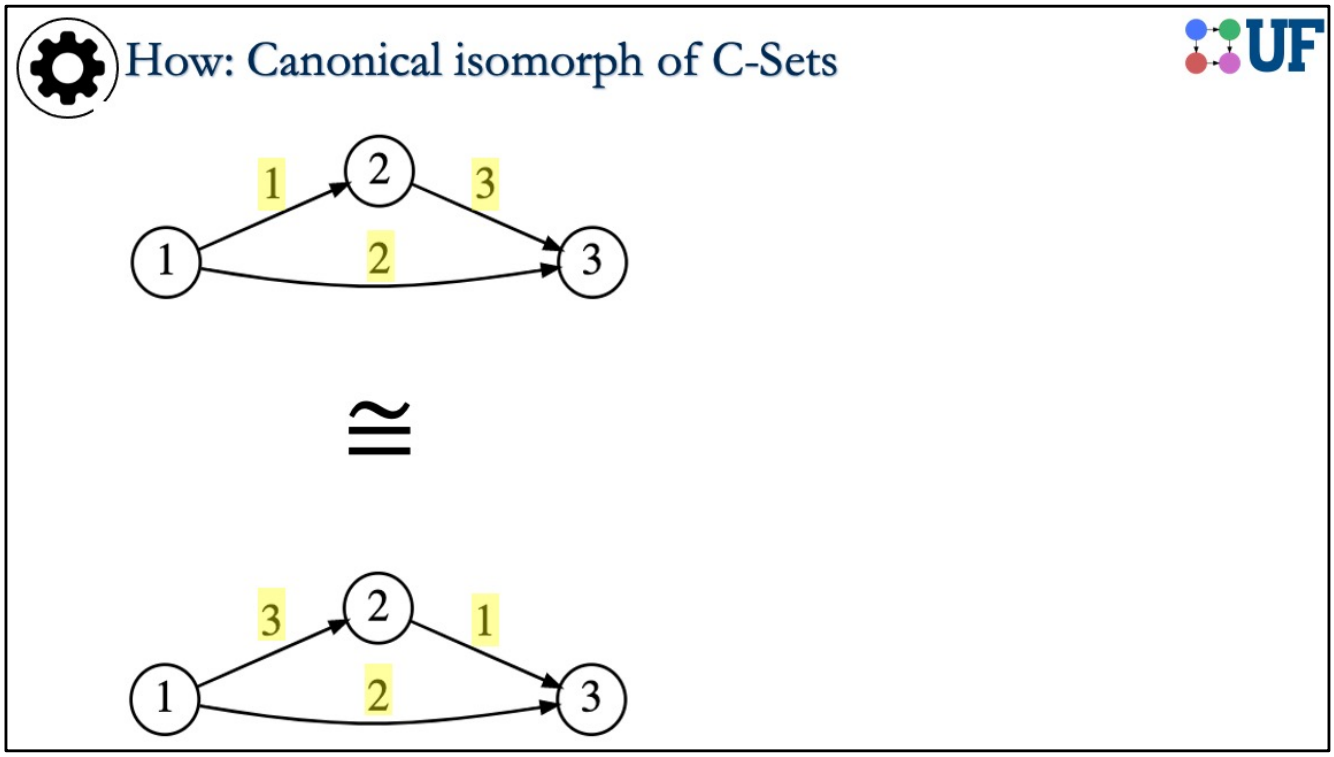

HIDDEN
Limits and colimits respectively, loosely correspond to the ideas of solving equations and gluing structures together.




HIDDEN

The chase is an algorithm which takes a model, call it ”I” there at the center of this cartoon, and it tries to make it satisfy some constraints which we’ll call sigma.

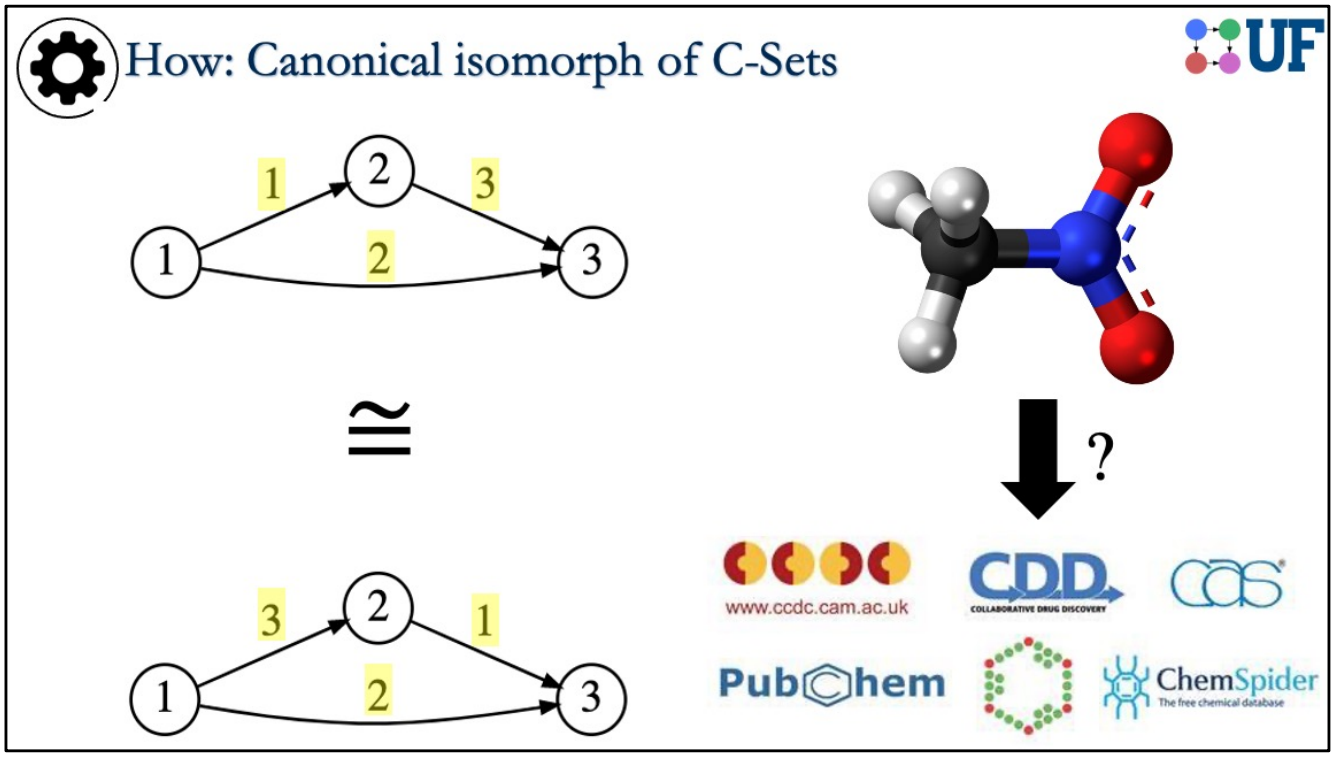
It doesn’t want just any model that satisfies this though. If you imagine these arrows as adding assumptions to a model, then the algorithm is finding the best possible way to make I satisfy the equations, by adding the least possible assumptions. This shows up in a surprising number of contexts.




HIDDEN

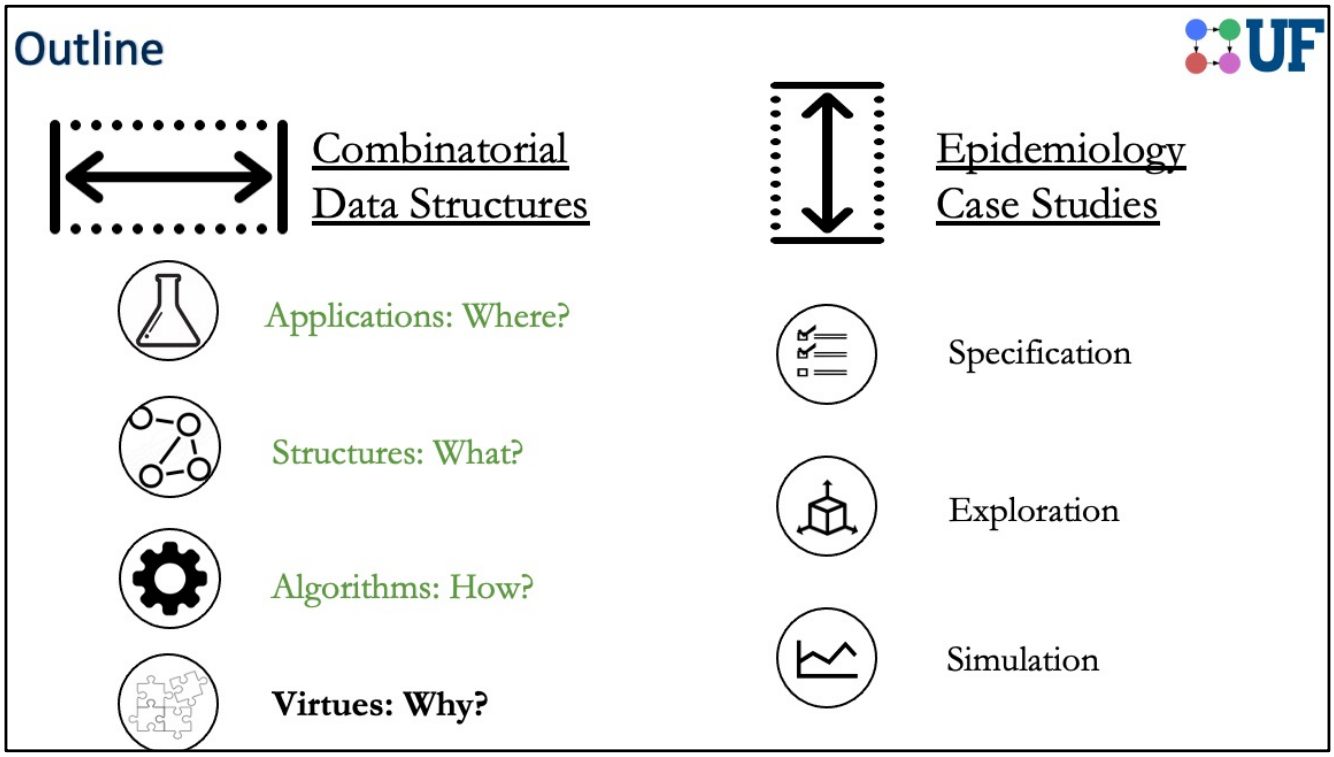
Lastly, there is this notion of symmetry where two models really are morally the even though they've been labeled differently. This is isomorphism.

This algorithm finds a specific labelling that is equal for all isomorphic C-Sets.



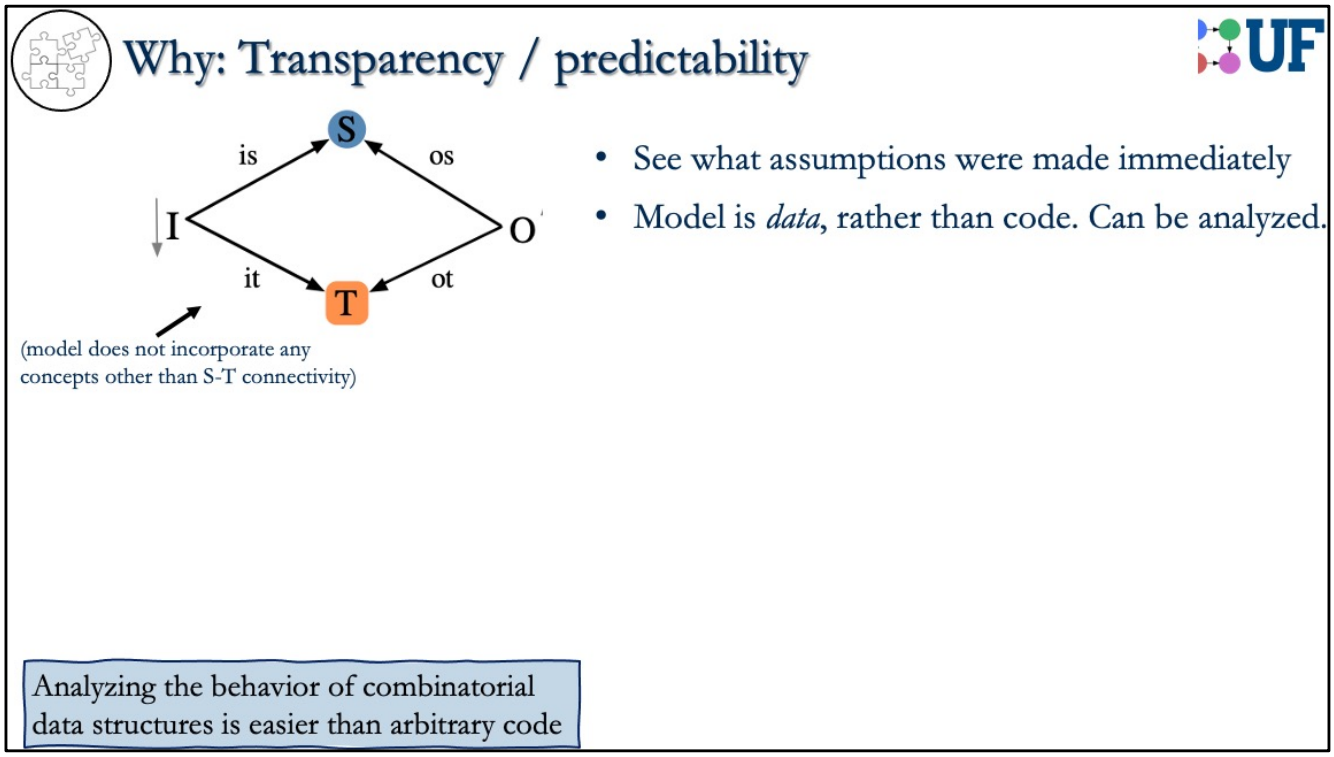

HIDDEN

This is relevant for scientists who, for example, might have a molecule and want to check if it exists in some massive databases. If we don't compute a canonical representation of ordering all the atoms and bonds, then it can be very inefficient to check, for each molecule in all these databases, whether or not there exists an isomorphism.




I'd rather focus on why these combinatorial structures are a healthier way of representing knowledge.

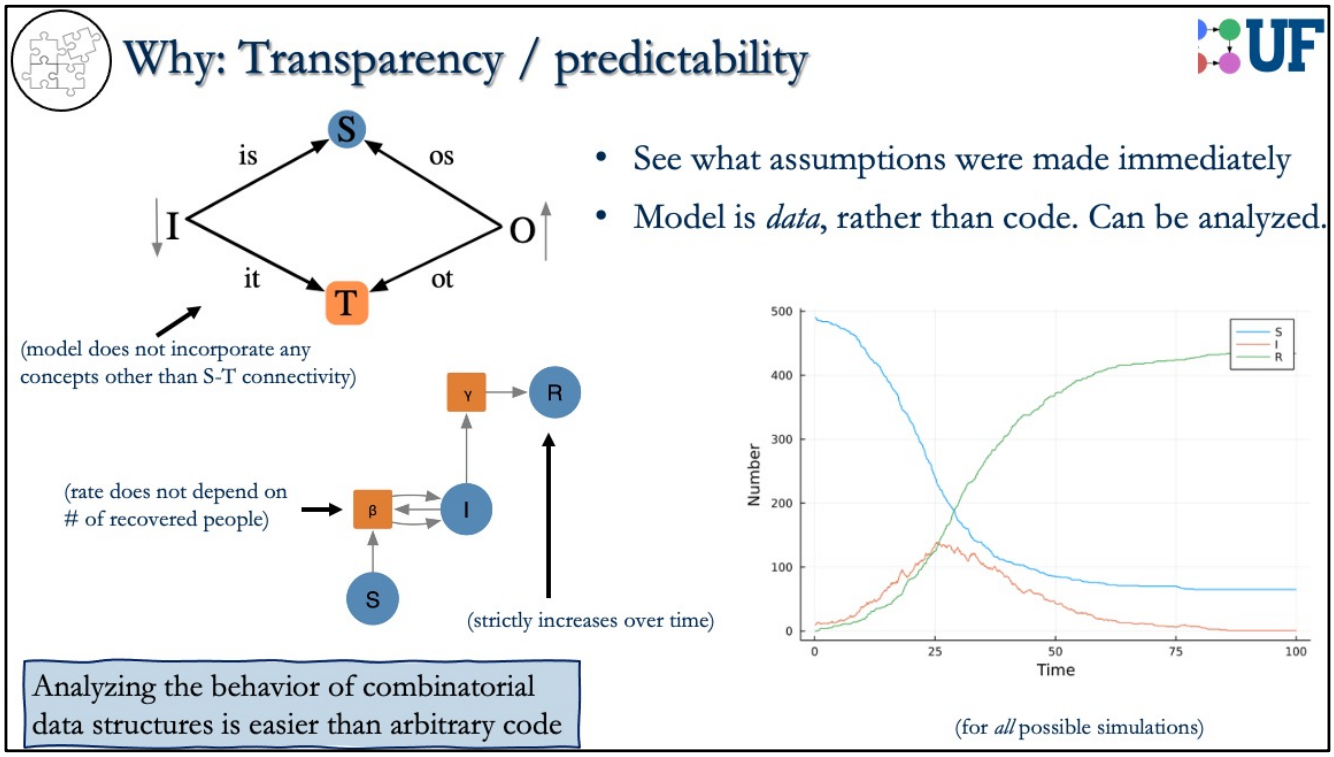
The theme of these virtues is that there are really nice things you want to do with models but cannot do them if models are code or math. You can only do them when you represent your model as these (rigorous) pictures.




So what are these benefits? The first one I call transparency or predictability.

By separating the structure of the model from its behavior, you can see assumptions that have been made.

Without even looking at a particular Petri Net, by knowing the structure of it's schema we can see what sorts of concepts it could possibly contain.




Then, looking at the structure of the Petri net, even a simple computer algorithm can deduce that the number of recovered people will strictly increase for all possible simulations.

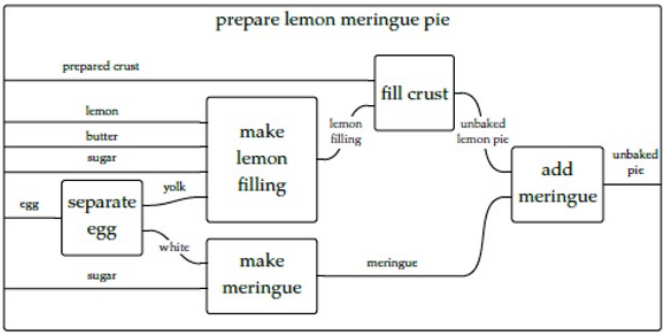
This is pretty simple stuff here, but when I talk later about encoding constraints into the structure, you'll see this is a powerful point.

In general, analyzing behavior is easier than with arbitrary code or arbitrary math.

# Why: Generality of scope

- Many different applications share the same syntax
- E.g. these are all applications of *wiring diagrams* (a particular kind of C-Set)

General processes




Explicit distinction between syntax and semantics makes it easy to generalize models

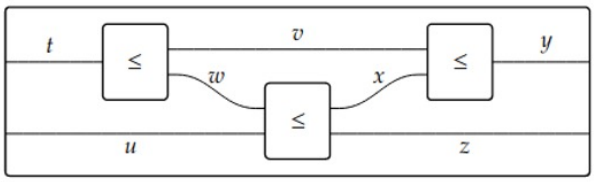
Fong and Spivak. "Seven Sketches in Compositionality" (2018)

So this slide visualizes a directed wiring diagram, which is actually a specific type of C-Set. You can see here it represents general processes such as making a pie.

# Why: Generality of scope

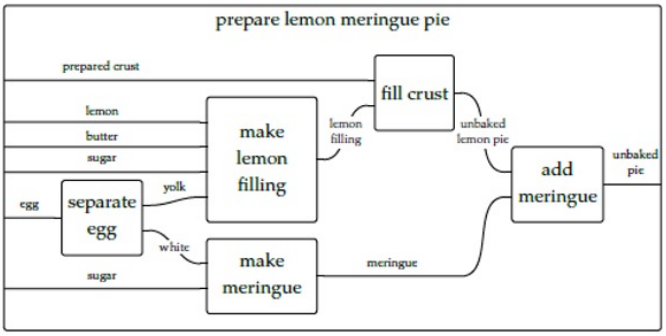
- Many different applications share the same syntax
- E.g. these are all applications of *wiring diagrams* (a particular kind of C-Set)

Inequality reasoning




$t \leq v + w$ and $w + u \leq x + z$ and $v + x \leq y$ implies $t + u \leq y + z$

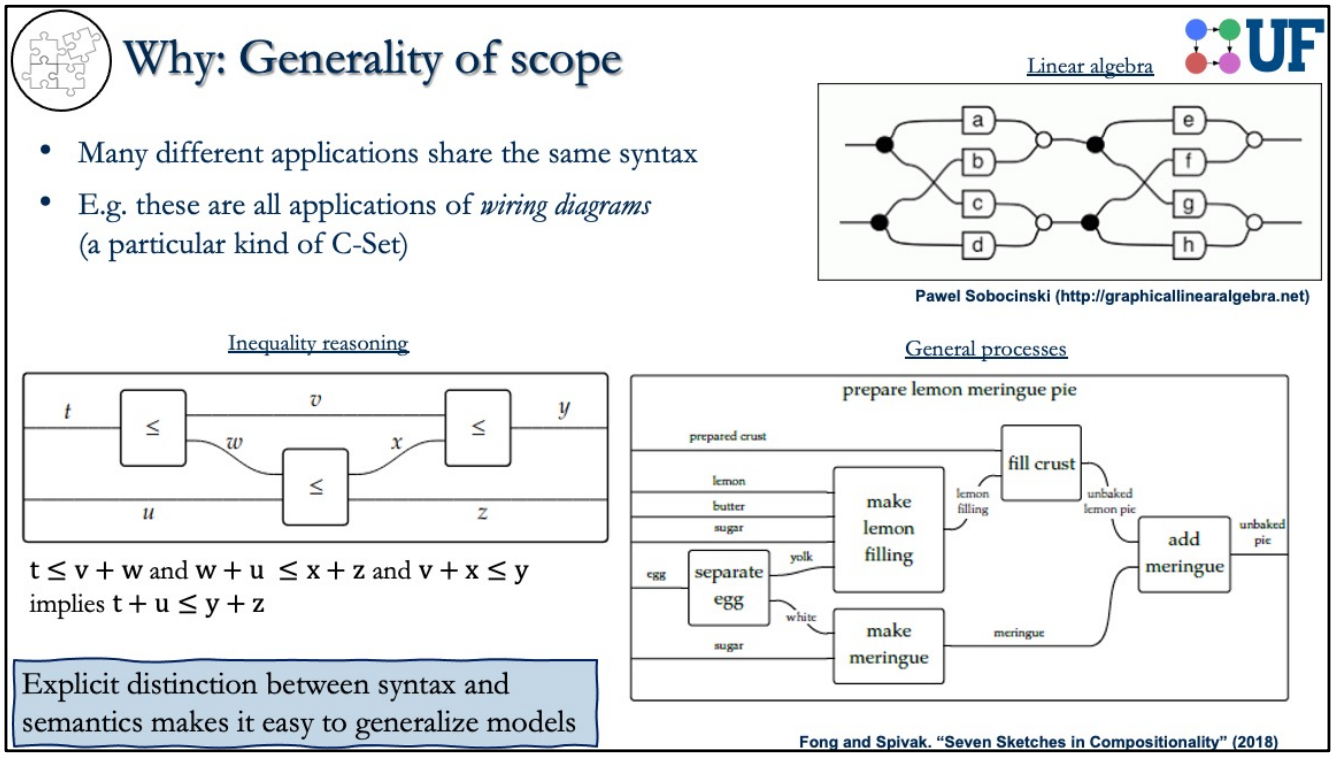
General processes




Explicit distinction between syntax and semantics makes it easy to generalize models

Fong and Spivak. "Seven Sketches in Compositionality" (2018)

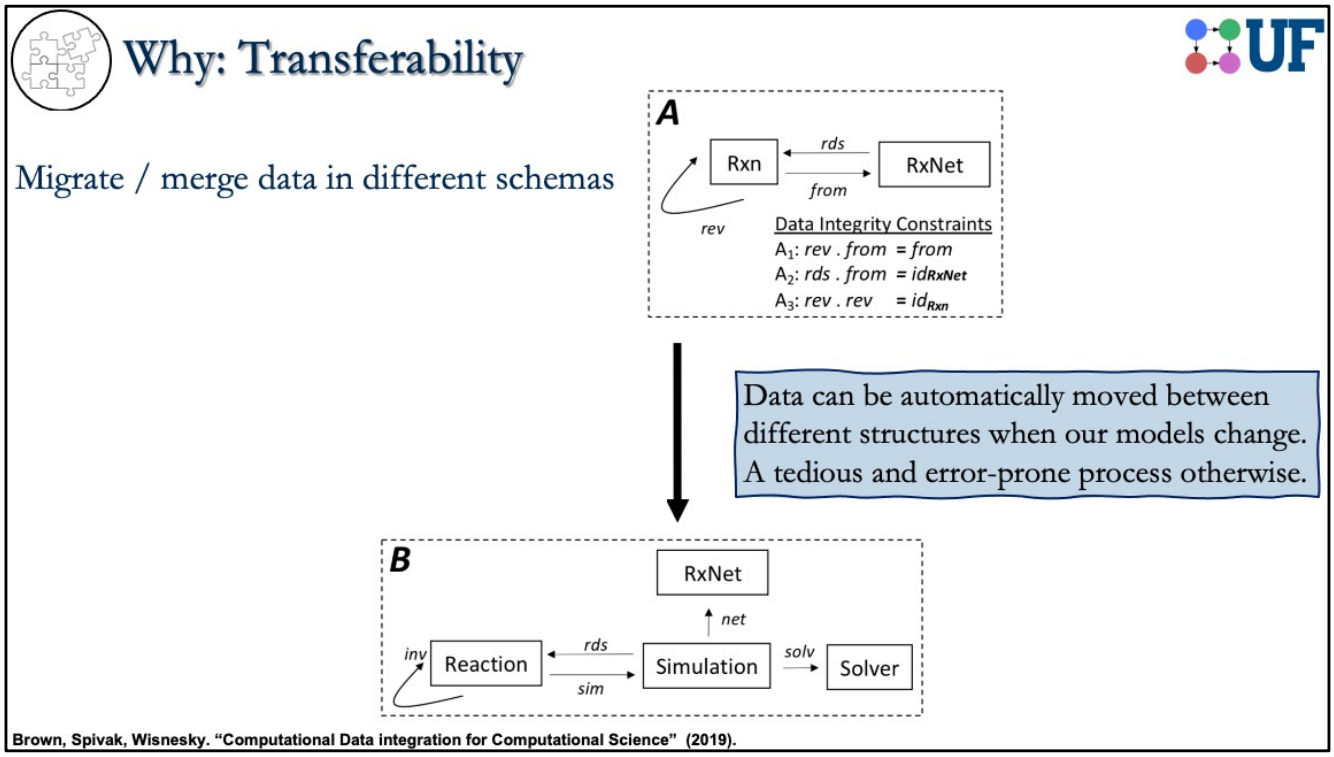
But this same syntax allows us to represent mathematical proofs of a certain kind.




And furthermore we can even represent linear algebra with these diagrams, which becomes a lot less mysterious when it is visual. (THERE'S A LEARNING CURVE.)

When you write code for wiring diagrams in general, it becomes applicable in a wide variety of circumstances (obviously, also code for C-Sets in general has wide applications).

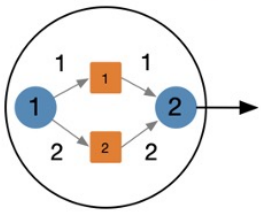
Basically, syntax and semantics are two independent knobs we often want to tweak, so keeping them separate makes it very easy to do the kinds of generalizations that we naturally want to do.




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

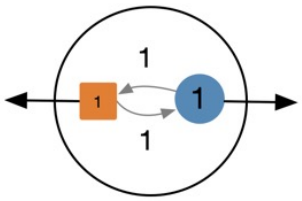
# Why: Hierarchical design

"Find all states which are the output of two transitions"




```
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
       …
```

"Find all catalysts and the reactions they catalyze"




```
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
```

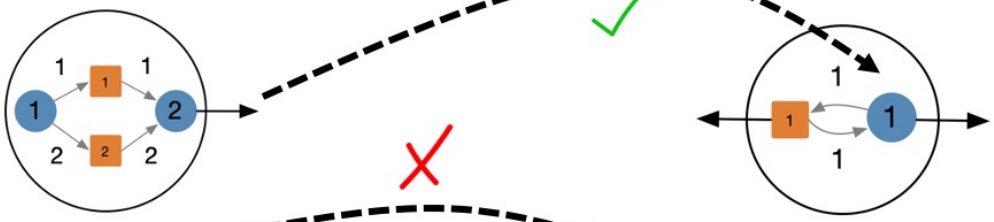
I want to highlight the hierarchical design virtue by considering querying a database. If you're not familiar with this problem, just imagine that, if have a gigantic petri net, we need to use a special language that can efficiently extract data from it. This language, SQL, is at the bottom here.

But we also have a special graphical syntax which corresponds to SQL queries, and this representation has an important advantage.

# Why: Hierarchical design

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"

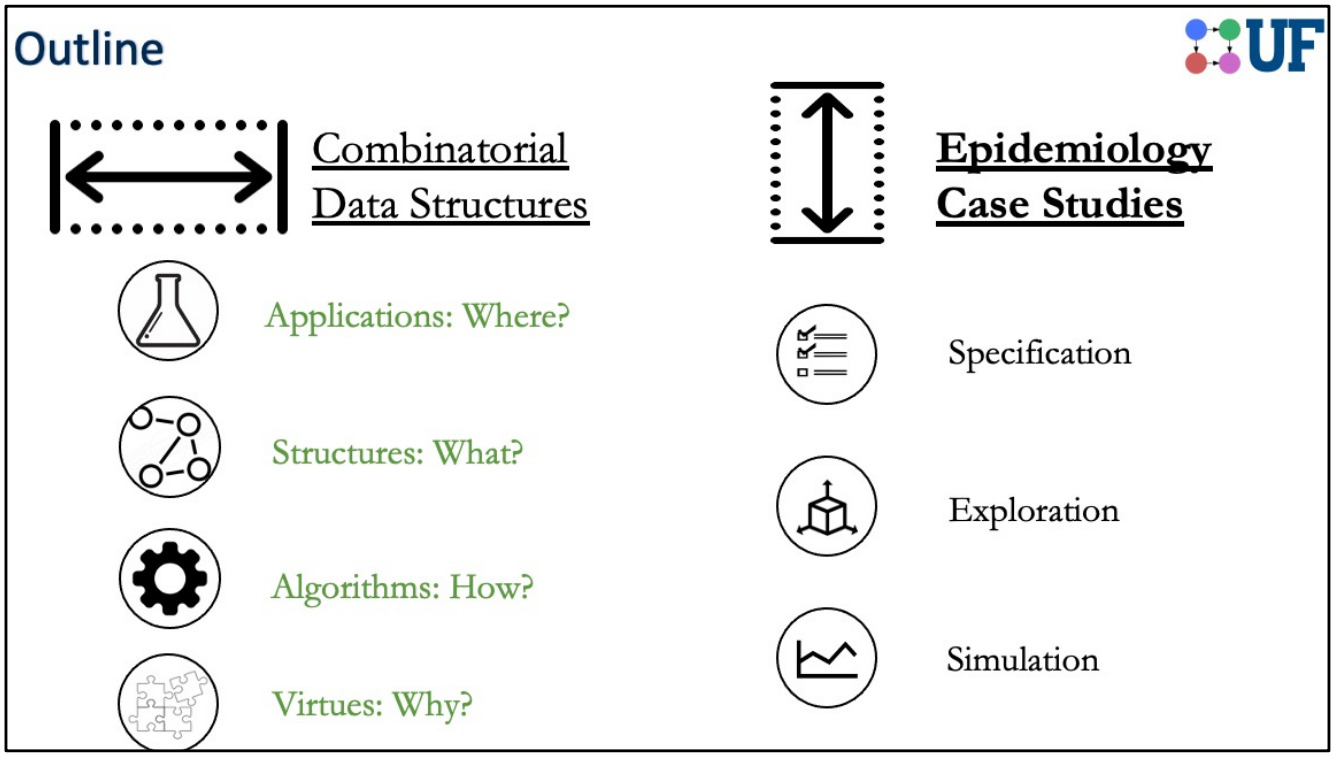



```
SELECT state2.id
FROM   S AS state1, S AS state2,
       T AS tran1,  T AS tran2,
       I AS in1,    I AS in2,
       O AS out1,   O AS out2
WHERE  in1.is  = state1.id,
       in1.it  = tran1.id
       out1.os = state2.id
       out1.ot = tran1.id
       …
```

```
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.

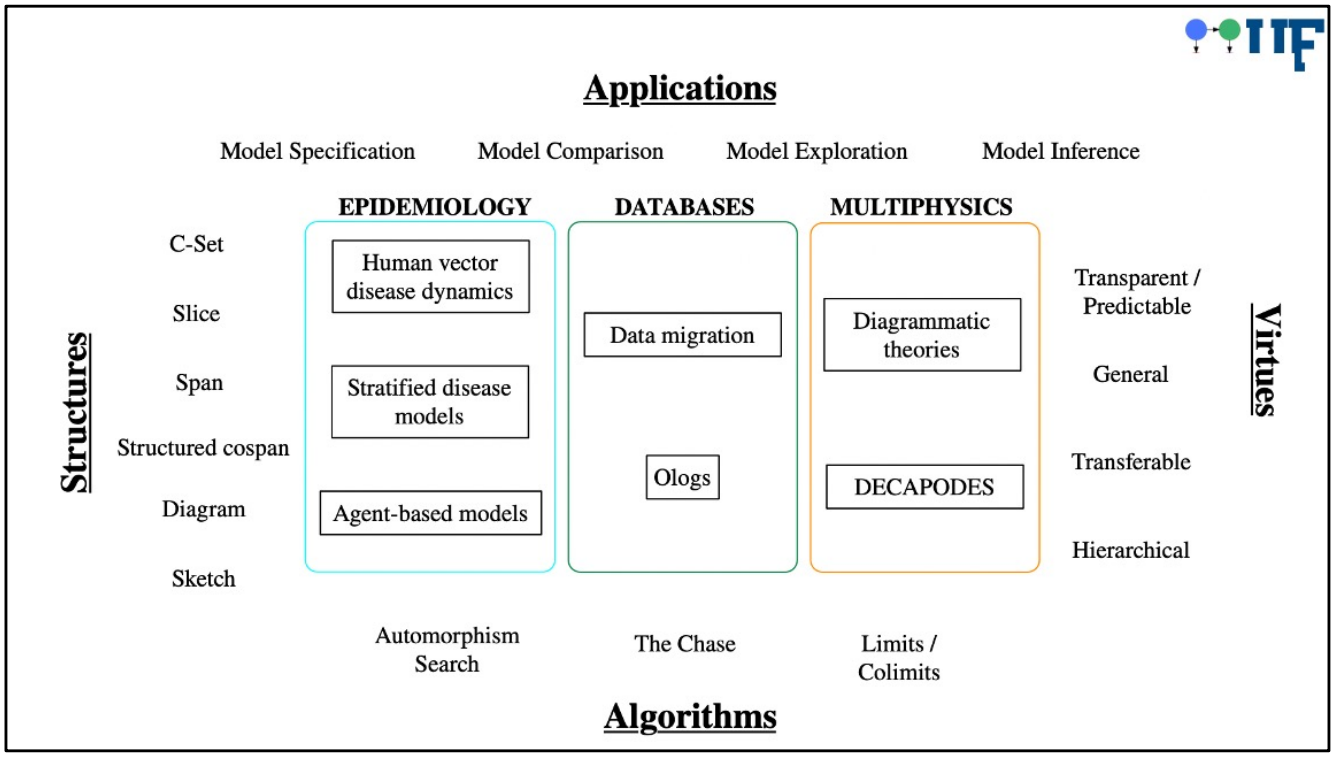
SQL code in contrast, doesn't let you do this kind of substitution (you'd have to write a special purpose algorithm to handle all the edge cases)



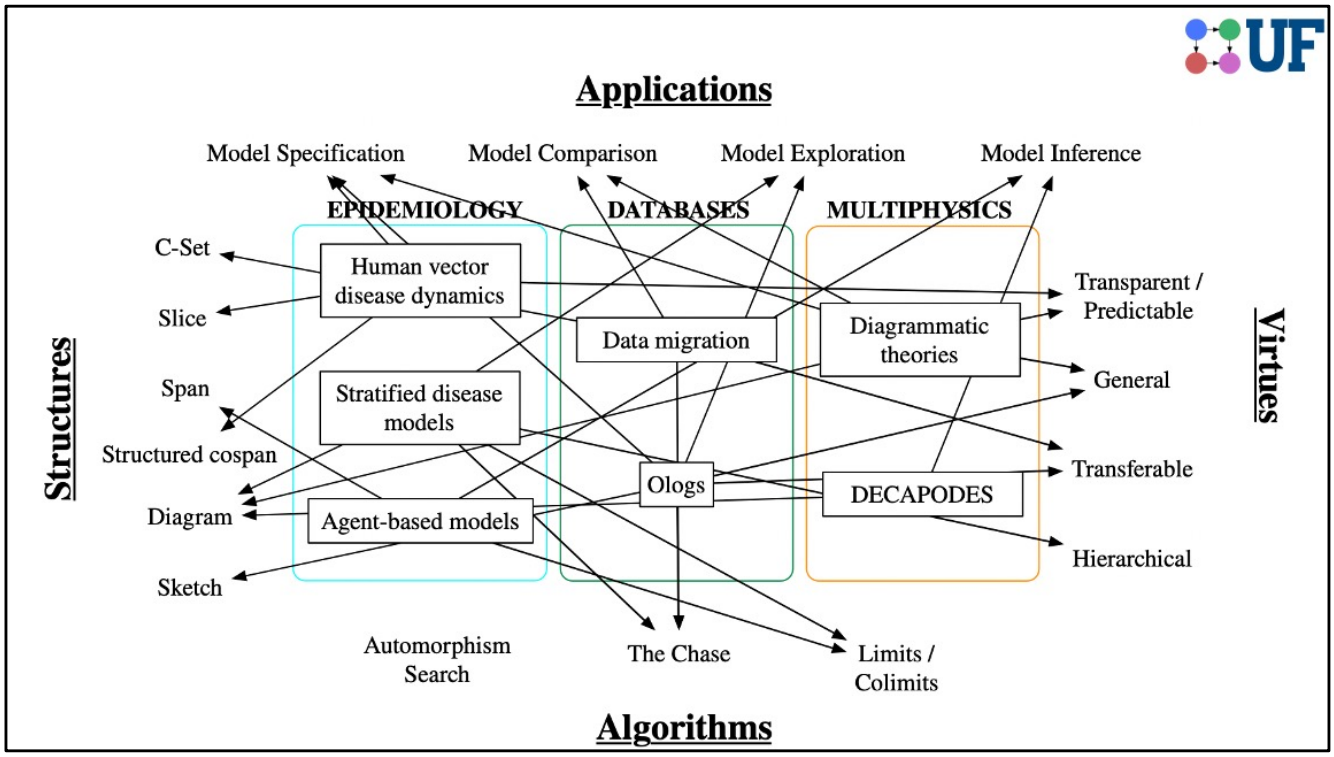

So that was very broad and very fast.

RECAP: lots of things you can't do with models as code that we highlighted

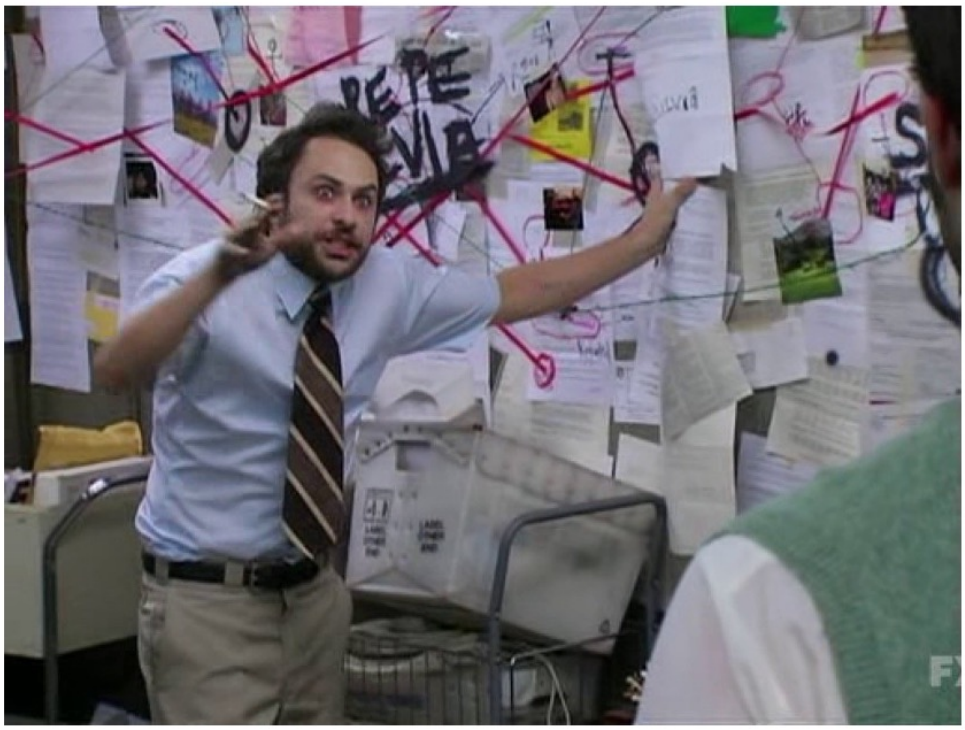
But I'll switch focus to some specific projects




I want to show three categories of projects with examples in each. I think of each of these projects as intersecting with each of the four topics I overviewed previously.

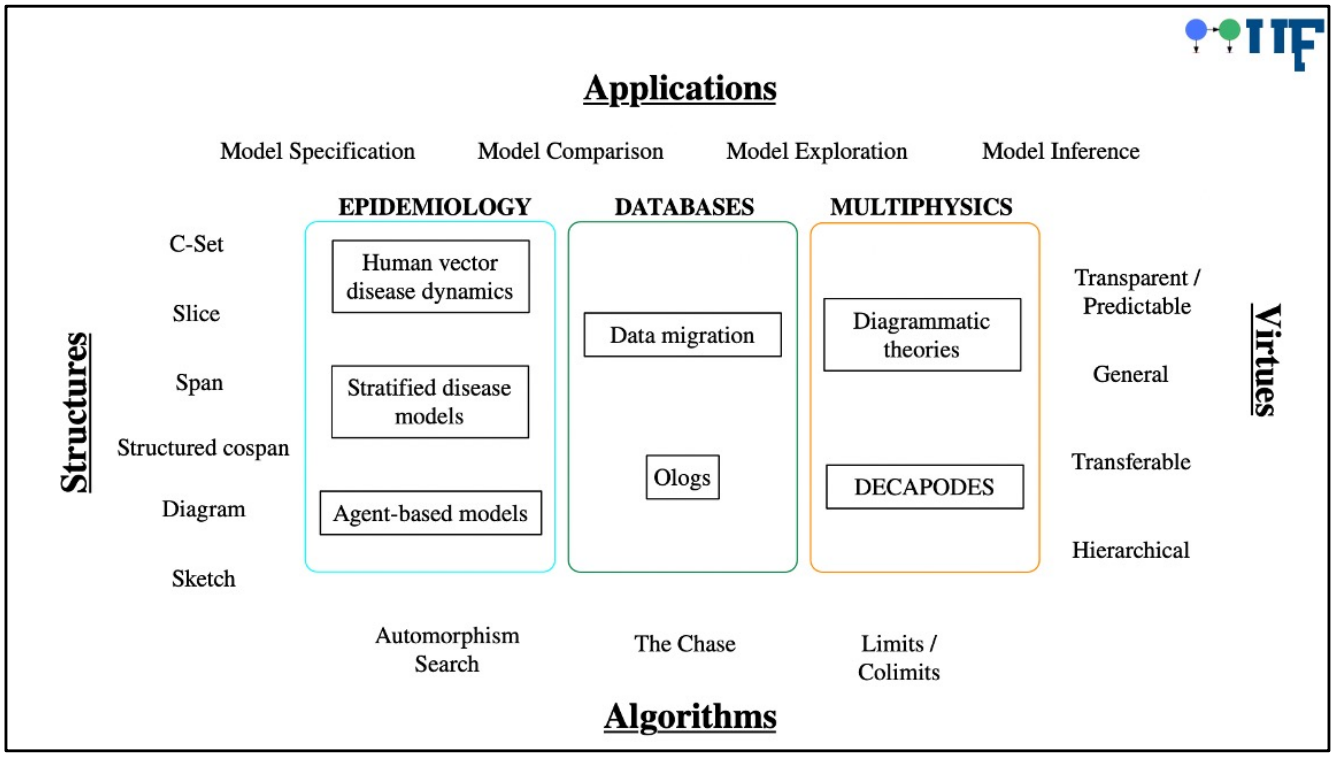



Looking at all the projects I want to talk about, we could draw all the connections.

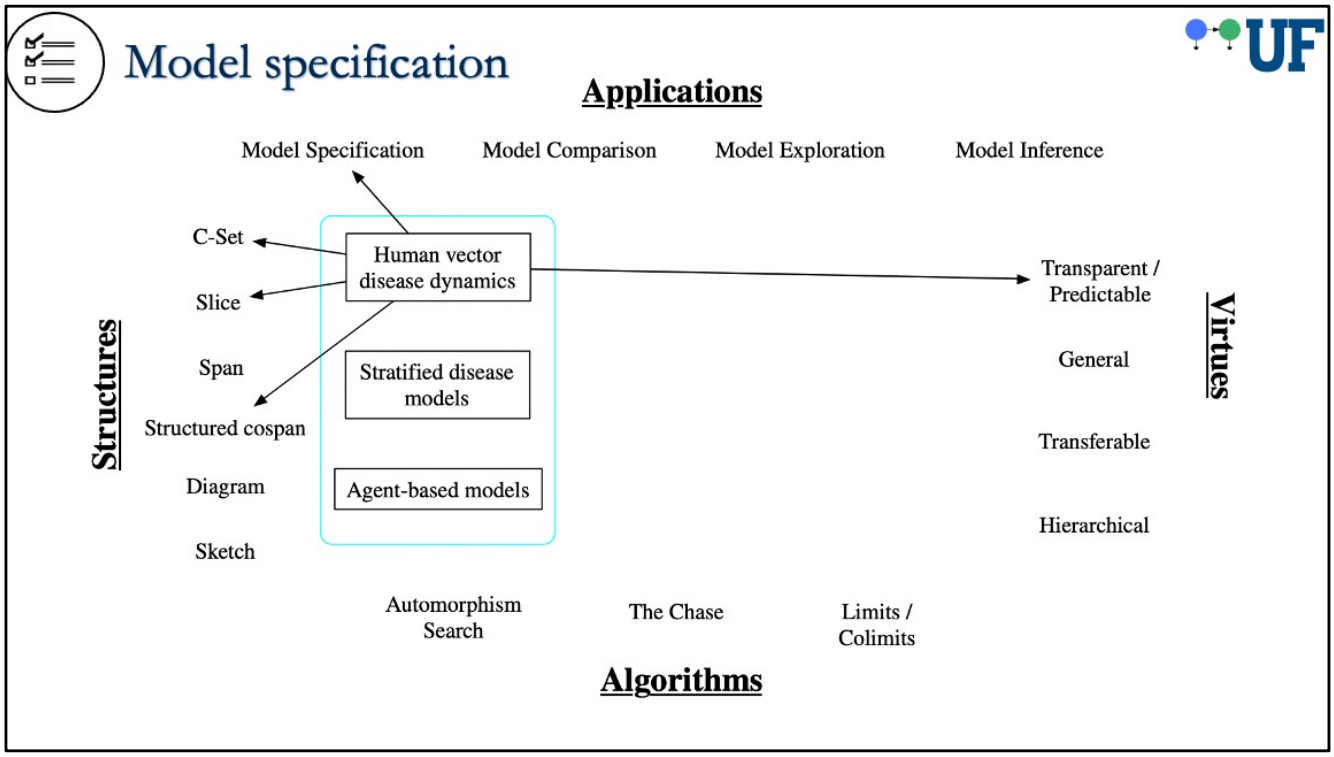


47

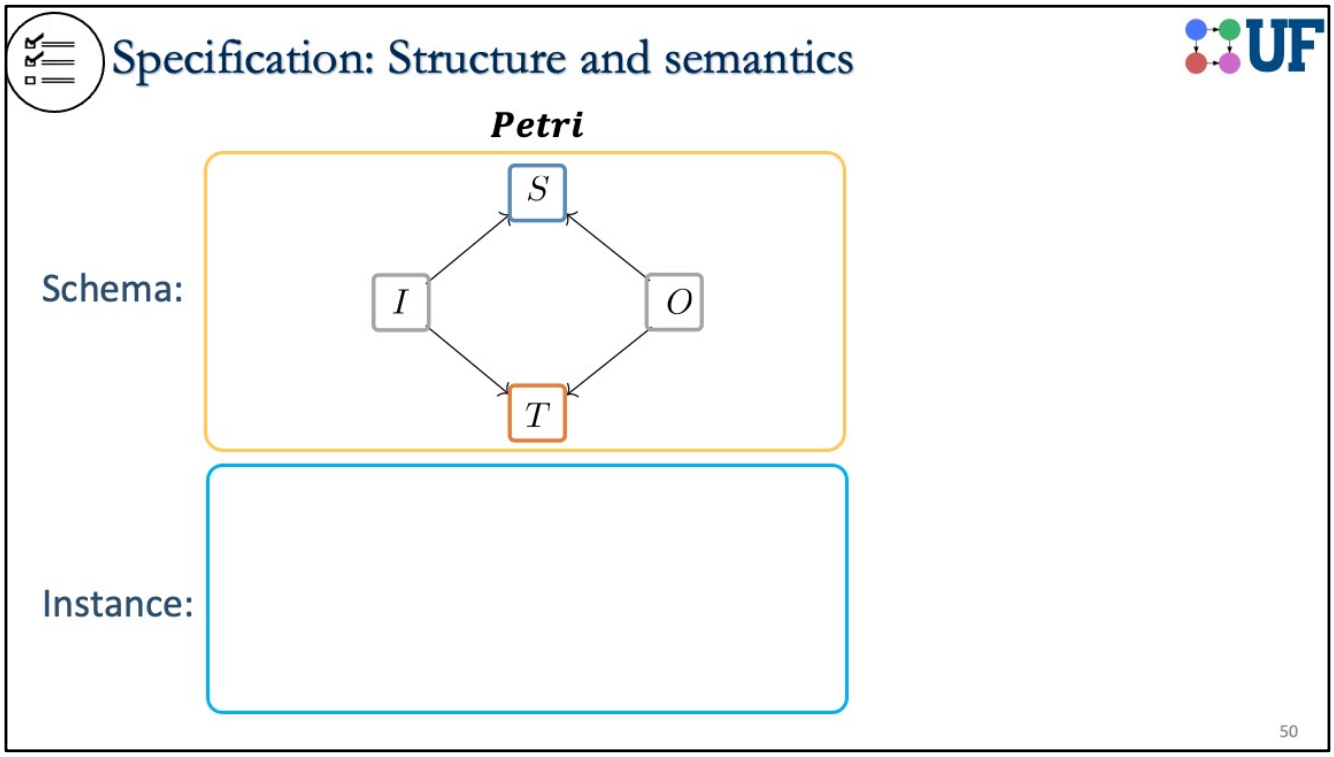
I've been strongly advised to not show all the connections at once.




In fact, we will just focus on epidemiology.




The first thing I will talk about is modeling disease dynamics.



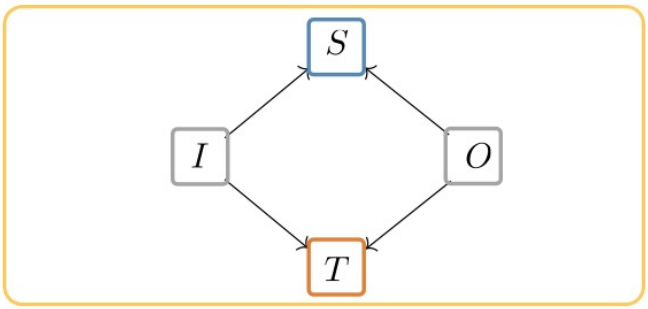

Now I'll reintroduce the concept of a Petri Net as a C-Set. A petri net is a nice way to separate the syntax (a kind of graph showing what reacts with what) from the semantics (which are dynamical systems).

We use a C-set with this shape of sets and functions to encode the fact that there are possibly many species flowing into and out of transitions.

# Specification: Structure and semantics

***Petri***

Schema:

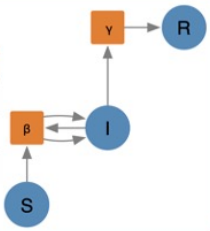



$$S + I \xrightarrow{\beta} 2I$$

Instance:

| Input | I_S | I_T |
|---|---|---|
| 1 | S | β |
| 2 | I | β |
| 3 | I | γ |

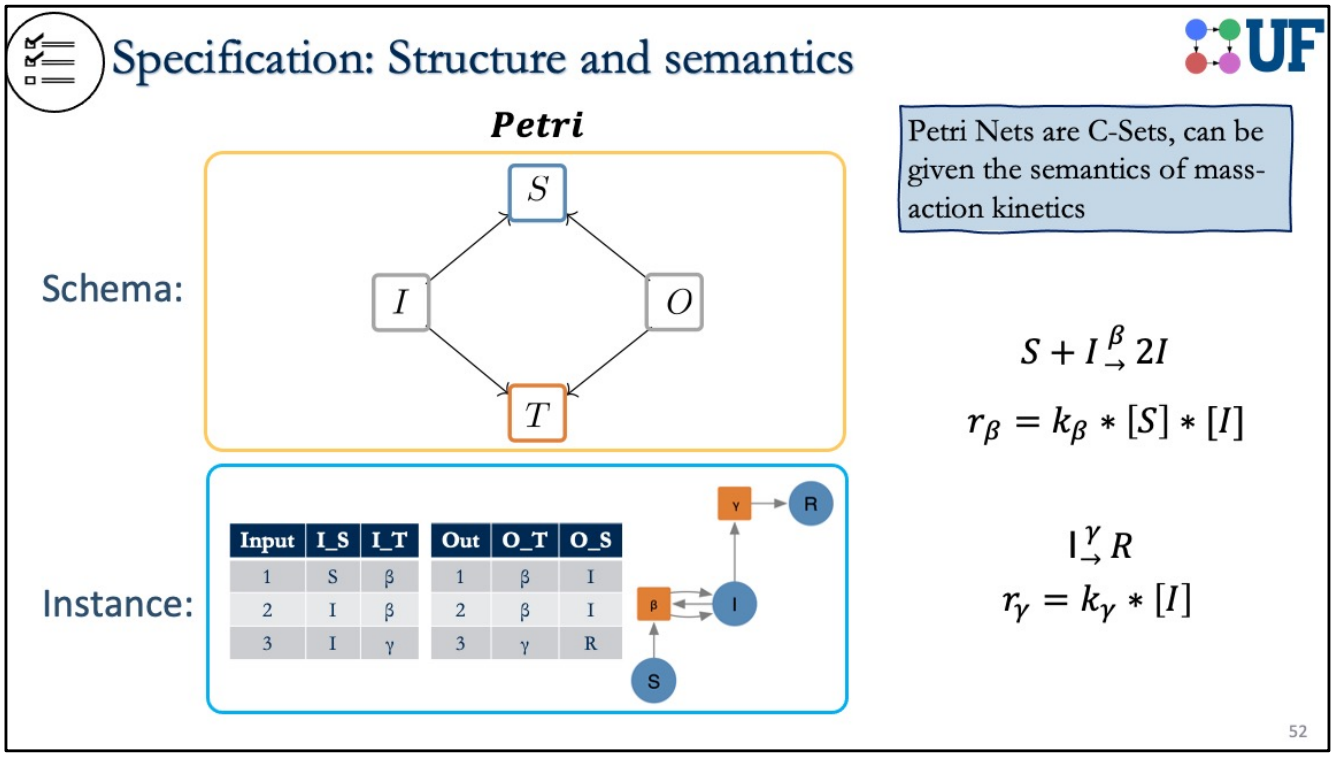
| Out | O_T | O_S |
|---|---|---|
| 1 | β | I |
| 2 | β | I |
| 3 | γ | R |




$$I \xrightarrow{\gamma} R$$

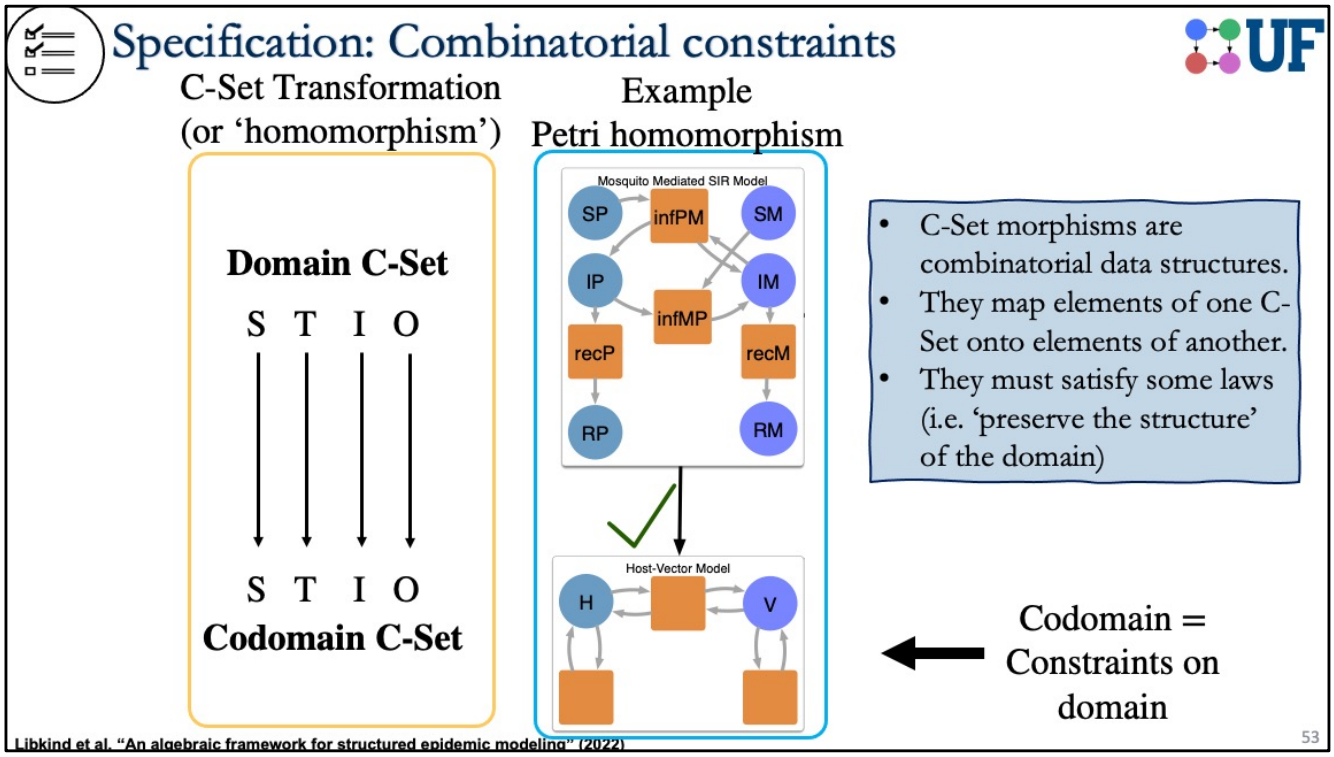
51

Here's SIR as an example on the bottom, with the equations that are represented by this model




We can assume mass-action kinetics, where reaction rates are given by a constant times the product of reactant concentrations.

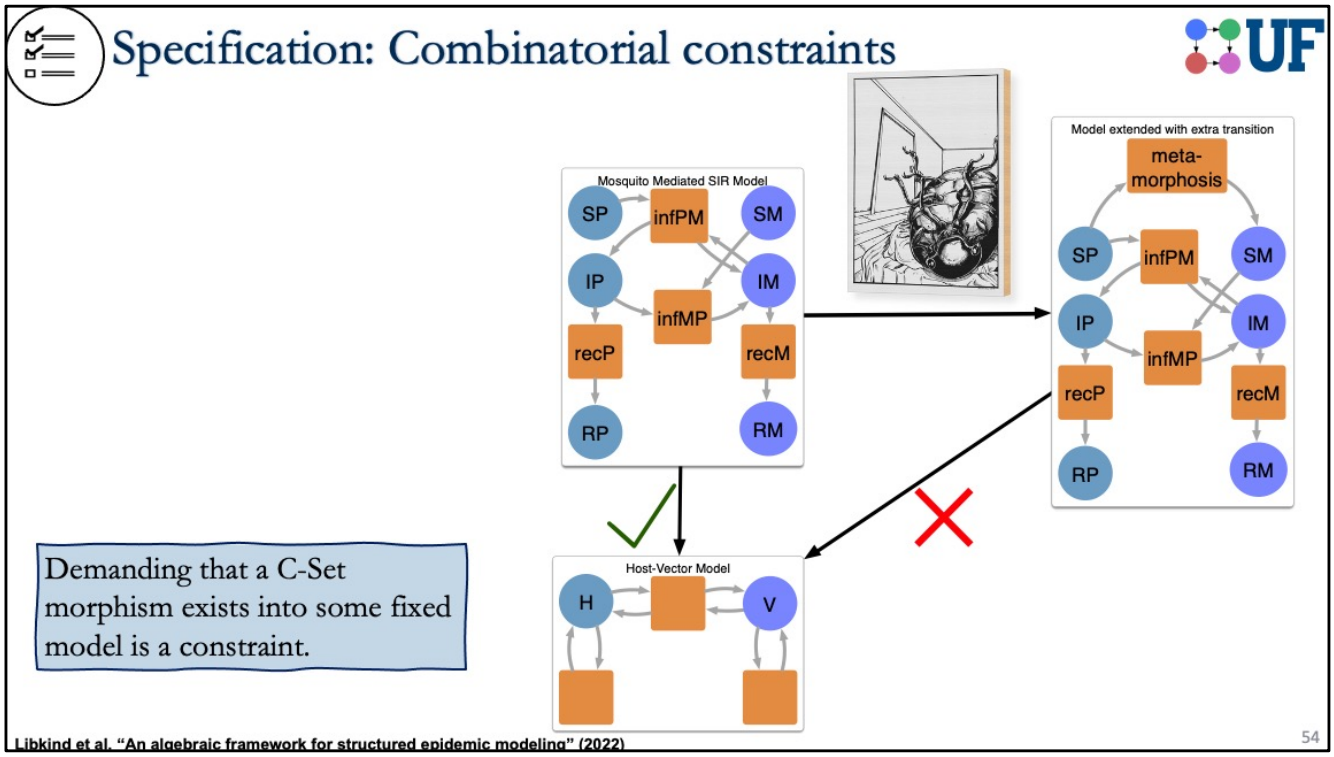
So we could stop here and say we're done defining our class of models, but this is pretty unrestricted. There's not much scientific knowledge encoded here other than mass action kinetics.




Now I want to return to those constraints I talked about earlier.
Just like a C-set is a combinatorial data structure, the transformations between C-sets are also combinatorial data structures. A C-Set transformation is like a meta-level function which maps C-sets (which are networks of sets and functions) in a way that respects structure.

Suppose we have a particular petri net which encodes very high level information about the kind of petri nets we're interested in. This one is an example taken from a recent paper of Sophie / Evan / James.

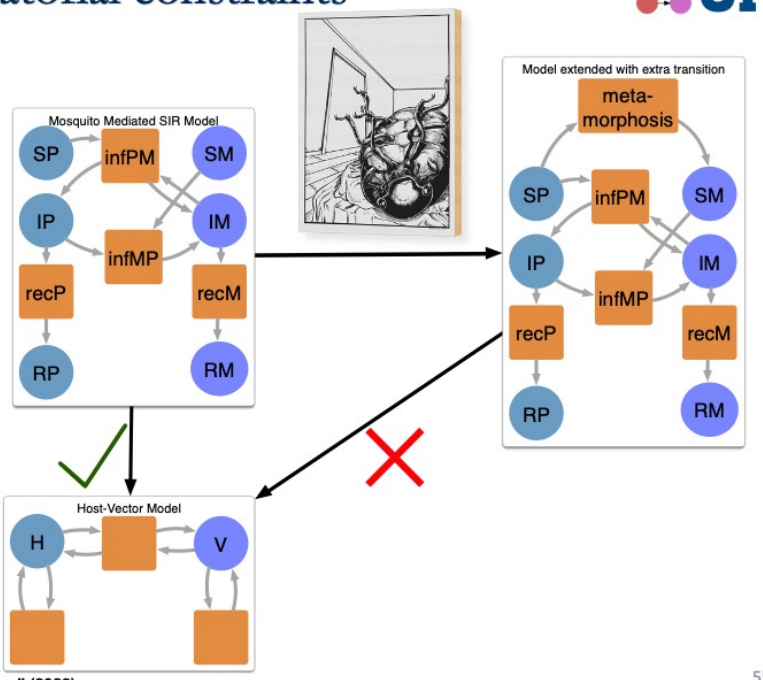
By restricting our petri nets to those which have homomorphisms into this one, we're in effect demanding that anyone who provides us a Petri also must declare that their species are kinds of either Humans or Vectors, and that all transitions are either two humans interacting, two vectors interacting, or a human and a vector interacting.




Why do I call this a constraint? Well not every petri net has a homomorphism into it. The constraint can be efficiently checked, so it's not possible for someone to submit a new model that has a transition converting a human into a vector. We rule out that (and many more nonsensical Petri nets) merely by changing our model type from Petri net to this homomorphism type (which is called a slice in category theory lingo).

# Specification: Combinatorial constraints

- Homomorphisms effectively assign "types" to elements of the domain model.

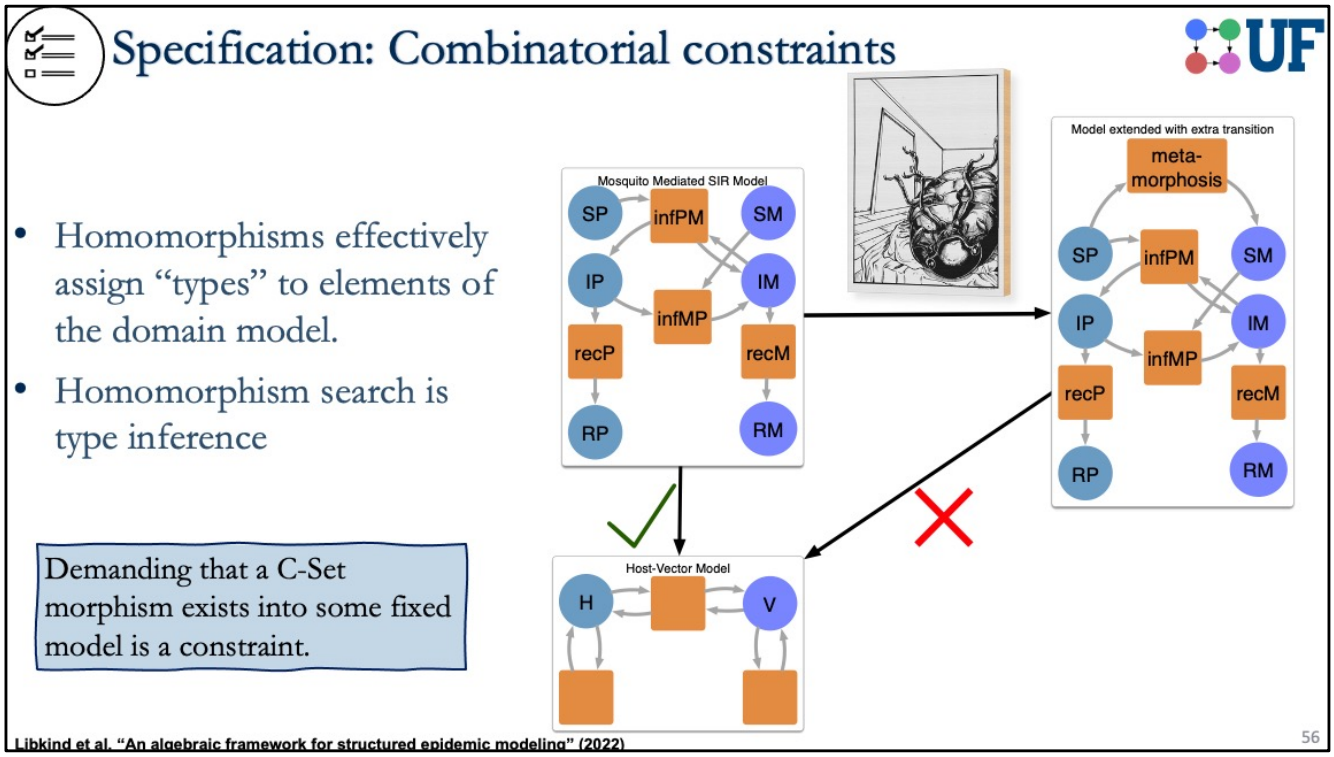



Demanding that a C-Set morphism exists into some fixed model is a constraint.

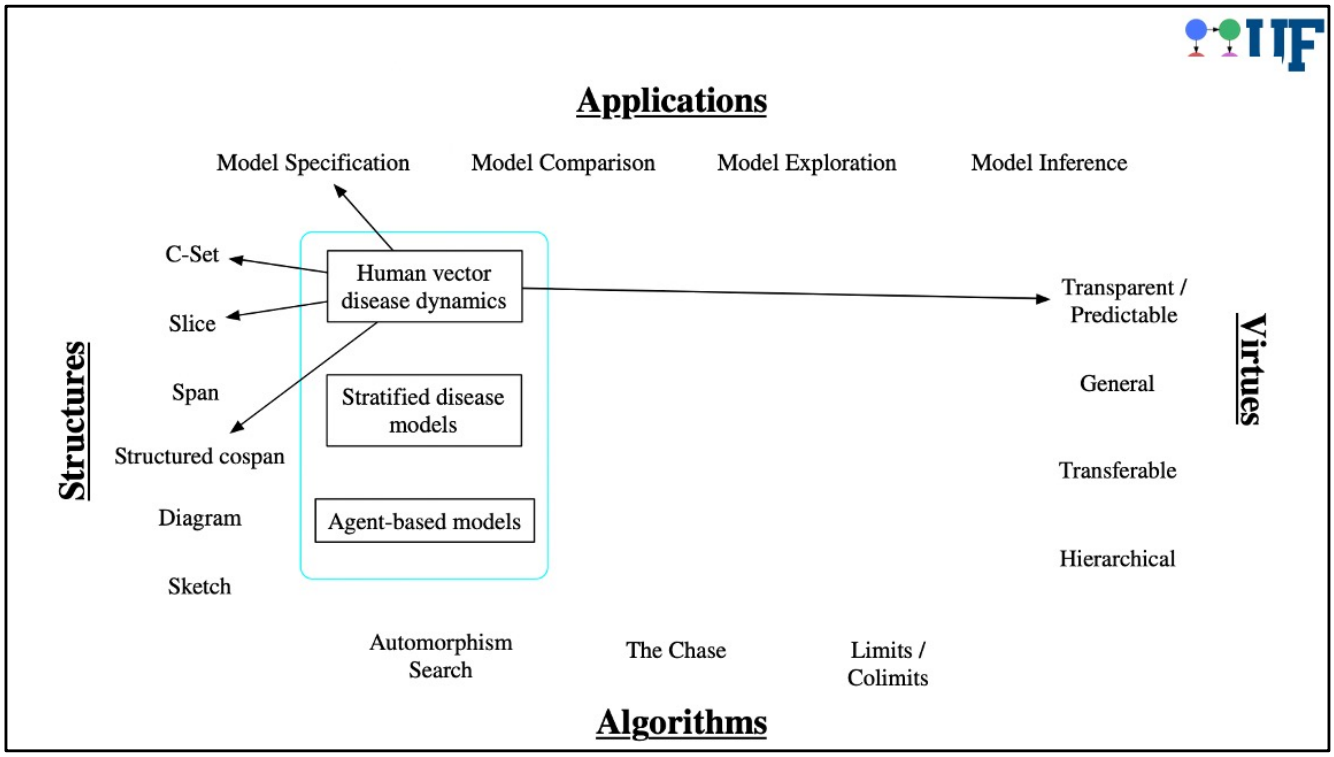
Libkind et al. "An algebraic framework for structured epidemic modeling" (2022)

55

Not only have we constrained our models, but we obtain a notion of typed elements that is completely transparent in the model: we know which states are humans and which are vectors.

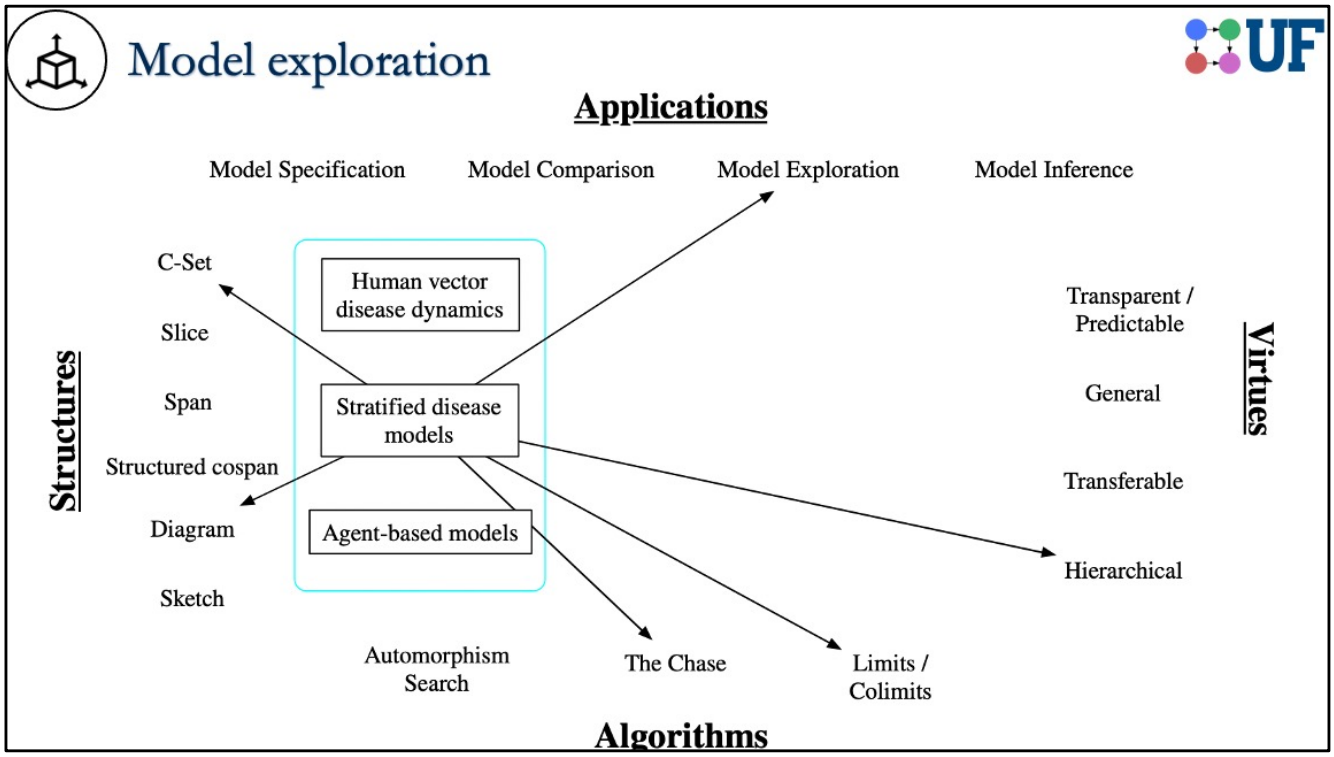



And one useful feature of having homomorphism search implemented generically (for all C-sets) is that we get for free an algorithm which takes a partially labeled Petri net and tries to find whether zero, one, or many type type assignments. This is good because it can be tedious to specify all this data manually.

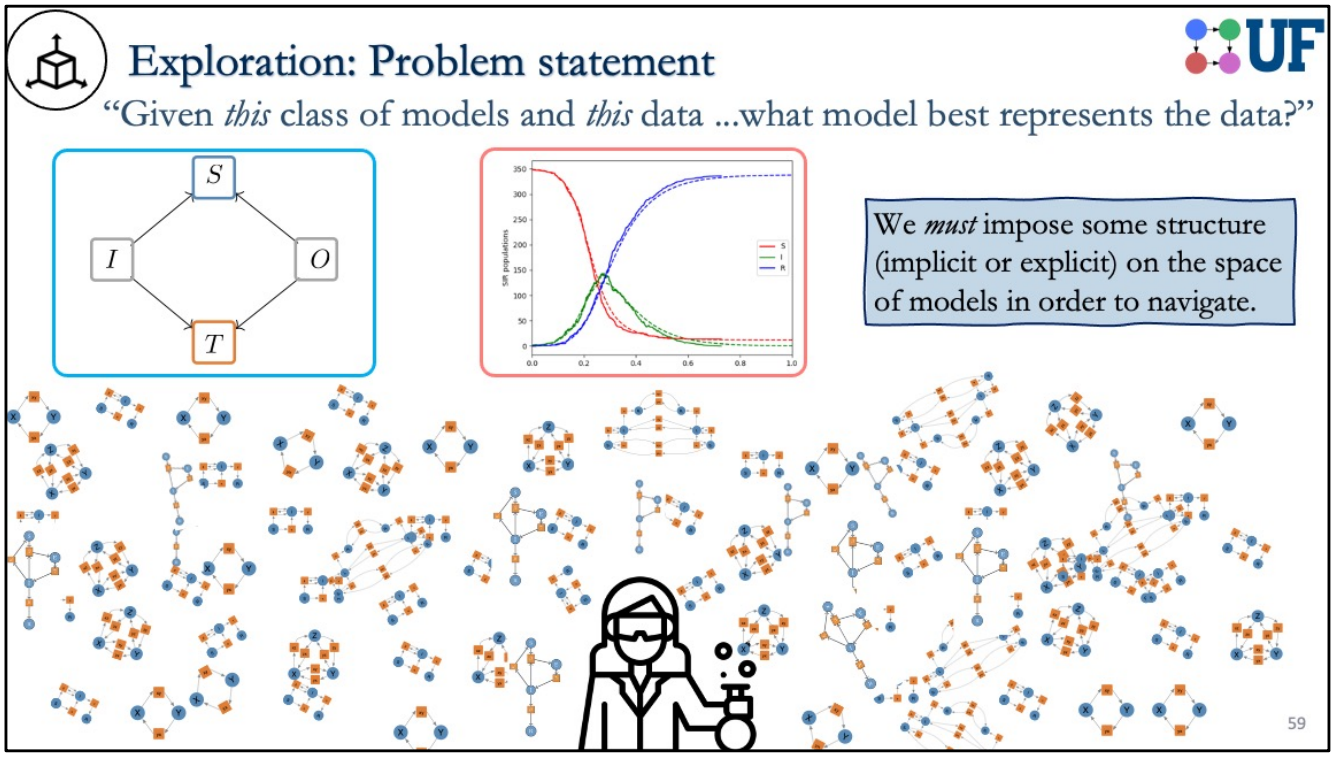



So in summary, we enable the representation of models encoding precise domain knowledge by using the combinatorial structure of slices.

I actually could use this story to demonstrate all four of the virtues here, but the key one to stress is the fact that model has become transparent and predictable from representing it combinatorially.




Now I want to return to the model exploration problem.
The key structure we'll be invoking is a diagram. In particular, we'll look at diagrams in the category of Petri Nets. (22:30)

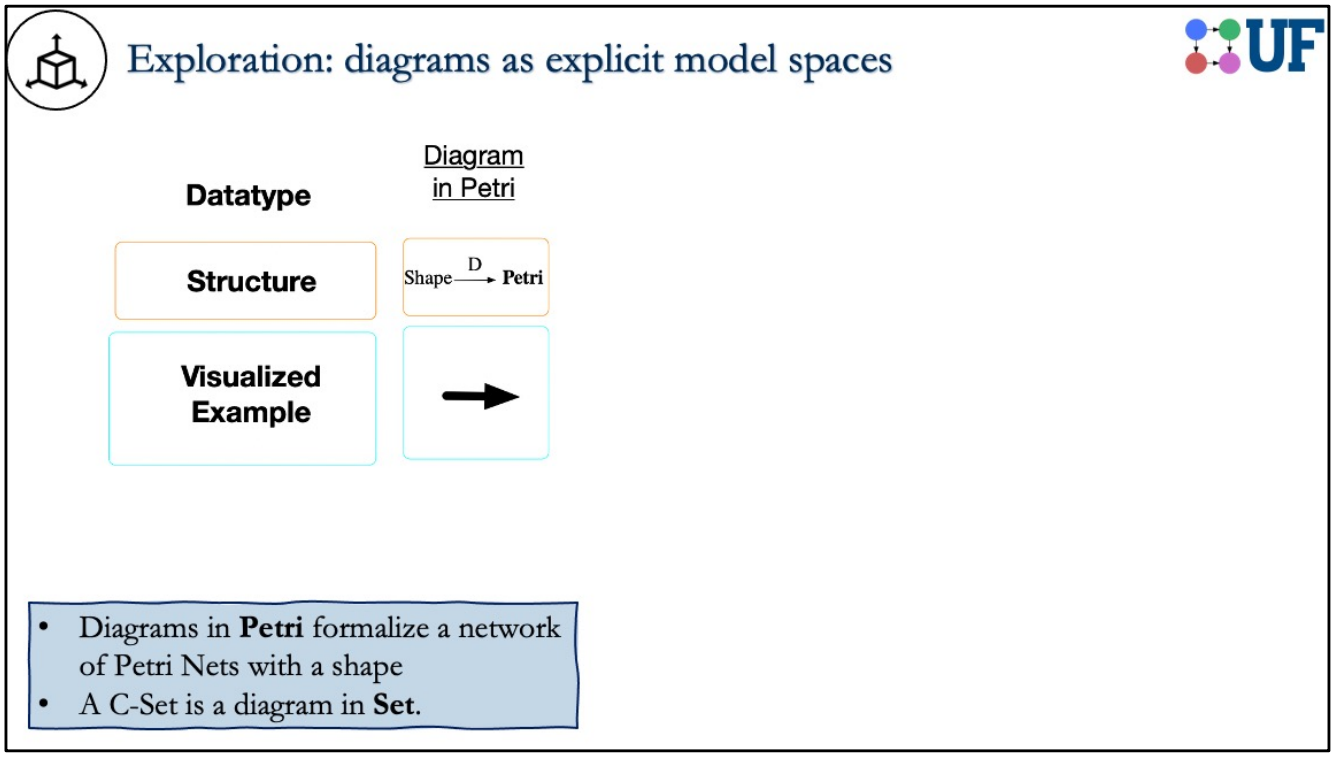



But first, what is our goal? Whatever problem we're trying to solve with the model, a scientist is confronted with the fact that there are too many possible models to do any kind of naive search.

And it's hard to explore if we don't have an actual structure through which allows us to visualize a space of Petri nets (rather than this big grab bag soup that comes from considering Petri nets as living in a big Set rather than something with more structure).

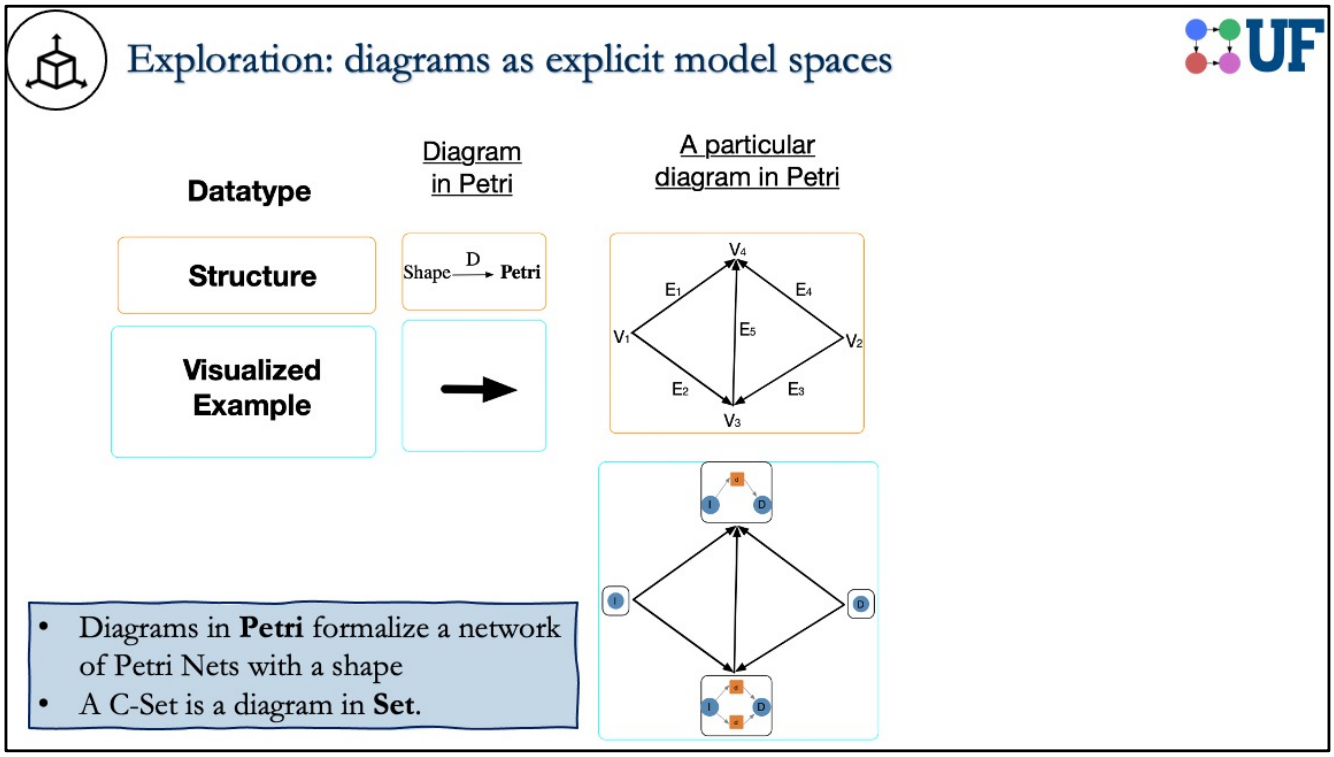
So I'll now introduce the category theoretic notion of diagrams as a potential solution to creating a space of models rather than a bag of models.

Diagrams give us a tool for cutting through this space in a meaningful (not brute force) way.

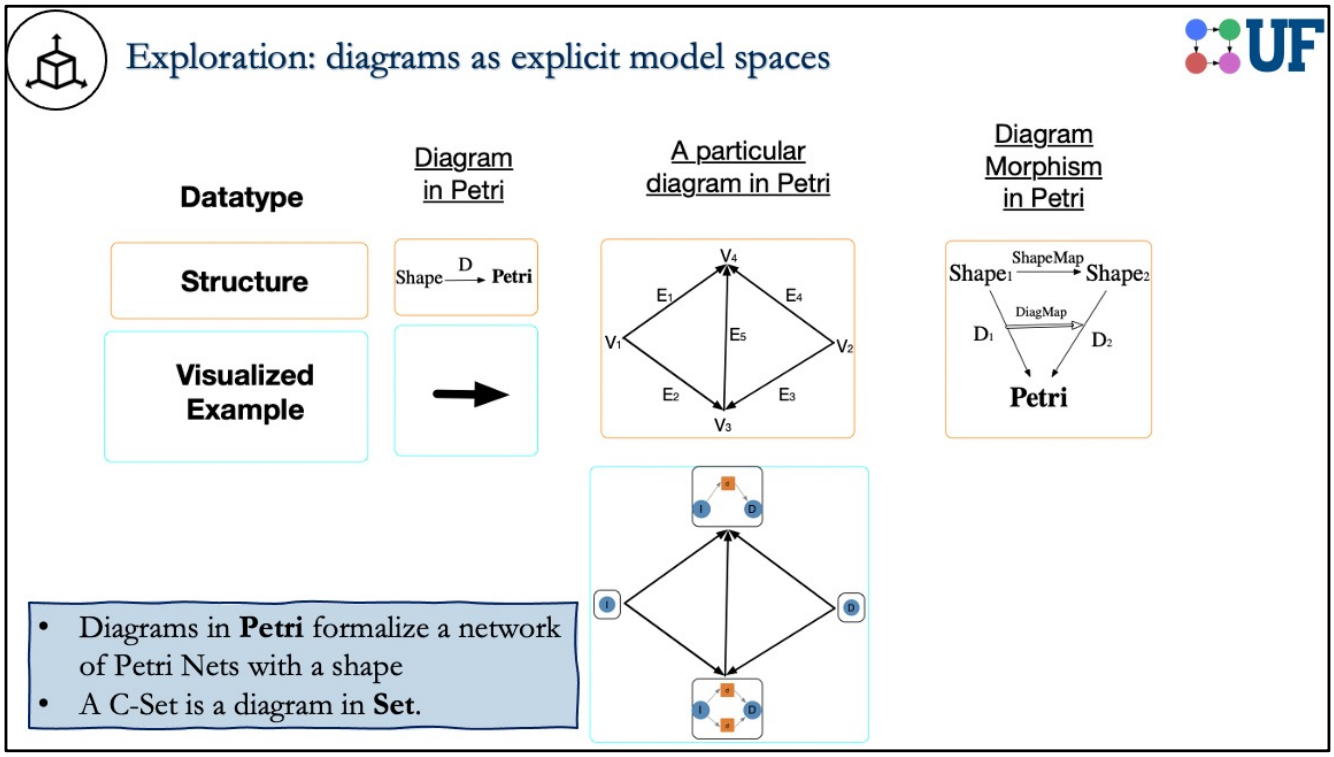



A diagram in Petri is a combinatorial data structure built on top of C-Sets. In essence, we have a shape which is a schema just like "C" was a schema for C-sets (it's like a directed graph with equations). And instead of the values we put on that structure being Sets and Functions, we put Petri Nets and Petri net morphisms.

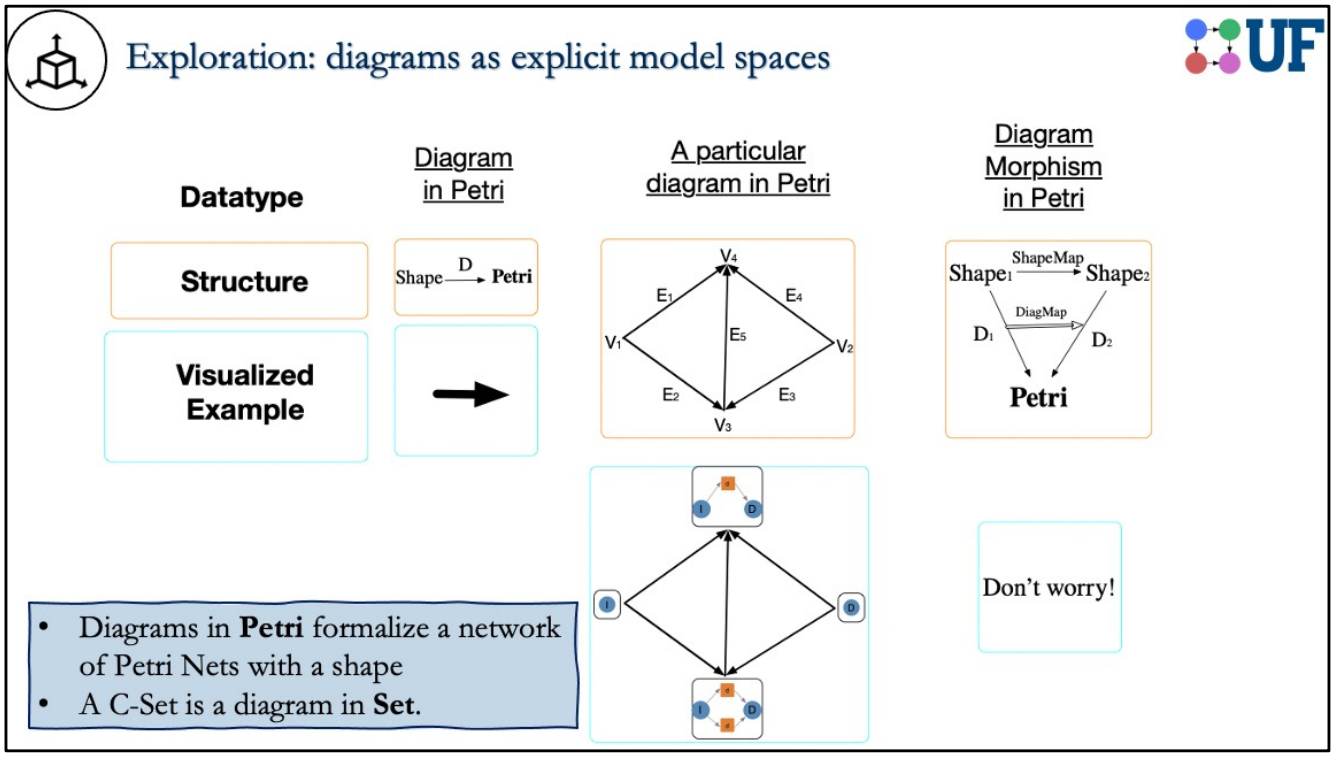
Note, given this definition, a C-set is itself tantamount to a diagram in Set.




Here's an example defining a triangle-pair-shaped network of Petri nets.

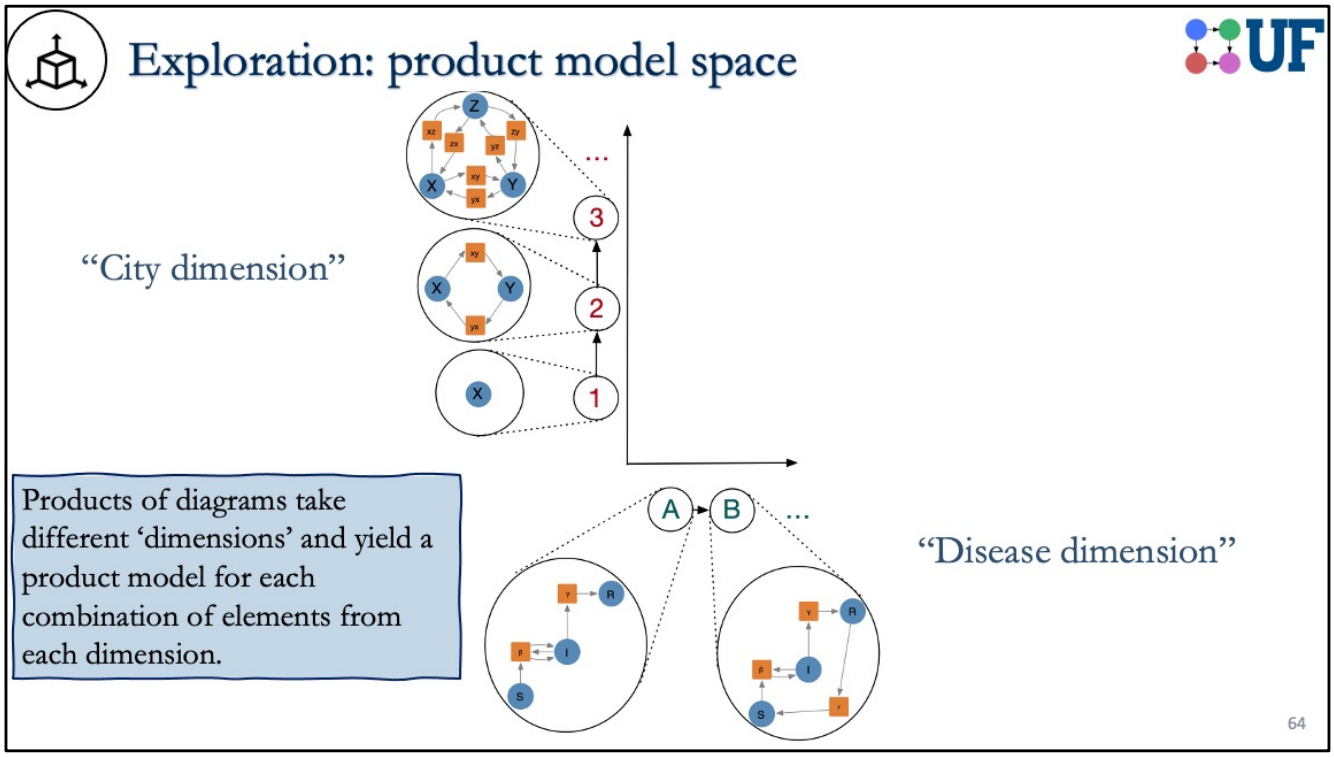



That's a very obvious generalization to make, but the subtlety lies in what the right choice of morphisms are between these objects. For C-sets, we only ever look at morphisms where C is fixed, but now this category of diagrams can have changing structure and changing data at the same time!




It's a bit too much data to visualize, but trust me that this is the right notion of morphism for our purposes.

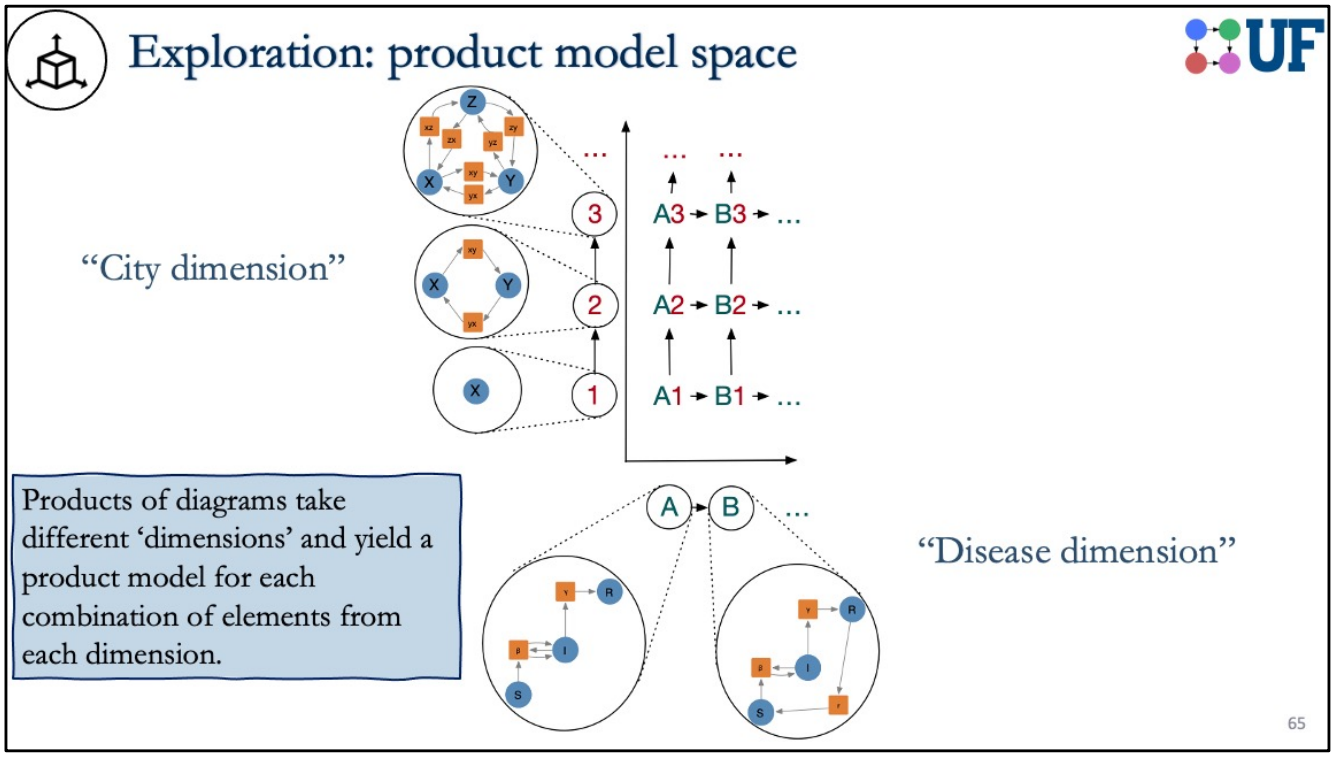
But now I want to address the question of why it was actually useful to recast our vague notion of a "model space" into something precise like this - what did we gain? Well, category theory has a lot to say about things you can do with diagrams, and it turns out some of those are useful at recasting our other vague notions into concrete algorithms that can be implemented.



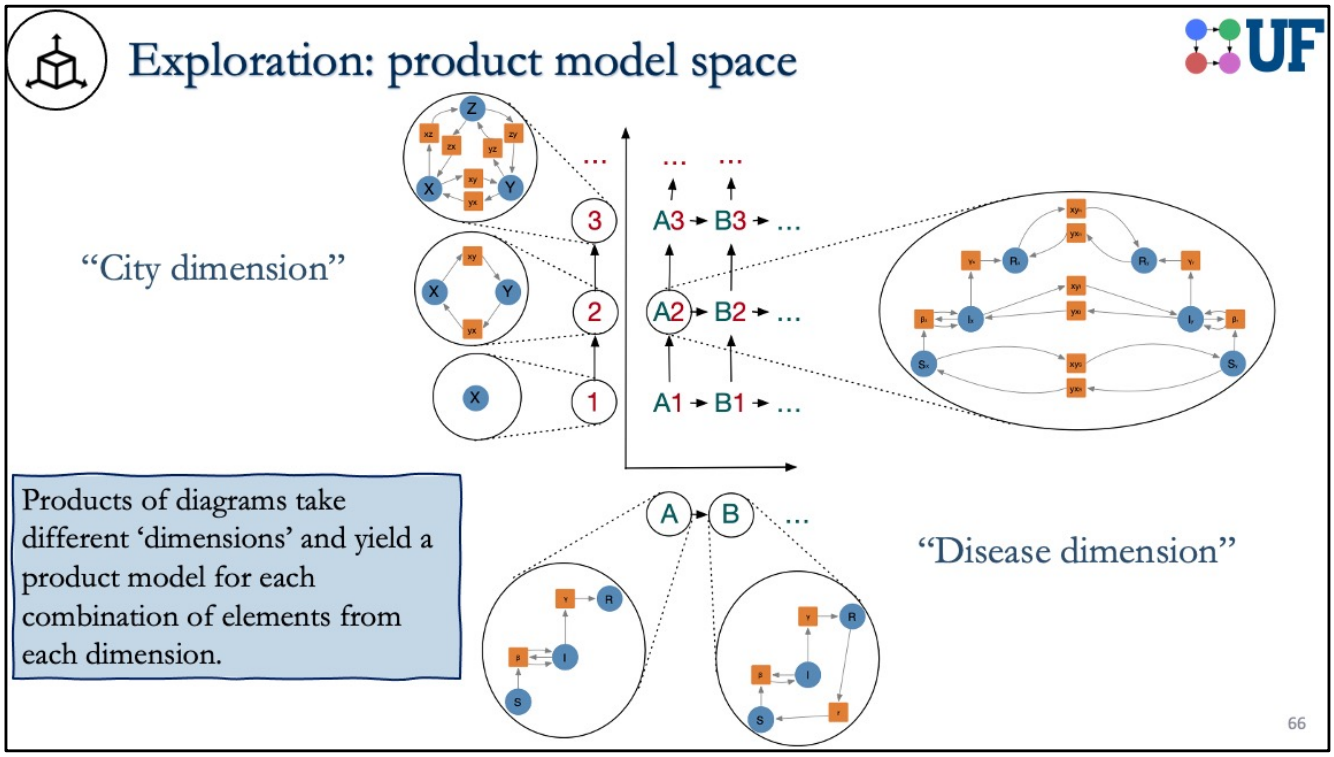

One thing we get is a way of combining small model spaces into larger ones. This is a reasonable problem to have, where the scientist has a few ideas of "dimensions" so to speak that they want to explore along, and they want their models to take features from each of these dimensions.

Here's an example of a pullback in the category of diagrams (though because pullbacks are a kind of generalization of products, I also call it a product of model spaces).

On the vertical axis, we have a 1-city model, a 2-city model, and so on. And on the horizontal axis we have some sequence of disease dynamics.

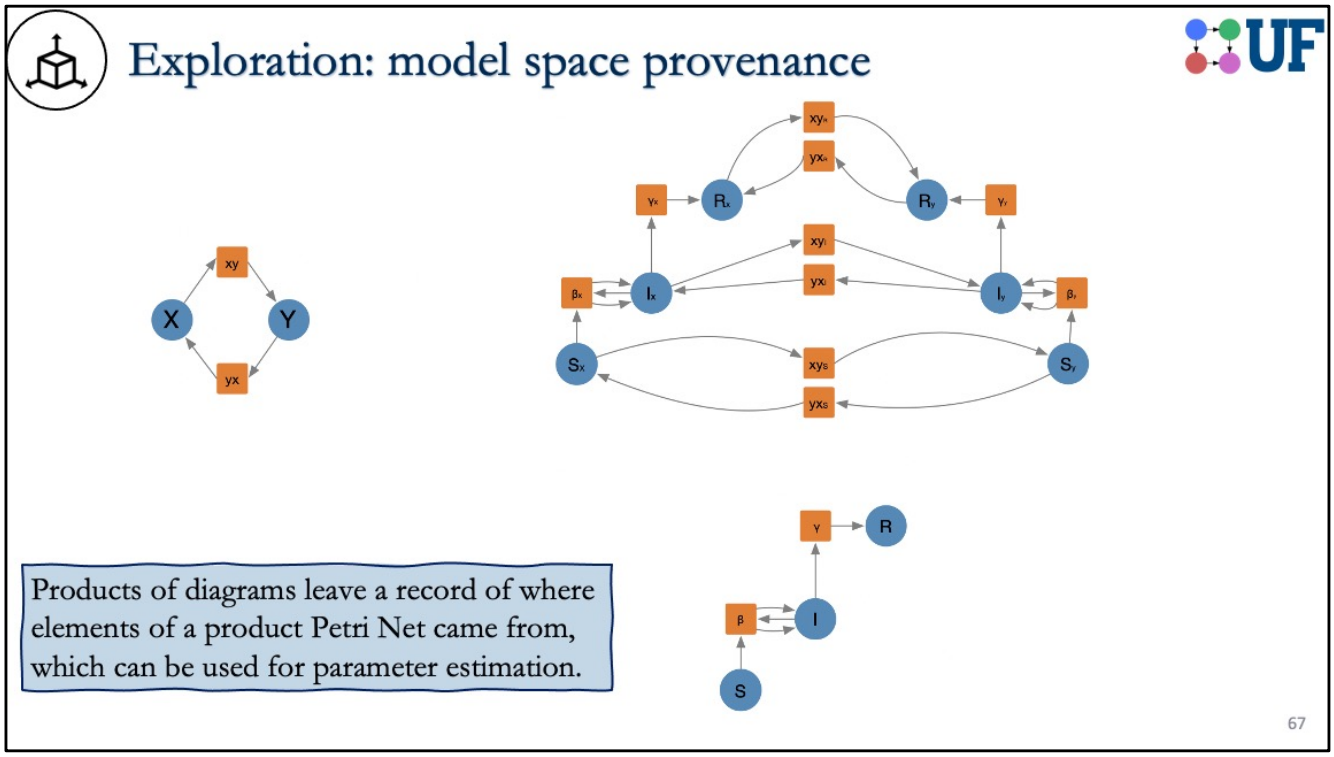



The resulting model space would look like this at the shape level. We'll pick one of these models to see what element of Petri that vertex got mapped to.

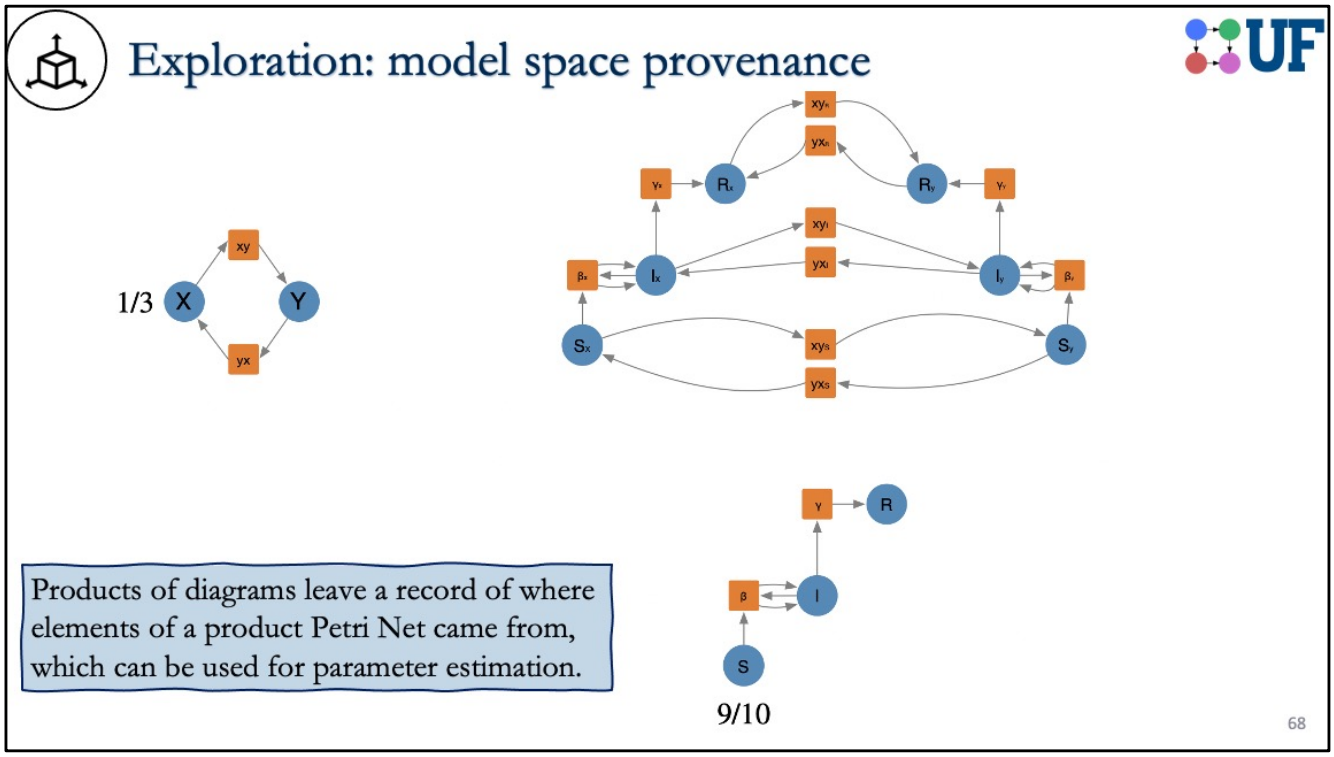



You see here we have the same disease dynamics occuring in two cities that interact with each other.

If this process were coded up in a script, it would look like a nested for-loop for each dimension. That works for a one off product, but it doesn't compose well with other ways of combining model spaces and the resulting product models do not have provenance.

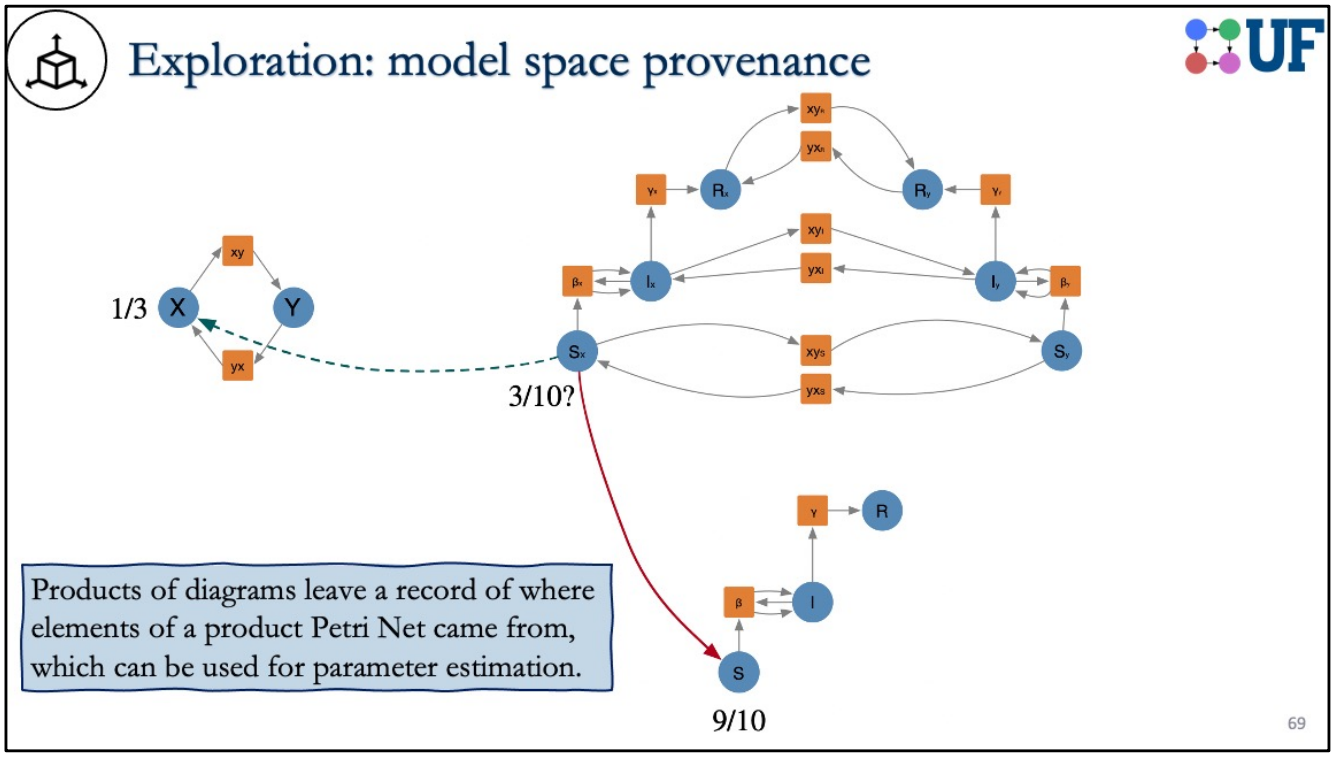



Computation of pullbacks gives a complete account of where features in a composite model come from. This can be useful in many ways .The one I will highlight is making initial guesses for parameters.



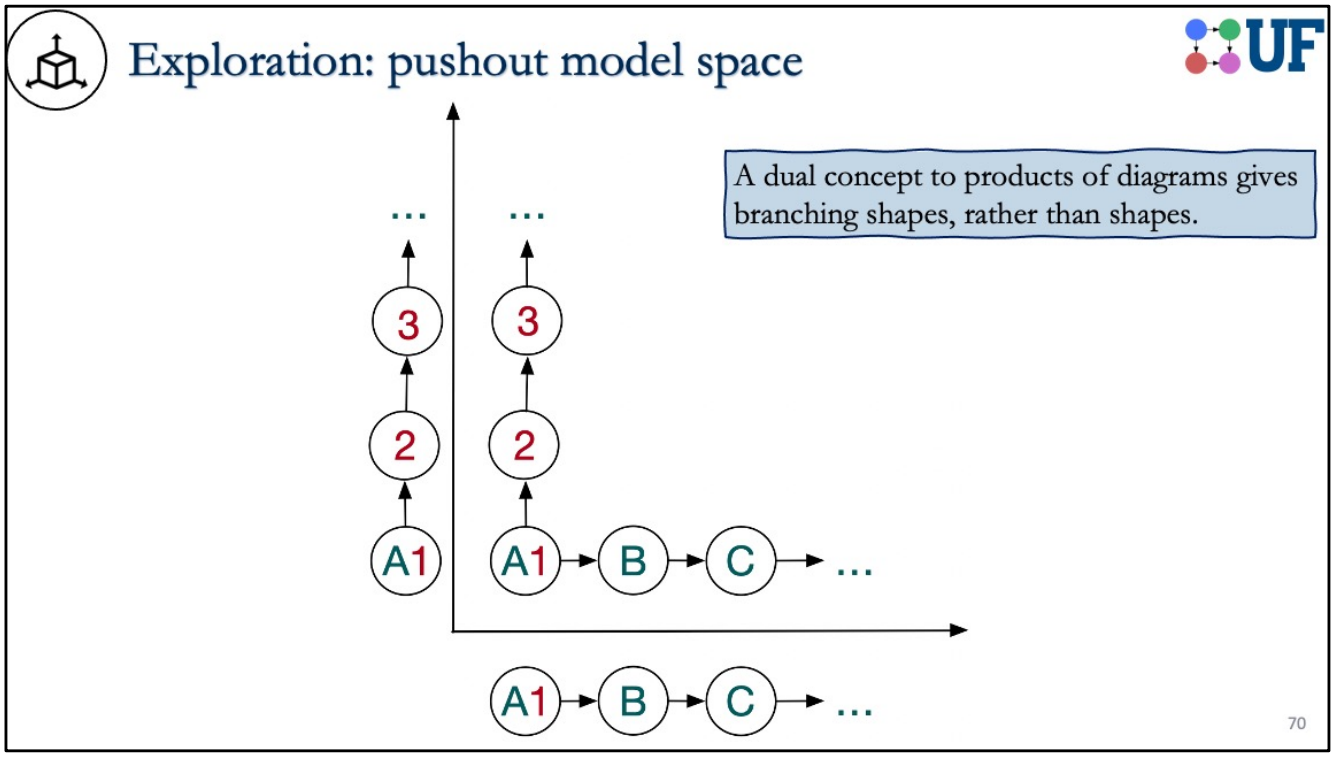

Suppose we fit SIR to our data and determine iniitially 9/10 people are Susceptible, and we collect some demographic data that 1/3 of people are in City X.

We really ought to deduce that, initially, 3/10 people are susceptible in City X in the product model.

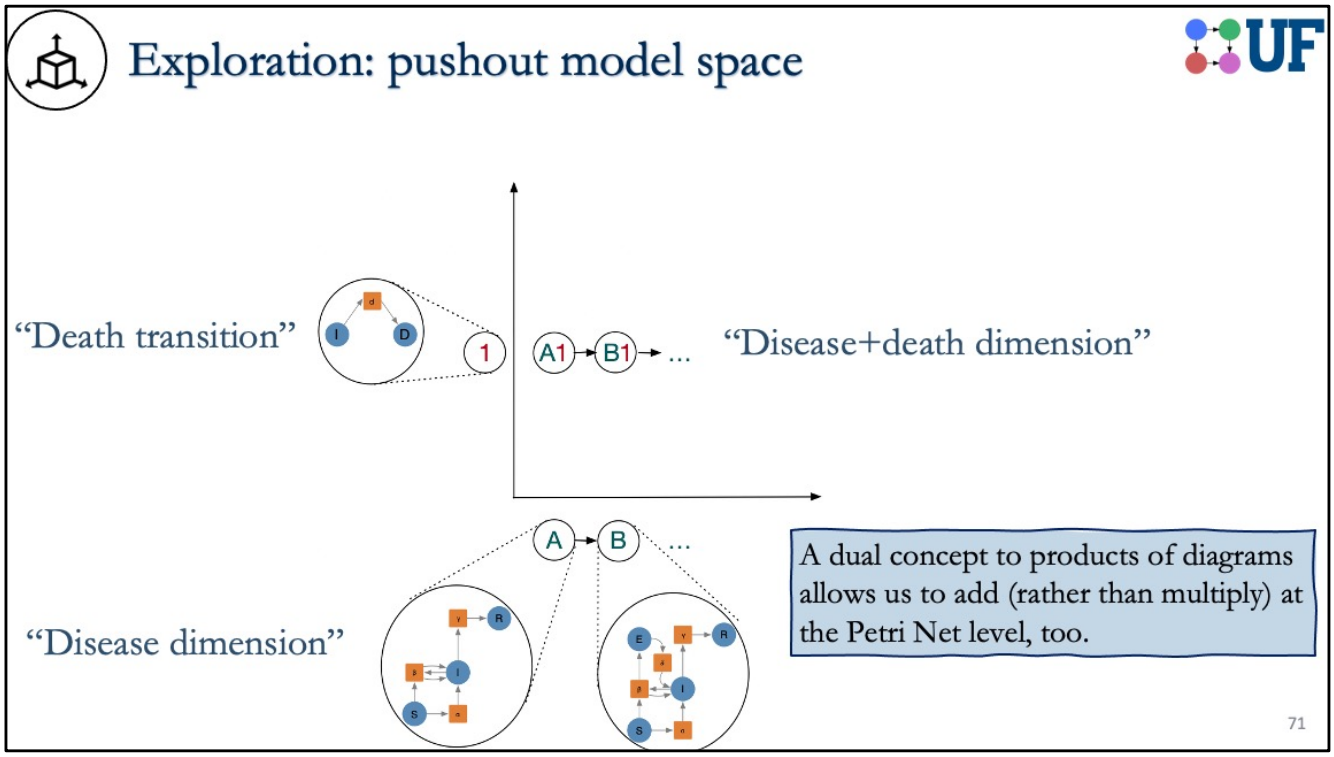



And you'll see that the data of a pullback actually tells us where that state was derived from, so we can compute this value!

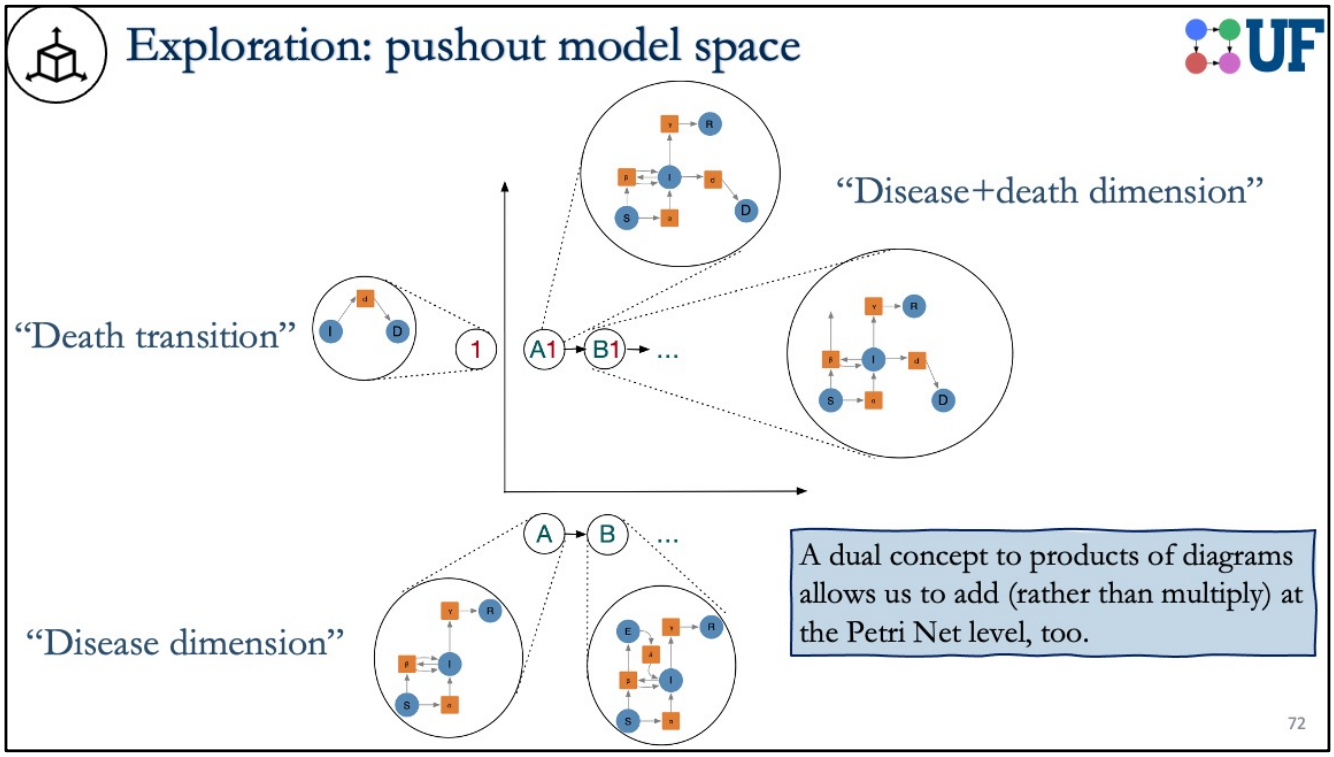
These types of computations assume a kind of statistical independence of the dimensions, but it is still useful as an initial guess for iterative algorithms.



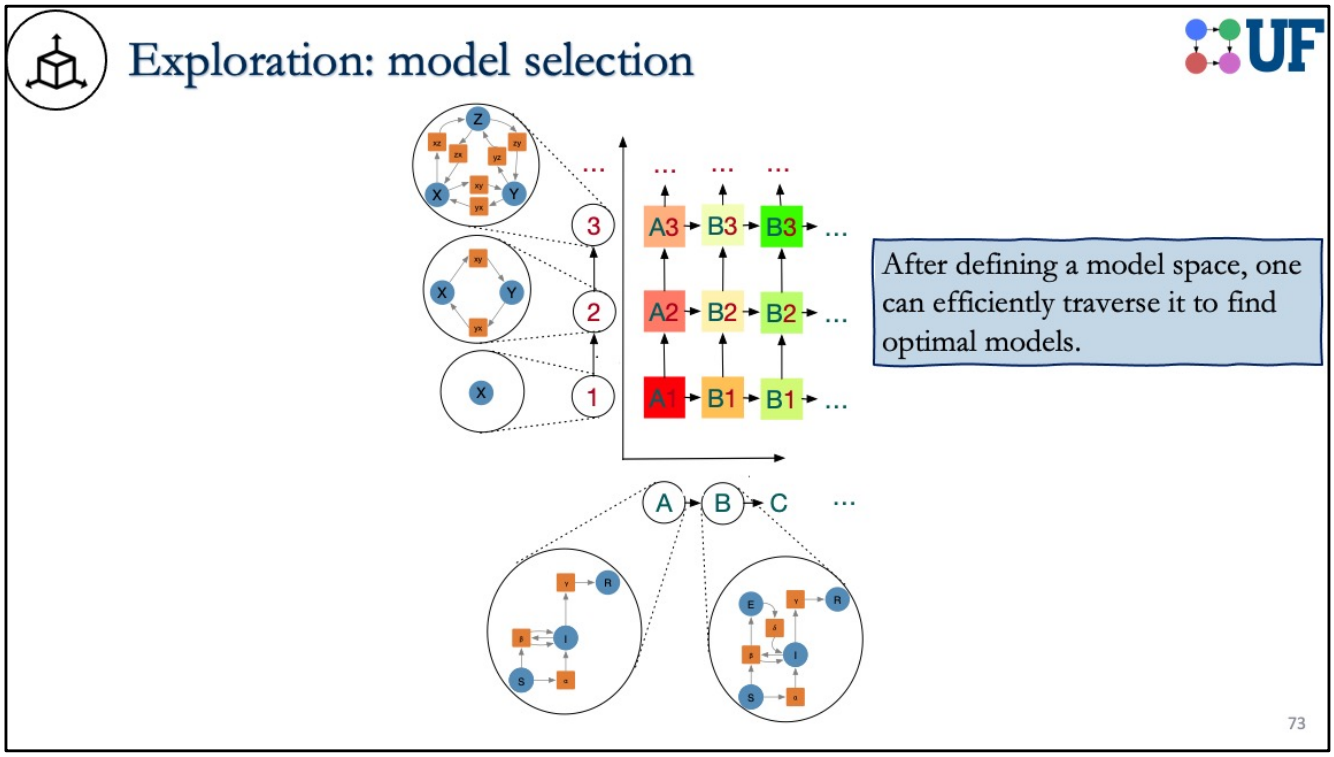

Just like we can take a product of two model spaces, we can also take a sum, which would produce more of a branching shape than a grid shape here.




And this other example which I'll gloss over is a an example of gluing together things at the Petri-net level, not the shape level. So we have an Infected to Death transition that we attach to a sequence of models in the bottom model space.

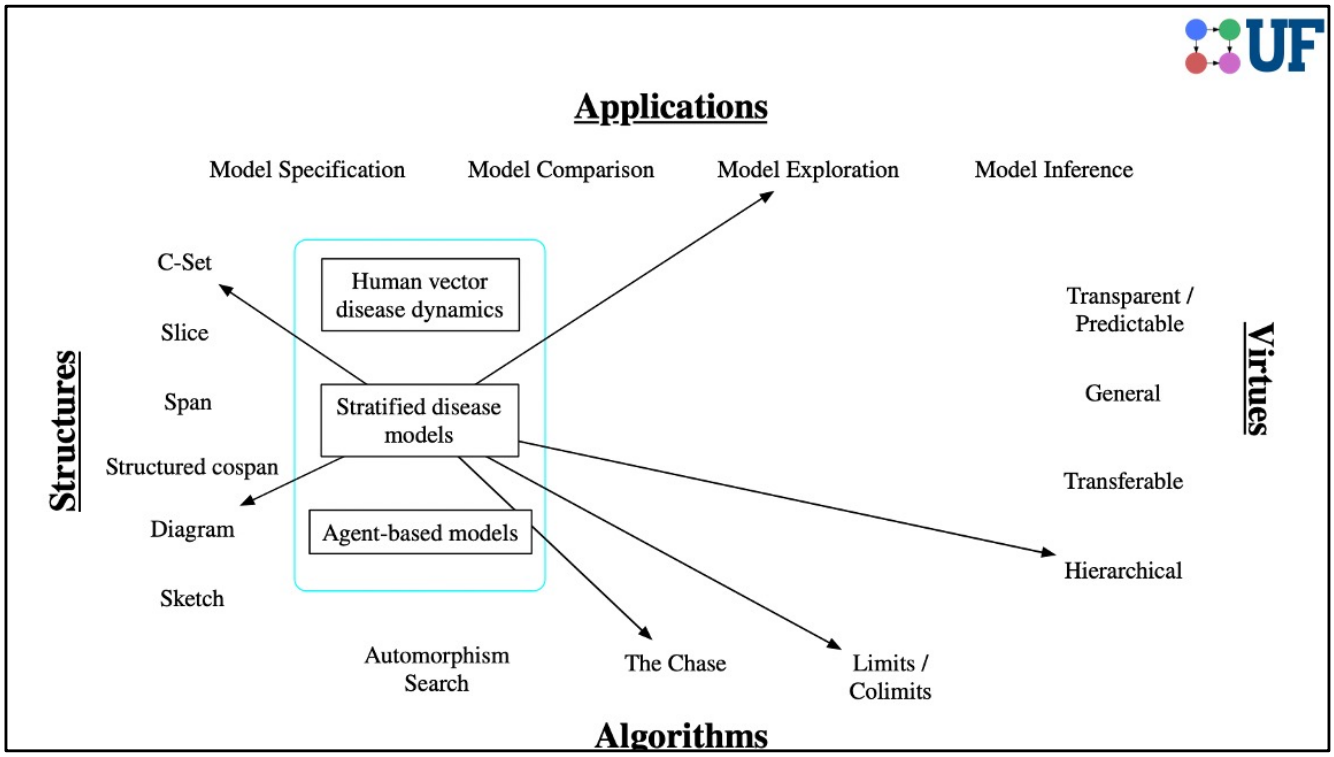



And specializing this very basic CT idea of a pushout in the context of diagrams computes the correct thing for free.

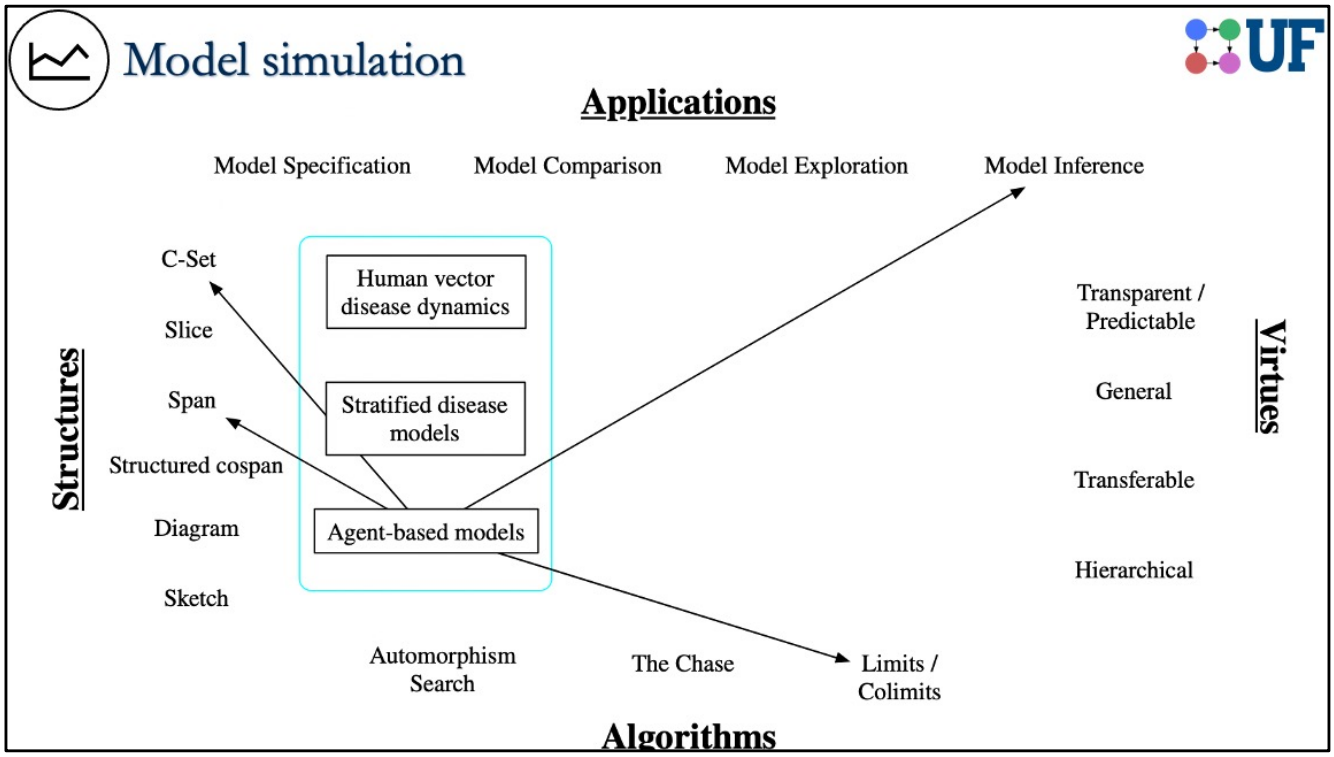



Ok, that was two very simple examples of two very basic ways to combine model spaces into larger ones. I want to switch focus to the problem of having defined the composite model space, you are now trying to select to pick which model is your favorite. Say you have a function that can give each model a score,.

These basic compositions introduce a geometry on the model space; we can also get a notion of gradient. We can navigate this space efficiently, rather than by brute force.



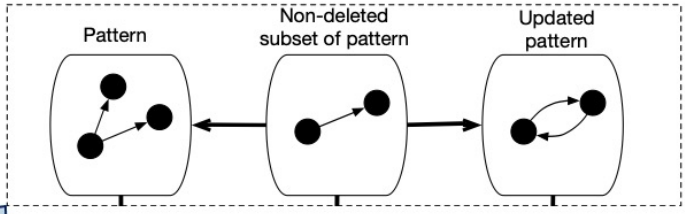

So that was the second project I wanted to highlight.




Lastly I want to talk about a model inference problem. Often the semantics of a scientific model are some kind of dynamical system. We'll focus on a flavor of discrete dynamical systems called rewriting systems.

## Simulation: rewrite rules

- In general, there is a **pattern match**, a **deletion**, and an **insertion**.
- Categorical rewriting captures all three with a *span* of morphisms



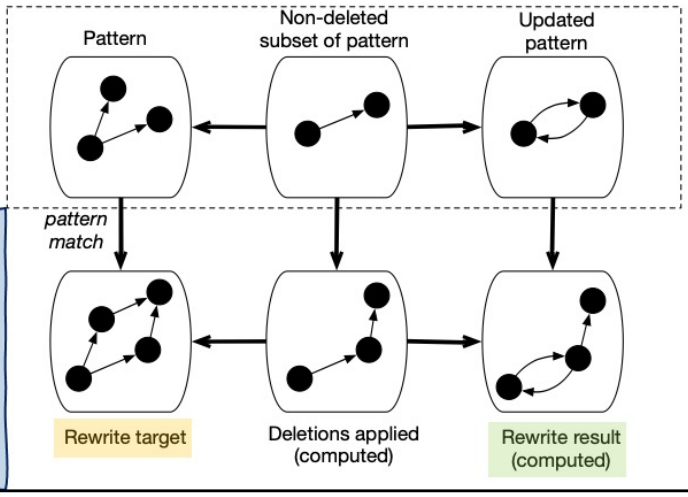

A recipe for performing a *dynamic* update can be represented by a *static* data structure.
The same few lines of code performs rewriting on *any* C-Set.

76

Main mechanism of discrete update in a simulation is applying a rewrite rule.
The structure of a rule is to have a pattern match, a deletion, and an insertion.
We break the rule down into a pattern that gets matched onto our model that we want to rewrite. We have a portion of the match that is preserved and we have a third structure which is what gets substituted into the location that was matched.

## Simulation: rewrite rules

- In general, there is a **pattern match**, a **deletion**, and an **insertion**.
- Categorical rewriting captures all three with a *span* of morphisms

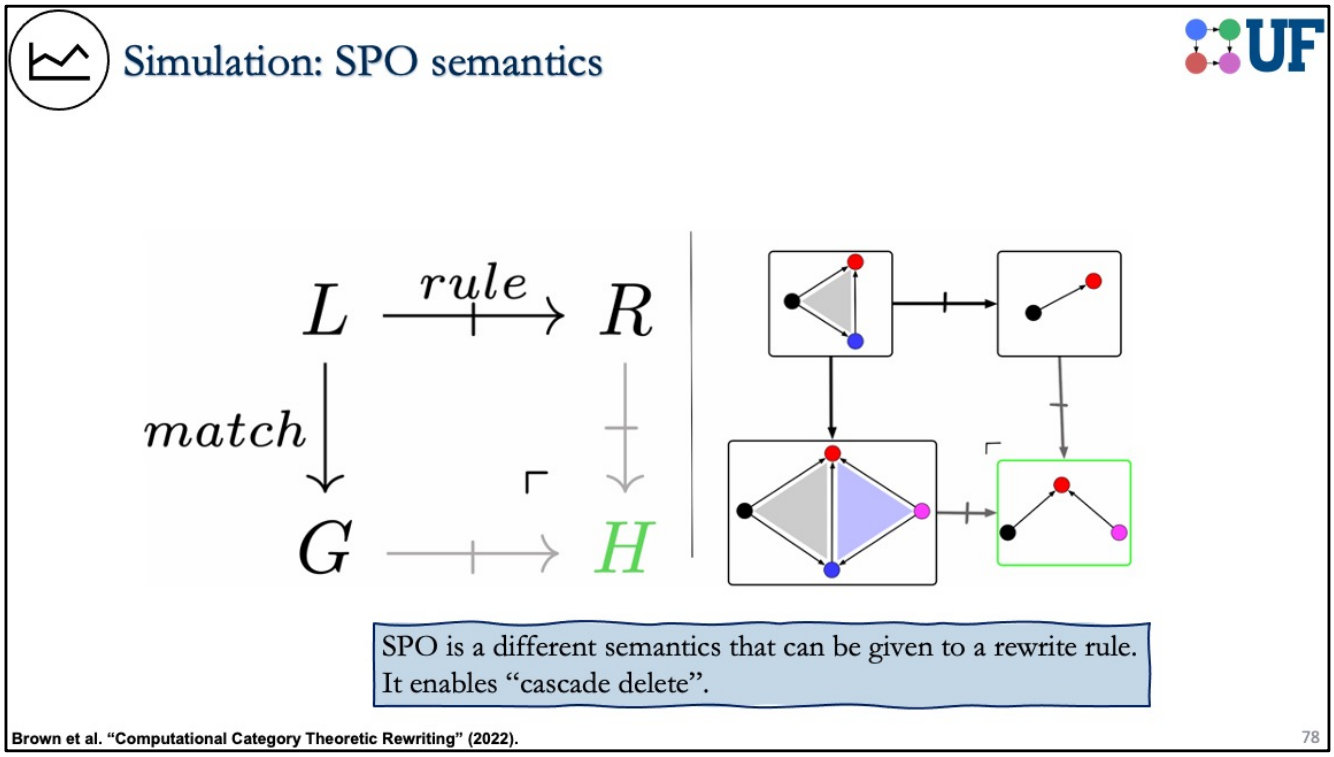



A recipe for performing a *dynamic* update can be represented by a *static* data structure.
The same few lines of code performs rewriting on *any* C-Set.

77

So you see here how this process applies to the bottom left graph and yields as the final result the graph on the bottom right. The actual implementation is defined in terms of categorical operations that make sense for all ACSets.

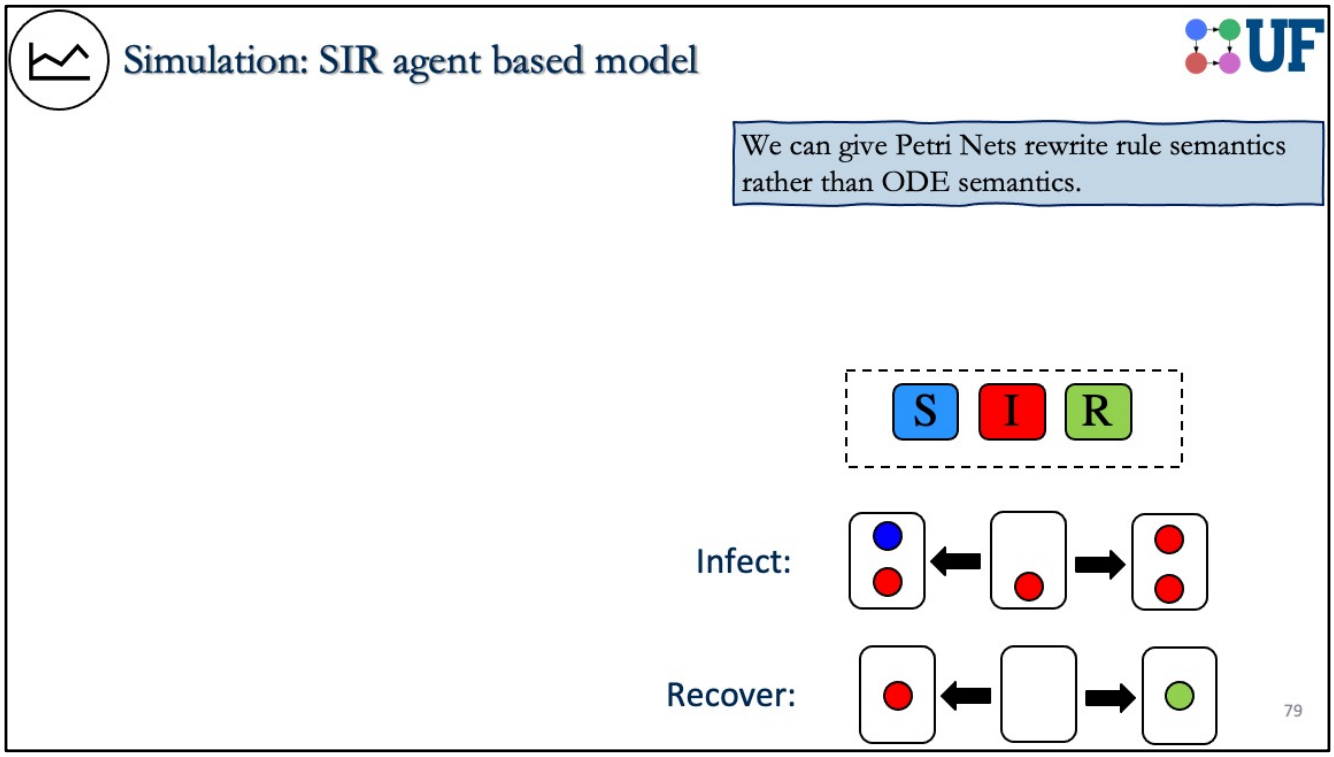
This is a lot to throw at you very quickly, but the conceptual point here is that we were able to express this inherently DYNAMIC process of rewriting with a purely STATIC structure, which is those three structures there in the box. We did not need to write a function in code or a differential equation.




What I just showed was double pushout rewriting (DPO). Single Pushout Rewriting is different from DPO because when you delete something, you implicitly delete everything that refers to it. This is like CASCADE DELETE in databases.

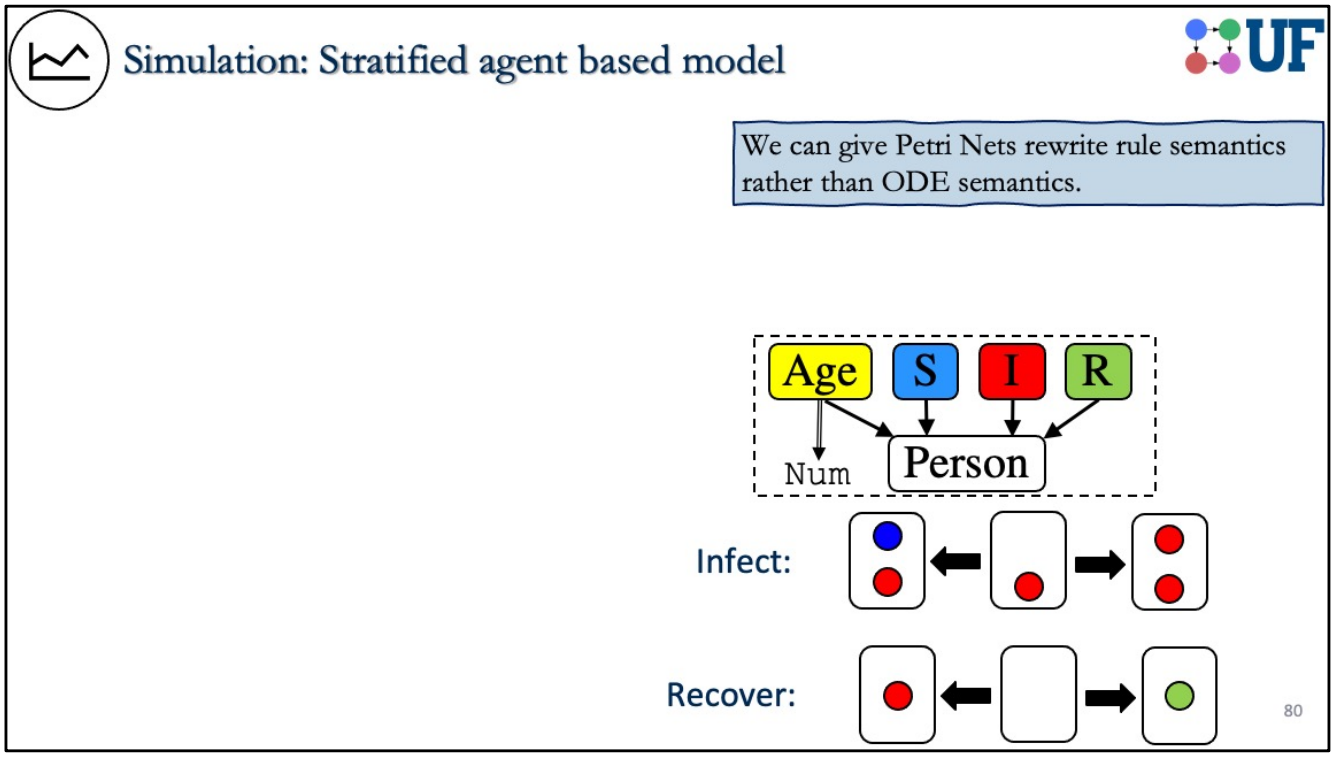
I shown an example of this using a different C-Set from Petri Nets just because it is easier to visualize. We delete the blue vertex, but the lavender triangle is referring to the blue vertex so it gets implicitly deleted, even though it's far removed from the part of the model that got matched by the pattern.

Here is another example of the same syntax (the rewrite rule) being given multiple semantics.

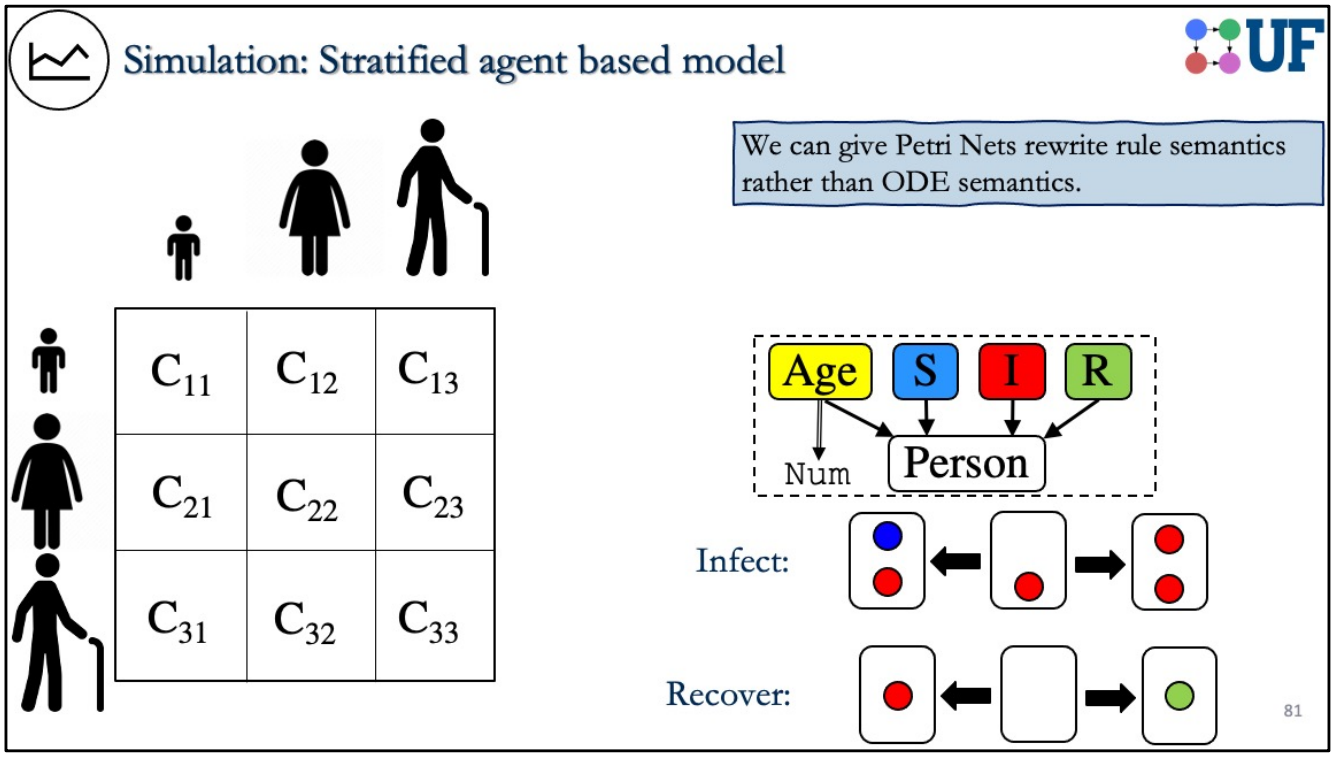



So taking the world state to be represented by a C-Set with just three objects and no morphisms, we can recapture the dynamics of SIR by these two rewrite rules.

These could be generated automatically from the Petri Net representation.

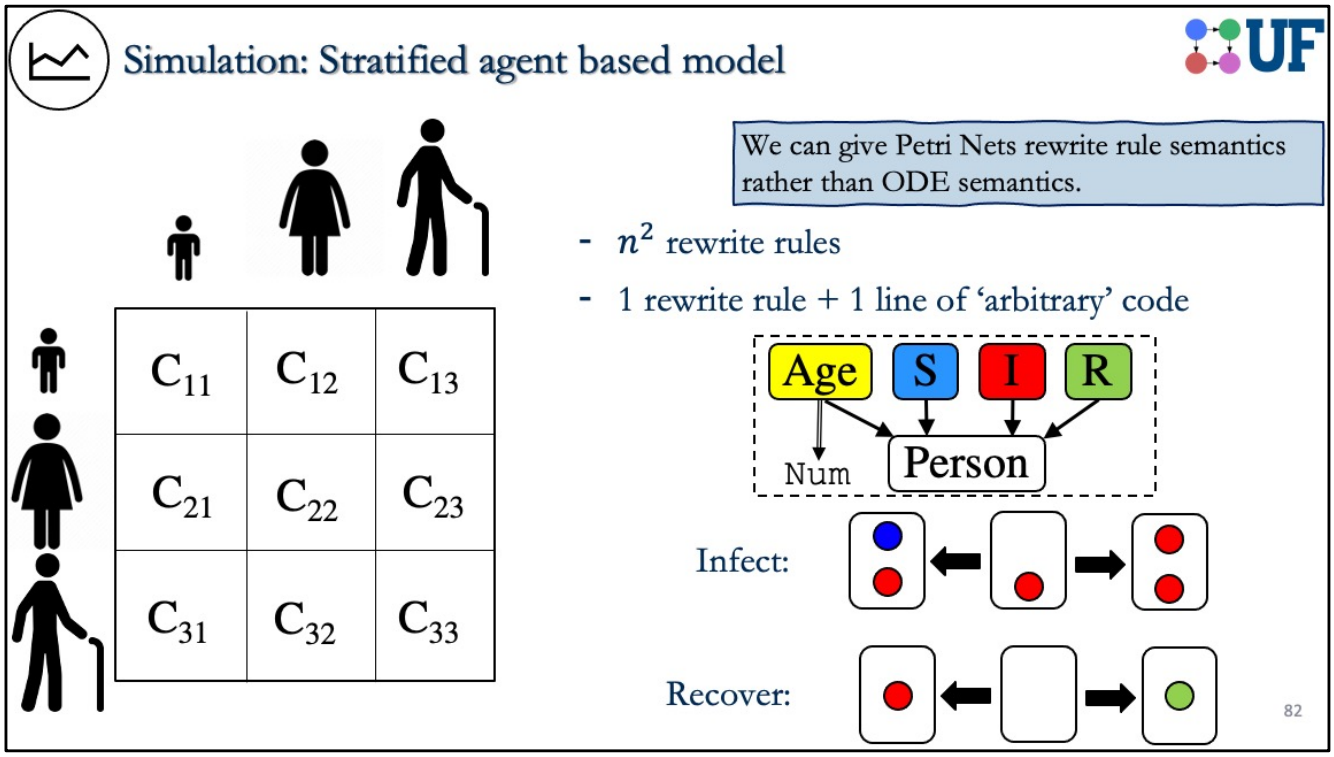



Moving beyond that toy example, we could add a numerical age to the schema.



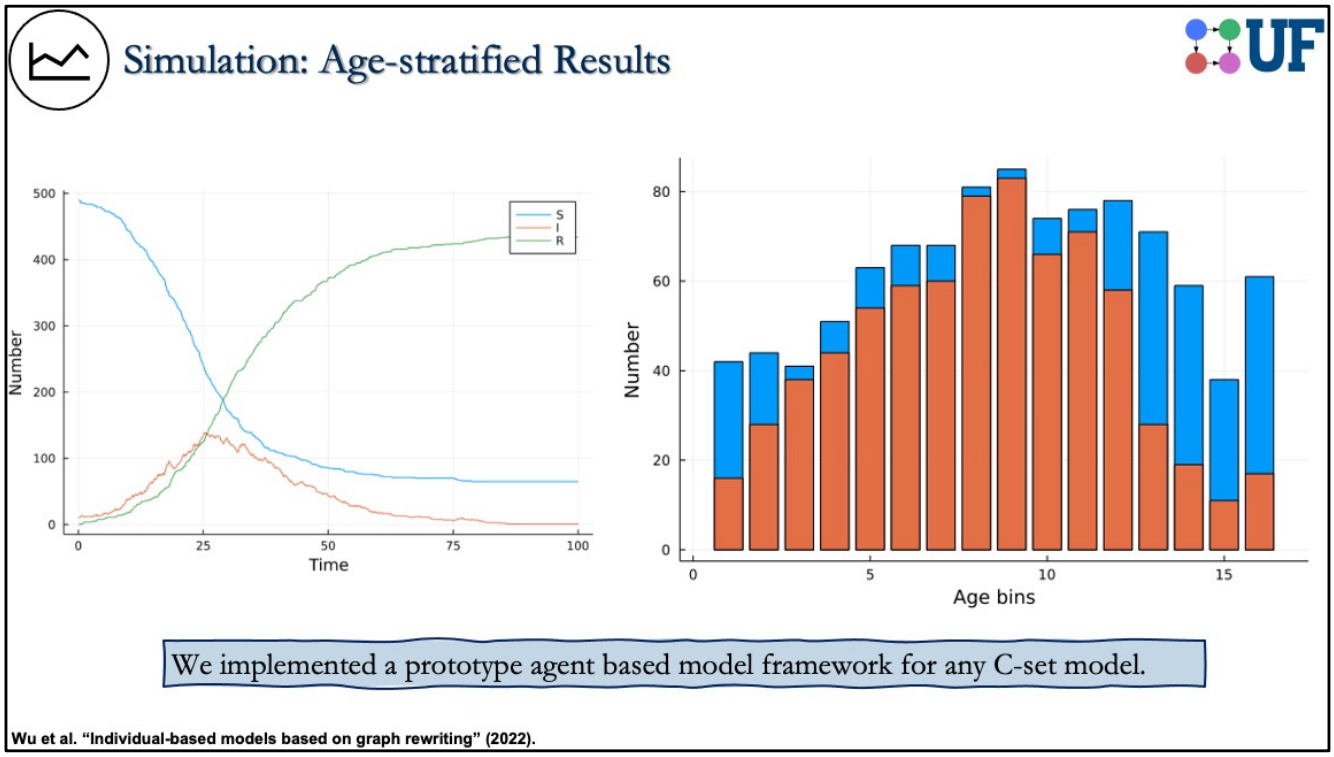

We might have some function which takes two ages and computes the probability of that infection happening.

One type of function would be to break down demographics into bins, and n^2 numbers characterize each group's propensity to infect another.




Here there is an interesting tradeoff where having n2 rewrite rules is completely transparent to a computer, but one rewrite rule + 1 line of arbitrary code feels more human interpretable.

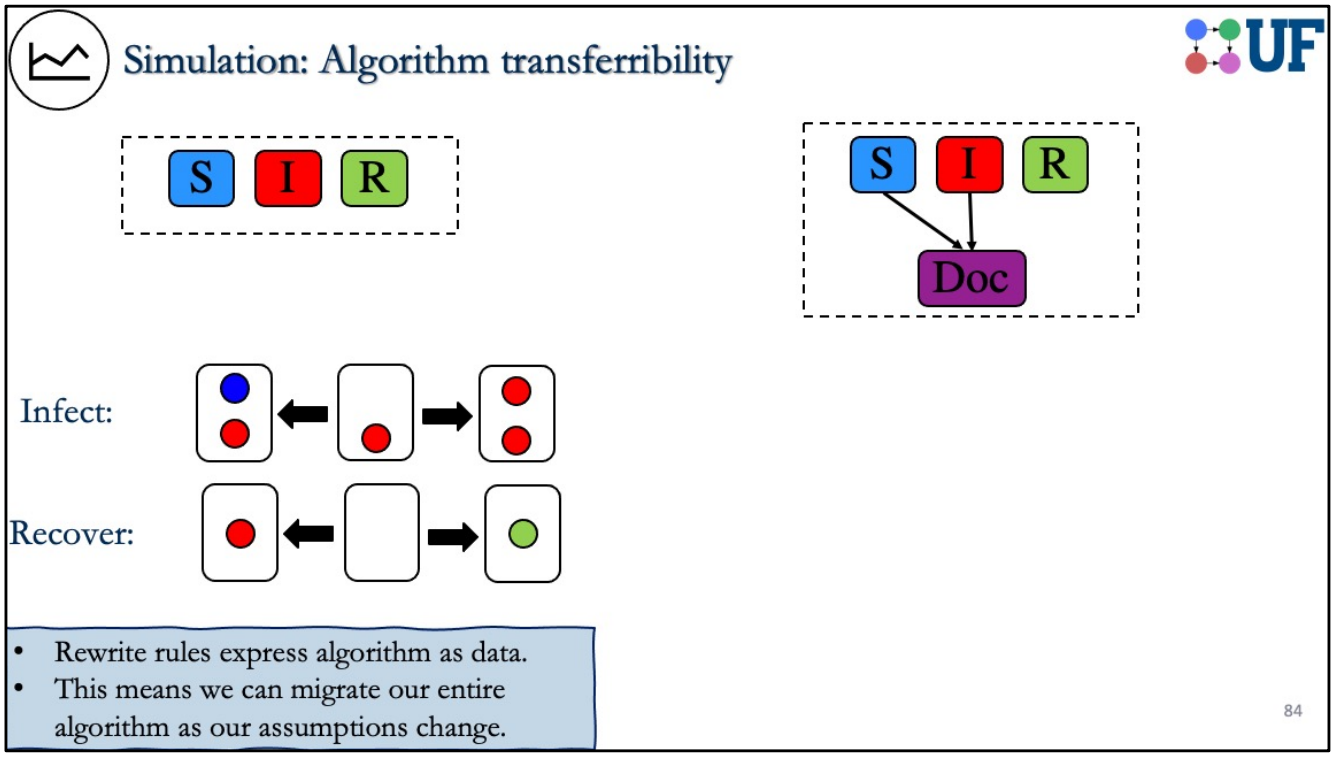
But we're in a position to pick which design makes the most sense.




So we used a real contact matrix based on research in Taiwan, and Sean Wu at the institute of Health Metrics and Evaluation was able to reproduce earlier results from Individual.jl by using C-set rewriting as the core engine.

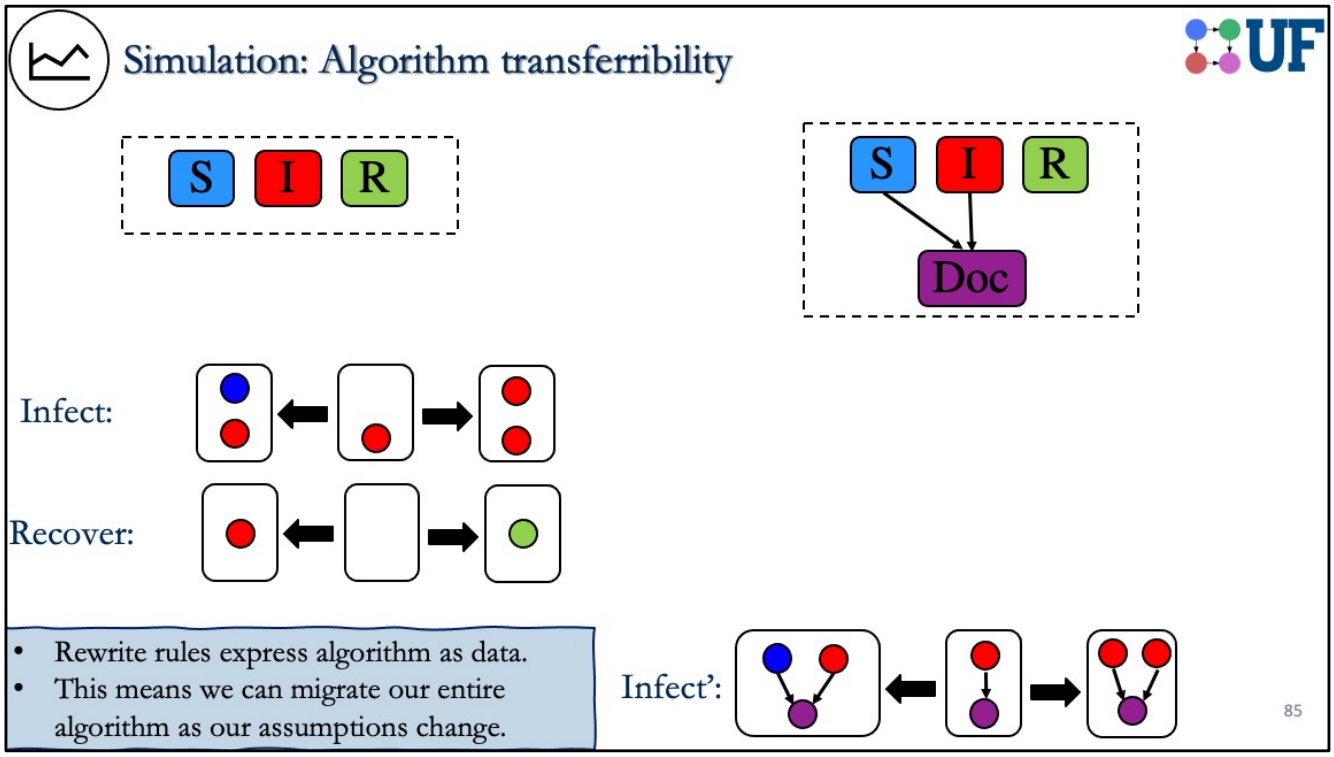
In the plot on the right, the orange fraction of the bar of an age group shows what fraction got infected over the course of the simulated pandemic.

You can see the elderly and young are relatively insulated from the working age population which bears the brunt of infections.



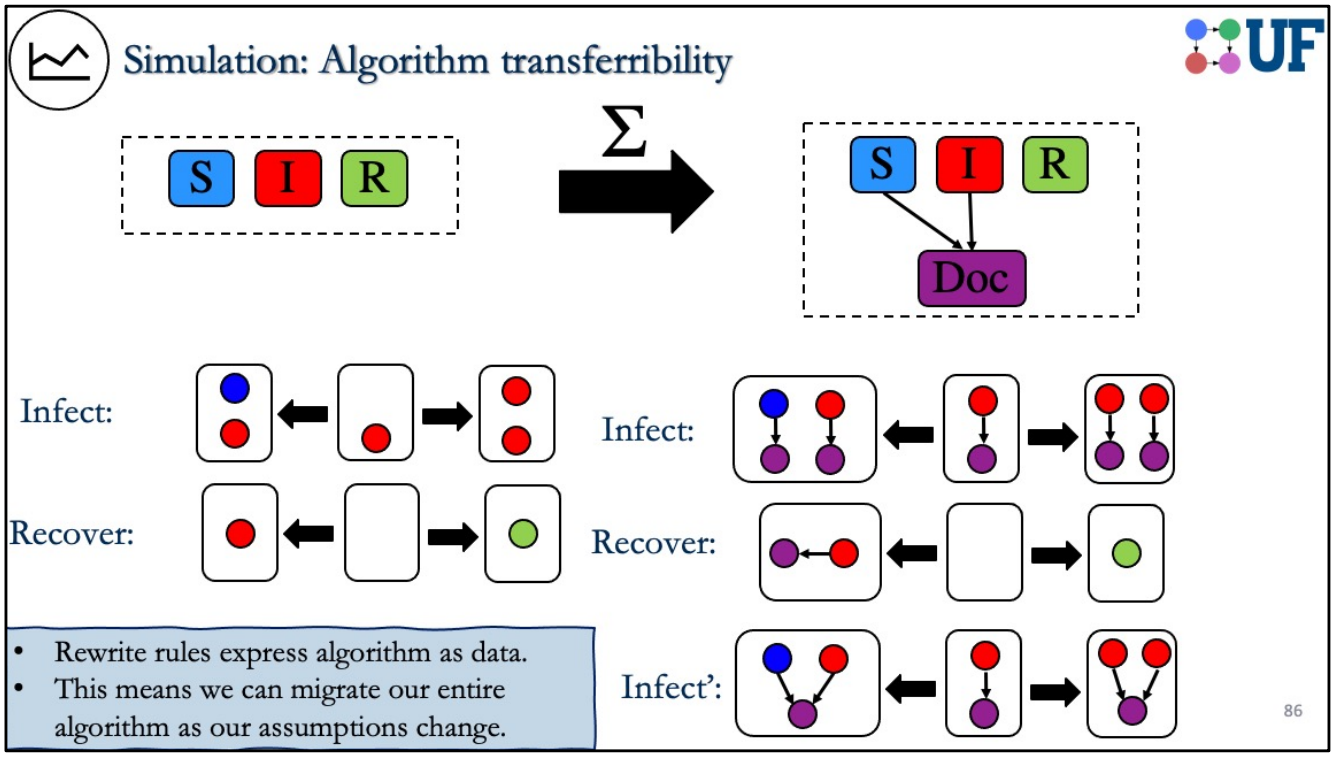

Here's one reason why one might prefer the machine interpretable route. I'm going to demonstrate the transferribility virtue, which we obtain by having our algorithm itself expressed combinatorially.

Suppose we decide to update our schema like so, where we model the fact that susceptible and infected people each are assigned a doctor from some set of doctors.




This might be because we deliberately want to introduce an alternate mode of infection that only applies to people who share the same doctor (maybe in the waiting room, or the doctor is a vector), which could be written like this.

But we might have a lot of infrastructure built around our old SIR model. How do we ”import” this into our new schema?




It turns out there is a very obvious functor from the left category into the right one, and by pushing the old rewrite rule into the new schema, we obtain the correct interpretation of it, which freely assigns a doctor to each person in the rewrite rule.

This is an example of migrating an algorithm, and likewise when we express queries combinatorially (e.g. by representing queries as homomorphisms), we can migrate our analysis as well.

This should be pretty mindblowing to people who think of algorithms as only expressible as general purpose code.

## Takeaways

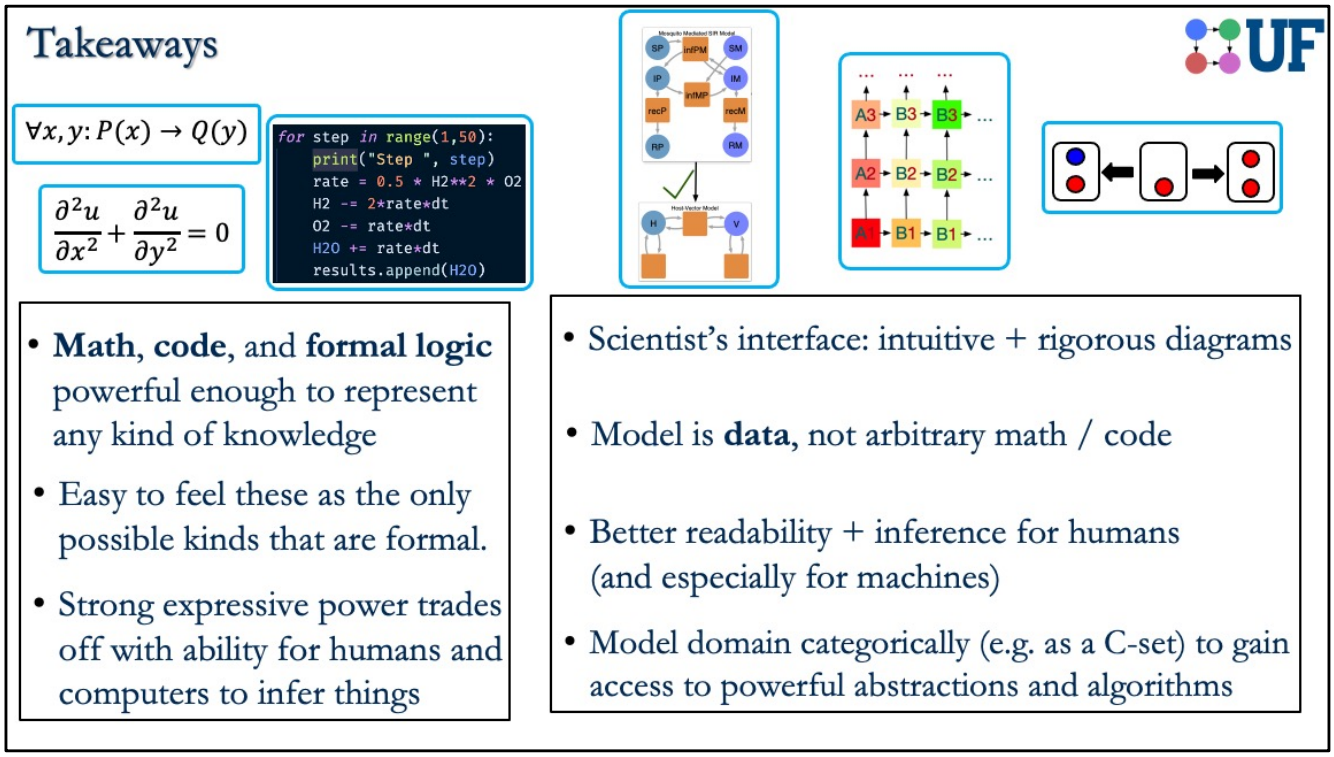
$\forall x, y: P(x) \rightarrow Q(y)$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

```
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)
```

- **Math**, **code**, and **formal logic** powerful enough to represent any kind of knowledge
- Easy to feel these as the only possible kinds that are formal.
- Strong expressive power trades off with ability for humans and computers to infer things

So despite the variety of examples, I think there was a central thesis to this talk. We're trying to present a different paradigm to scientists who think that math, code, or formal logic are the only ways to rigorously represent their domain knowledge. These are incredibly flexible and powerful tools, but that is a double edged sword because they are very difficult to reason about, for a human or especially a computer.

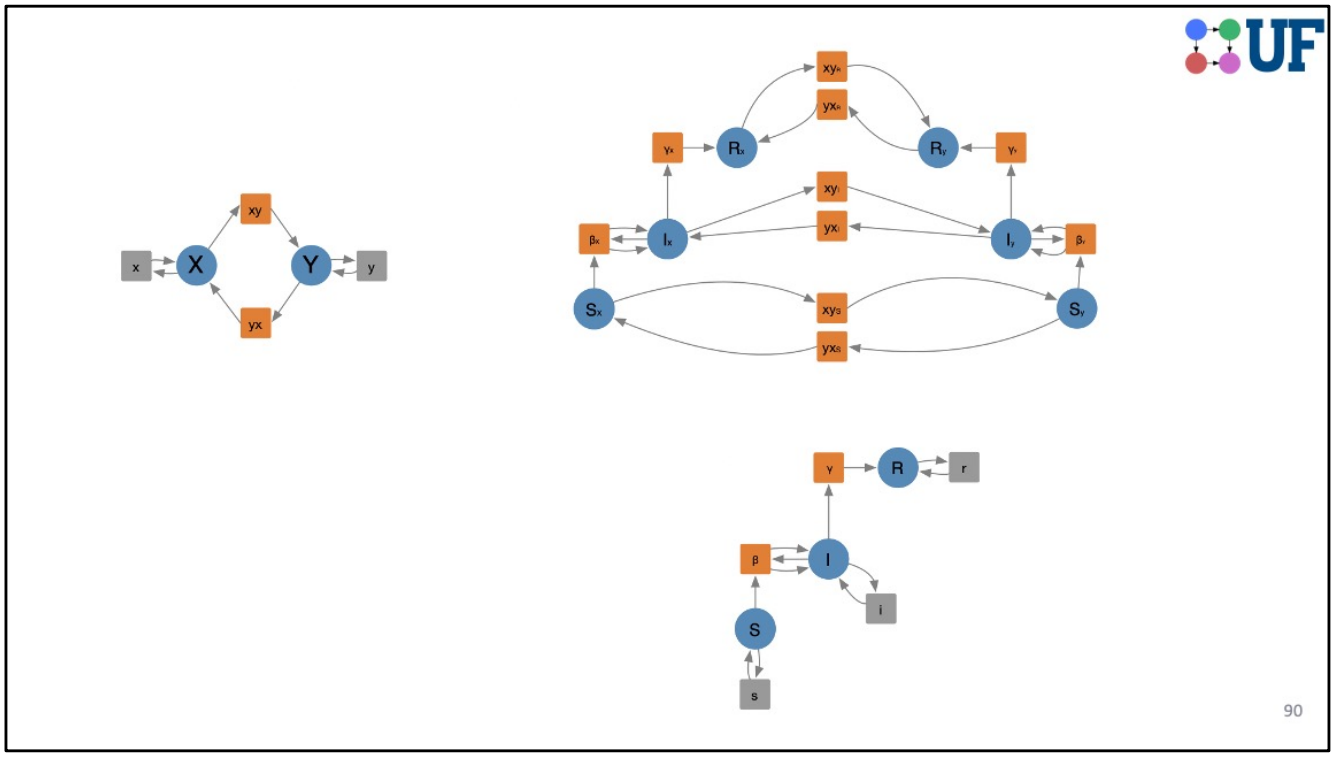



On the other hand, combinatorial data is inherently visualizable, making the barrier to rigorous modeling lower.

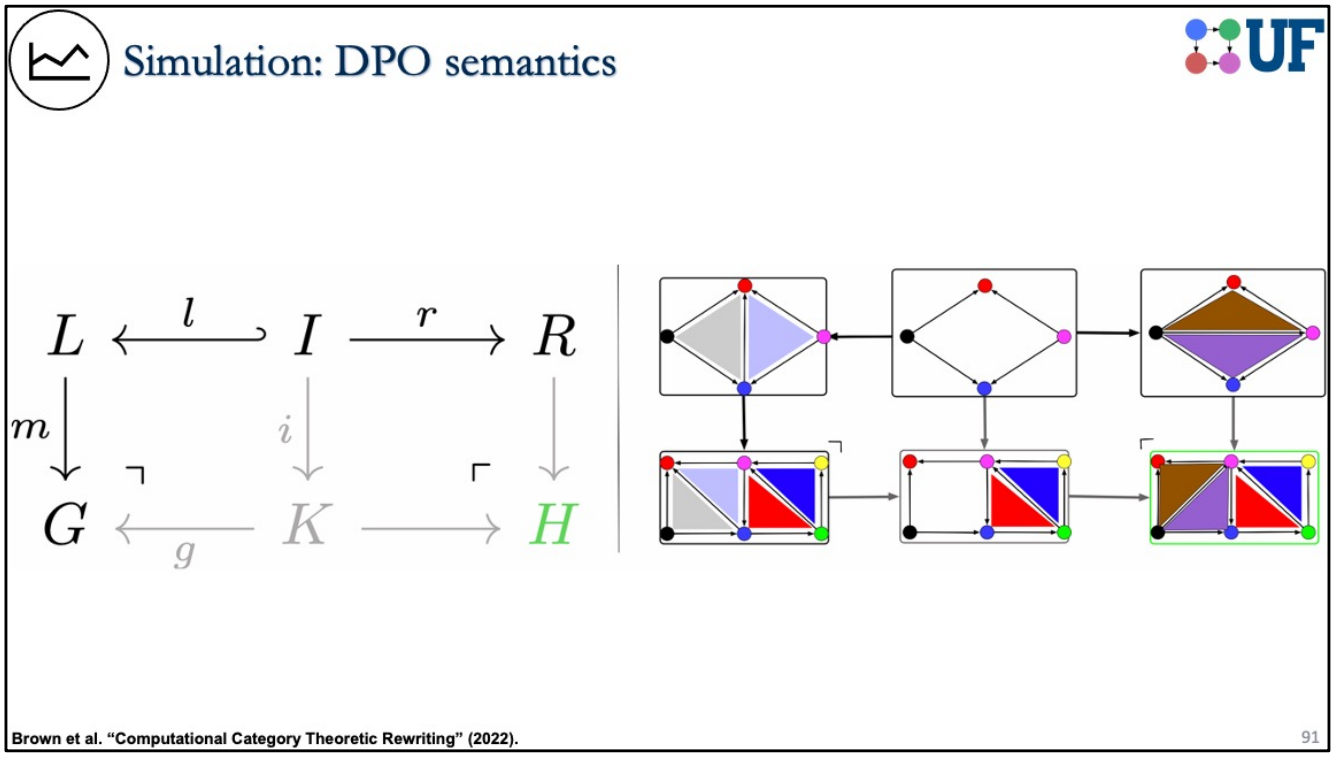
We showed lots of examples of why it's nice when computers can automatically reason about your knowledge, such has migrating information and algorithms from one domain to another in an automated way.

We're particularly interested in C-sets which have a low barrier to entry, wide generality, and many powerful abstractions and algorithms to use with them.


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

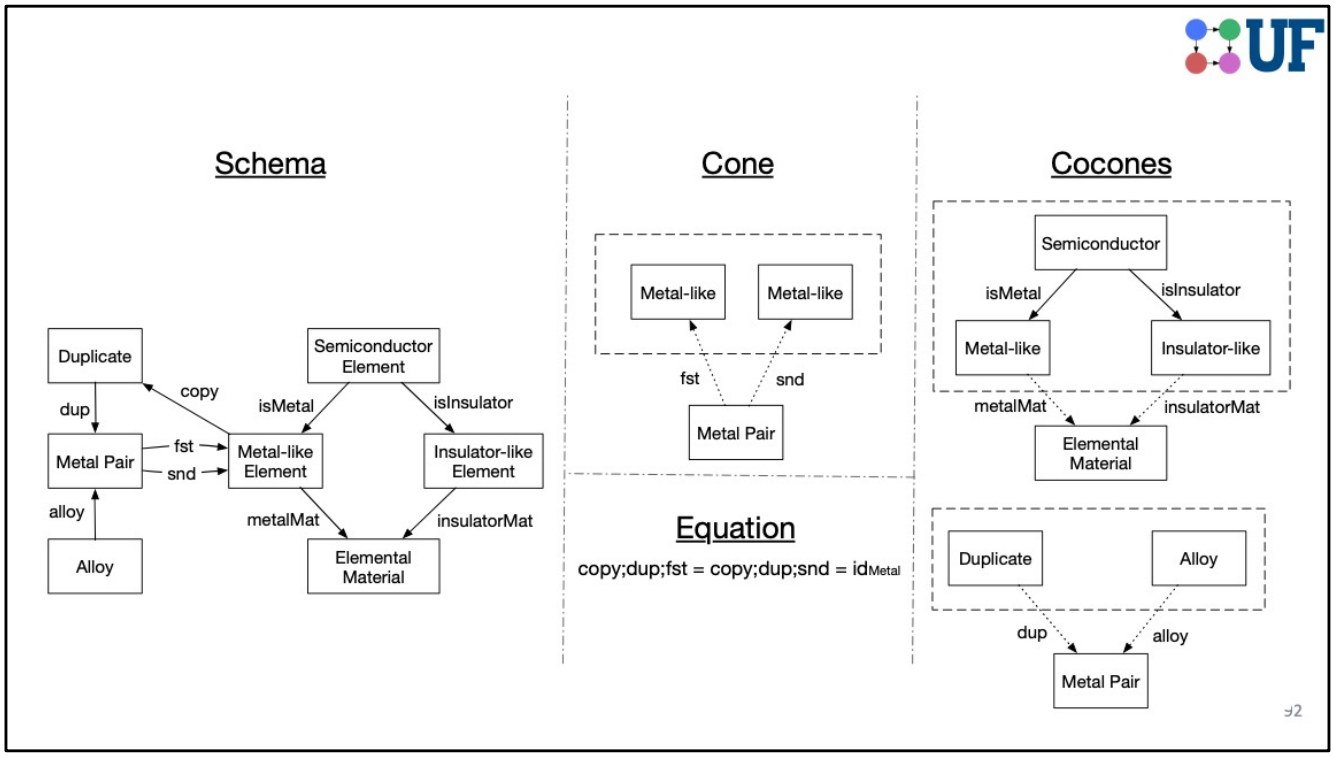



HIDDEN
The REAL pullback (slightly more annoying to specify)



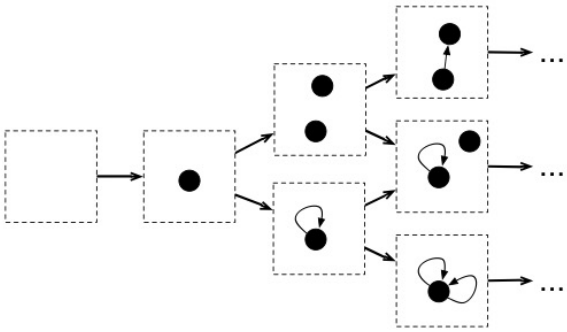

HIDDEN

Here's another DPO example using the C-set of triangles I showed near the beginning of the talk, where we flip a quadrilateral.




HIDDEN
Example of scientific knowledge encoded by finite (co)limit sketches

## II. Defining model space: enumeration



| Semigroup order | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| # of C-sets | 1 | 16 | 19683 | 4294967296 |
| # of semigroups | 1 | 5 | 24 | 188 |

- If constraints were arbitrary code, we'd have no choice but to generate C-sets and filter.
- With a restricted language of constraints called *sketches*, we can greatly prune the search space.

- Pro: finds "unknown unknowns"
- Con: massive search space, results have no semantic meaning

93

HIDDEN

In some sense it's the most general but least interpretable way to explore models.

I think sometimes we do want this - it can find structures we weren't consciously expecting.

And you might learn there's something wrong with your framework if this procedure produces models that seem fundamentally wrong. I'll show an example of that soon.

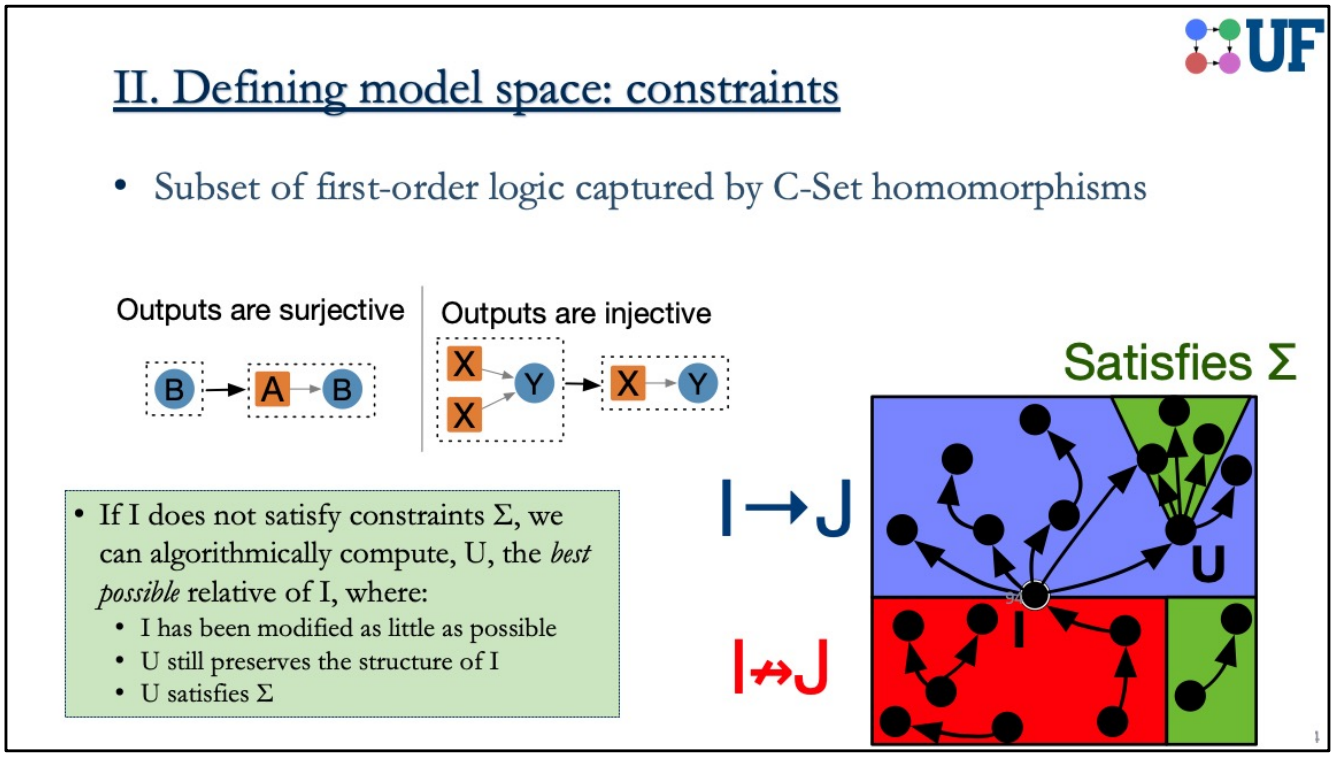
But the search space is huge, and unlike the other base-level ways of introducing generators, when your model has contents that get ultimately traced back to originaiting from enumeration, there's not much you can say about them, semantically.

Computationally, things were actually pretty easy when enumerating graphs because every valid C-set was a valid graph.

What about something with just a bit more structure, like a semigroup (which is an associative binary function).

It turns out that a naïve enumeration algorithm will be useless for finding semigroups of a reasonable size.

So we want to constrain our search by something that has some structure that we can use to efficiently prune the space. It turns out there's a category-theoretic notion of a "sketch" which is a very expressive constraint language while still being computationally tractable.



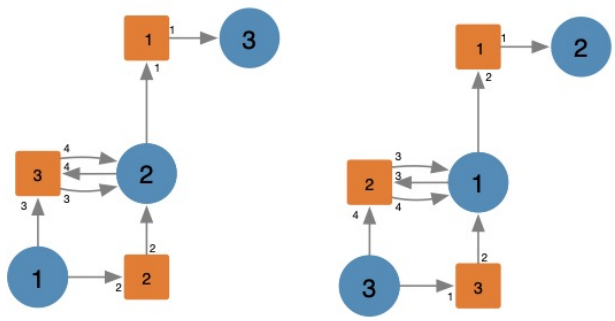

HIDDEN
Example of some constraints encoded by C-set homomorphisms

For example, if I want to say the output relation of a Petri net is surjective, then I want to say, for every state "B", there exists a transition "A" that has "B" as its output. For injectivity, I want to say, for every two transitions which output to the same state, those two transitions are actually the same. Together these would allow us to enforce that each species has a unique transition that produces it, if we wanted to do that for some reason.

You're specifying this complicated constraint specification and repair process, not with 1000 lines of opaque code, but just elegantly with these few structures.

## III. Exploration of model space: note on symmetries

- We can arrive at the 'same' C-set via multiple paths
  - Rewriting rules, enumeration, products and sums, applying constraints
- # distinct C-sets on a given schema >>> # of isomorphism classes
- Distinctness due to implementation details: we want to ignore them



| n | # C-Sets | # Iso classes | Time |
|---|---|---|---|
| 1 | 1 | 1 | Instant |
| 2 | 256 | 17 | Instant |
| 3 | 531441 | 505 | Minutes |
| 4 | 4294967296 | ? | ? |

95

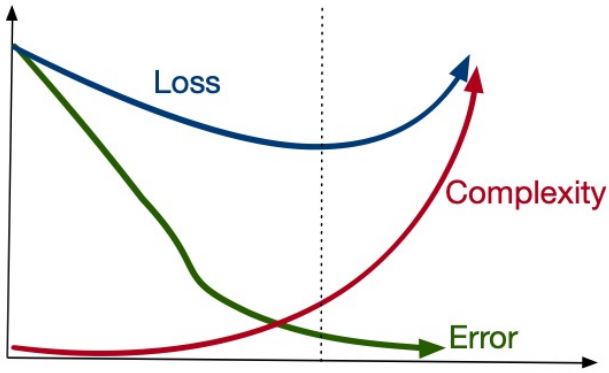
HIDDEN

it's possible to have models which we feel ought be considered to be equivalent actually seen as distinct. This makes our model space MUCH larger than it ought to be.

For example, to the computer, these two Petri Nets look completely different.
For a given pair, we can use a CSP algorithm to detect the existence of an isomorphism. But what if we have seen 1,000 models so far and want to check whether a freshly generated model is new? Rather than do 1000 pairwise tests, instead the solution is to just always work with a canonical permutation of the ACSet, which takes some effort to compute.

I have a blog post specifically about this problem and the generalization of a graph algorithm to ACSets, but the takeaway here is that when we explore models we are going to consider them only up to isomorphism.

# Exploration: navigating a model space



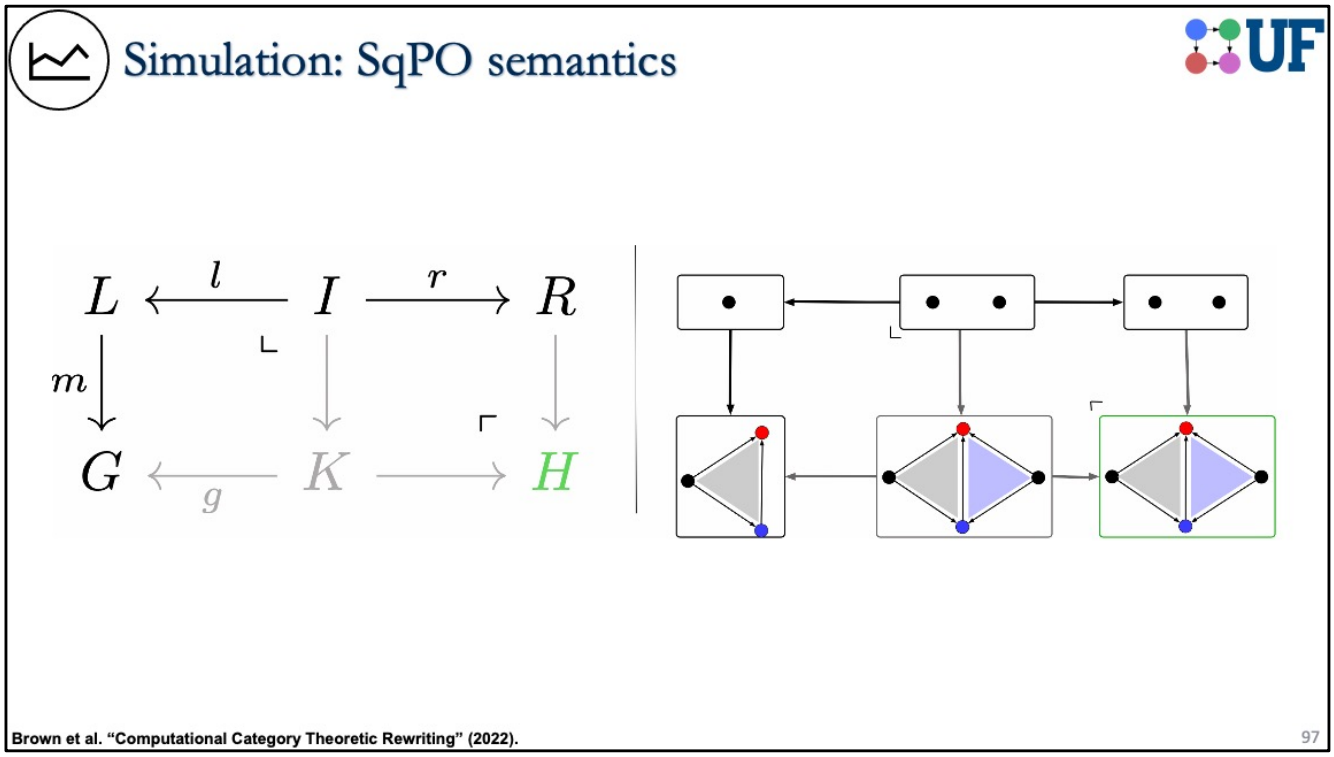

- With monotonicity assumptions, we can find the optimal tradeoff of performance vs complexity without evaluating every model.

96

HIDDEN

If we pick our structure of our models and our loss function in a compatible way, then our loss profile will look smooth and predictable like this. That means we know when to stop exploring, once loss starts going up agan.

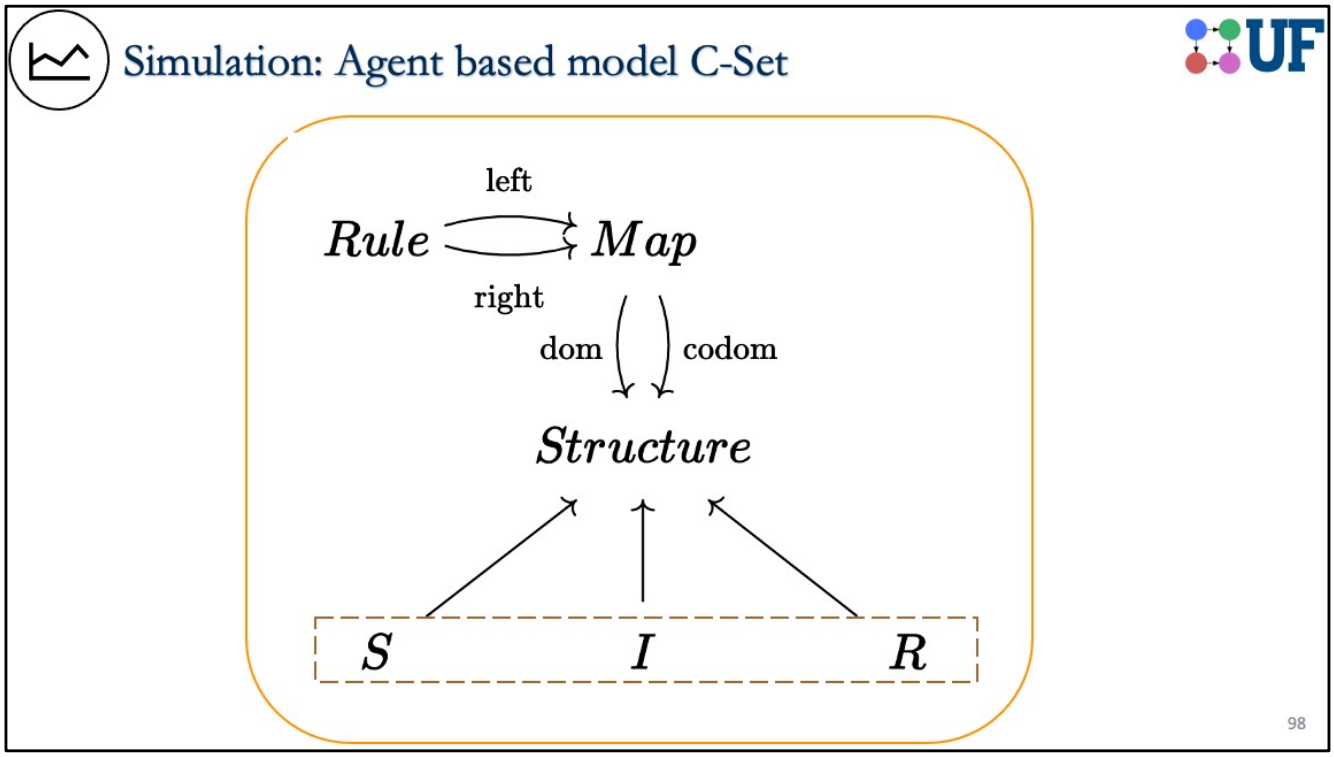
This is a nice property to have, though not all loss functions will satisfy it.




HIDDEN

Sesqui pushout rewriting adds a similar ability to SPO where you can add things which implicitly add other things.

We can define one rewrite rule, which is a syntax, and freely give it any of these semantics.

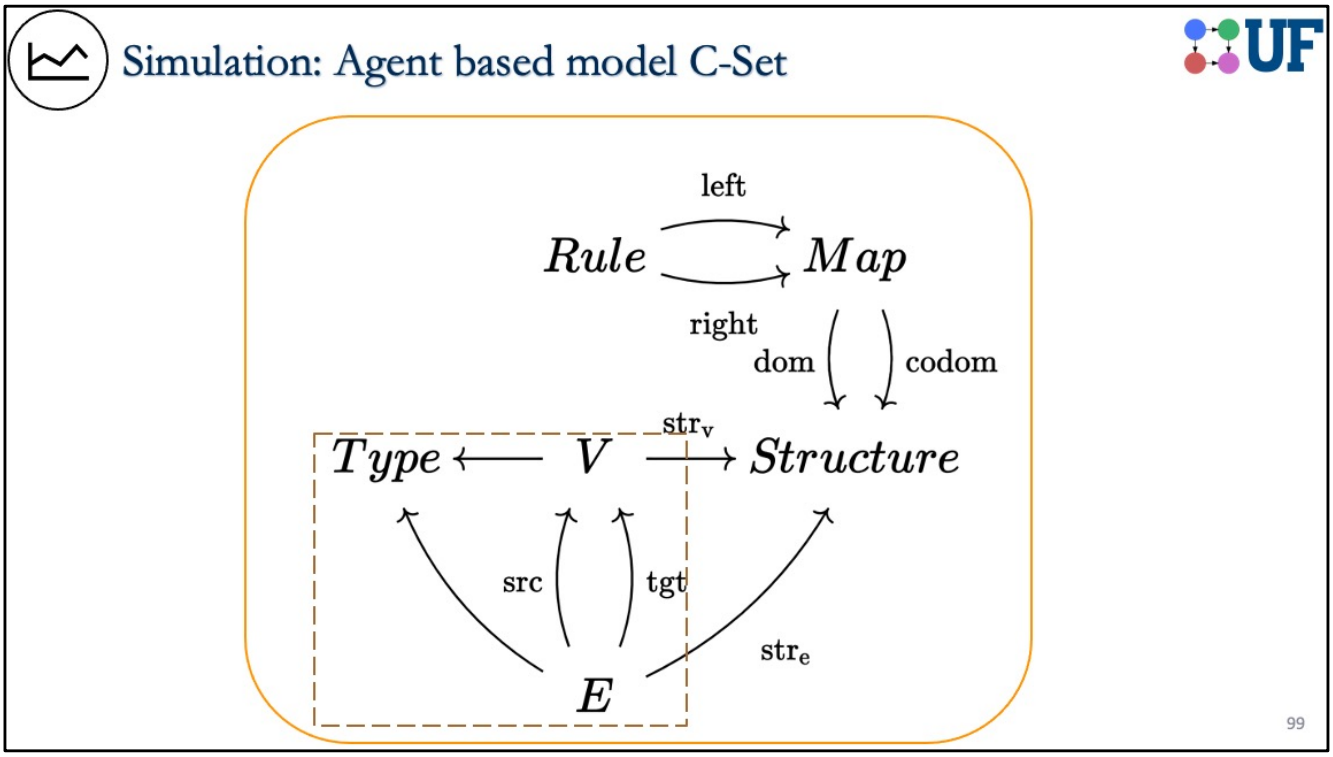



HIDDEN

The data of a very basic agent-based model is captured in the following C-Set. It has multiple rewrite rules, each which has the data of a C-set homomorphism.

Note we could have any type of C-set instead of this SIR C-Set by substituting a different C-set in the dotted box.

(We also want to add attributes to the rules to help us schedule them when running a simulation.)




HIDDEN

Here's an example of an agent based model with the state of world being represented with a typed graph.