AlgebraicABMs: An Emerging Next-Generation Categorical ABM Platform

Most diagrams from John Baez, Evan Patterson, Kris Brown, Xiaoyan Li, Nathaniel Osgood

Separation of Concerns

Ontology (in physics, “kinematics”): Specifies what is changing over time (e.g., via a schema): The state

Governing processes (in physics, “dynamics”): Specifies the rules by which the situation changes

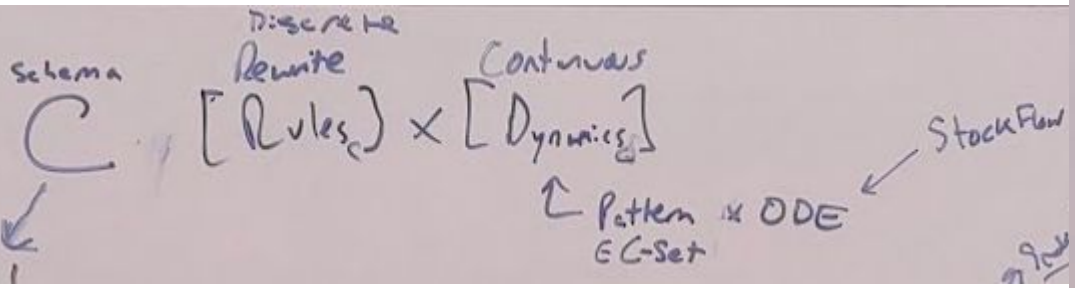
Runtime trajectory (in physics, “trajectory”)**:** Evolution of the system for a single realization

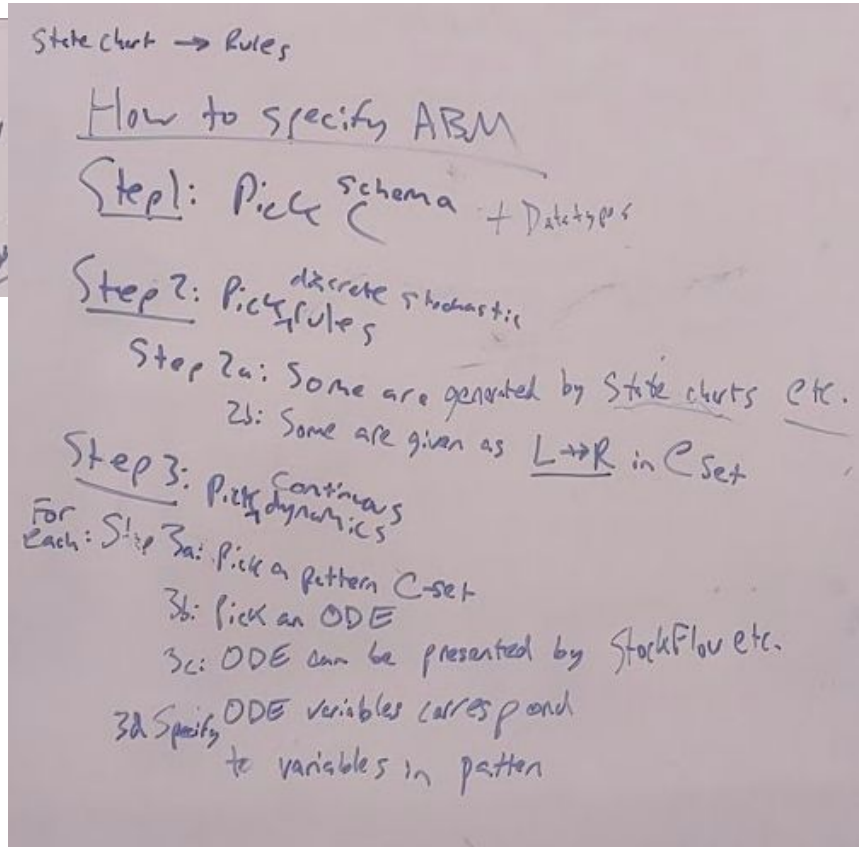
Runtime trajectory distribution (in physics, “trajectory”)**:** Evolution of the system; will be different for different purposes (e.g., single realization run, ensemble, sensitivity analysis, Particle Filtering/Particle MCMC, calibration)

Notable Characteristics of Technical Approach

- Continuous time setting; discrete time updates can occur at equidistant time points
- Intermixing of Discrete Events (CSet rewriting) with continuous (ODE) semantics
  - Talk to each other via attributes
- Metalinguistic abstraction: Each of discrete updates and continuous updates can be specified with multiple types of formalisms
- Declarative characterization of both continuous and discrete governing rules
- Minimize need to declare explicit rewrite rules via widespread use of higher level components
- Eliminating messages as ways of enabling actions based on context
- Weaving of relevant governing rules (continuous & discrete) into areas of the state schema via patterns, with dynamics updating attributes at those areas
- Data migration to translate rules between levels of context & with model evolution
- Planned: Piecing together continuous dynamics with OpenDynam, with dynamical systems coming from stock & flow, petri nets with mass-action semantics, AlgebraicDynamics (?)
- Statecharts.jl to encode statecharts
  - Statecharts as initially compiling to rewrite rules, but more restricted in semantics; potential for supporting explicitly via other high-efficiency mechanism
- Relational/contextual situations recognized as first-class entities for updating

Model Characterization





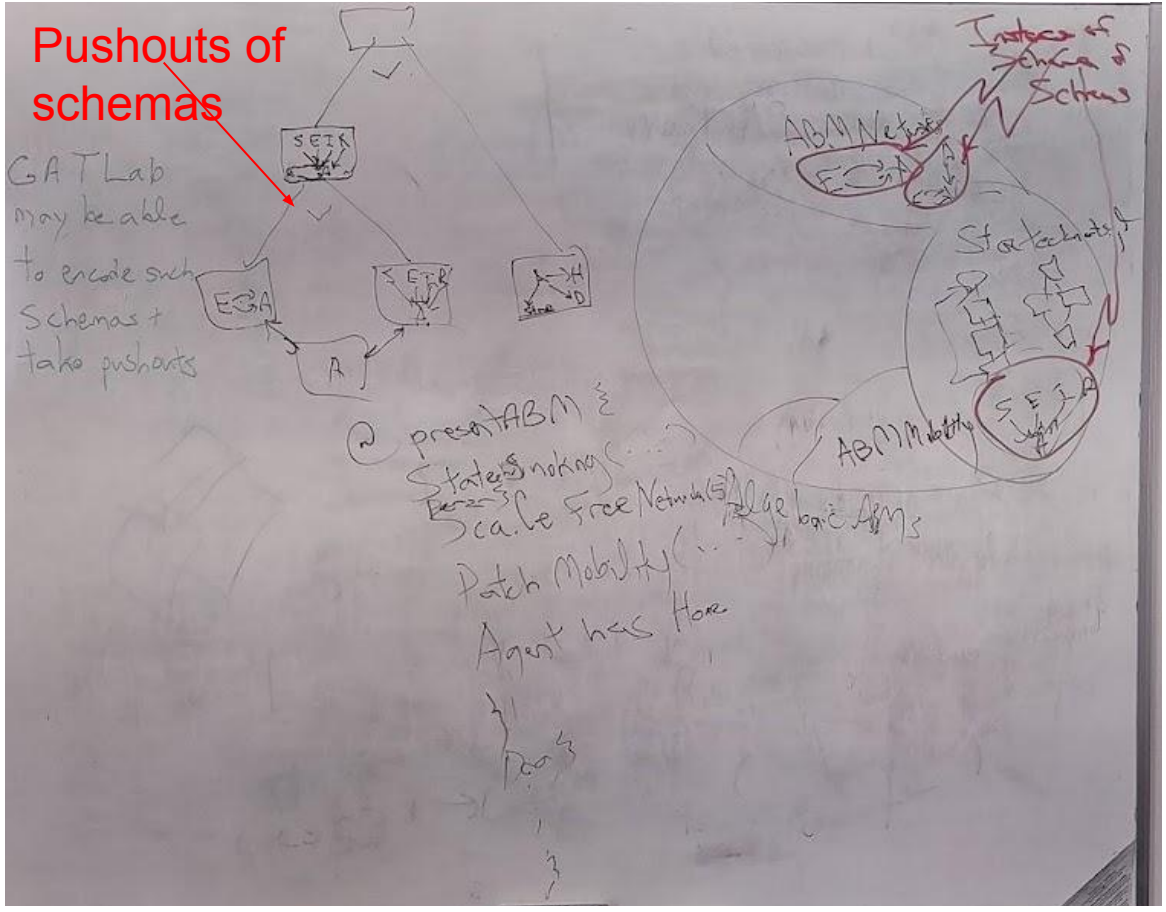
Current Code (in ABMs.jl in AlgebraicABMs.jl)

```
An agent-based model.
"""
@struct_hash_equal struct ABM
  rules::Vector{ABMRule}
  dyn::Vector{ABMFlow}
  ABM(rules, dyn=[]) = new(rules, dyn)
end
```

Layering in Context Via Schema Pushouts -- Migrating Logic as Schema Grows

While GATLab can take pushouts of different theories, for legacy reasons, it cannot yet take pushouts of schemas for the same theory.

For now, Xiaoyan has implemented a schema of schemas. This will be replaced when GATLab's mechanisms are available.

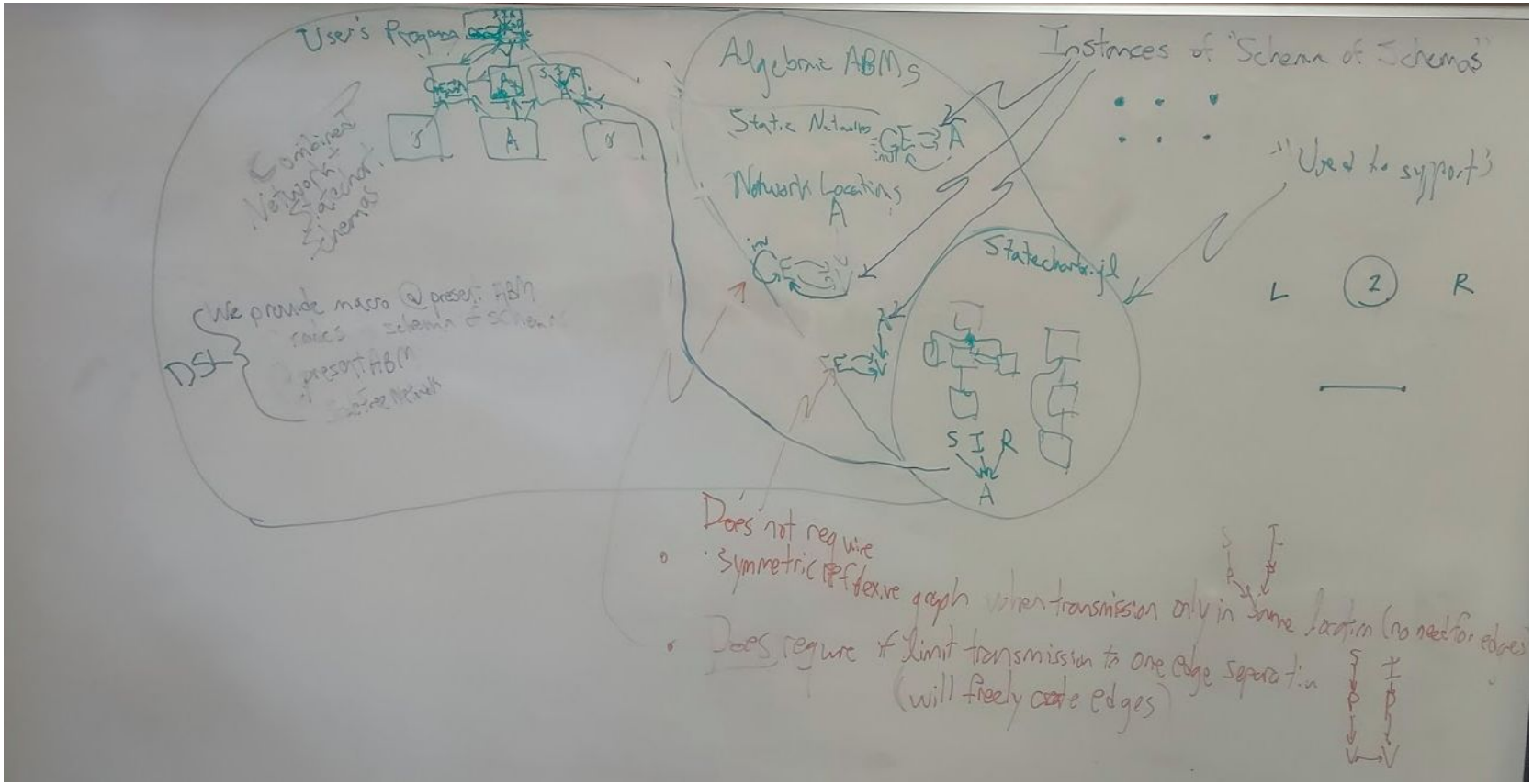


Use of C-Sets Supports Representing Higher Simplices

Assembly of Models out of Pre-Built Parts + Custom Pieces

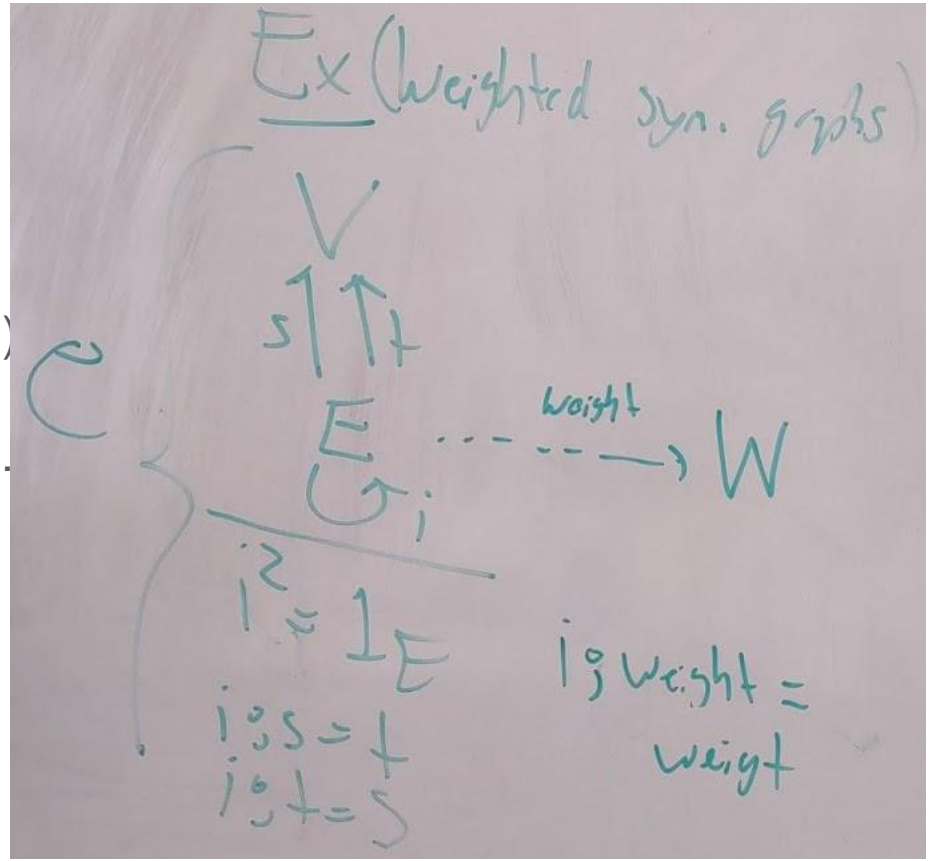
```
@abmAssembly {
                    elementalPerson(...)
                    home::Ob,
                    personHome::Hom(person,home),
                    NicotineUse(...),
                    HeartDisease(...),
                    Diabetes(...),
                    GeographicSpace(...),
                    DistanceBasedNetwork(...))

}
```

Affordances in Some Relevant Areas of AlgebraicJulia



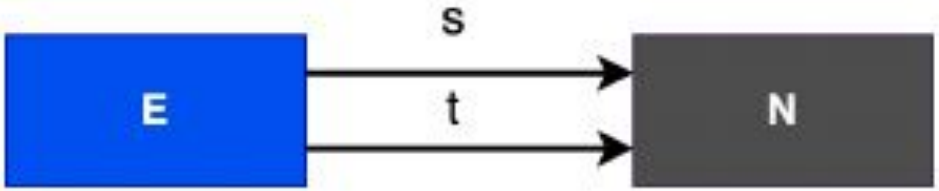
Issue -- Handling Equations

Our current workaround strategy of implementing our own schema-of-schemas (pending GATLab support for pushouts of presentations) does not support equations within schemas. For now, we need to put in place manually “fixups” that enforce such equations.



Fundamental Building Block: C-Sets

Schemas & their Instances 1



Schema category $\mathscr{C}$

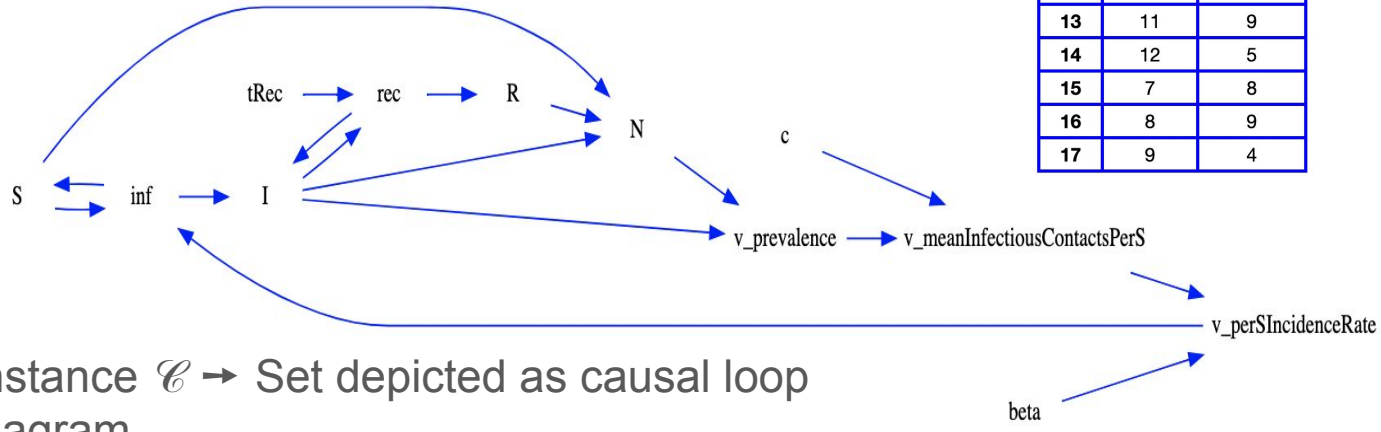
Instance $\mathscr{C} \to$ Set depicted as database

E	s	t
1	1	6
2	2	6
3	3	6
4	6	7
5	2	7
6	1	4
7	2	5
8	4	2
9	5	3
10	4	1
11	5	2
12	10	8
13	11	9
14	12	5
15	7	8
16	8	9
17	9	4

S

t

N	nname
1	S
2	I
3	R
4	inf
5	rec
6	N
7	v_prevalence
8	v_meanInfectiousContactsPerS
9	v_perSIncidenceRate
10	c
11	beta
12	tRec

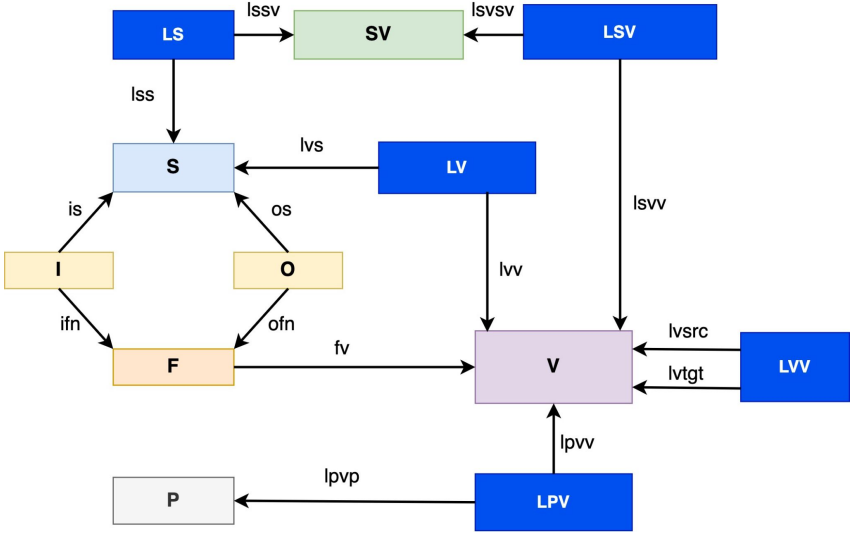


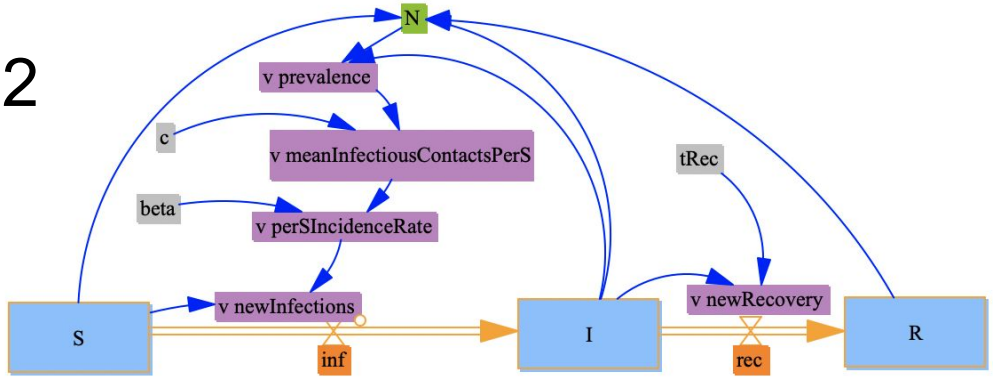
Instance $\mathscr{C} \to$ Set depicted as causal loop diagram

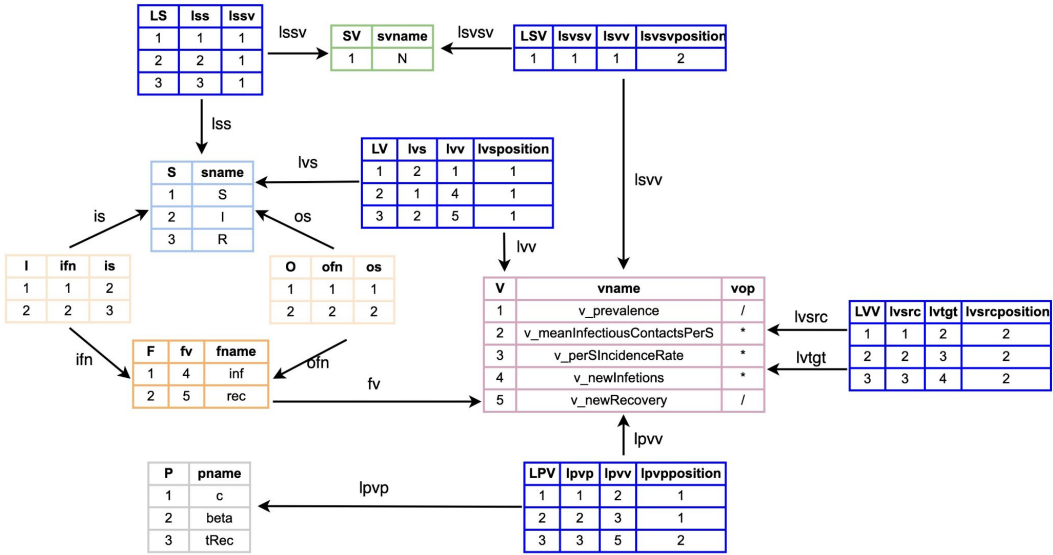
Primitive Causal Loop Diagram Schema in AlgebraicJulia

```
@present TheoryCausalLoop(FreeSchema) begin
  E::Ob
  N::Ob

  s::Hom(E,N)
  t::Hom(E,N)

  # Attributes:
  Name::AttrType

  nname::Attr(N, Name)
end
```

Schemas & their Instances 2







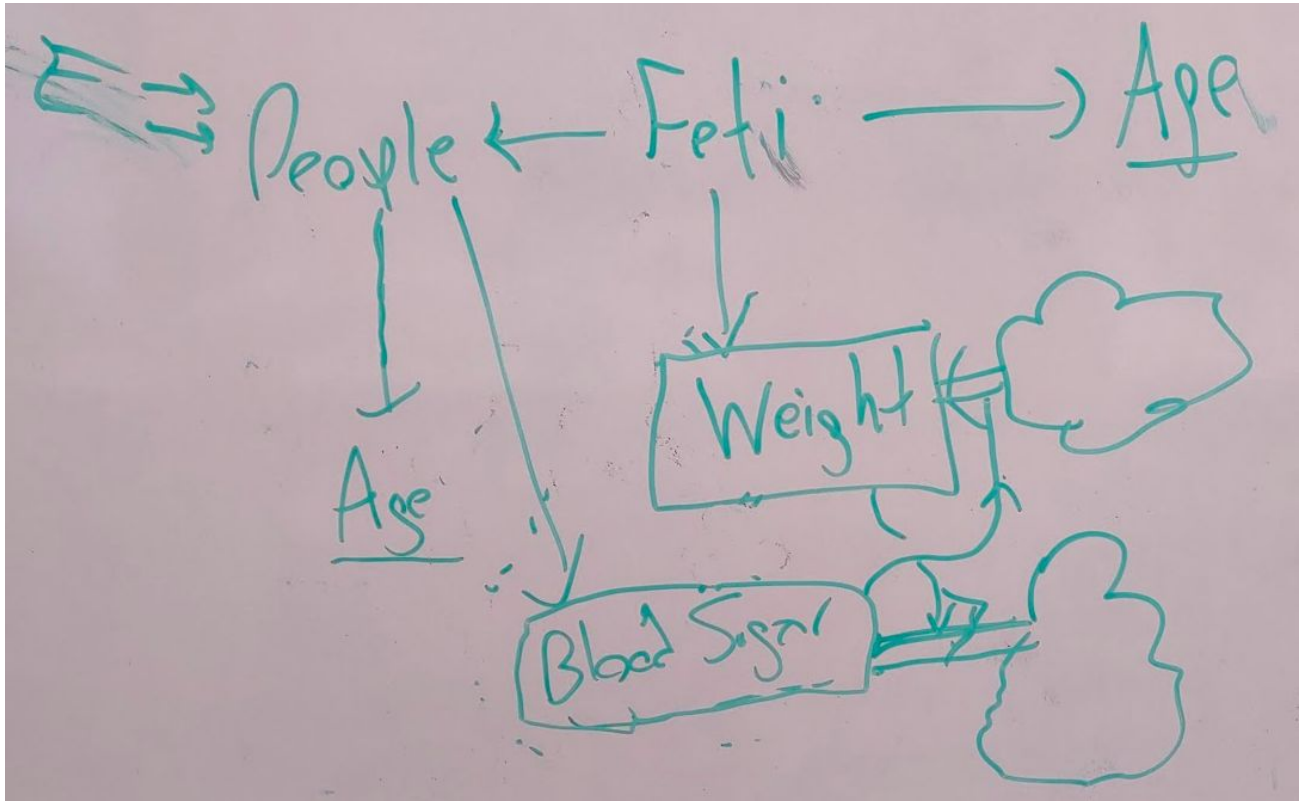
Representing Stock-Flow Schema in AlgebraicJulia

```
@present TheoryStockAndFlow(FreeSchema) begin
  S::Ob
  SV::Ob
  I::Ob
  O::Ob
  F::Ob
  V::Ob
  P::Ob
  LS::Ob
  LSV::Ob
  LV::Ob
  LVV::Ob
  LPV::Ob

  lss::Hom(LS,S)
  lssv::Hom(LS,SV)
  ifn::Hom(I,F)
  is::Hom(I,S)
  ofn::Hom(O,F)
  os::Hom(O,S)
  fv::Hom(F,V)
  lvs::Hom(LV,S)
  lvv::Hom(LV,V)
  lsvsv::Hom(LSV,SV)
  lsvv::Hom(LSV,V)
  lvsrc::Hom(LVV,V)
  lvtgt::Hom(LVV,V)
  lpvp::Hom(LPV,P)
  lpvv::Hom(LPV,V)
```

```
# Attributes:
  Name::AttrType
  Op::AttrType
  Position::AttrType

  sname::Attr(S, Name)
  svname::Attr(SV, Name)
  fname::Attr(F, Name)
  vname::Attr(V, Name)
  pname::Attr(P, Name)
  vop::Attr(V, Op)
  lvsposition::Attr(LV, Position)
  lsvsvposition::Attr(LSV, Position)
  lvsrcposition::Attr(LVV, Position)
  lpvpposition::Attr(LPV, Position)
end
```

Associated attributes are as above.

C-Sets can encode Continuous & Discrete Attributes, Relational, Networks



Rewriting Essentials

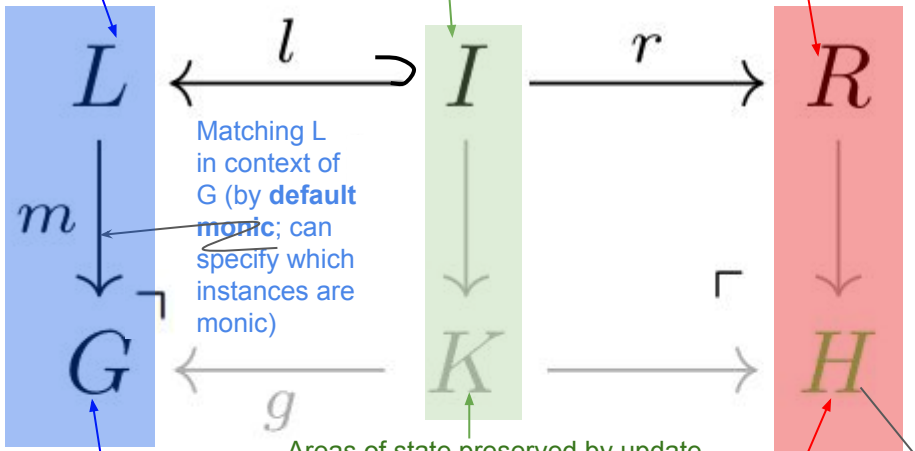
Basics of Rewriting: Double Pushout

Basic Rewriting Context

Match (everything here but not in I is deleted)
For attributes, *variables* can be included, so as to capture the value of that attribute & use in R

Features in L retained in R

Replacement (everything here but not in I is added)



Areas of state preserved by update

All objects are C-Sets, all arrows are C-Set homomorphisms

xmap

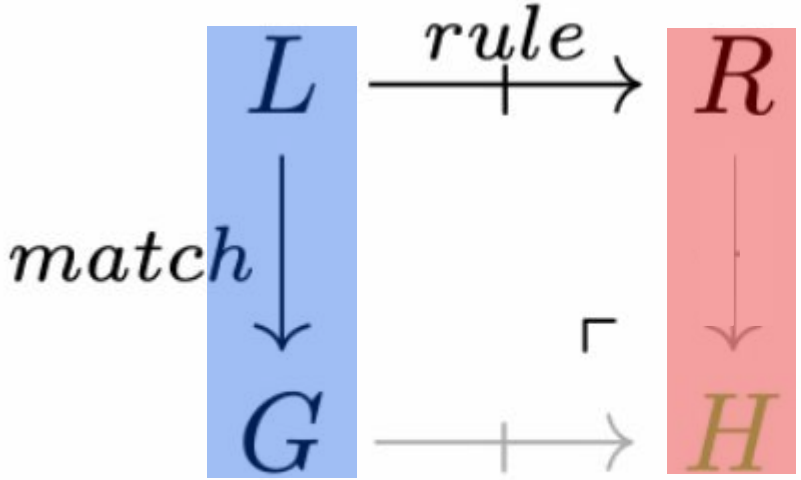
H'

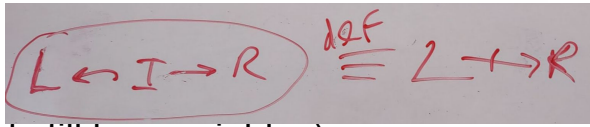
Original state (X_t)

Updated state (X_{t+1}; incorporates R but still has variables)

Updated state (X_{t+1}'; incorporates R but and all variables replaced w/values)

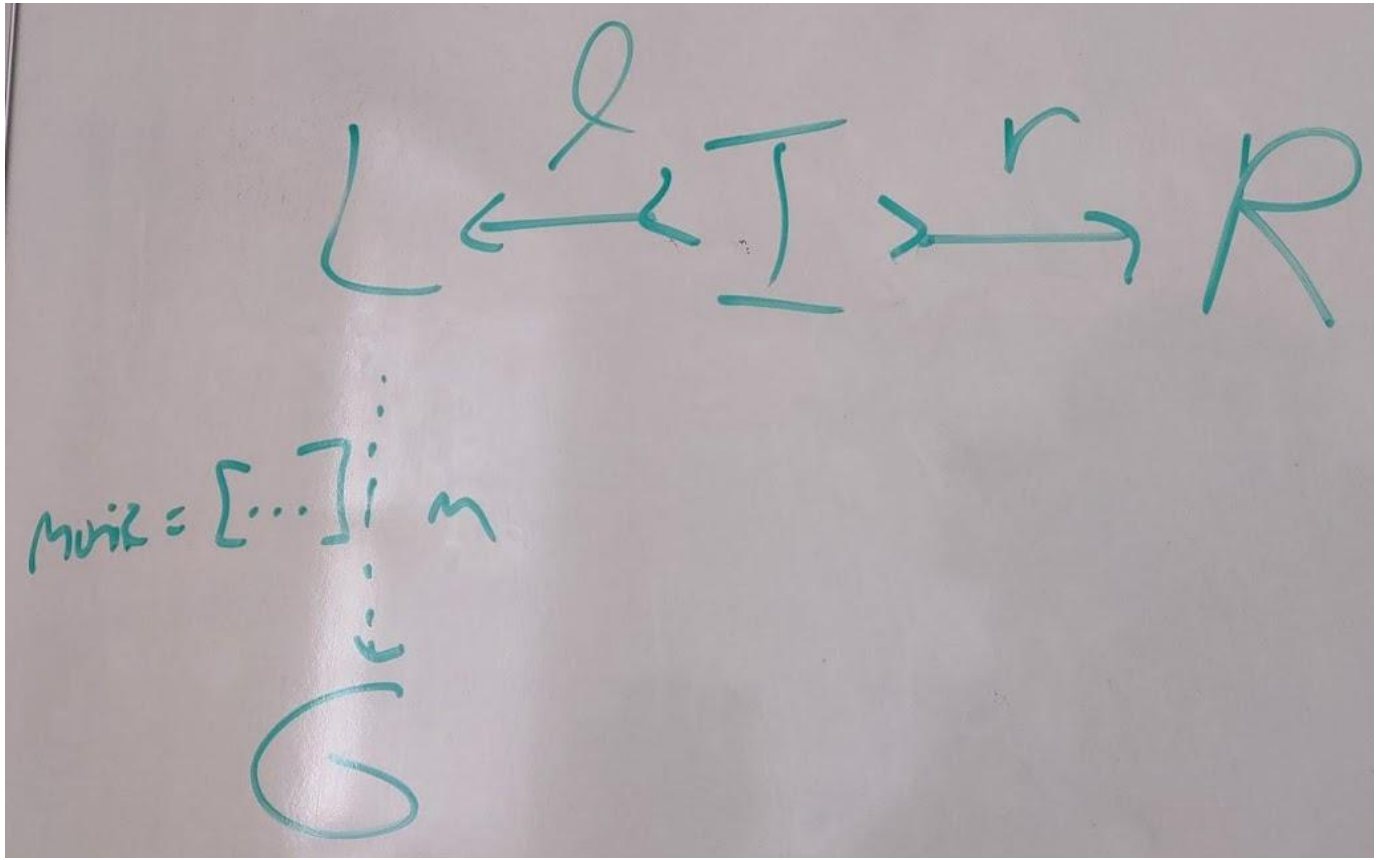
Considered as Partial Maps



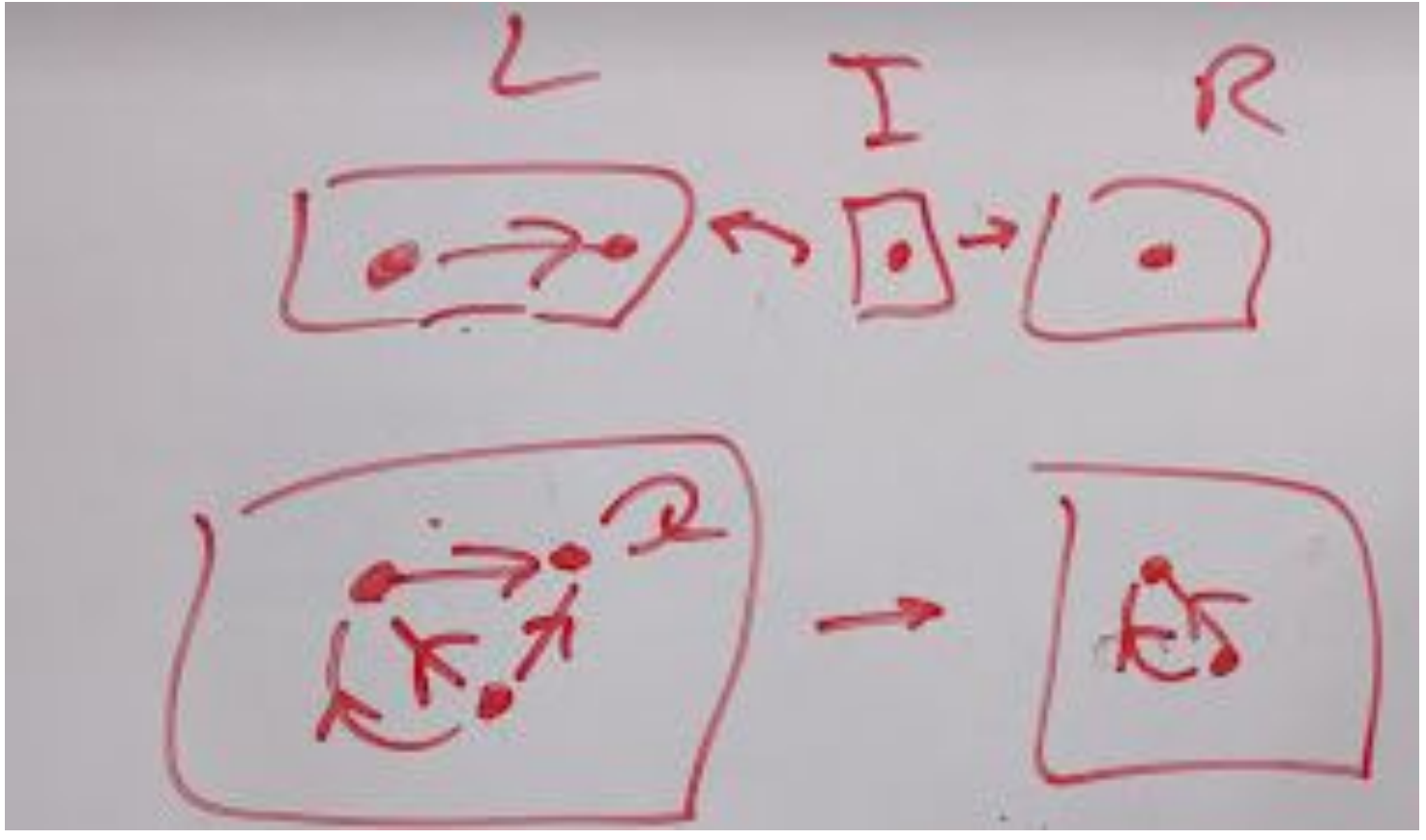


Images adapted from Brown, K., Patterson, E., Hanks, T. and Fairbanks, J., 2022. Computational Category-Theoretic Rewriting. In International Conference on Graph Transformation (pp. 155-172). Springer, Cham. https://arxiv.org/pdf/2111.03784.pdf

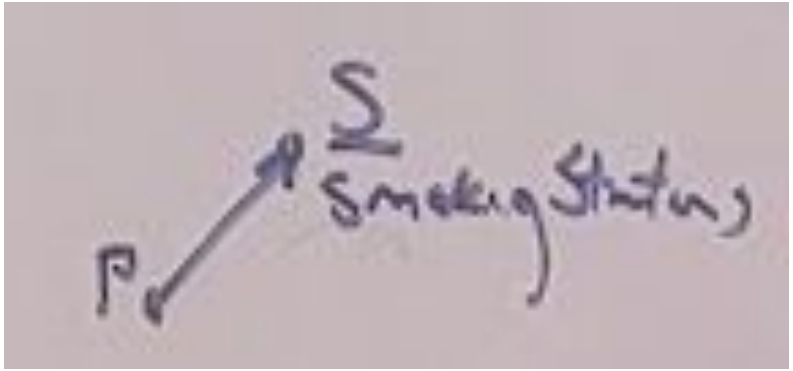
Selectively Monic Morphisms L -> G



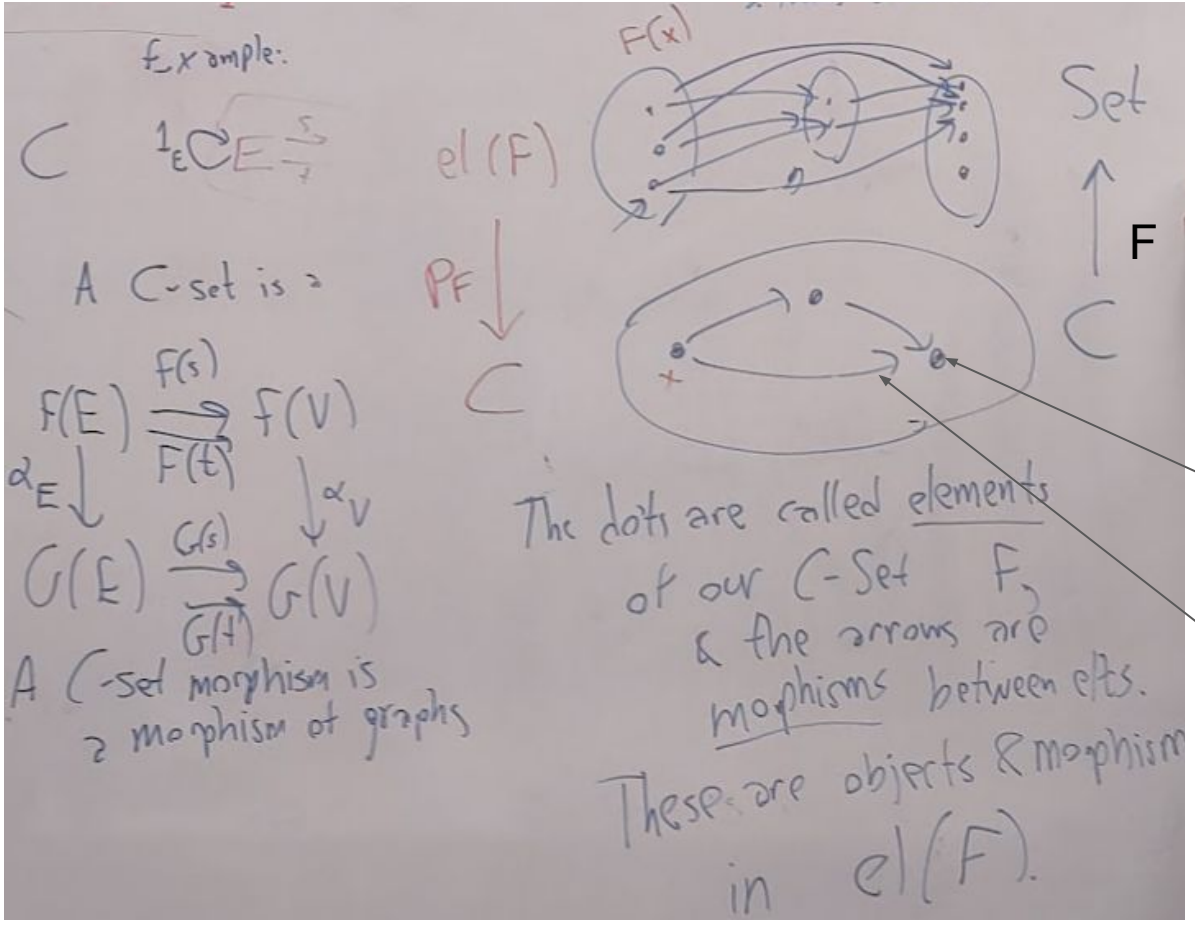
Commonly Rule Application Yields Multiple Matches



Simple Interpretation of Diagrams



The **Category of Elements** el(F) for a C-Set F: $\mathscr{C} \to$ Set

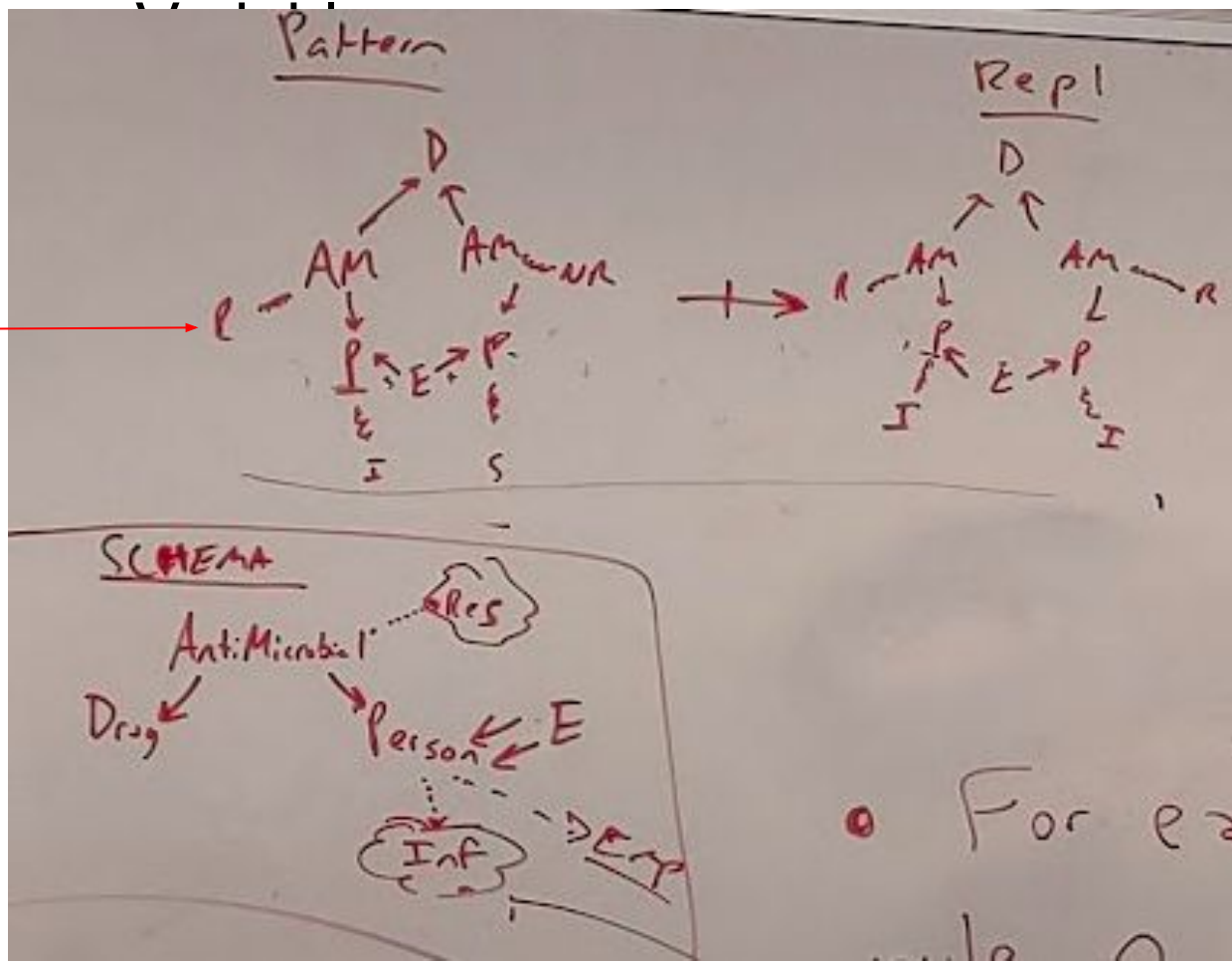


Object

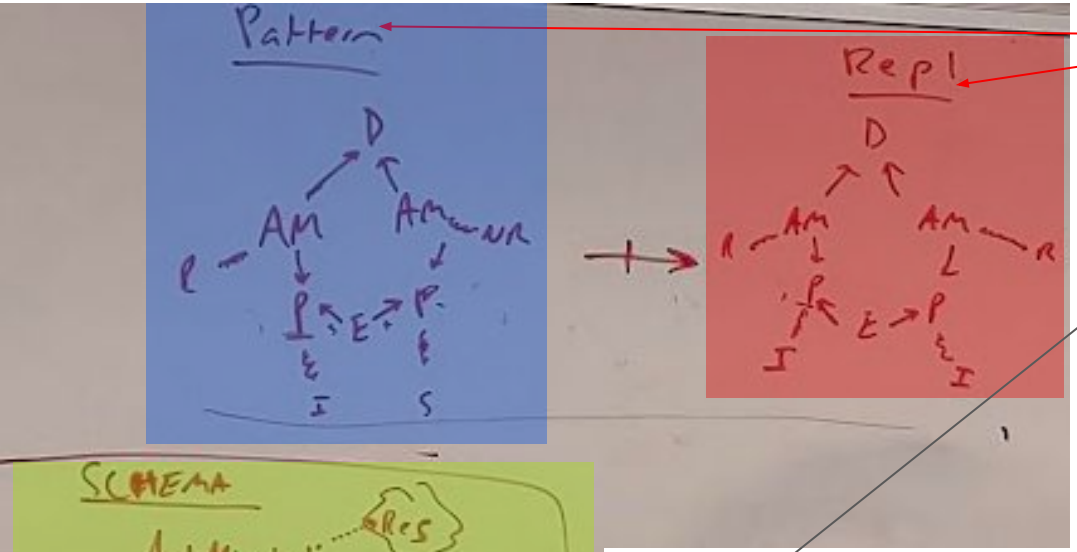
Morphism

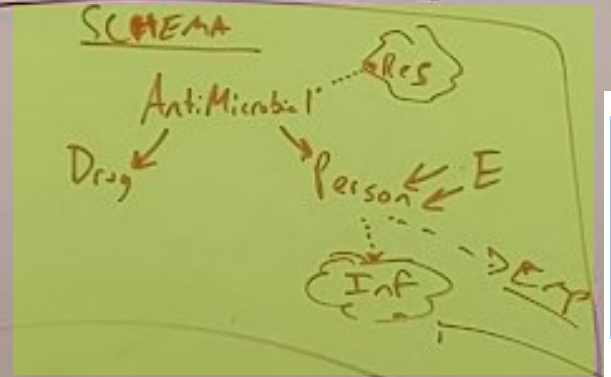
Patterns for Matching As Drawn from Category of Elements &

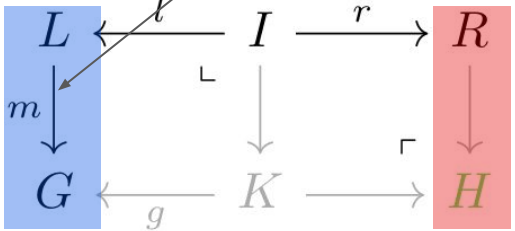
Note because these are C-Sets, the **minimum that we can represent in this pattern is a representable.**
In general, this is a **colimit of representables.**



We Draw our Pattern & Replacement in the Category of Elements for C-Sets in the Schema Category



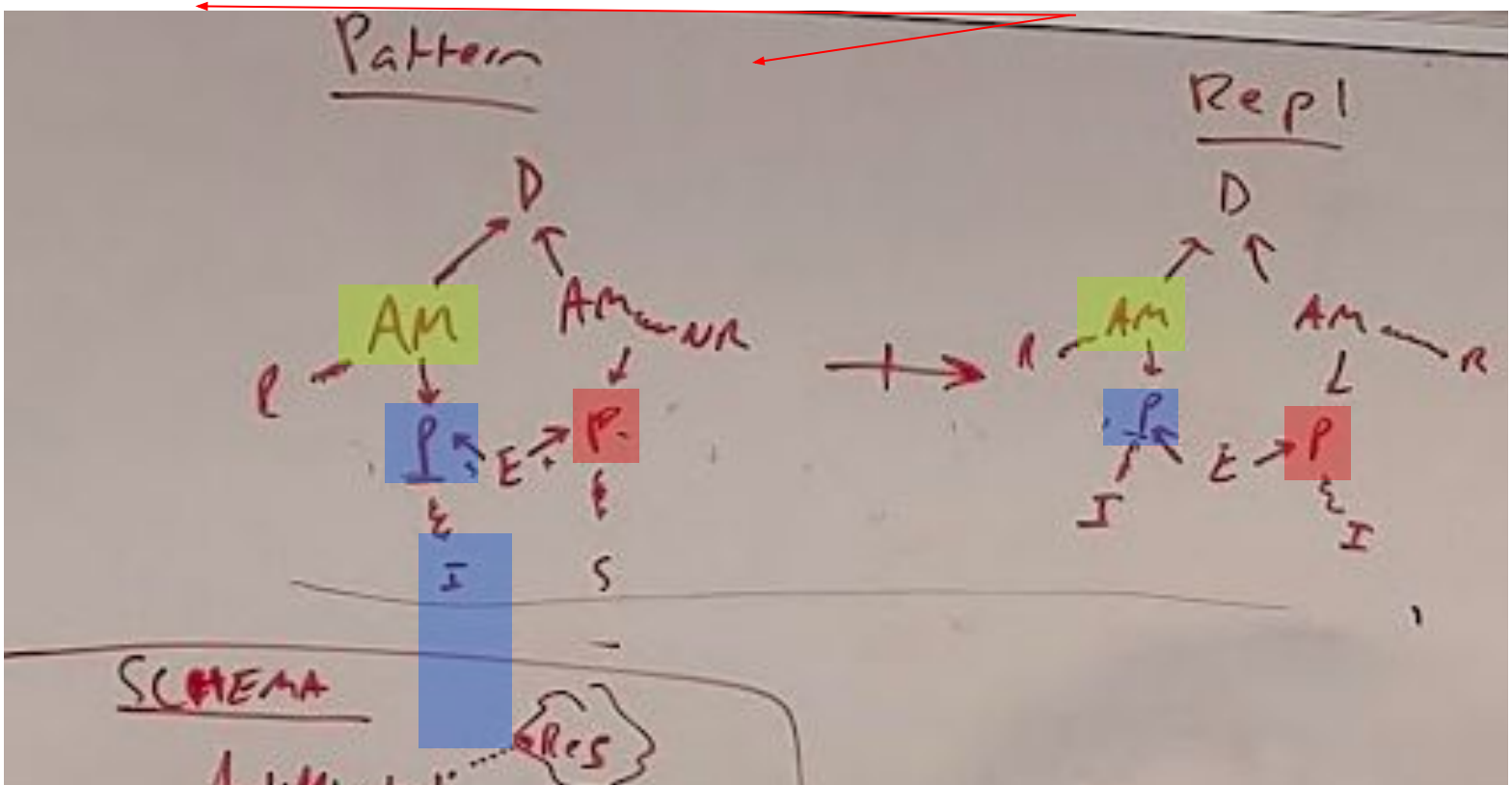




These are drawn in the category of elements for the schema; matching rewriting rules then searches for **homomorphisms** (structure preserving mappings) from such a C-Set into the state of the model (also encoded as a C-Set).

In practice, we can also use variables to capture the value of attributes in the match pattern, and use such values in formulating the replacement.

We Draw our Pattern & Replacement in the Category of Elements for C-Sets in the Schema Category



Interpretation?

Schema category $\mathscr{C}$

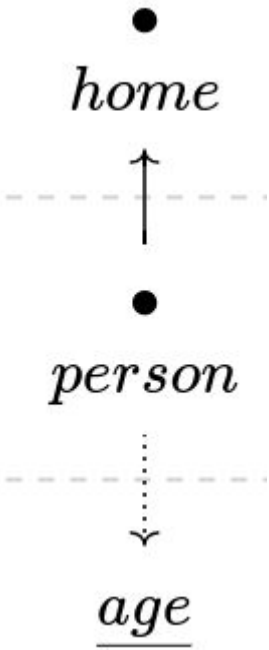
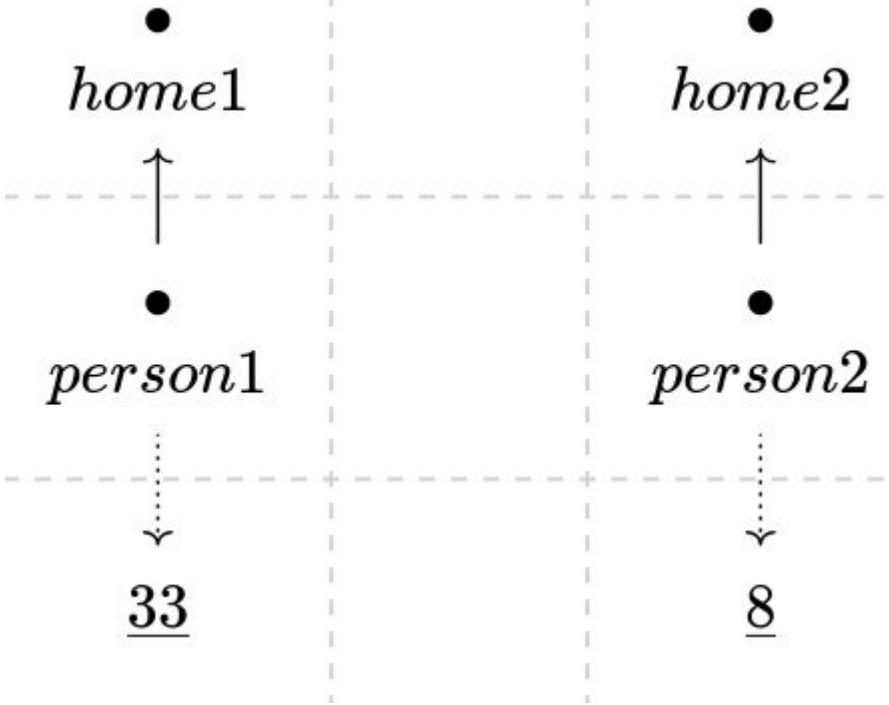


Diagram in category of elements



Interpretation?

Schema category $\mathscr{C}$

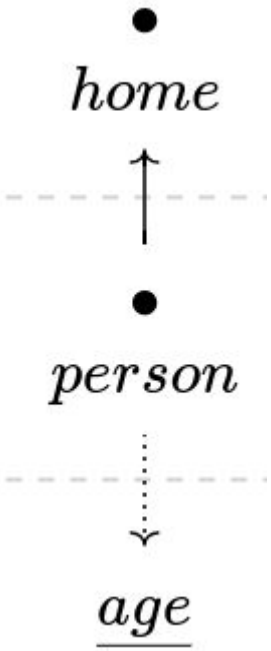
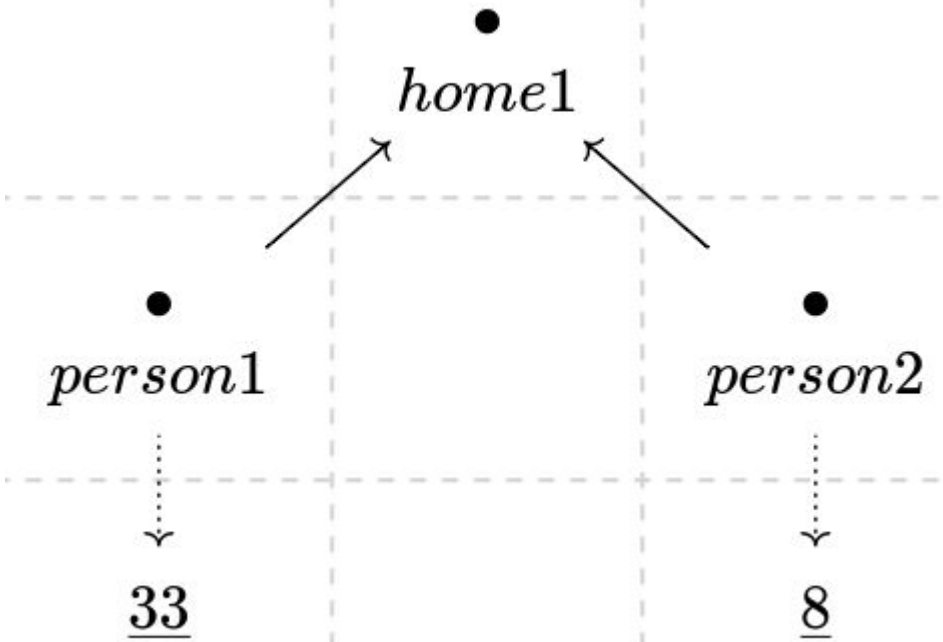


Diagram in category of elements



Interpretation?

Schema category $\mathscr{C}$

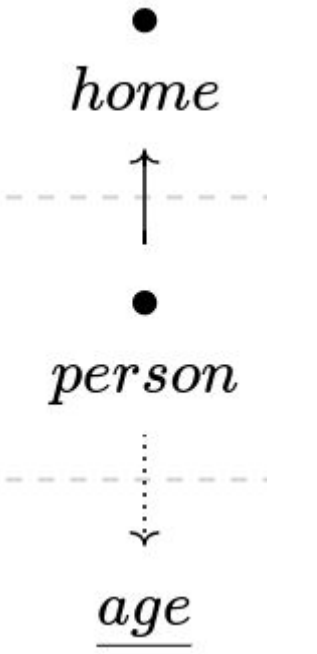
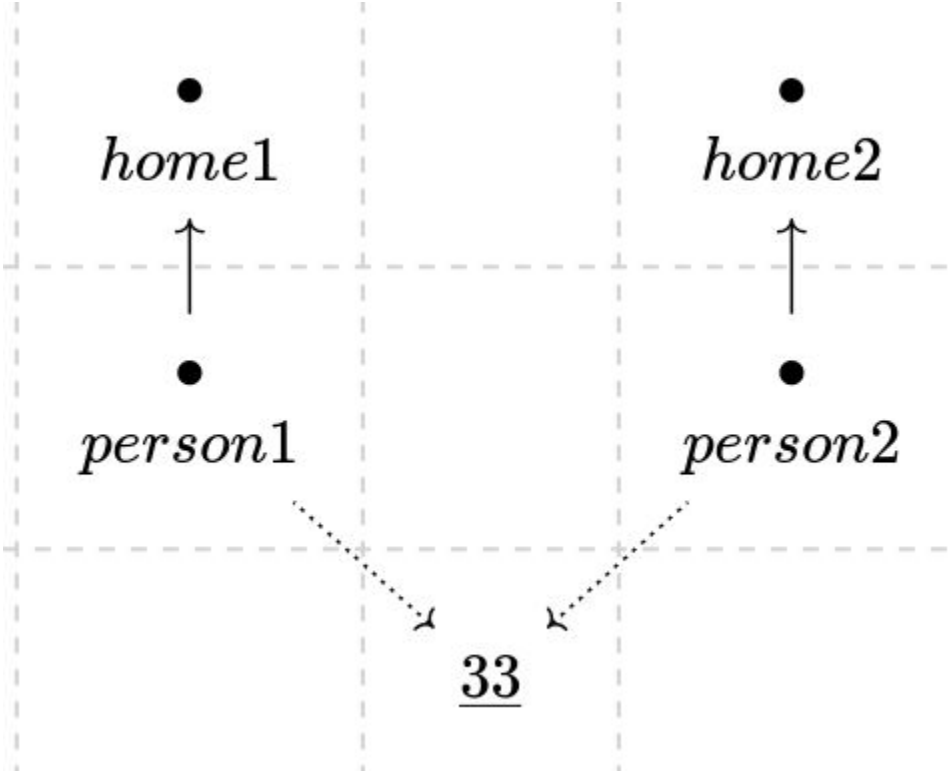


Diagram in category of elements



Interpretation?

Schema category $\mathscr{C}$

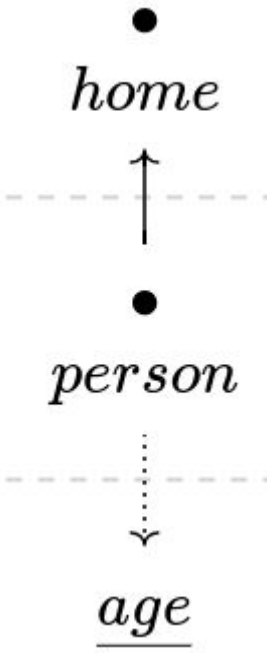
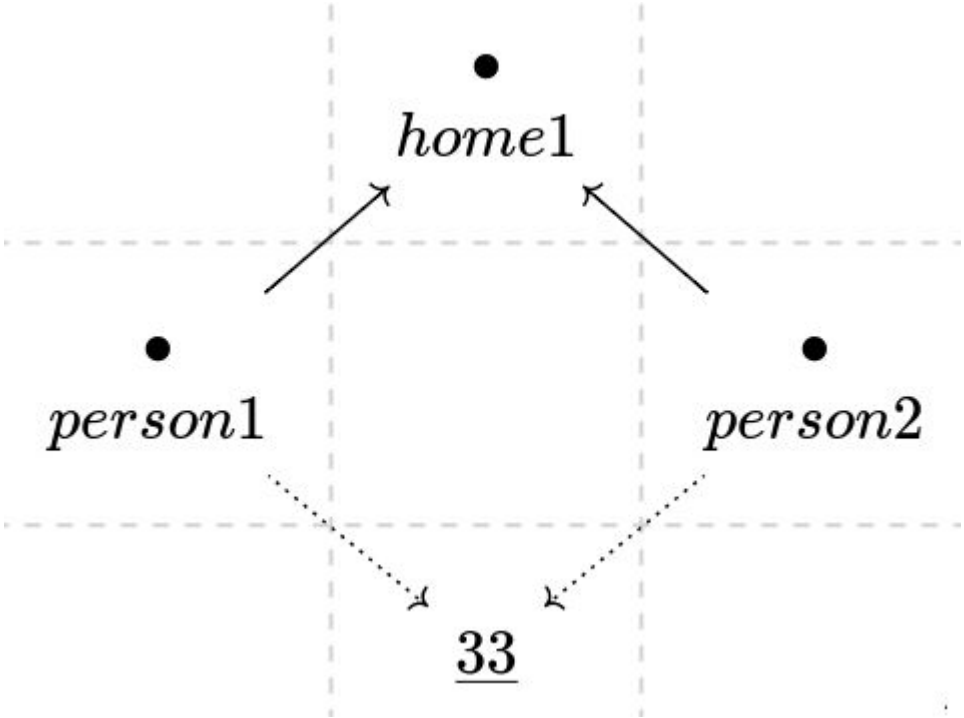
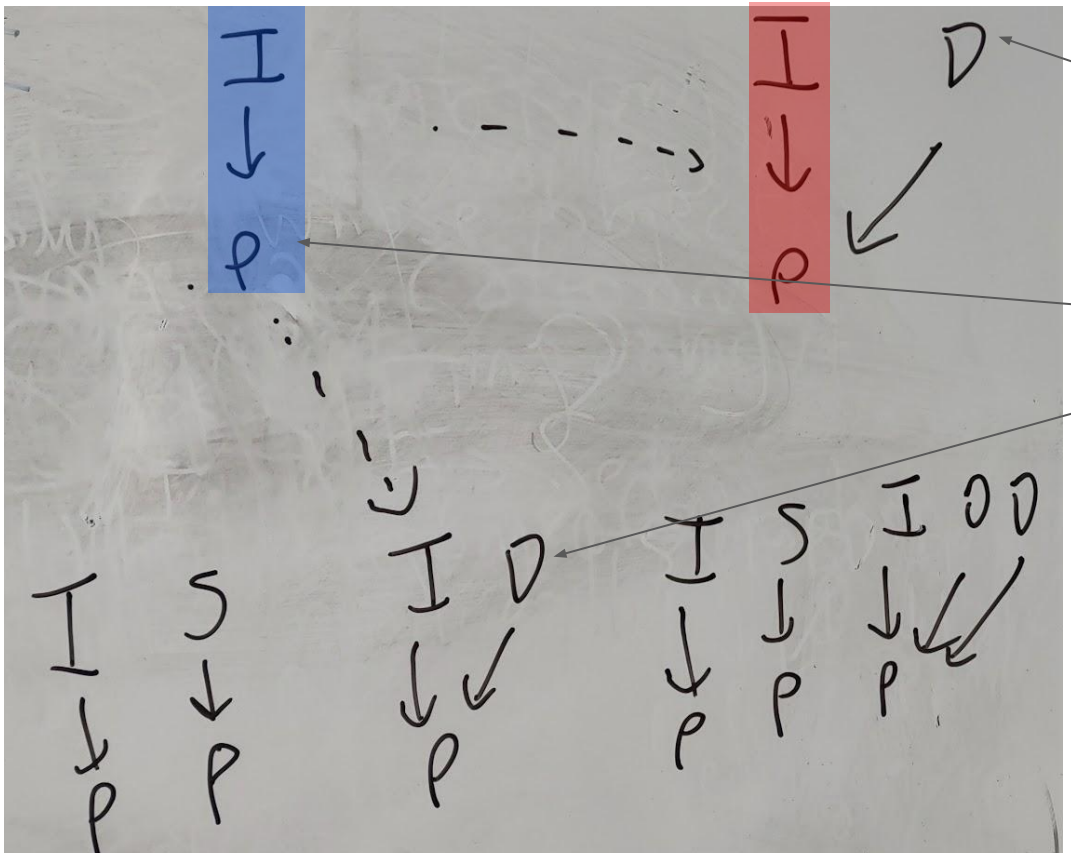


Diagram in category of elements



Understanding Pushouts of Patterns -- Take Pushout of the Image of the Match and State X_t Fibered by the Match



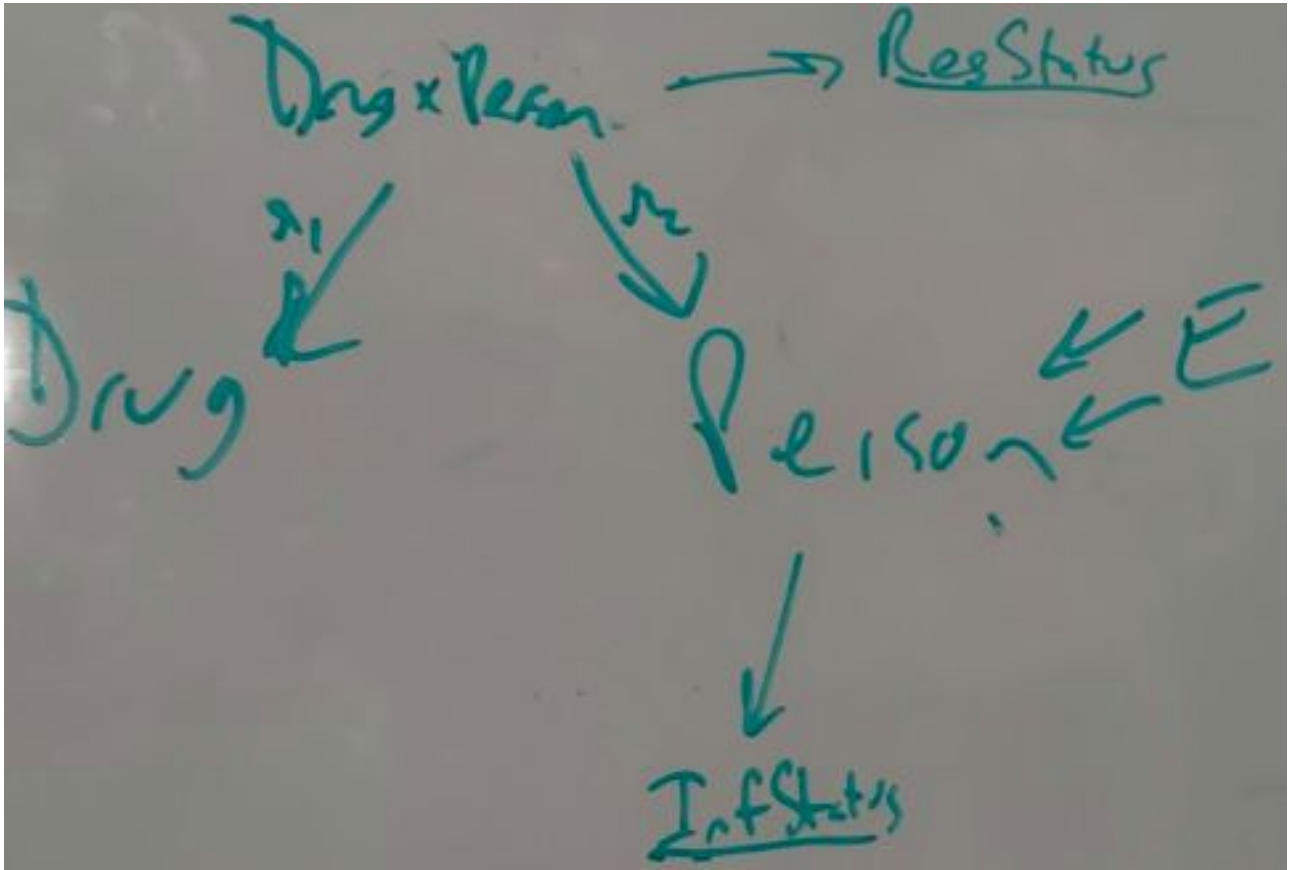
Even though have D in pattern B, and in X_n, because these are not in the homomorphism match I -> P , get an addition of two dogs

Patterns Can Use *Attribute Variables* to Bind & Propagate Values

Categorical Alternatives Offered

Traditional Construct	**Our Construct**
Messages	Context specific pattern matching
Agent as updated entity	Situations as matched
Nesting of agents	Relations between agents in patterns

Schema Capturing Drug-Specific Resistance for Person



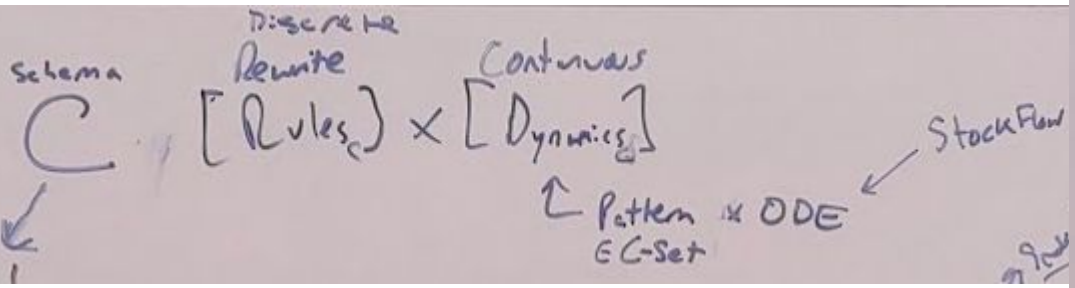
Recall: Use of ACSet_colim Eases Declarative Specification of Patterns

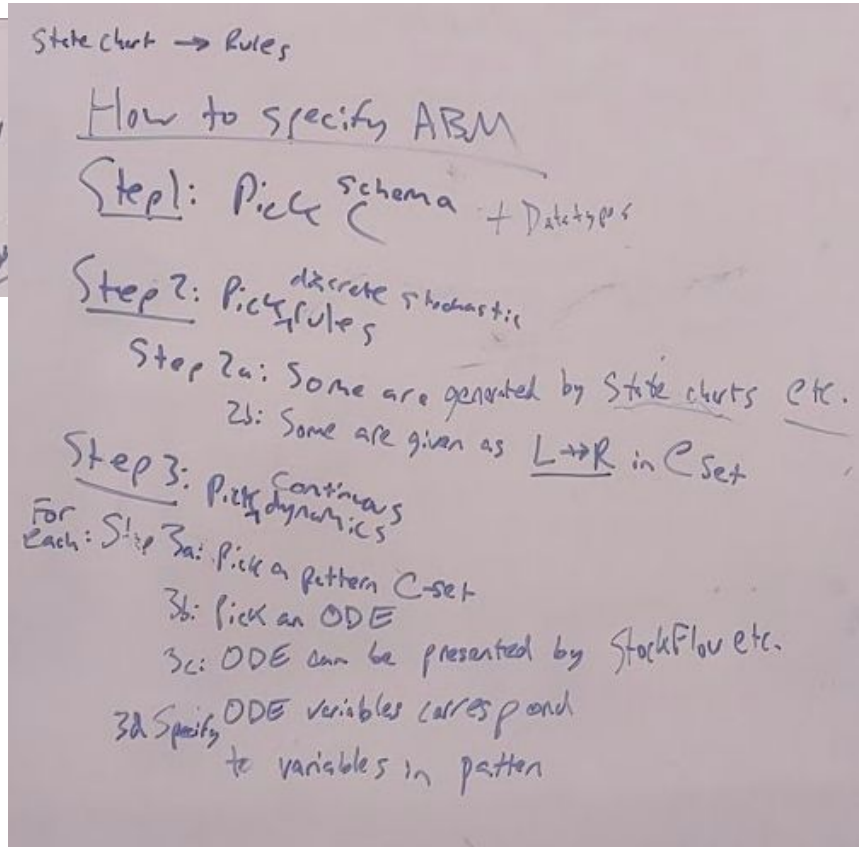
```
s_eat_pac = @acset_colim yLV begin s::Sheep; countdown(sheep_loc(s)) == 0 end;
```

```
w_eat_l = @acset_colim yLV begin
  s::Sheep; w::Wolf
  sheep_loc(s) == wolf_loc(w)
end;
```

Recall: Model Constituents

Model Characterization



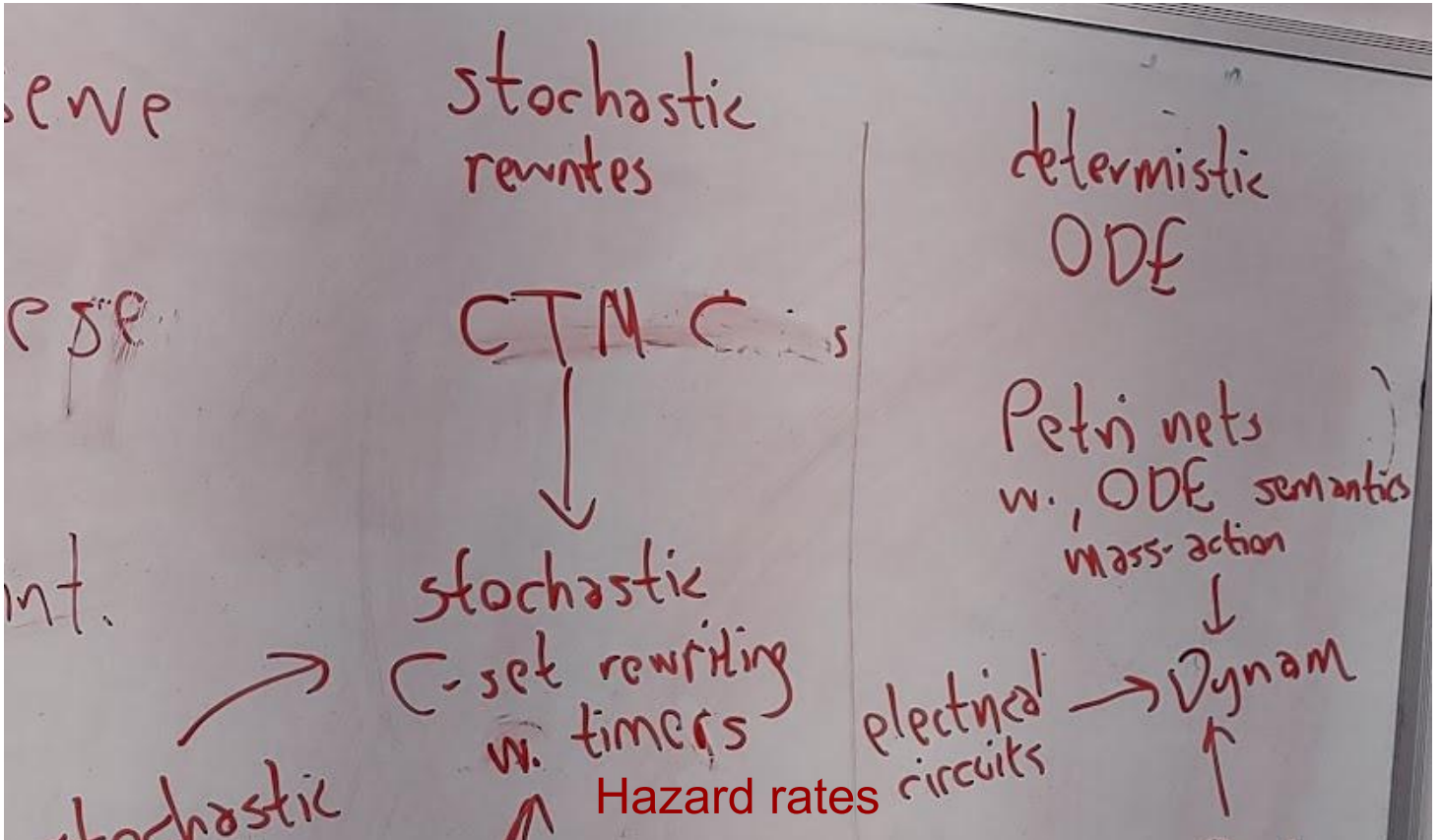


Current Code (in ABMs.jl in AlgebraicABMs.jl)

```
An agent-based model.
"""
@struct_hash_equal struct ABM
  rules::Vector{ABMRule}
  dyn::Vector{ABMFlow}
  ABM(rules, dyn=[]) = new(rules, dyn)
end
```

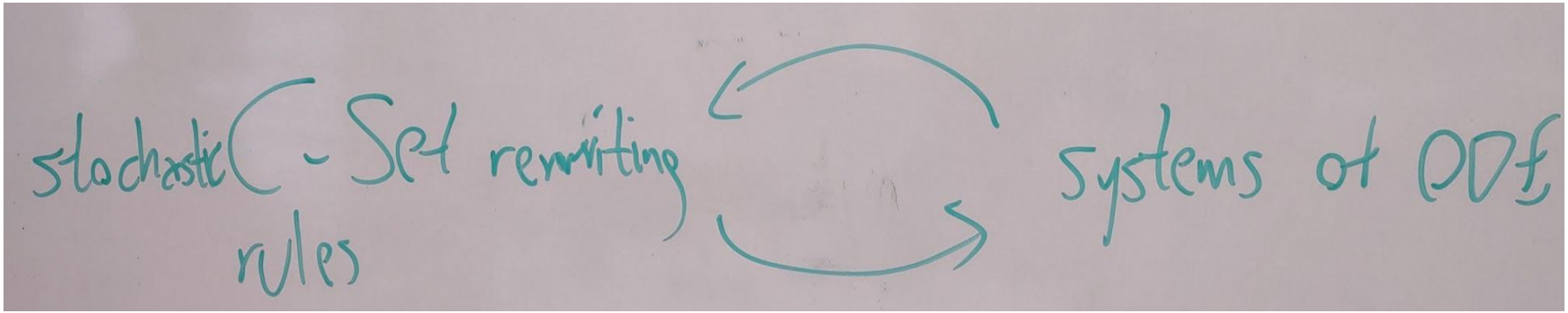
Basic Types of Dynamics

A Pluralistic Framework



Two Types of Dynamics

- **Continuous over time**: Ongoing updates; Underlying framework: ODEs, e.g., through direct ODE specification, stock & flow, or Petri Net mass-action semantics
- **Discrete events**: Events occur at an instant, undertake atomic update to new state.



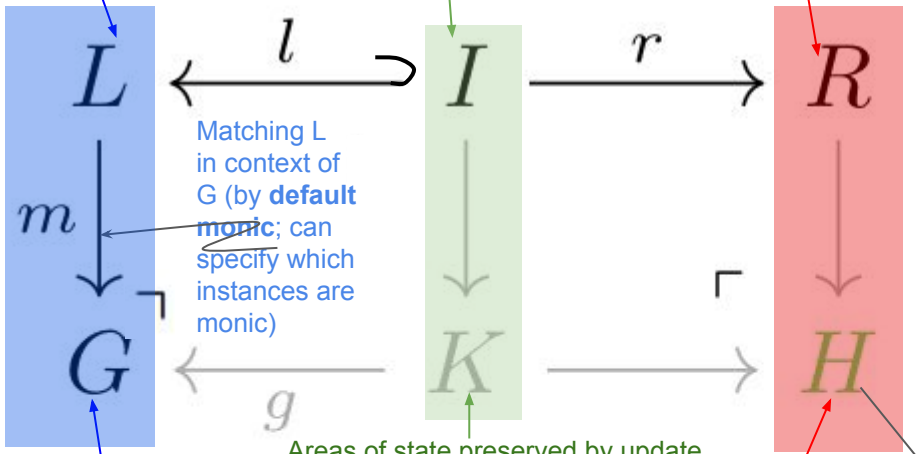
Rewriting Rules Triggered via Hazard Rates

Basic Rewriting Context

Match (everything here but not in I is deleted)
For attributes, *variables* can be included, so as to capture the value of that attribute & use in R

Features in L retained in R

Replacement (everything here but not in I is added)



Matching L in context of G (by **default monic**; can specify which instances are monic)

Areas of state preserved by update

All objects are C-Sets, all arrows are C-Set homomorphisms

xmap

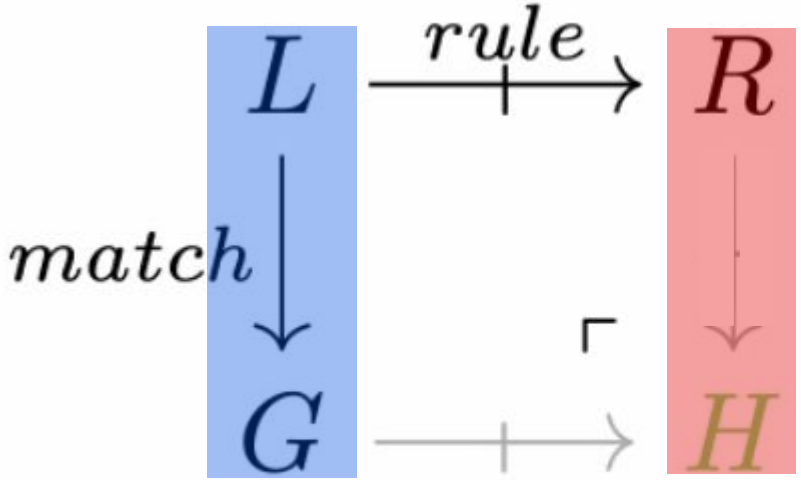
H'

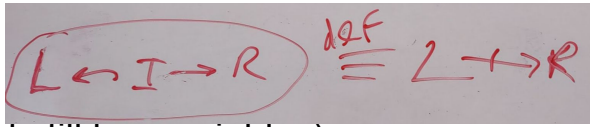
Original state (X_t)

Updated state (X_{t+1}; incorporates R but still has variables)

Updated state (X_{t+1}'; incorporates R but and all variables replaced w/values)

Considered as Partial Maps

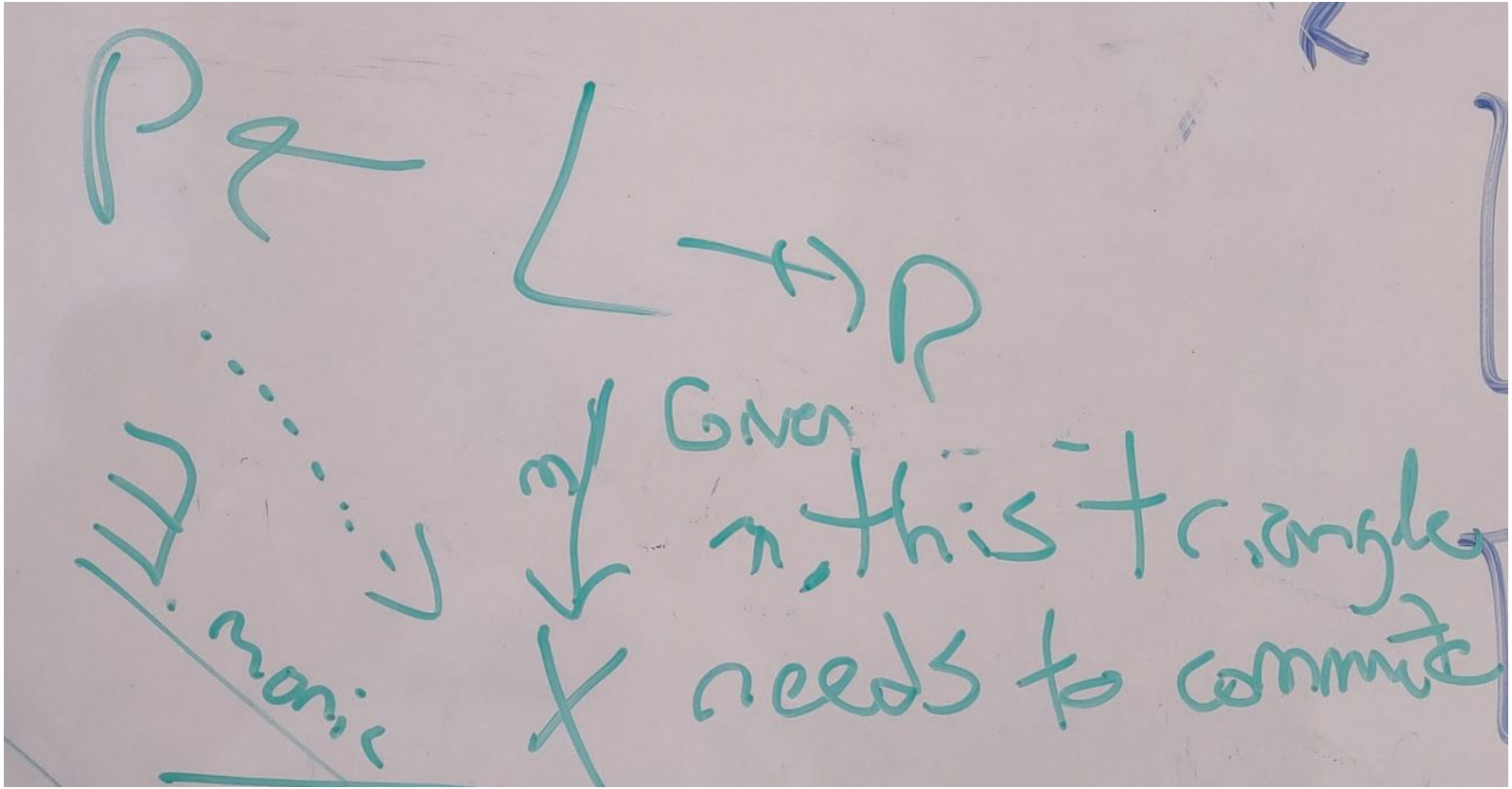




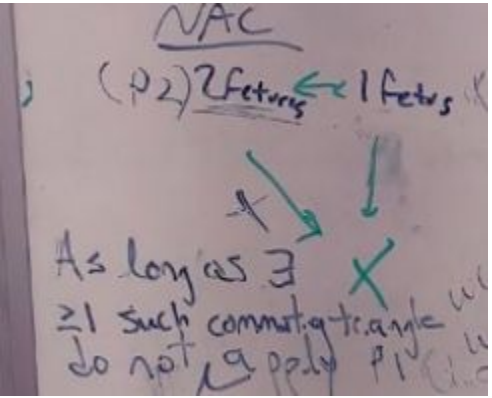
Images adapted from Brown, K., Patterson, E., Hanks, T. and Fairbanks, J., 2022. Computational Category-Theoretic Rewriting. In International Conference on Graph Transformation (pp. 155-172). Springer, Cham. https://arxiv.org/pdf/2111.03784.pdf

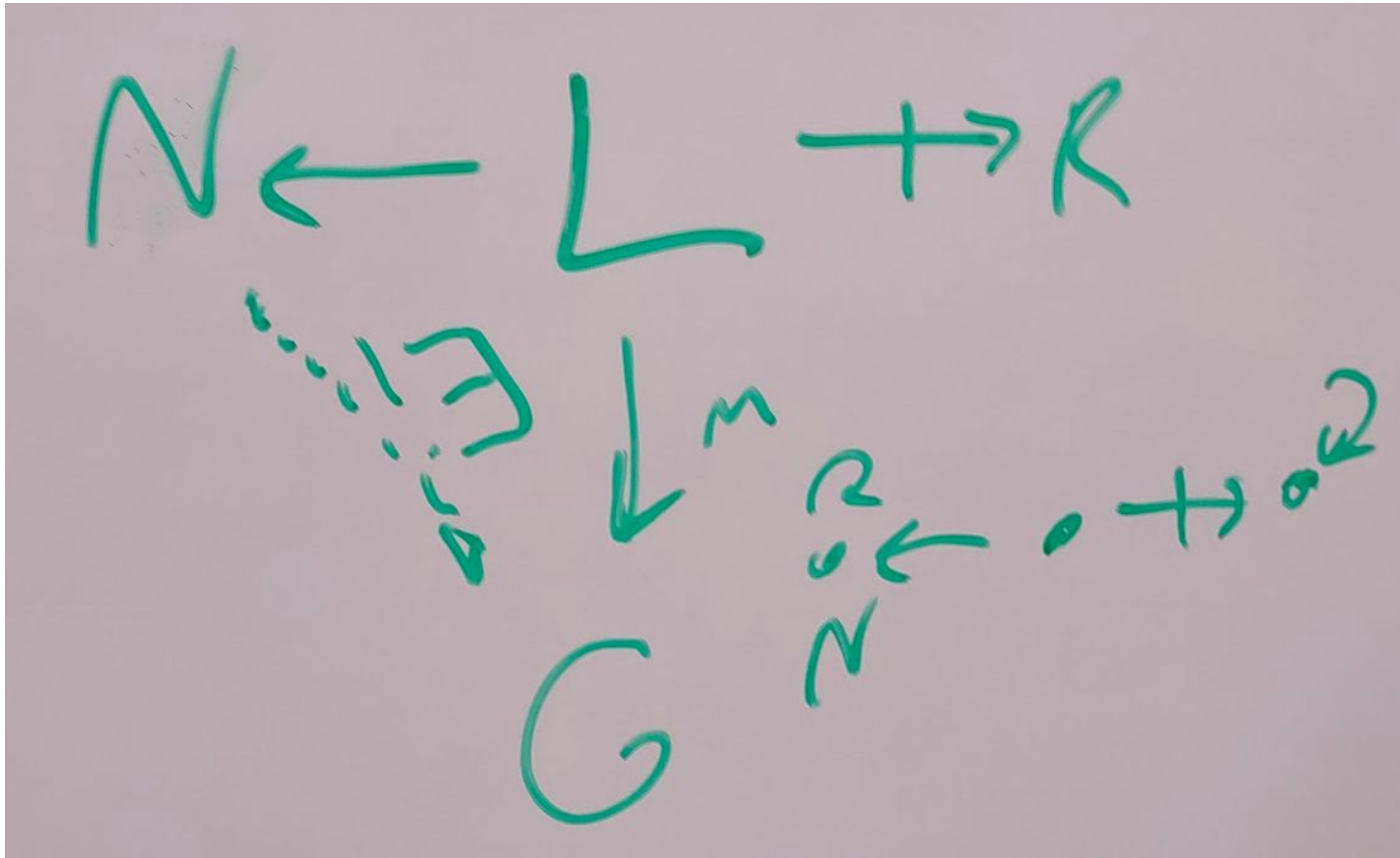
Application Conditions

Positive Application Conditions

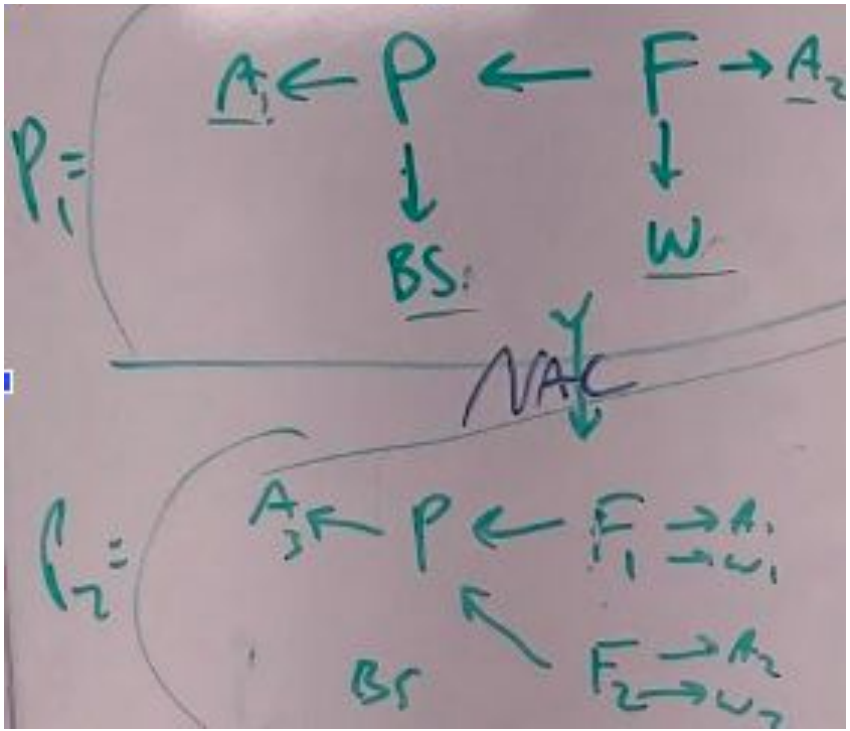


Negative Application Conditions

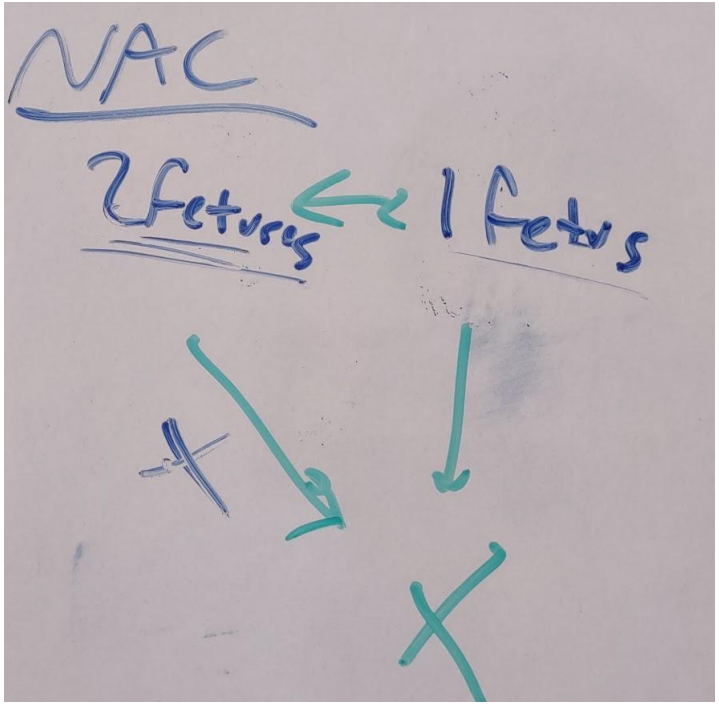




Example Need for Negative Application Condition -- P1 would normally apply if P2 matches, but NAC prevents this



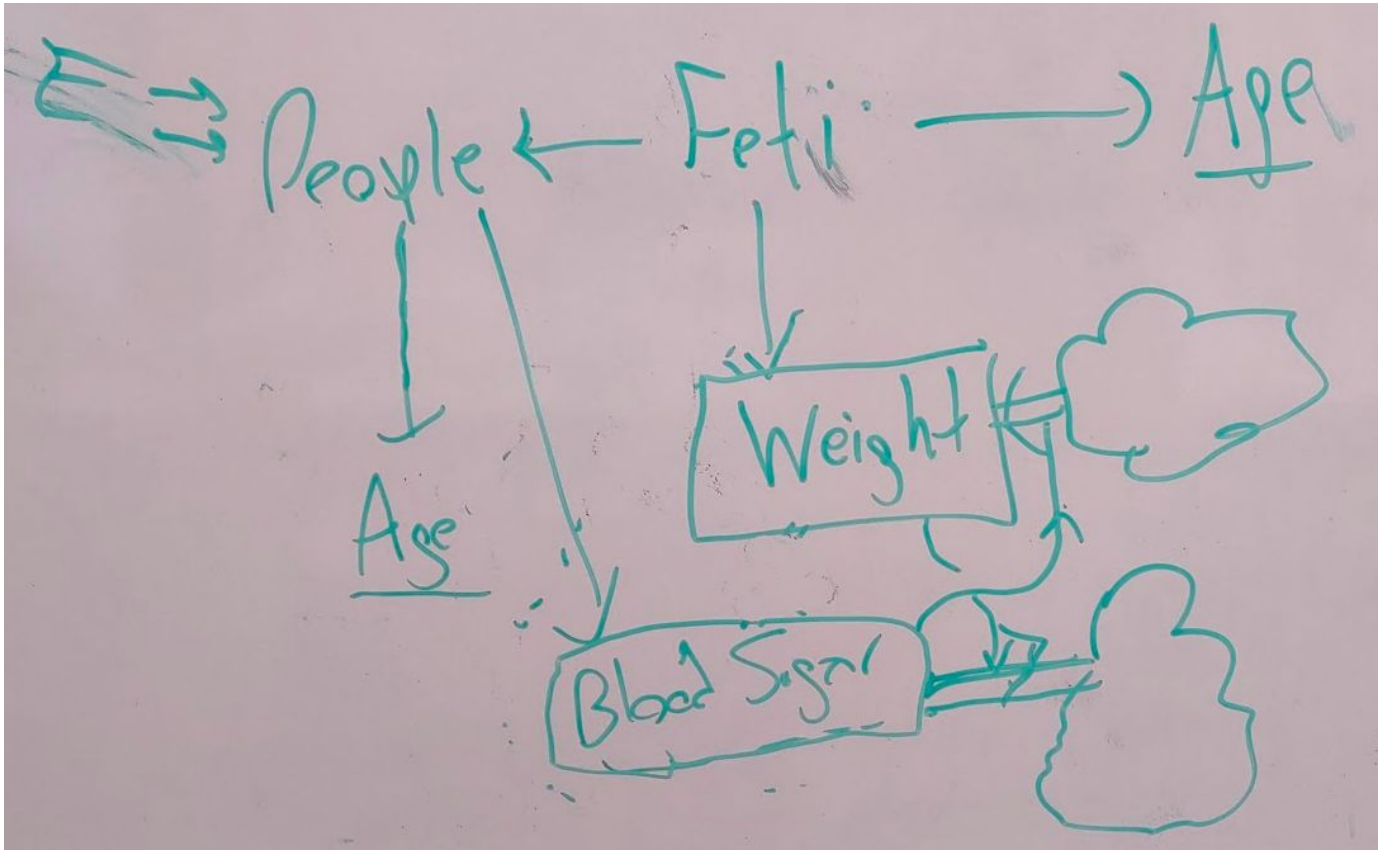
Example of Negative Application Condition



Underlying Rewriting Mechanisms -- Match & Schedule

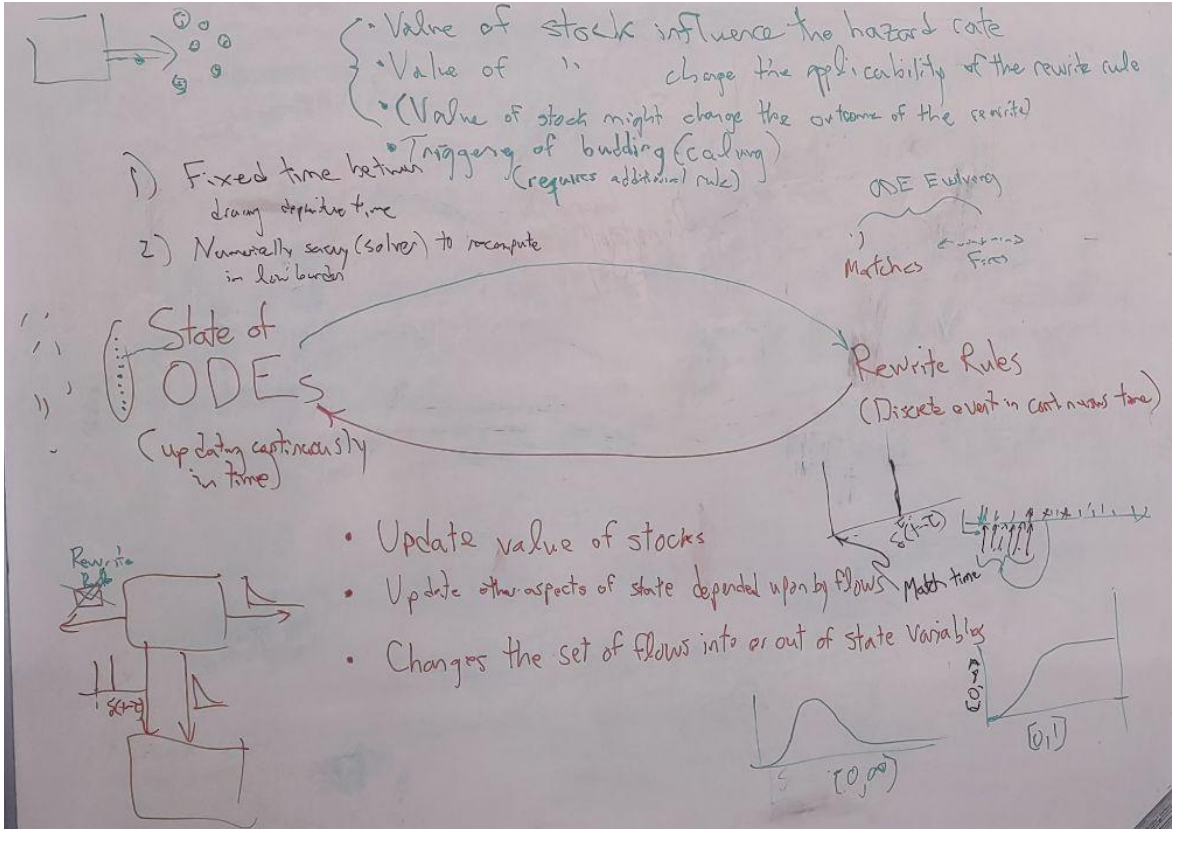
- Especially because there is no notion of “each agent” needing updates, we need to **match** pattern to find situations that can trigger dynamics via matching (e.g., that an agent is in a given state)
- These situations do not generally need to fire **immediately** -- for example, if an agent has entered a state; it could exhibits a hazard rate or timeout transition for leaving
- Using hazard rates (not PDF) lets us adjust firing rate as state evolves

Stock & Flow: Logic for Mother & Fetus



Interactions Between Types of Dynamics

Types of Interactions Between Rewriting & ODE Dynamics



C-Set rewrite impacts on evolution of ODE state variable X

- Directly updating the value of X
- Changing the value of attributes representing quantities depended upon by such flow functions (differentials) associated with X, so as to alter the value one or more flows compared to the value that it otherwise would have had; this could include e.g., other people who are neighbours of
- Changing (indirectly or directly) the value of other state variables depended upon by the flow functions (differentials) associated with X, so as to alter the value of one or more of the flows of compared to the value that they otherwise would have had
- Updating the collection of flows into or out of X

Pathways of ODEs Impact on C-Set Rewriting

- Affecting whether the pattern matches
- Affecting hazard rate
- Budding agents

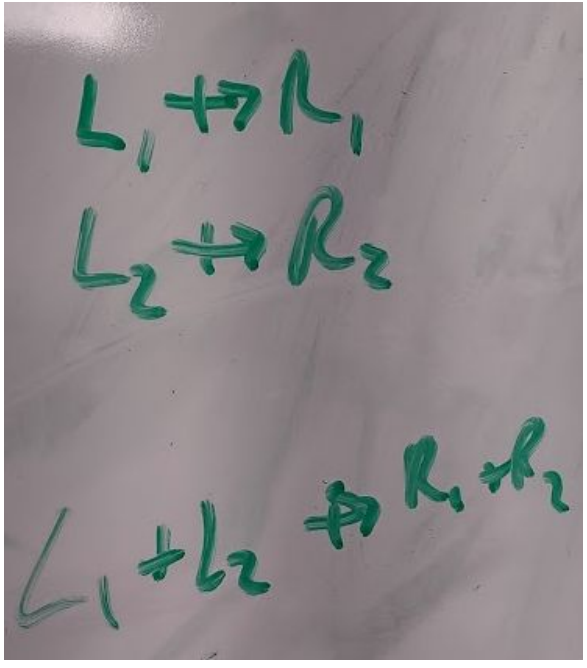
Handling Contemporaneous Events

Different Notions of “Clashing” Rules for Discrete-Time ABMs

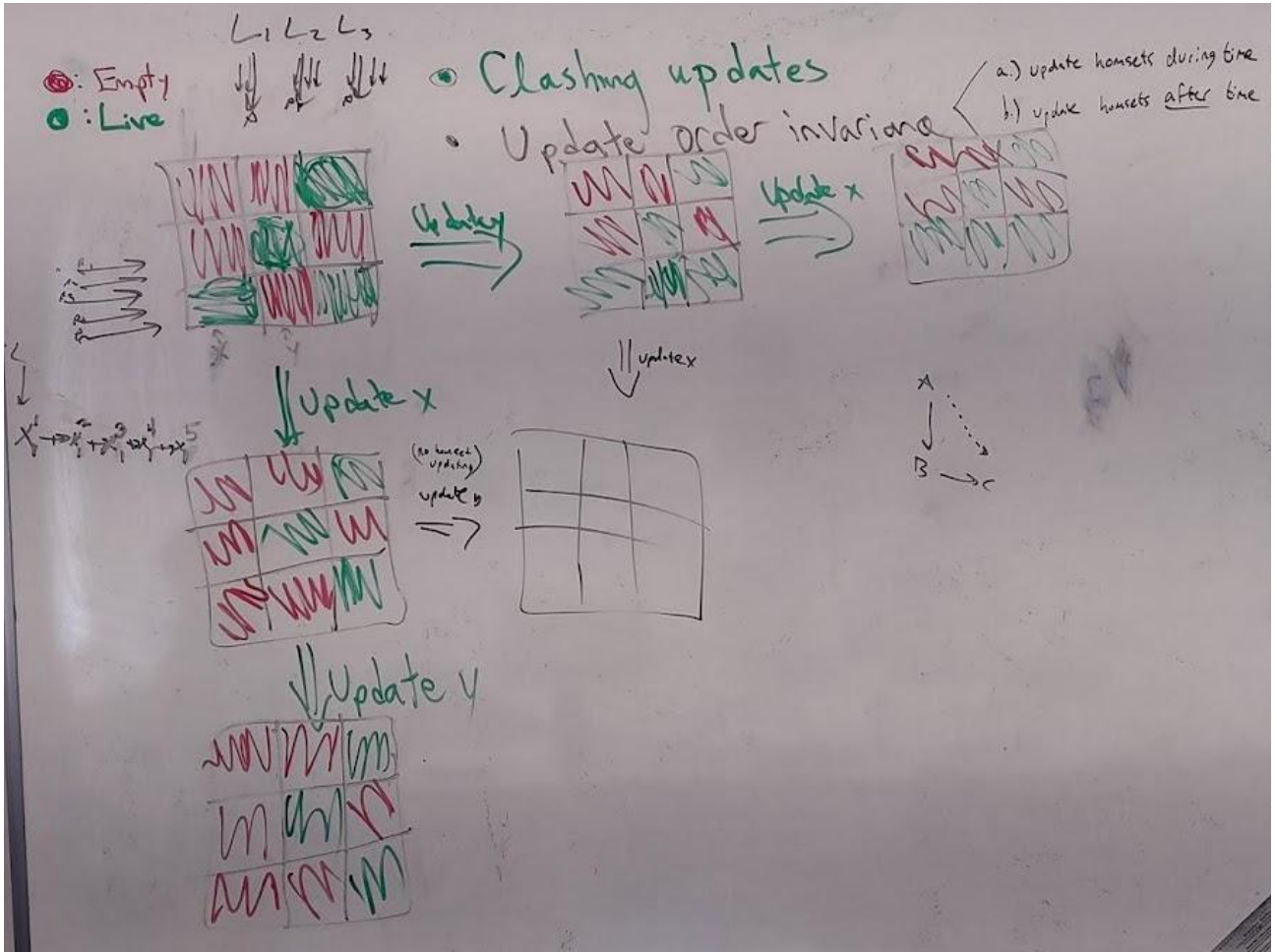
- Update order invariance for events that occur at the same time
  - Our framework achieves this because the homsets are updated after the time, allowing the updates to occur in lockstop
  - Note that the Game of Life is not itself naturally order invariant
- Clashing updates -- the rules issue clashing update requests for the same state (e.g., the same cell)
  - This is a more problematic issue, than can be **detected** by this framework with an error message given to the user, but ultimately needs to be handled by the user

Contemporaneous Events Adhere to Lockstep Dynamics

- The formulated dynamics achieve capturing lockstep discrete time updates occurring at same time (e.g., for Game of Life), contingent on no clashes occuring



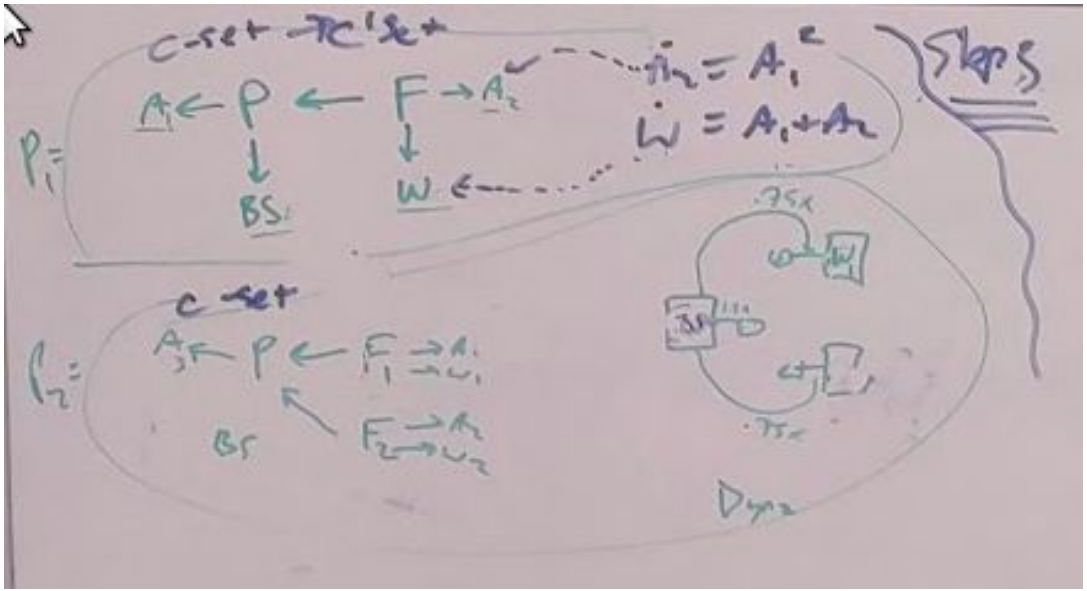
Invariance of Simultaneously Updating Rules



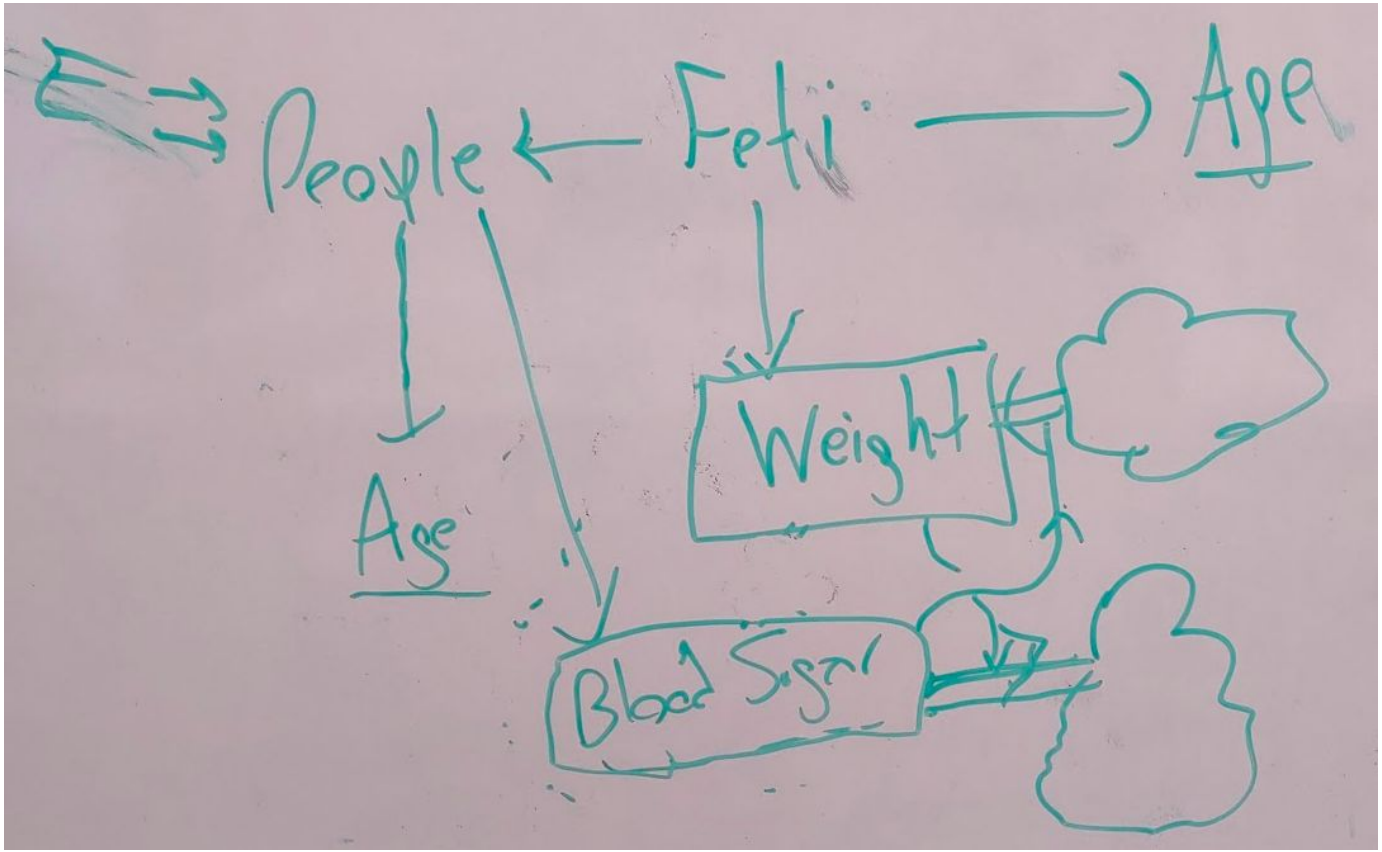
Pattern Matches Govern When & Where Continuous Dynamics is Placed within Running ABM

Matches Weave Governing Process Into Chosen Contexts

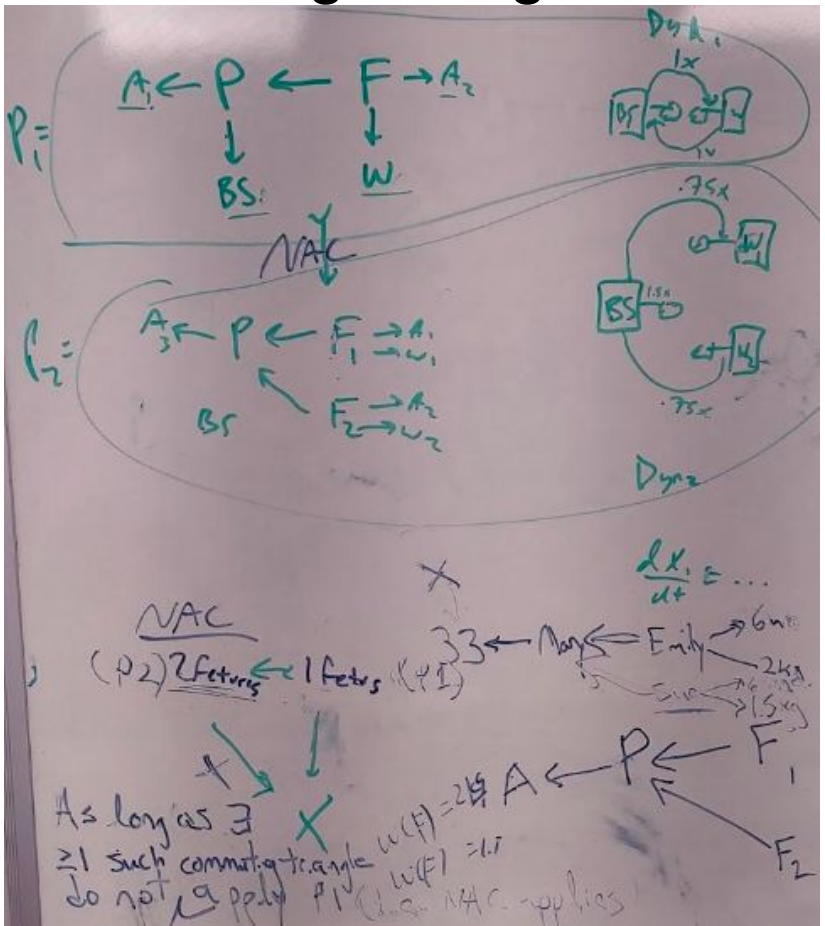
- We weave in dynamics at certain points based on where matches occur
- This can be done for either continuous or discrete event dynamics
- The mechanisms that govern this dynamics
- This allows for changing the governing dynamics as state evolves.



Another Example: Weaving Together Logic for Mother & Fetus



Application Conditions Support Added Refinement When Weaving in Logic



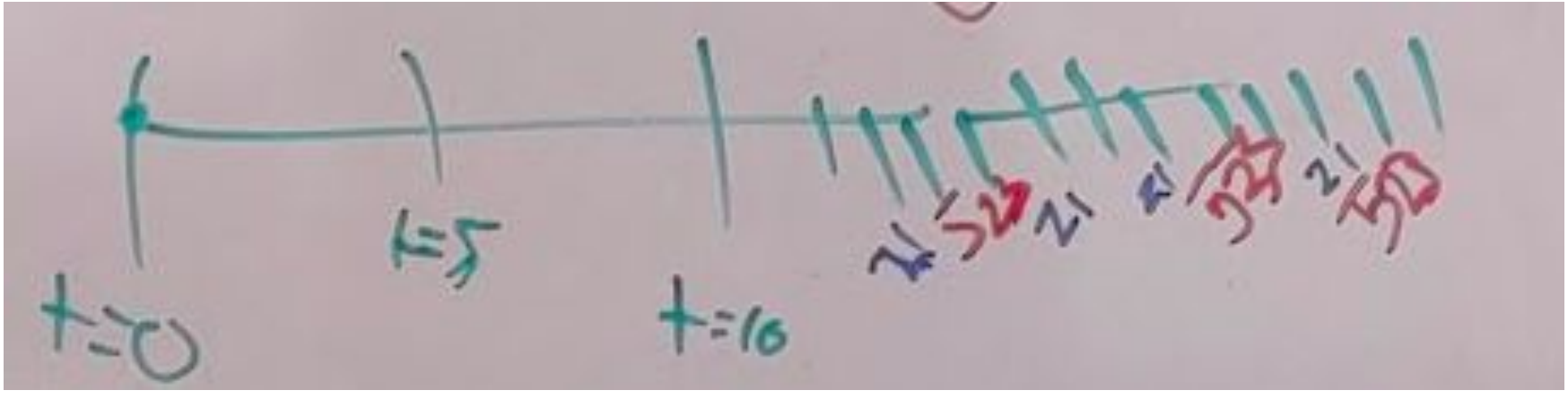
Pre-scheduled and Dynamically Scheduled Events

Underlying Rewriting Mechanisms -- Match & Schedule

- Especially because there is no notion of “each agent” needing updates, we need to **match** pattern to find situations that can trigger dynamics via matching (e.g., that an agent is in a given state)
- These situations do not generally need to fire immediately -- for example, if an agent has entered a state; it could exhibits a hazard rate or timeout transition for leaving
- Using hazard rates (not PDF) lets us adjust firing rate as state evolves
- [General case] For *state-dependent* events (hazard rates), we avoid **ongoing** need to reschedule by incrementally computing Bernoulli draws for the probability of firing, for a certain timestep
- For *state-independent* hazard, we **schedule** trigger, Check if match still applies when fires
- By default, if match still remains after rewriting, create new event in the schedule
    - A key exception -- when leaving state in a statechart (know that won’t match again)

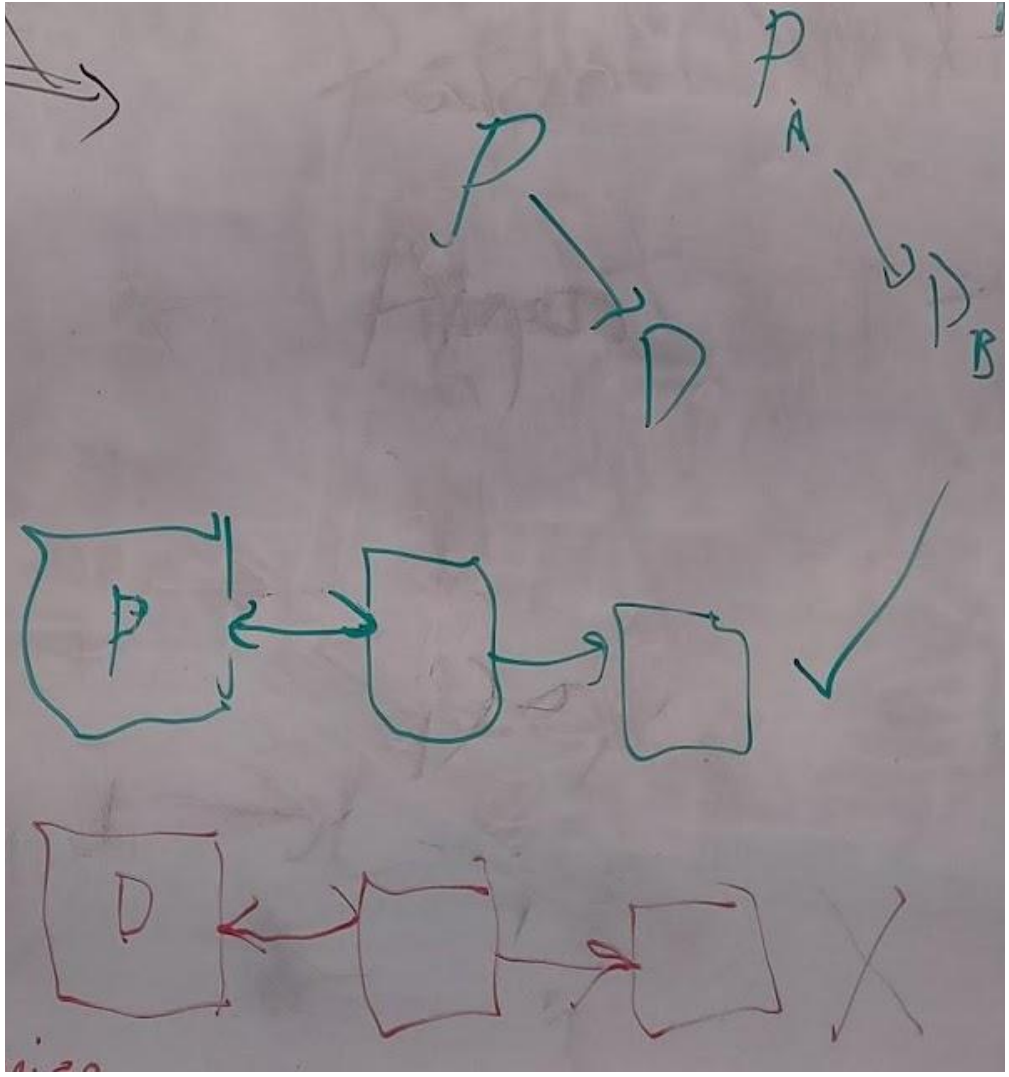
Hybrid (Static-Dynamic) Scheduling
The **Supporting** Role of an Event Schedule (Queue)

- **Model state independent timing:** Pre-scheduled (both memoryless & memoryful)
- **Timing dependent on Model state:** Events whose scheduling is state dependent, dynamically schedule (Bernoulli draws as to firing)
- Regardless of type, scheduled events can be descheduled if they are preempted eg
  - Agent dies
  - Agent leaves because of matching a different pattern

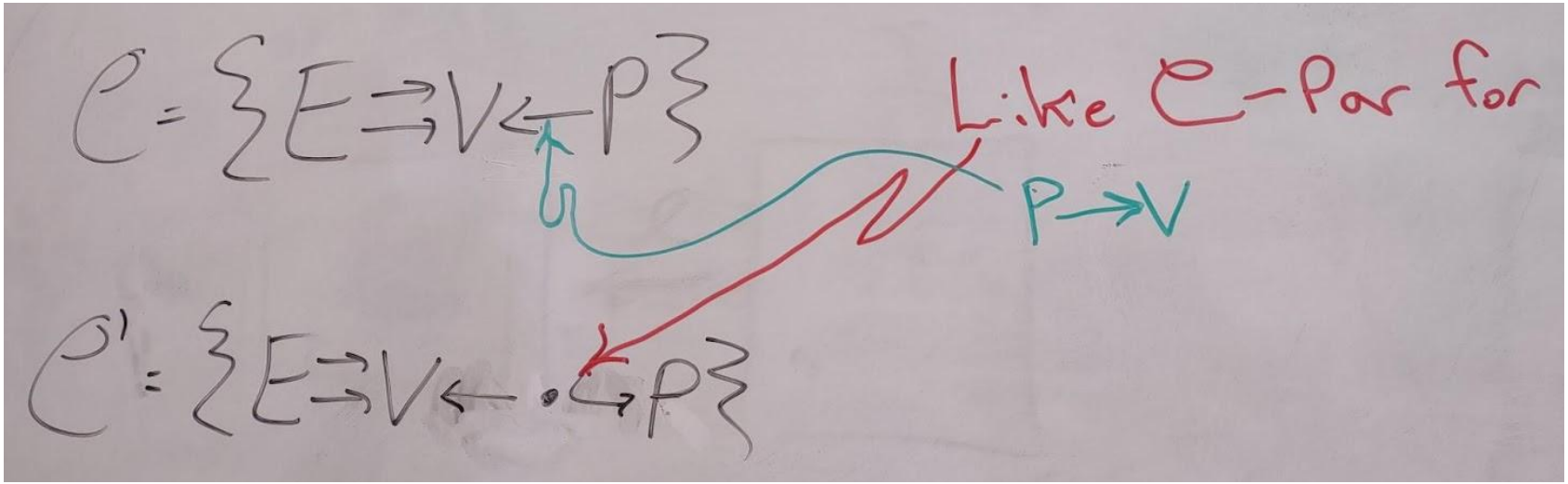


Problem: Naive Rewrite Rule Apices Could Violate CSet Rules

Consider: Referential integrity constraints on rewrites



Currently Planned Solution: Tweak Schema to Include Relation, rather than Direct Dependence so I can have no Link

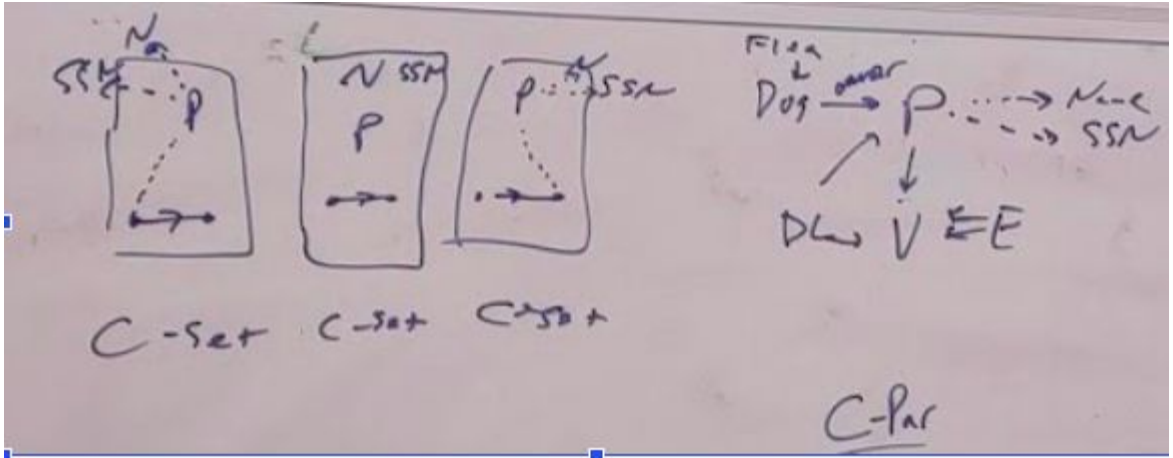


Possible Eventual Desire for Moving Rewriting to C-Par

We currently have a problem with rewriting needing to violate rules of valid CSet within conserved area I

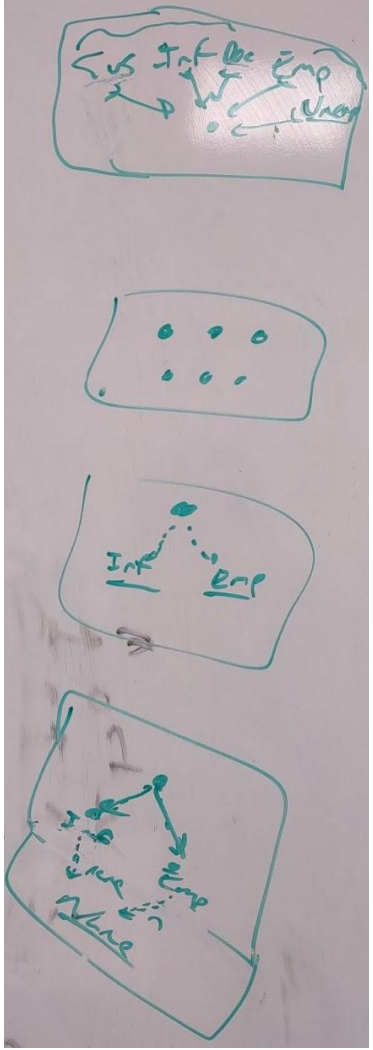
This problem crops up in each type of rewriting

Operating in C-Par allows for eliminating this problem



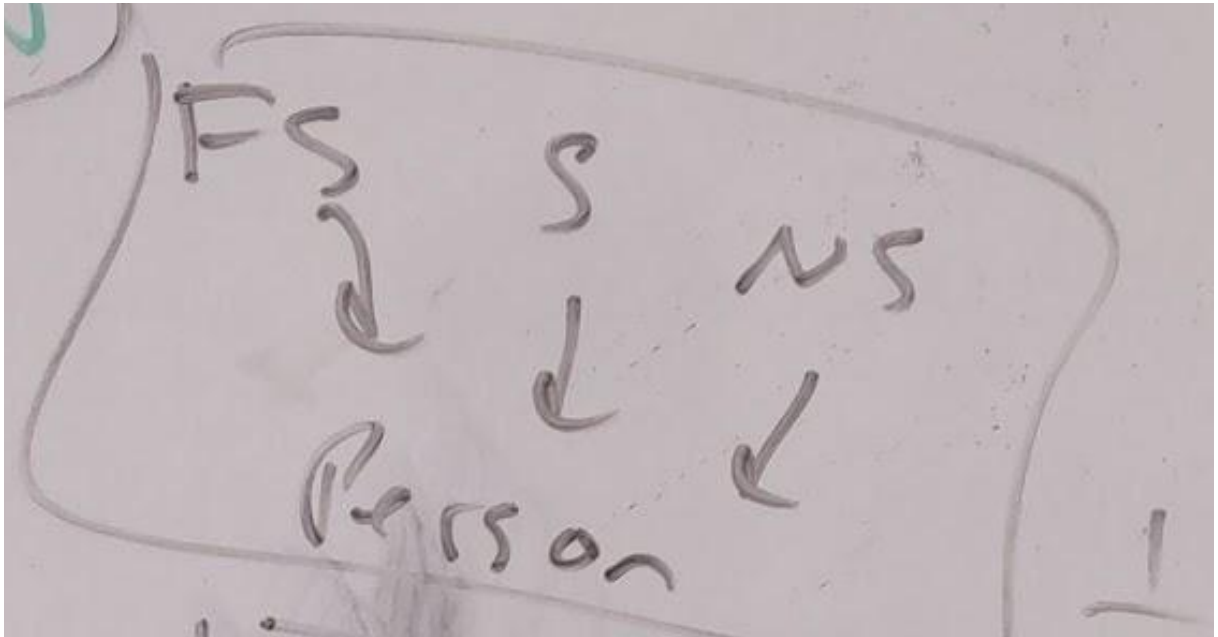
Dealing with Heterogeneity in Characteristics & State

4 Ways of Capturing Heterogeneity



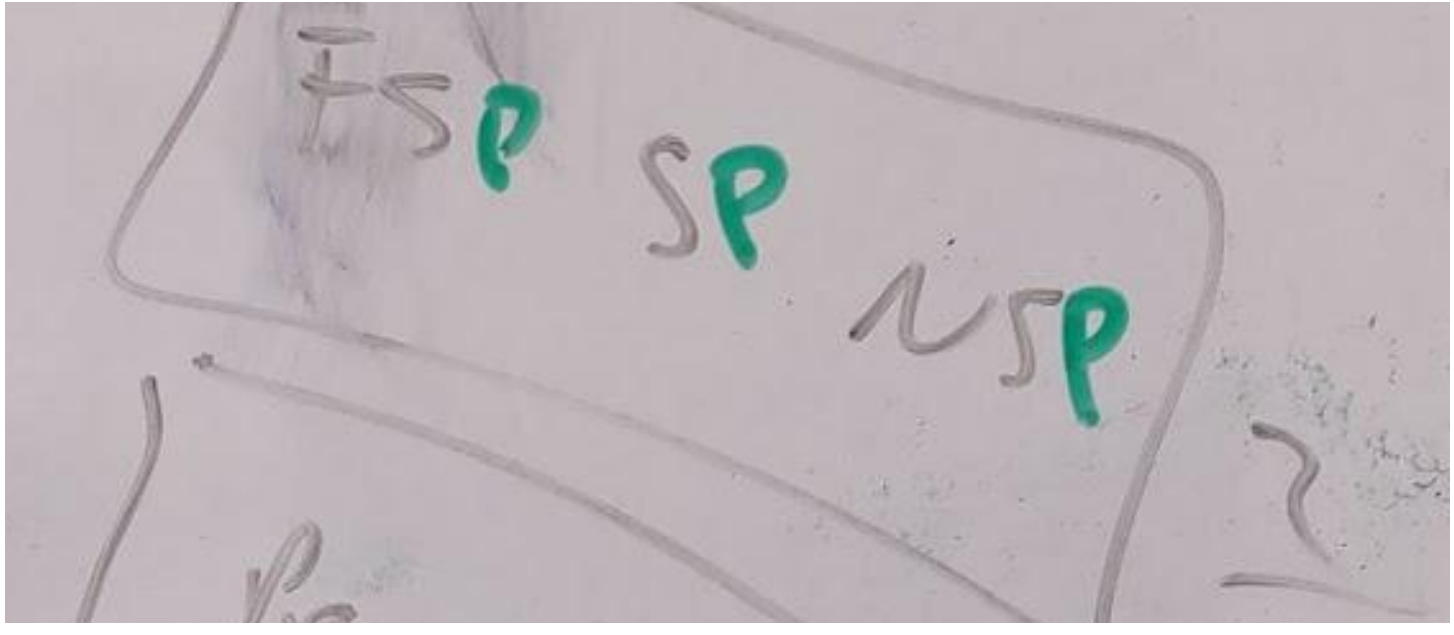
3 different ways of capturing nominal state information: 1

- Separate objects mapping into complete set
  - **Advantages**: Scales well with multiple dimensions of heterogeneuity
  - **Disadvantages**: does not make clear the mutually exclusive and collectively exhaustive nature of the update



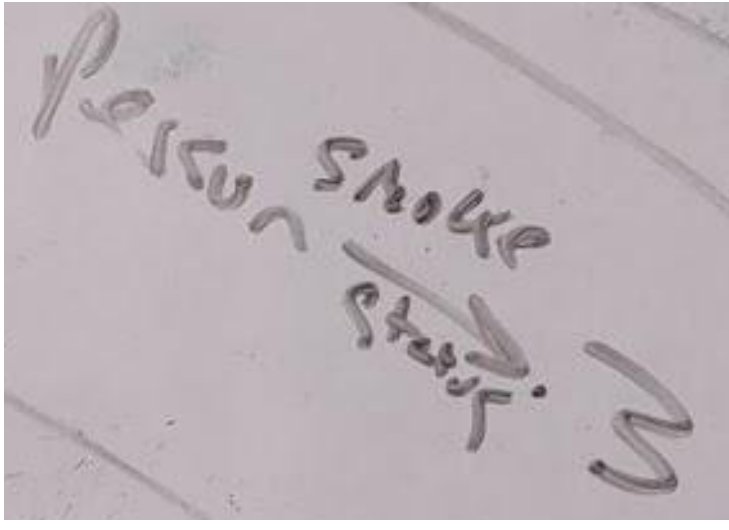
3 different ways of capturing nominal state information: 2

- Separate objects for each division of person
  - **Advantages:** Does make clear that covers all persons
  - **Disdvantages:** Suffers from Curse of Dimensionality with other heterogeneity dimensions



3 different ways of capturing nominal state information: 3

- Attributes
  - **Advantages**: Handles both
    - curse of dimensionality
    - collectively exhaustive & mutually exclusive nature of the data
  - **Disadvantages**:
    - Not as declarative in matching (but could be with enums?)
    - I believe that may not currently be as efficient for certain matching?? Ask Kris, but he seemed to think that it could be brought to a level that was as efficient.



Performance Investments

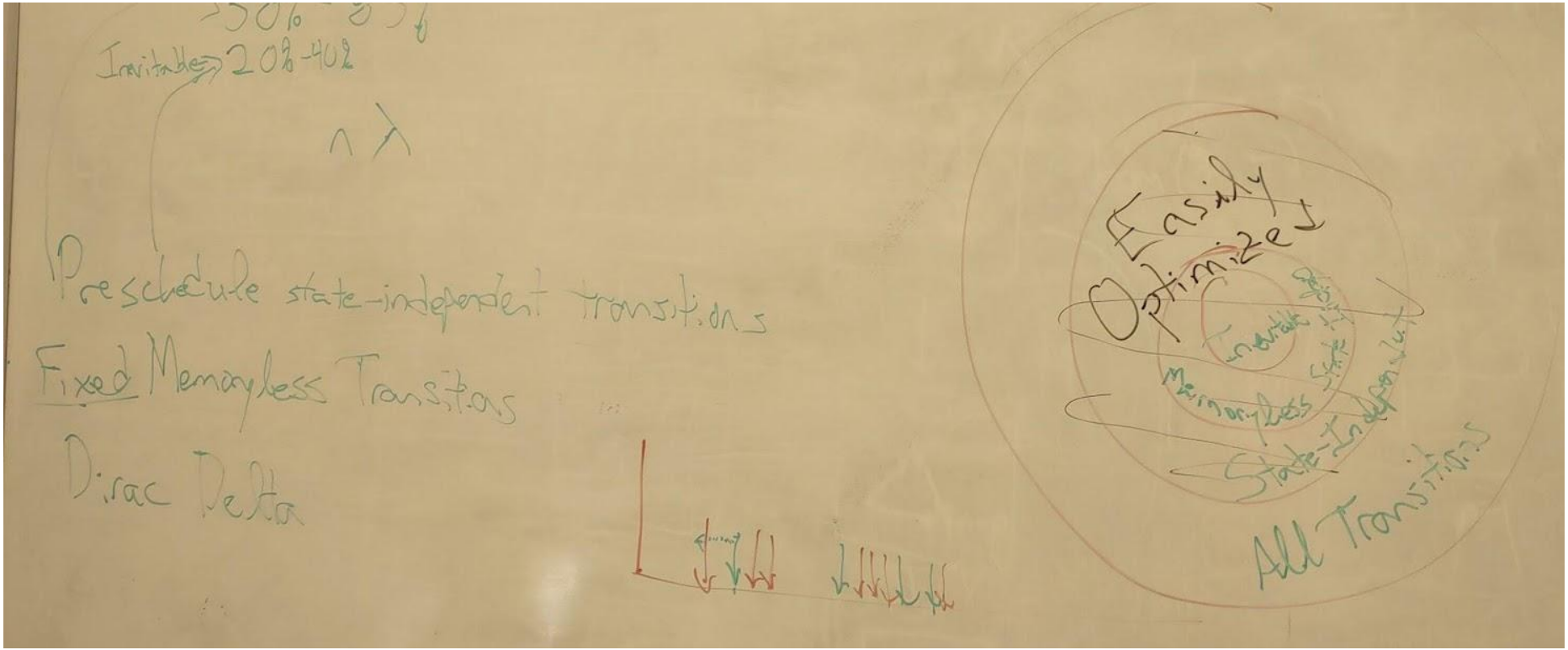
Performance Investments

- Incremental matching (“incremental hom search”) -- Alert rules when changes can trigger a pattern
  - Discrete event actions
  - More complicated with continuous dynamics
  - Descheduling events that no longer apply
- Staged computation: Build-time computation of constraints to optimize hom search
- Special match handling for
  - **Coproduct of Representables**: Due to Yoneda Lemma, know the number of matches (perhaps similar handling of each?); to match colimit, presumably need some match for each, so by Yoneda Lemma, rule out match if set mapped to by any of the representables whose colimit is taken are 0; detect change if count of any of “parts” has changed
  - Discrete category: This is a coproduct
- In-place rewriting
- Separation of positive application condition from patterns (reduces size of rewrite)
- Planned: Exploiting characteristics of statecharts
- Memoryless processes: Rather than scheduling n with rate λ, draw one with rate $n\lambda$
- Descheduling all events for a match when match fails

Successively Richer Opportunities for Optimization

- All transitions
- State independent transitions
- Memoryless state state independent transitions
- “Inevitable” memoryless state state independent transitions

Layering of Optimizations



Performance Investments

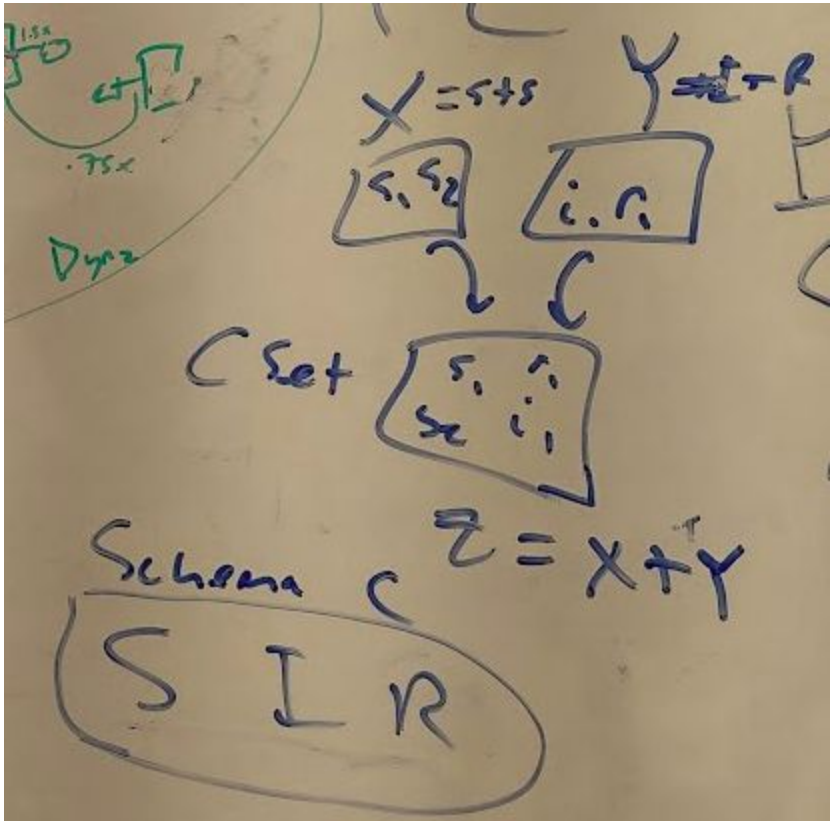
- Incremental matching (“incremental hom search”) -- Alert rules when changes can trigger a pattern
  - Discrete event actions
  - More complicated with continuous dynamics
  - Descheduling events that no longer apply
- Special match handling for
  - **Coproduct of Representables**: Due to Yoneda Lemma, know the number of matches (perhaps similar handling of each?); to match colimit, presumably need some match for each, so by Yoneda Lemma, rule out match if set mapped to by any of the representables whose colimit is taken are 0; detect change if count of any of “parts” has changed
  - **Discrete category**: This is a coproduct
- In-place rewriting
- Planned: Exploiting characteristics of statecharts
- Separation of positive application condition from patterns (reduces size of rewrite)
- Memoryless processes: Rather than scheduling n with rate λ, draw one with rate $n\lambda$
- Descheduling all events for a match when match fails
- HomSet type?

Flagging Co**products** of Representables for Optimization

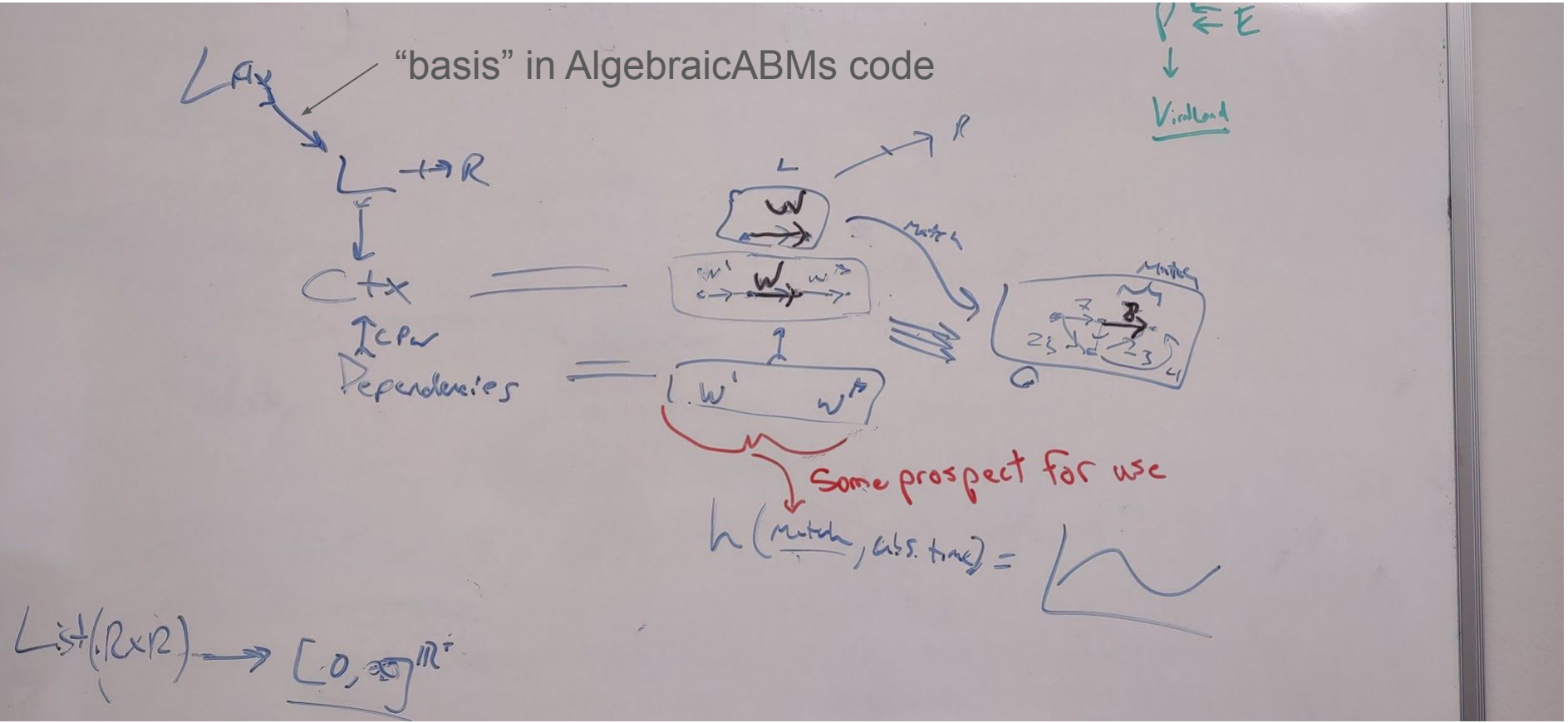
```
A pattern match from a coproduct of representables is just a choice of parts
in the codomain. E.g. matching L = •→• • •  is just a random choice of edge and
two random vertices.

The vector of ints refers to parts of L which are the counits of the left kan
extensions that define the representables (usually this is just wherever the
colimit leg sends 1, as there is often just one X part in the representable X).

WARNING: this is only viable if the timer associated with the rewrite rule is
symmteric with respect to the disjoint representables and has a simple
exponential timer.
"""
@struct_hash_equal struct RepresentableP <: PatternType
  parts::Dict{Symbol, Vector{Int}}
end

Base.keys(p::RepresentableP) = keys(p.parts)
```

Discrete Case

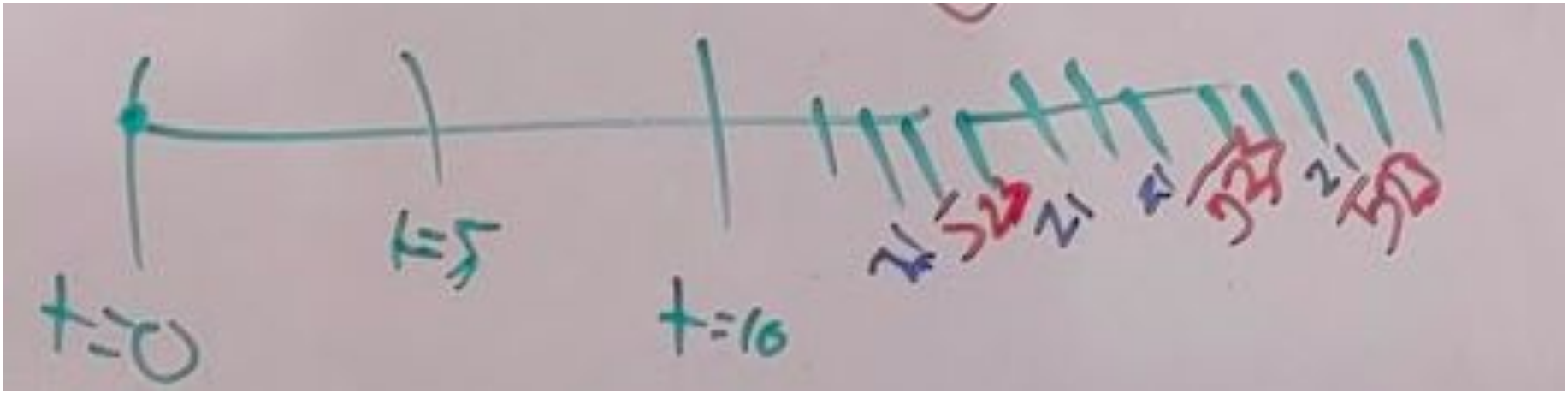
I believe the idea is perhaps that the coproduct of discrete cases is itself discrete?



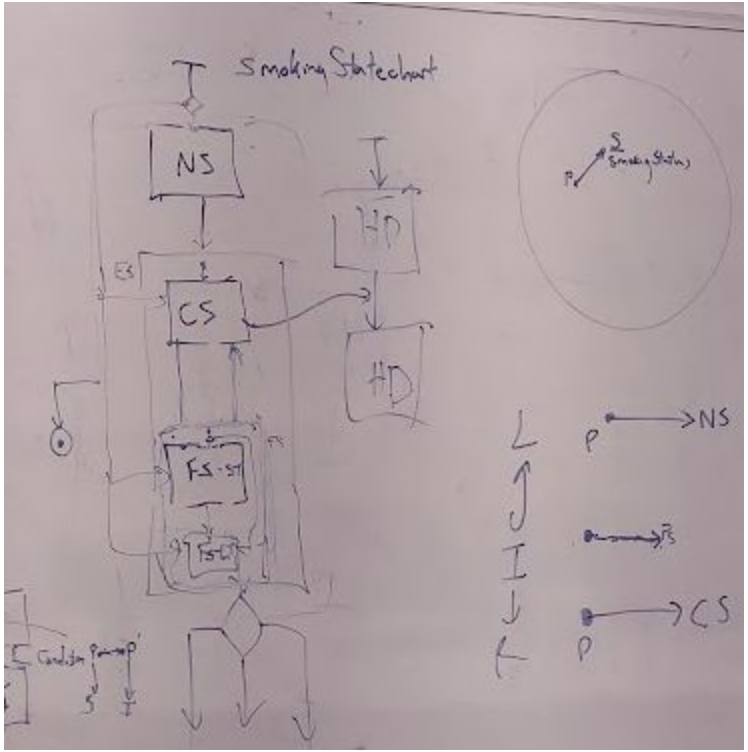
L_{fix} limits what Changes Are of Concern in Pattern Matching



Recall: Role of an Event Schedule (Queue)



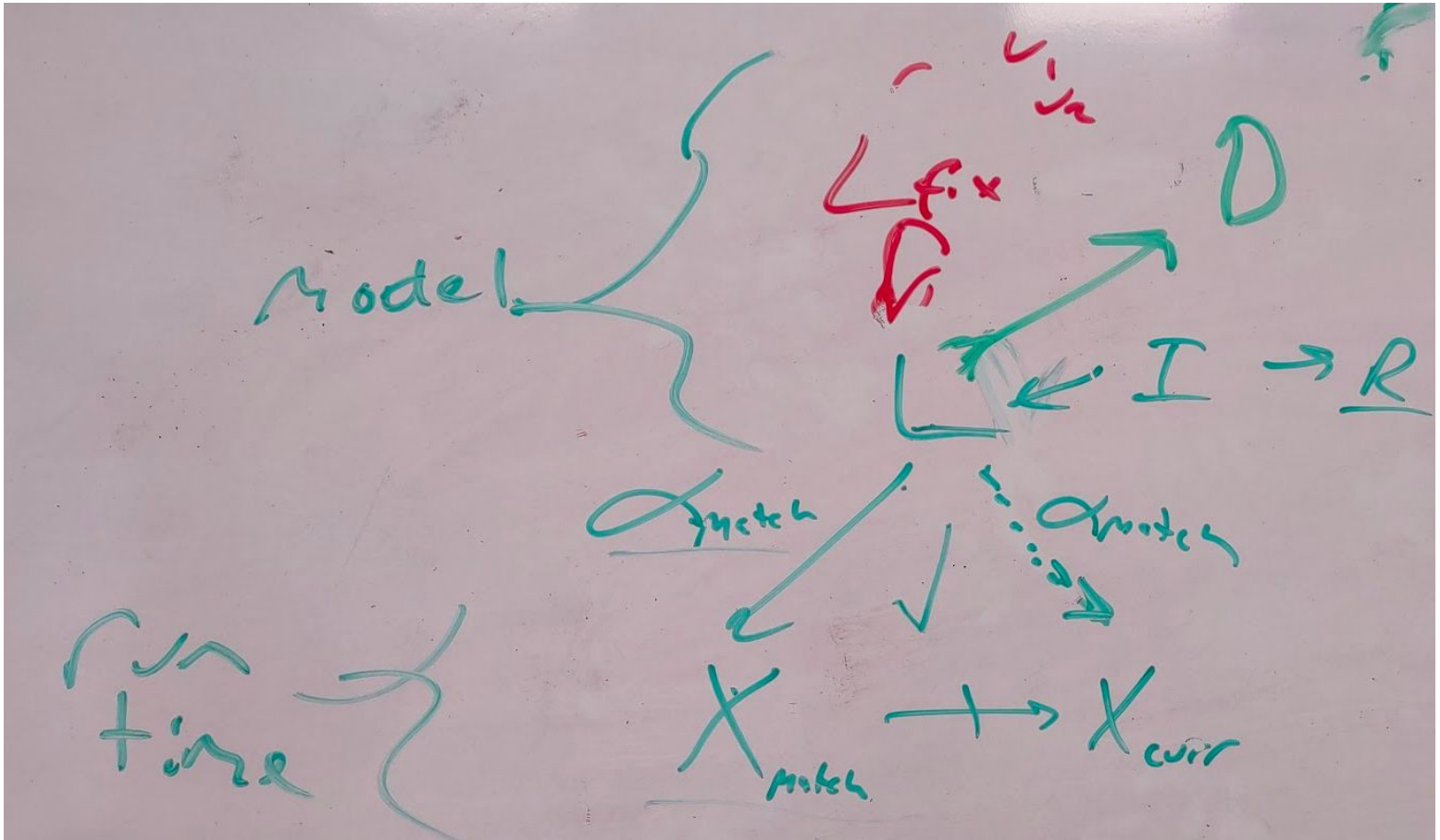
Statecharts Can Compile to Rewrite Rules



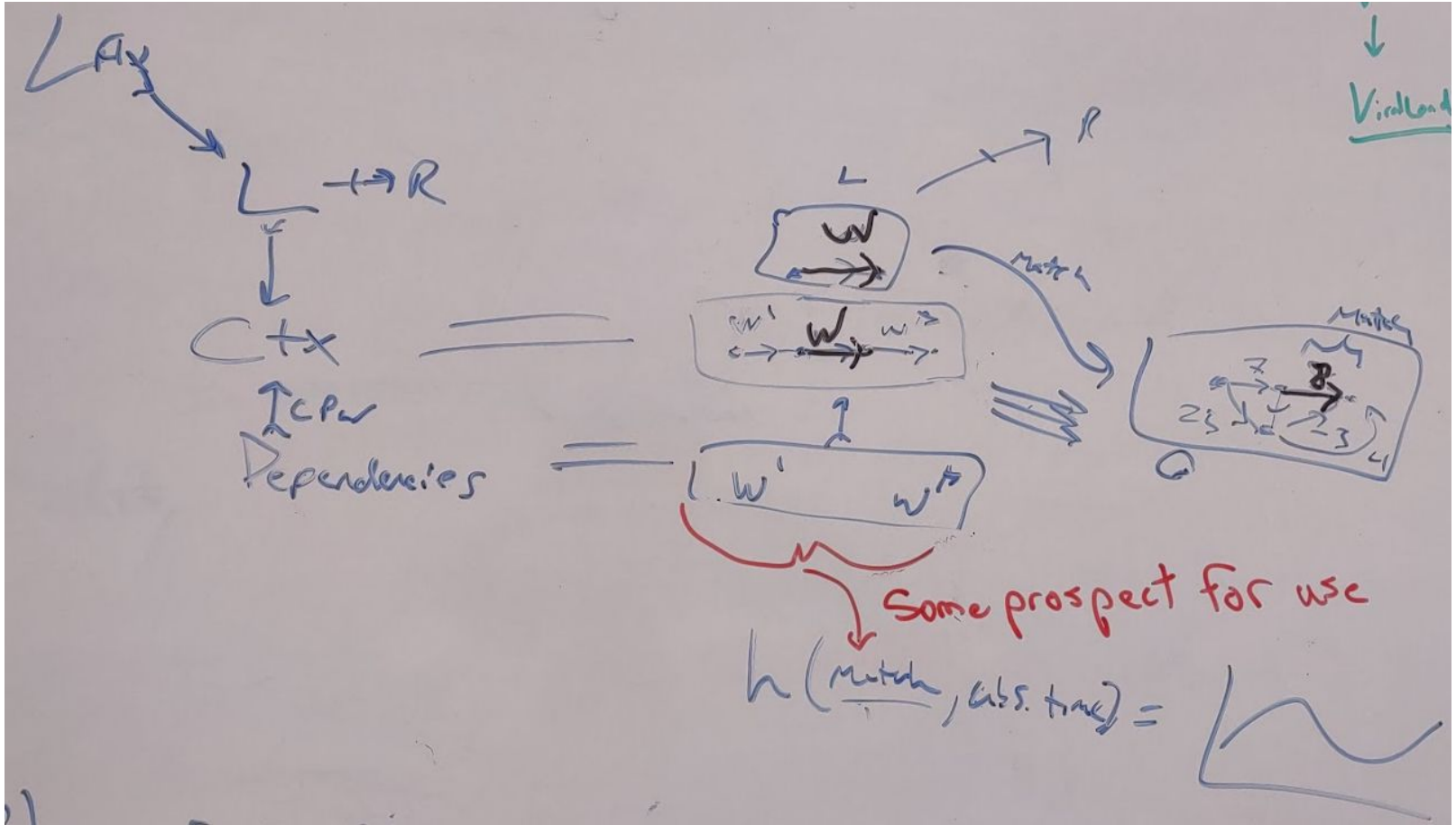
Anticipated Optimizations

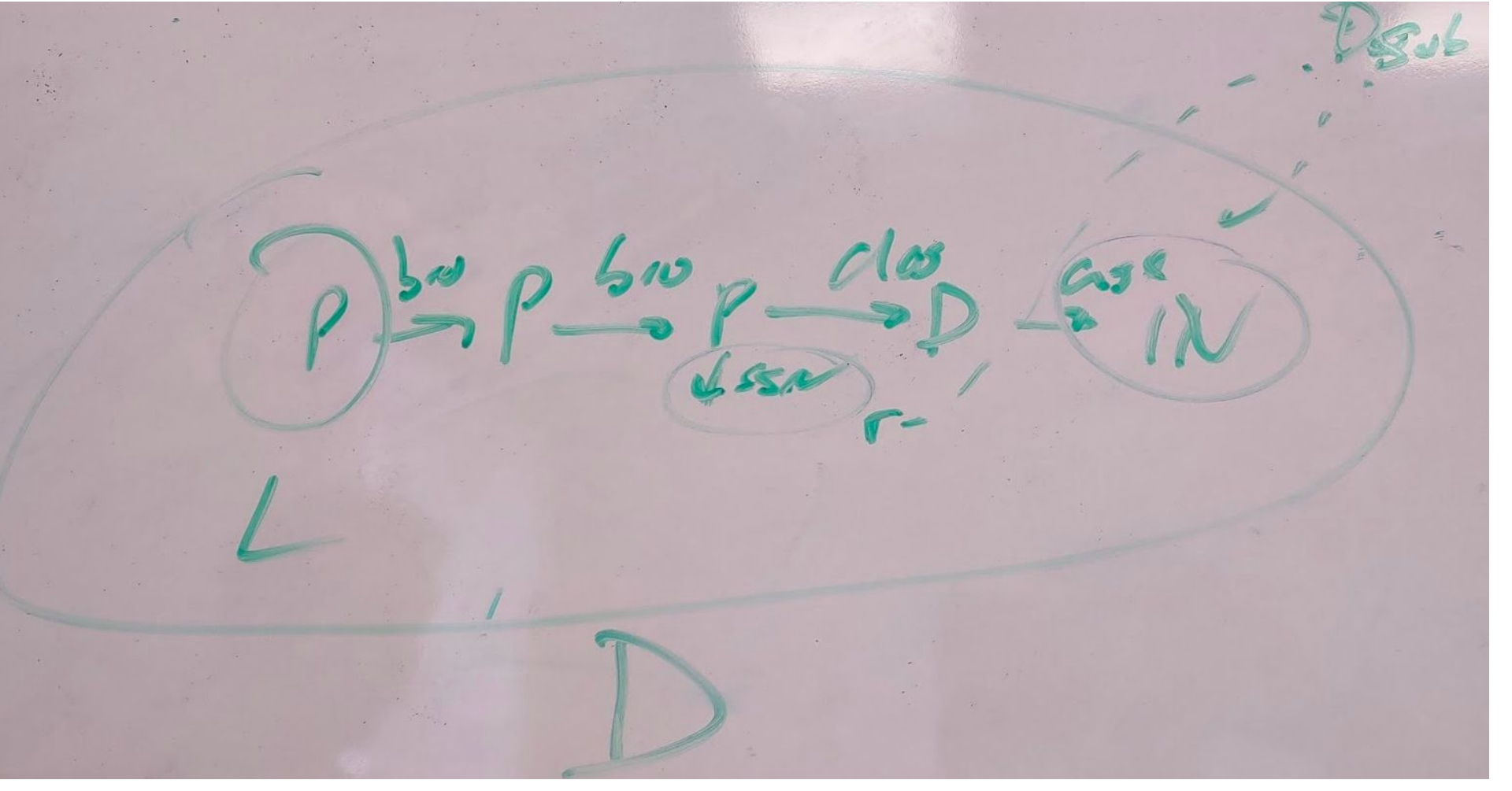
- Moving to observation processes rather than all changes in state

Separating out Context that Need not be Matched or Rewritten



Extracting Further Information on Selected Dependencies




P
P
P
D
IN
L
D

Statecharts

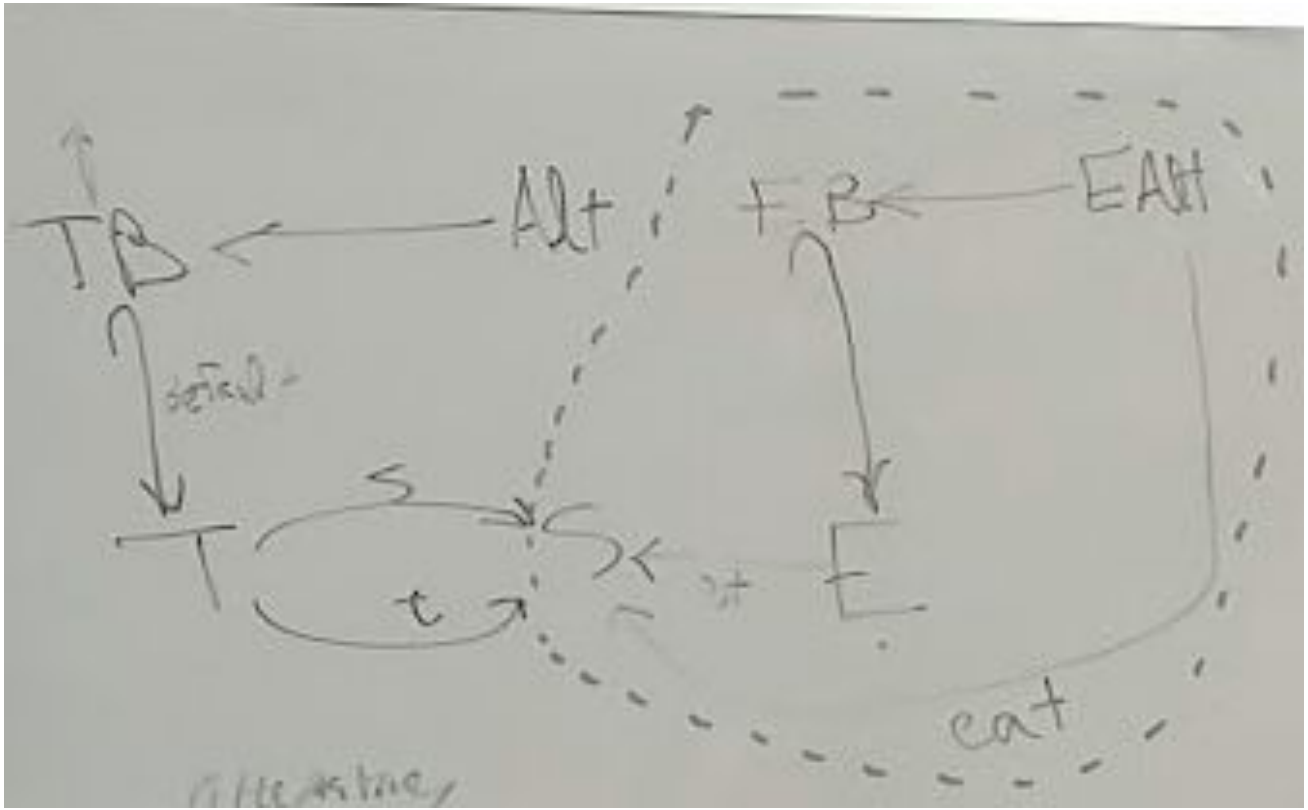
Currently Planned Statechart Transition Types

- Condition (fires immediately upon pattern match)
- Arrival (fires immediately upon arrival)
- For each of {Hazard rate, Timeout [Dirac delta]}, variants that have the following; guards here can make use of both positive & negative application conditions
  - Plain
  - Eligibility Guard
  - Triggering Guard
  - Both Eligibility & Triggering Guard

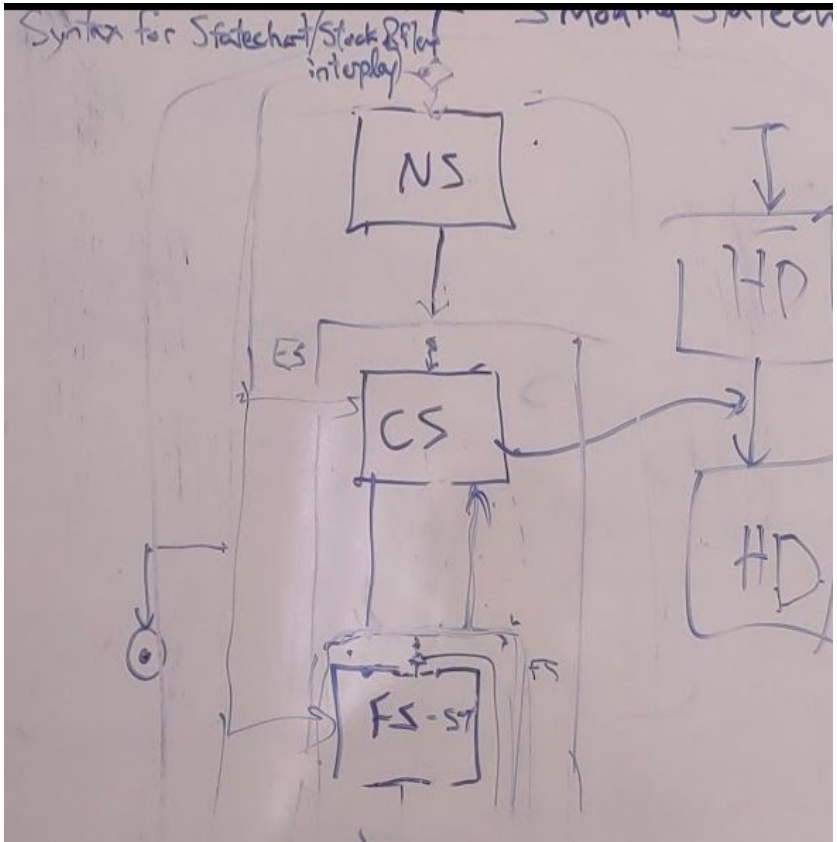
Statechart Affordances for Optimization

- All transitions emerging from state S are conditional on remaining in state S
- Can readily dequeue events departing a state if leave via other mechanisms
- For events without conditions other than presence in the originating state
  - Can determine events candidate timing for all departure transitions, and pre-schedule only earliest of them -- not all
  - Need not check any conditions at start than presence on state, and nothing when leave since would be dequeued if left state
- When have only eligibility-guarded transition, by definition need only check that condition when first matching; would be removed if left before firing, and fires unconditionally if remain
- When have only eligibility-guarded transition, by definition need only check that condition when first matching; would be removed if left before firing, and fires unconditionally if remain
- When have only triggered-guarded transition, need only check presence in state as match, and then check condition when would fire; would be removed if left before firing
- When have both eligibility-guarded and triggered-guarded transition, we have the usual case: Need to check full match initially (including that in state), and then check match when would fire
- Don’t have to reschedule a new firing time when departing state -- know won’t match

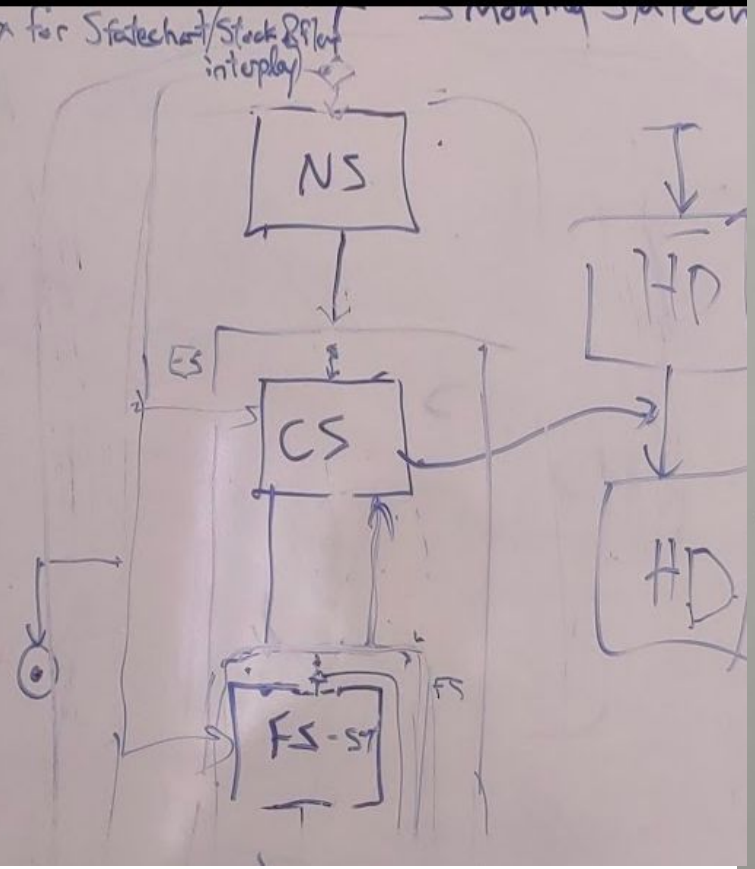
Early Idea for Statechart Schema

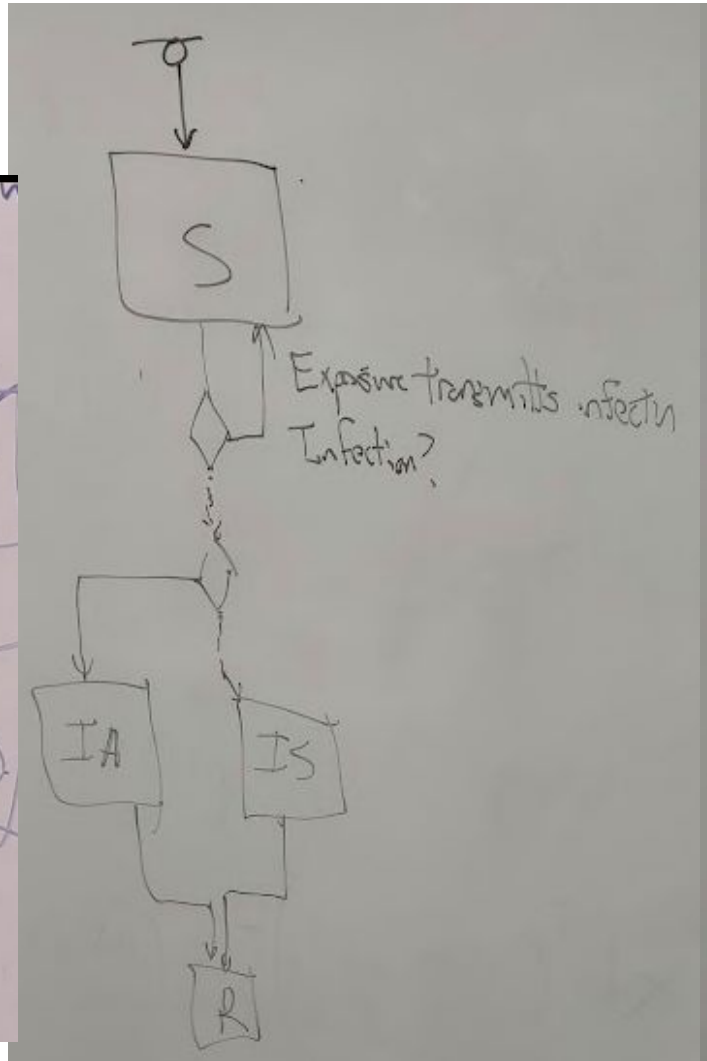


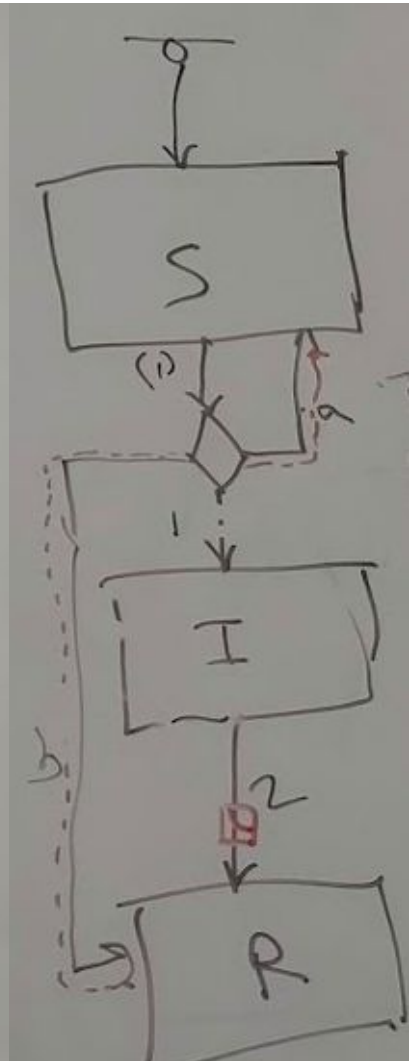
Forgoing Hierarchical Statecharts for Now



Example Statecharts







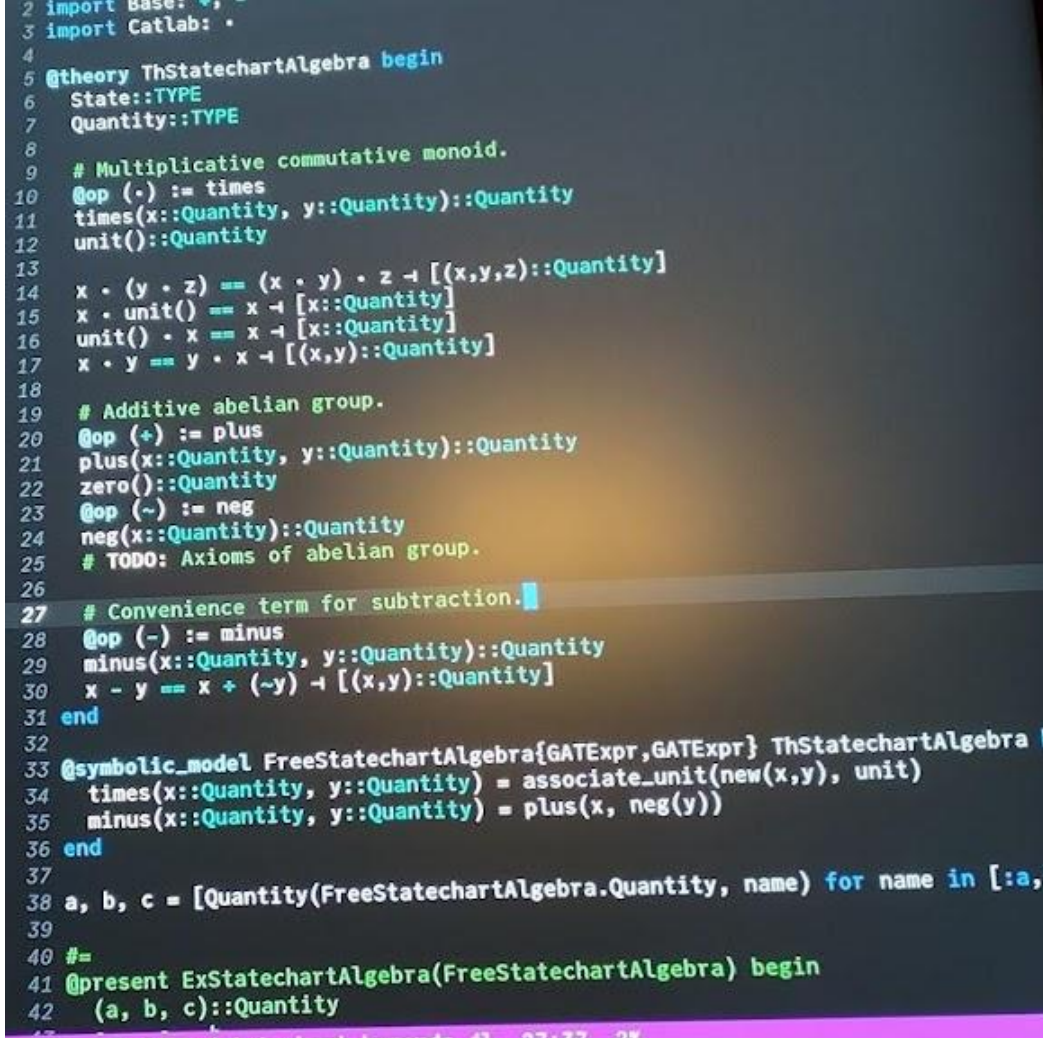
Statecharts and Dependencies

- Pattern matching rules for contextual dependencies not CONSUMING
  - Will seek negative or positive application condition alongside a rule that would otherwise fire
- application conditions can be nicely combined with statecharts when don't require resource to be consumed -- can't just take the boolean negation of boolean for application
- for contextual dependencies dependent on an agent's state that are required for a transition which require CONSUMING or CHANGING other aspects of state, we can often either a
  - petri net
  - arbitrary rewrite conditoons

Statechart Hazard Rate Expressions via GATLab

- We will represent the hazard rate rules for statecharts via GATLab
- A GATExpr (and subtypes) can be represented as an attribute in schemas
- Using GATLab in this way will let us track added semantic information -- e.g., dimensions/units, domains of values
- GATLab is already planned for explicit use in taking pushouts of schemas

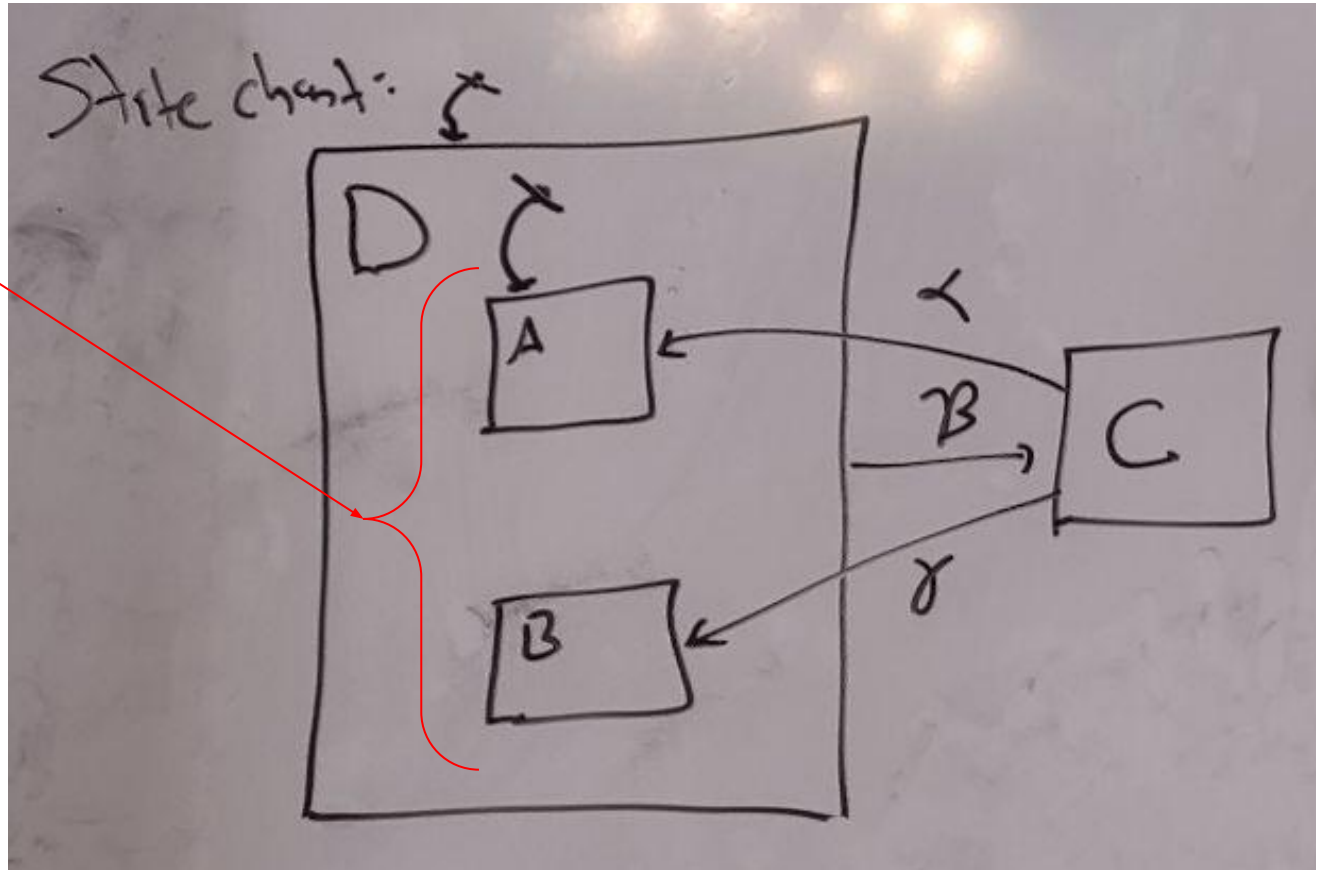
Example Starting Theory for Statechart Schemas



```
import Base: *, ⋅
import Catlab: ⋅

@theory ThStatechartAlgebra begin
  State::TYPE
  Quantity::TYPE

  # Multiplicative commutative monoid.
  @op (⋅) := times
  times(x::Quantity, y::Quantity)::Quantity
  unit()::Quantity

  x ⋅ (y ⋅ z) == (x ⋅ y) ⋅ z ⊣ [(x,y,z)::Quantity]
  x ⋅ unit() == x ⊣ [x::Quantity]
  unit() ⋅ x == x ⊣ [x::Quantity]
  x ⋅ y == y ⋅ x ⊣ [(x,y)::Quantity]

  # Additive abelian group.
  @op (+) := plus
  plus(x::Quantity, y::Quantity)::Quantity
  zero()::Quantity
  @op (~) := neg
  neg(x::Quantity)::Quantity
  # TODO: Axioms of abelian group.

  # Convenience term for subtraction.
  @op (-) := minus
  minus(x::Quantity, y::Quantity)::Quantity
  x - y == x + (~y) ⊣ [(x,y)::Quantity]
end

@symbolic_model FreeStatechartAlgebra{GATExpr,GATExpr} ThStatechartAlgebra
  times(x::Quantity, y::Quantity) = associate_unit(new(x,y), unit)
  minus(x::Quantity, y::Quantity) = plus(x, neg(y))
end

a, b, c = [Quantity(FreeStatechartAlgebra.Quantity, name) for name in [:a,

#=
@present ExStatechartAlgebra(FreeStatechartAlgebra) begin
  (a, b, c)::Quantity
```

Theory for Hierarchical Statecharts (Evan Patterson)

D := A + B



Distributive Categories -- Category $\mathscr{C}$ with

- Finite coproducts
- Finite products
- Products distribute over coproducts: For all A,B,C \in Ob($\mathscr{C}$) have isomorphism $(C \times A) + (C \times B) \leftrightarrows C \times (A + B)$
  - A canonical forward map is present for any category
  - For a distributive category, the reverse map must be the inverse of the forward map

Guaranteed, Canonical Map $(C \times A) + (C \times B) \to C \times (A + B)$

- As a map out of a coproduct, $(C \times A) + (C \times B) \to C \times (A + B)$ is equivalent to the pair of maps
  - $(C \times A) \to C \times (A + B)$
  - $(C \times B) \to C \times (A + B)$
- $(C \times A) \to C \times (A + B)$ is given by
  - $\pi_C : (C \times A) \to C$
  - $\pi_A ; \iota_A : (C \times A) \to A + B$
- $(C \times B) \to C \times (A + B)$
  - $\pi_C : (C \times B) \to C$
  - $\pi_B ; \iota_B : (C \times B) \to A + B$

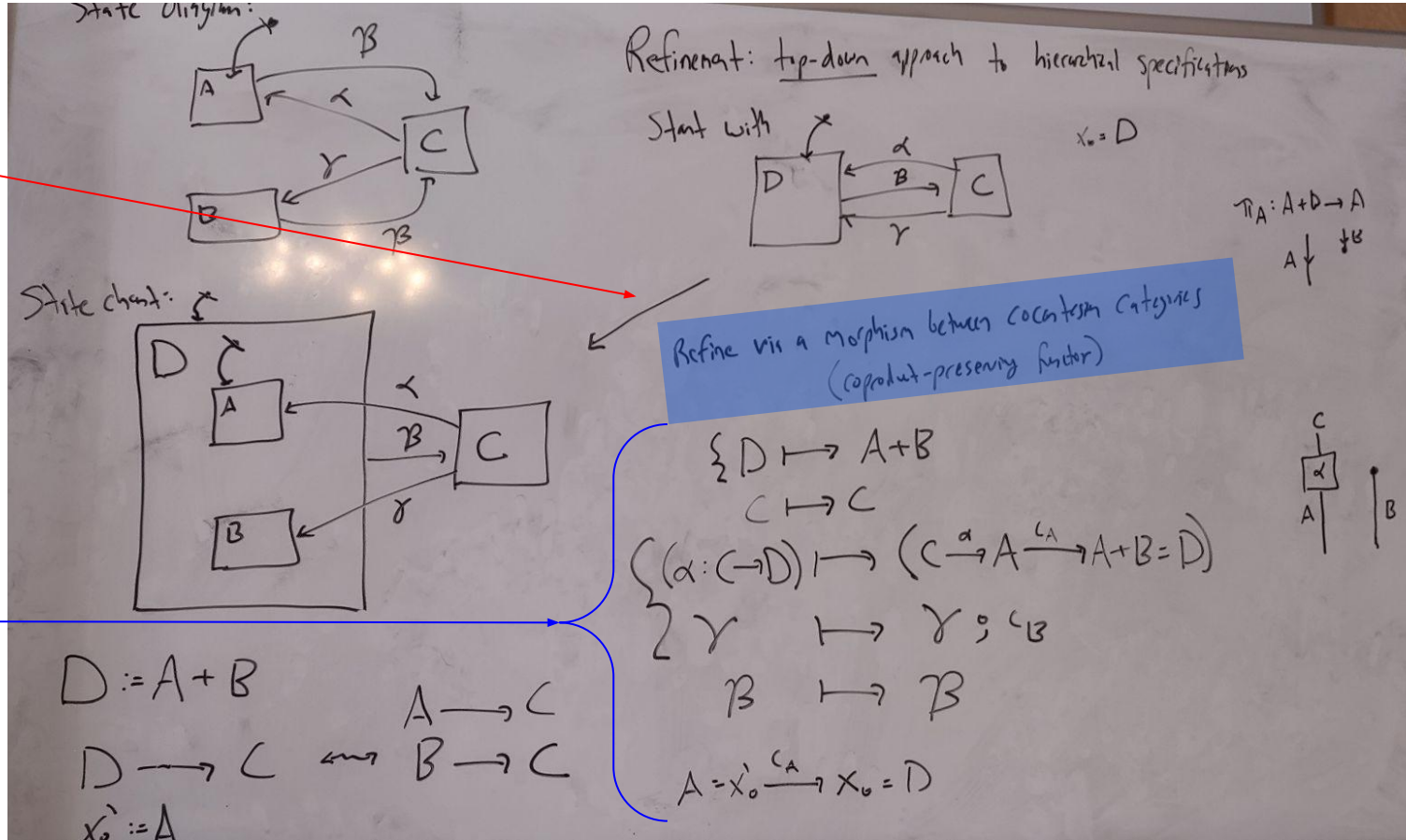
Statecharts as Distributive Categories: “Top-down” Expansion

- Multiple (mutually exclusive and collectively exhaustive) substates within a state (or overall statechart) as combined via **coproduct**
- Parallel statecharts combined via **product**
- **Morphisms**: Coproduct-preserving “refinement functor” maps coarser statechart to more detailed statechart (expanding hierarchically to do so)
- Additional features
  - Shared transitions (inputs) between statecharts
  - Initial states

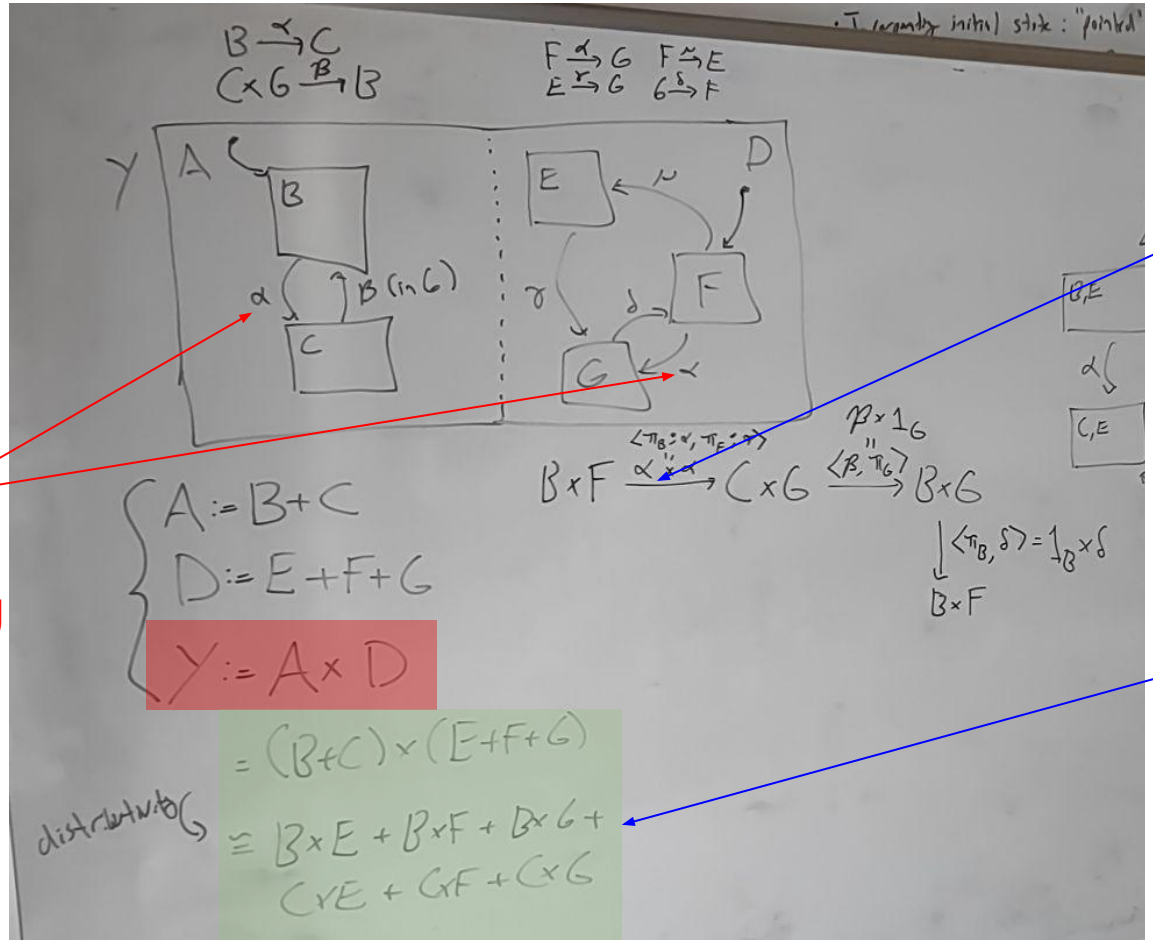
Co-Product Components: “Refinement” Relation

Coproduct preserving “refinement functor” maps coarser diagram into finer-grained (“refined”) diagram

Mapping of objects & morphisms



Product Components: Parallel Statecharts

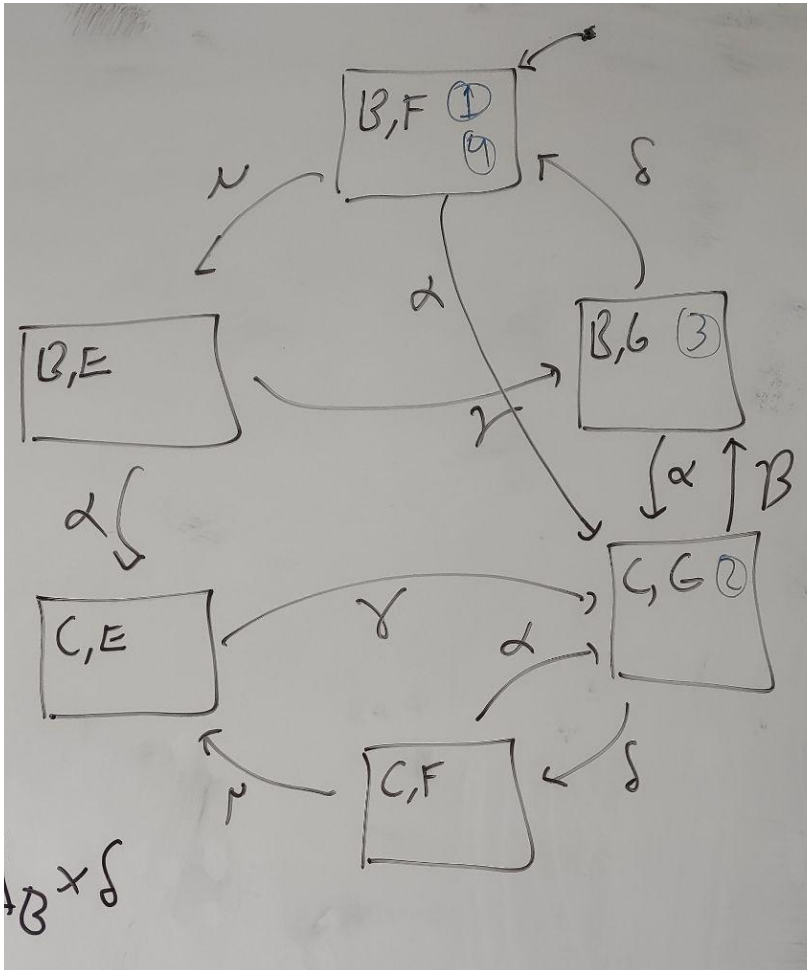


Note shared transition -- can think of as being triggered by same input.

Transitions from one state to another

Application of distributive law

Equivalent Fully Expanded (Non-Hierarchical) Statechart

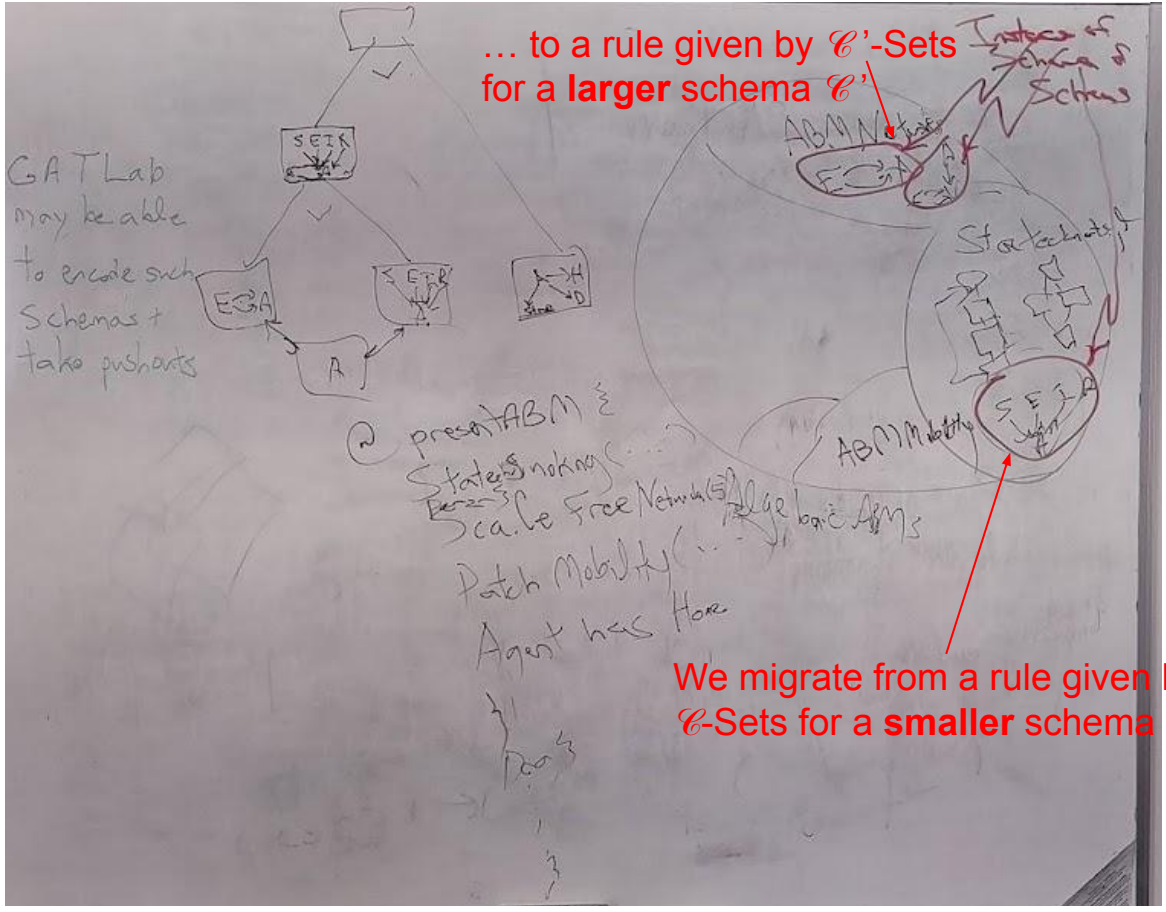


Lifting of Rewrite Rules & Patterns to Successively Larger Schemas

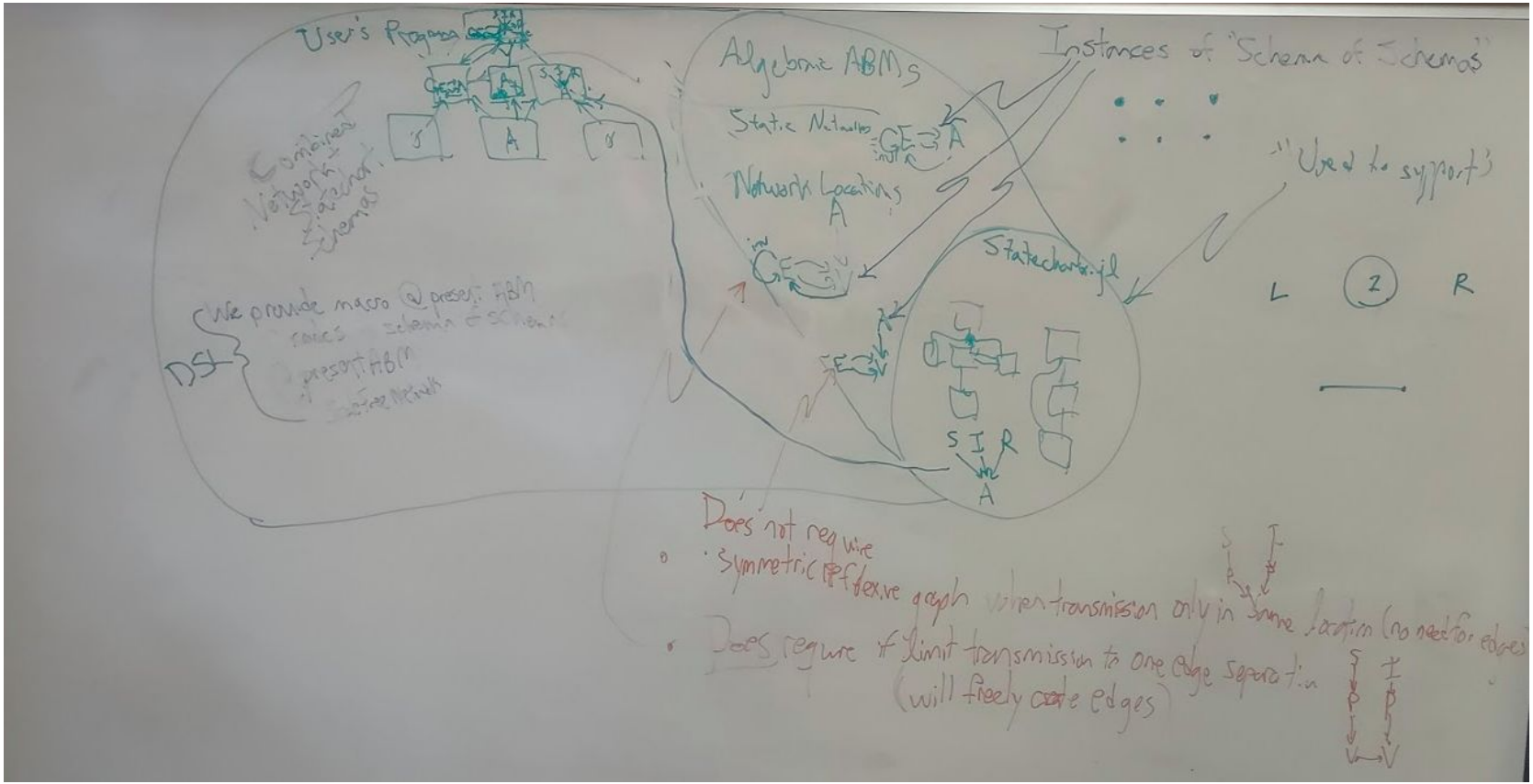
Layering in Context Via Schema Pushouts
-- Migrating Logic as Schema Grows

While GATLab can take pushouts of different theories, for legacy reasons, it cannot yet take pushouts of schemas for the same theory.

For now, Xiaoyan has implemented a schema of schemas. This will be replaced when GATLab's mechanisms are available.



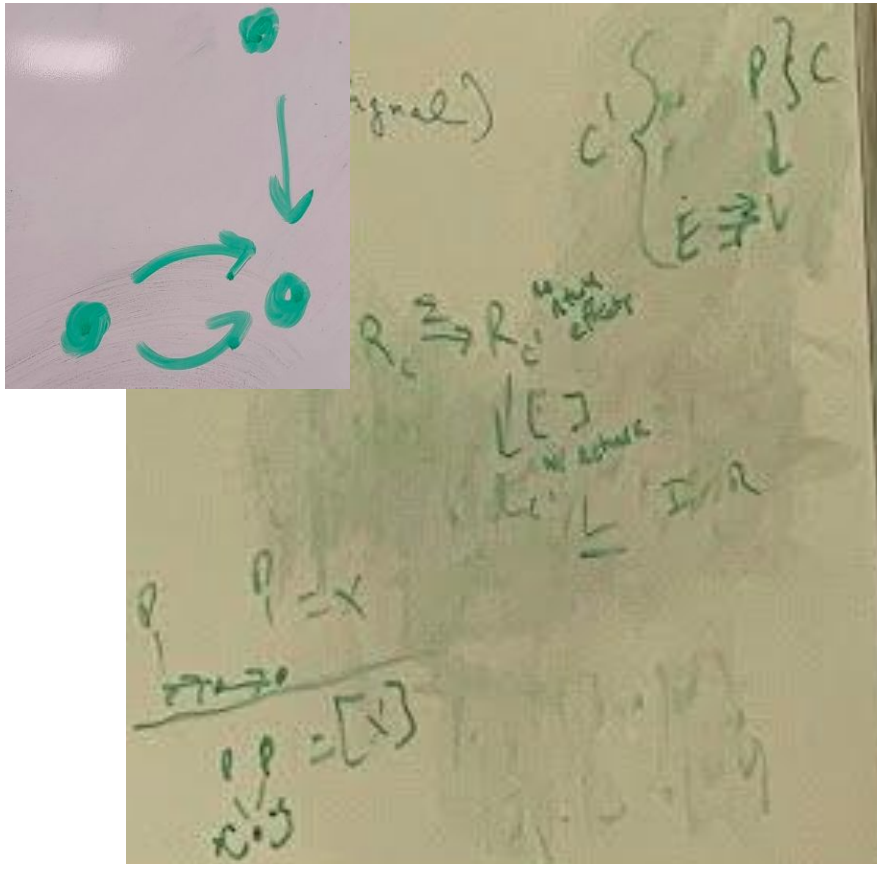
Successive Layering



Weaving in Common Location Logic into an Arbitrary Rule

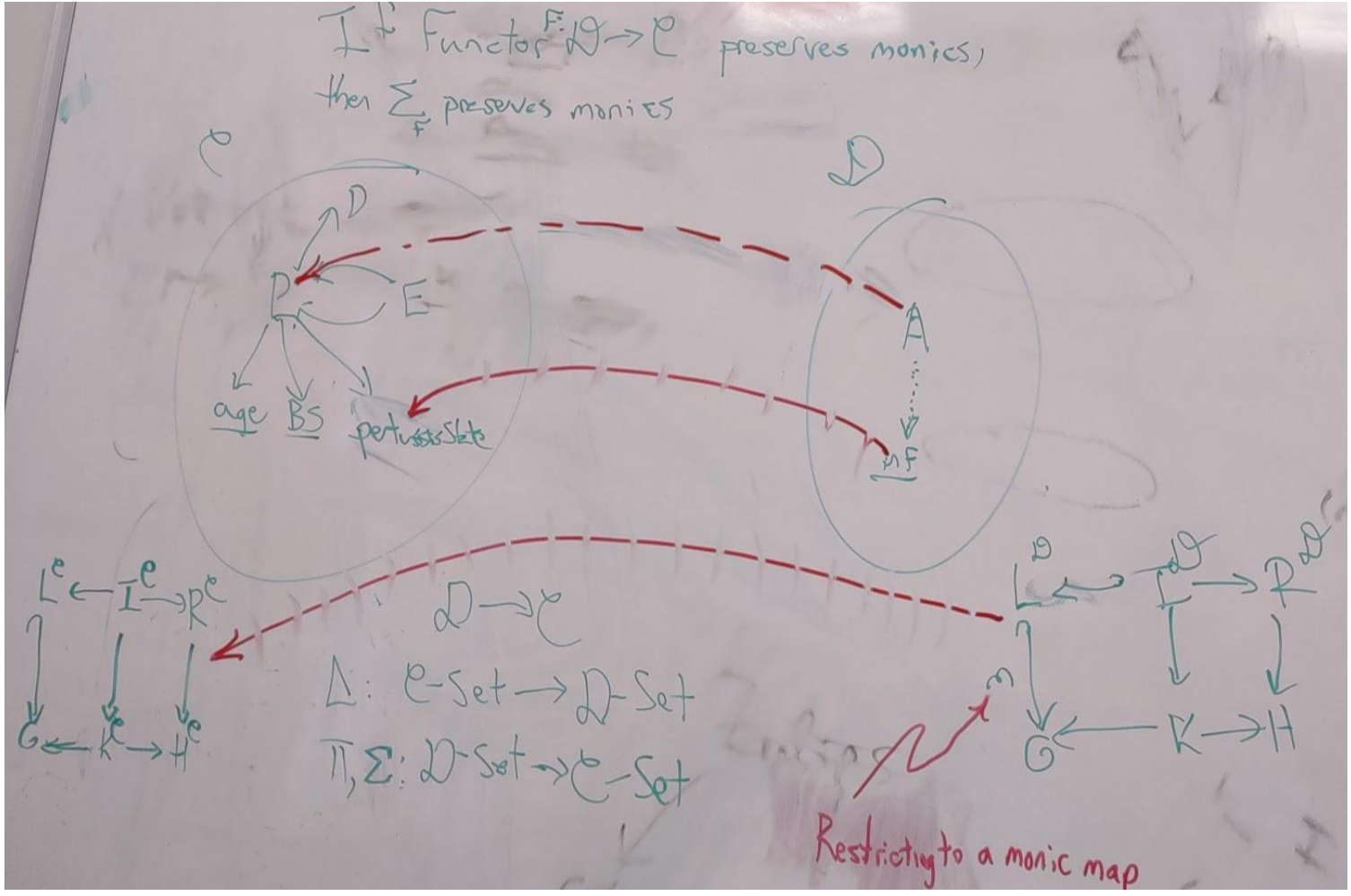
For example can readily weave rewrite rule logic for these cases:

- without regards to network
- at the same location
- for agents with n distance of each other



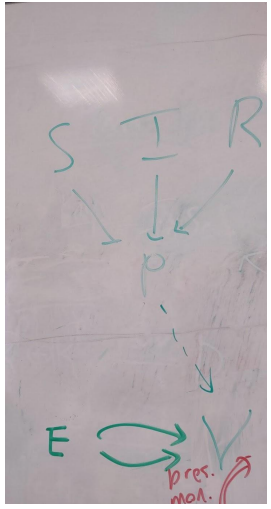
Data Migrations -- Canonical and Otherwise

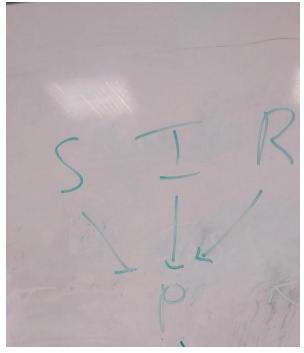
Whilst such canonical data migrations are useful, there are many times we will seek other forms of data migrations that cannot be expressed in terms of simple maps between schemas. DataMigrations.jl gives us far greater flexibility.

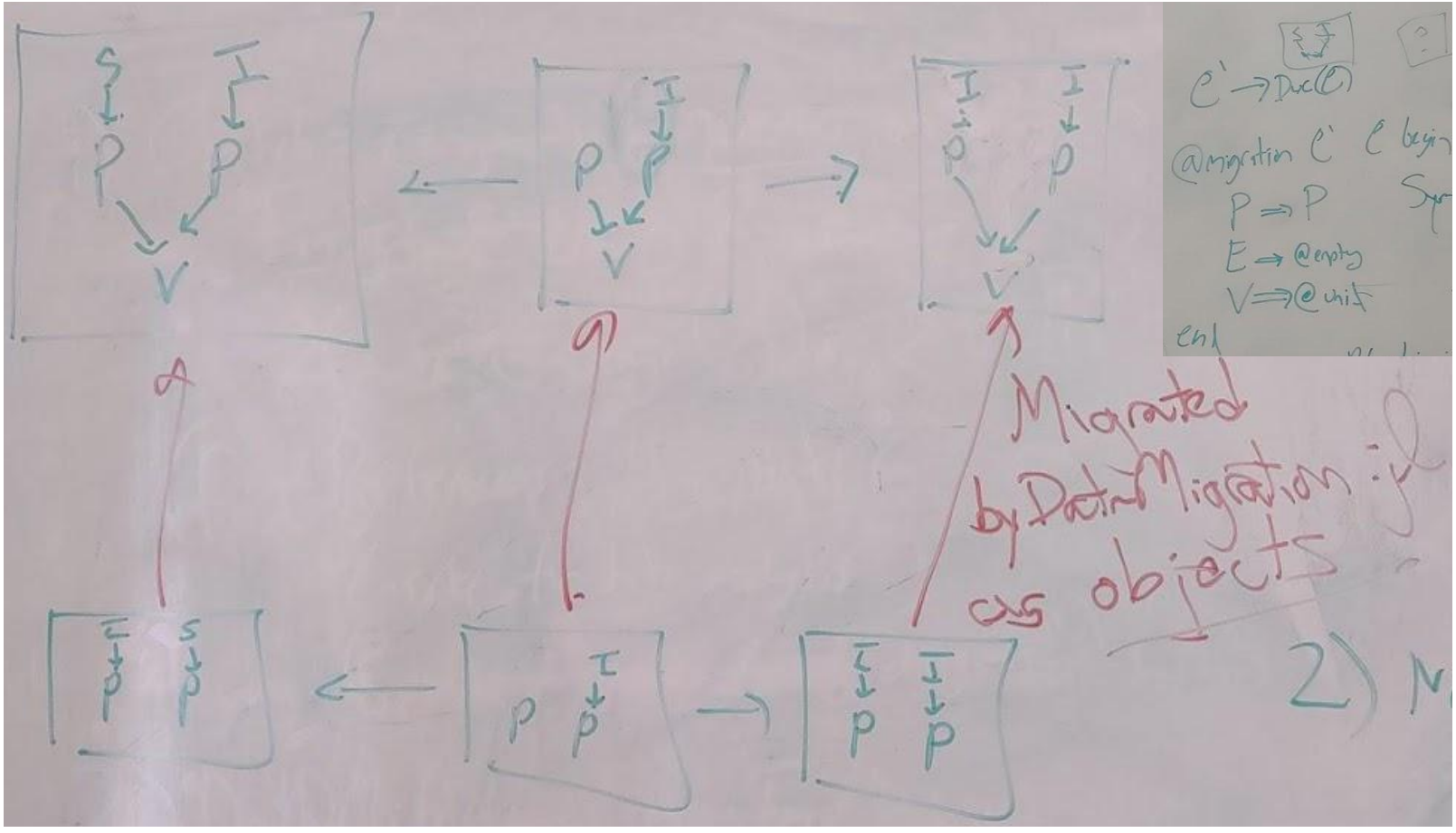


Migrating to Default Network Behaviour

Schema

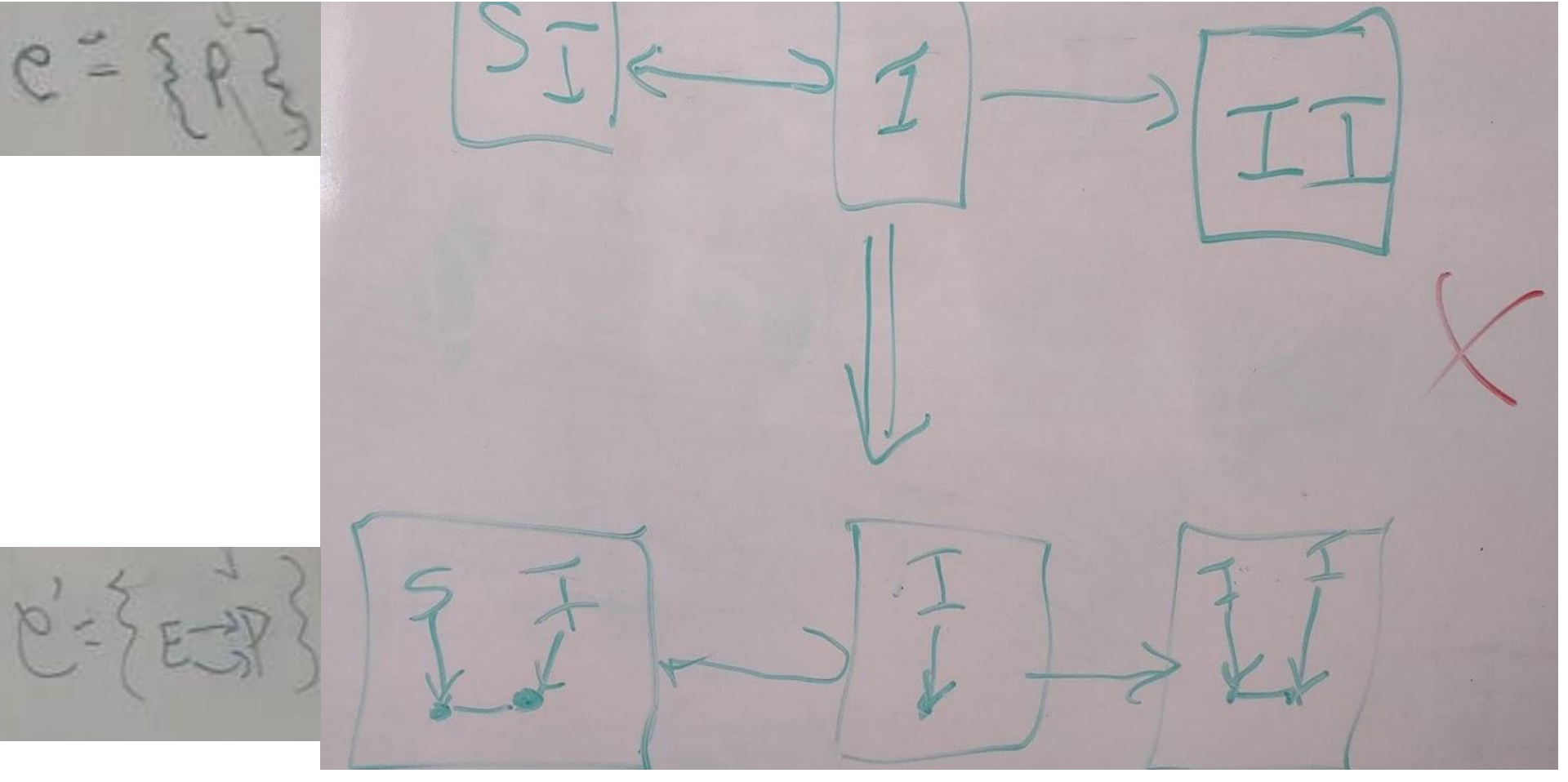






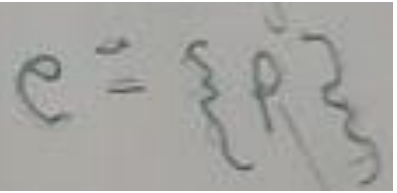
Logical Limits on Behaviour: Infeasible Migration

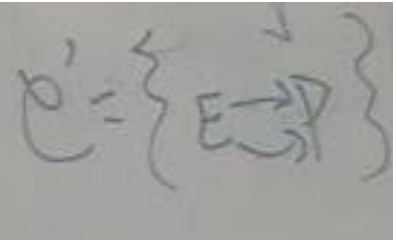
Schema

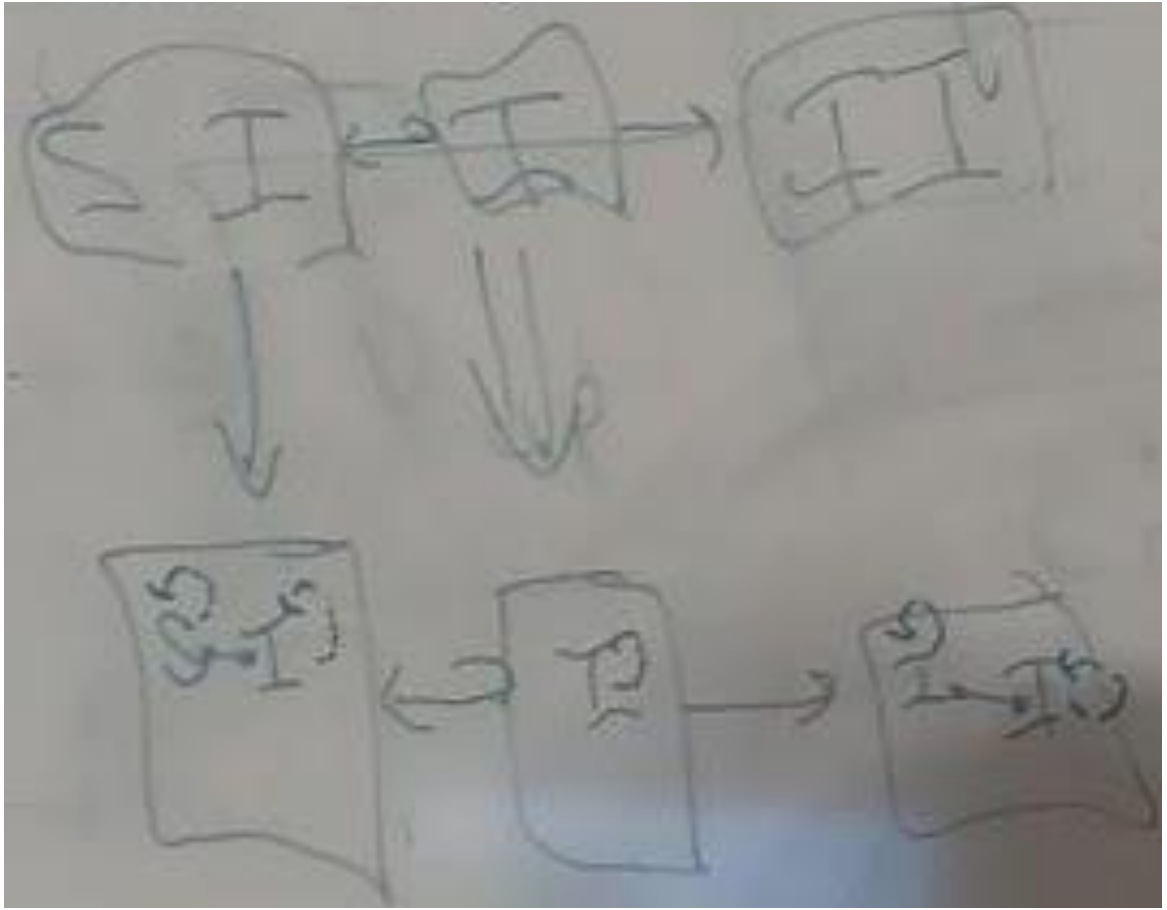


Logical Limits on Behaviour: Feasible Variant

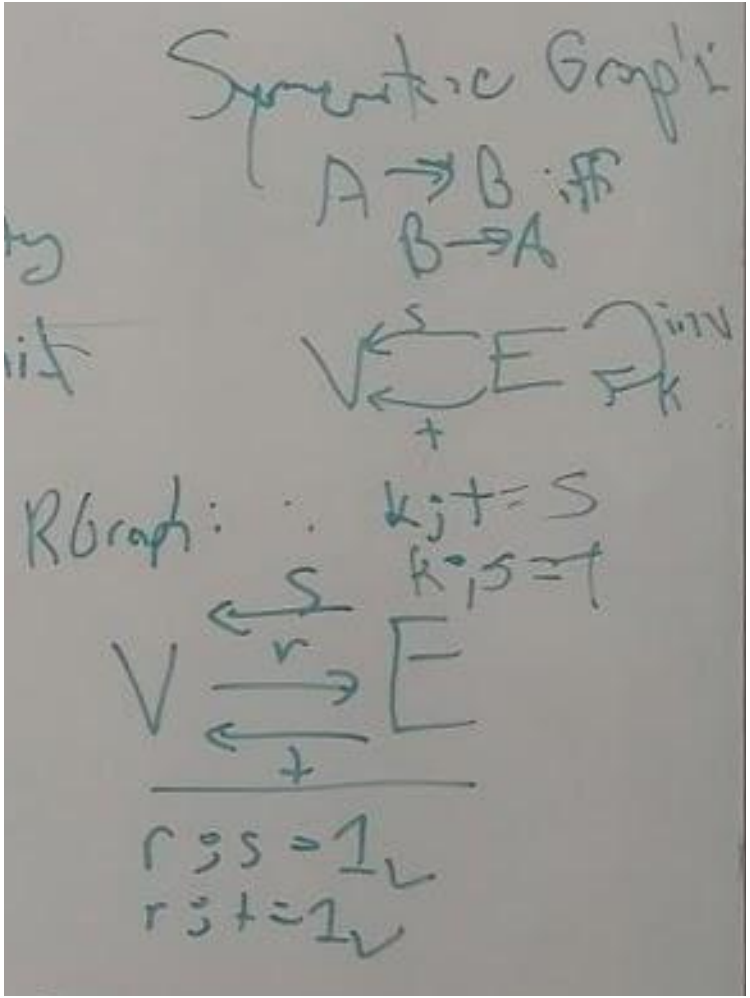
Schema



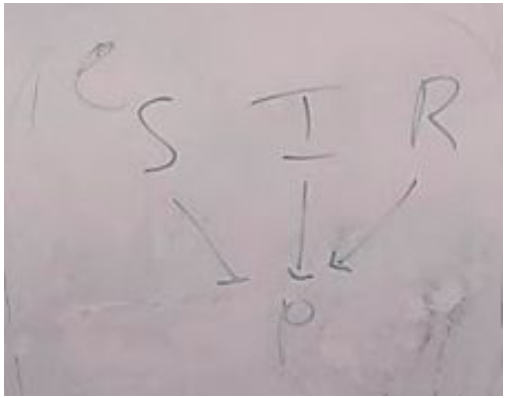


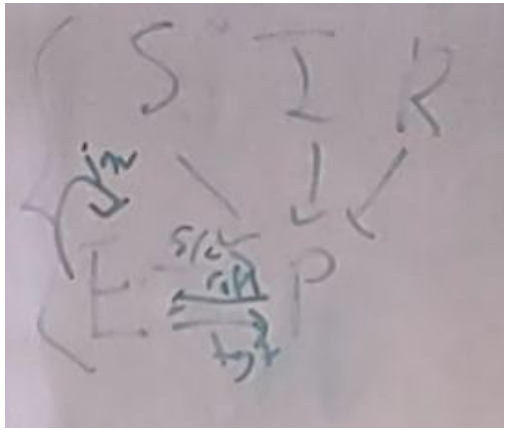


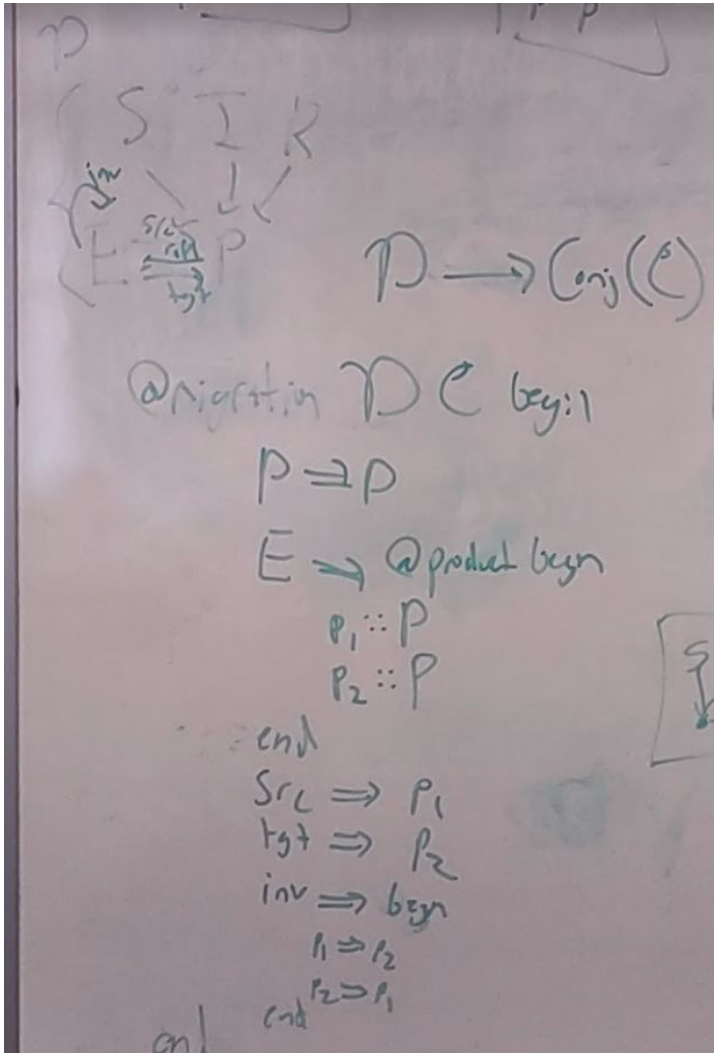
Suggestion: Make what would otherwise be artefactual behaviour (and could be missed in rewriting patterns) standard, so that it is assured in all matches



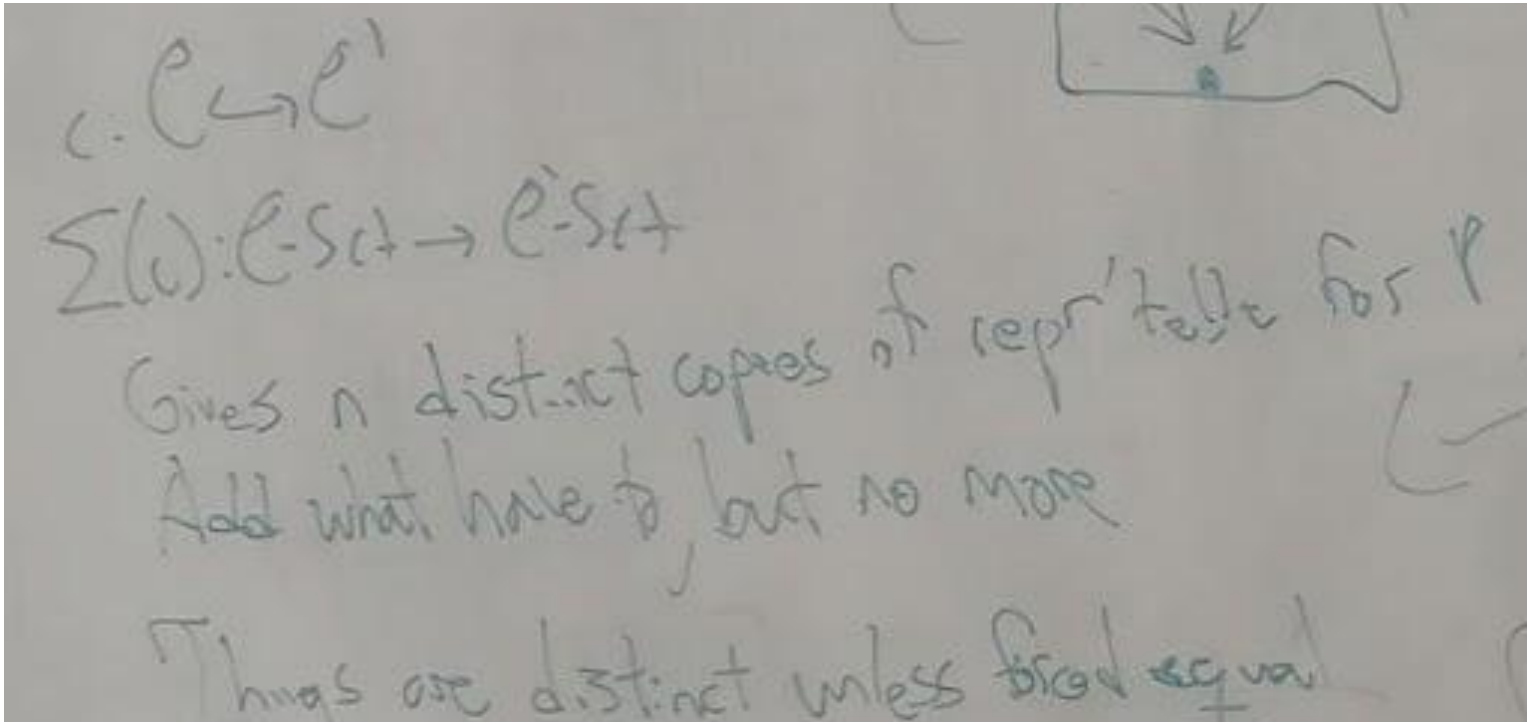
Additional Detail





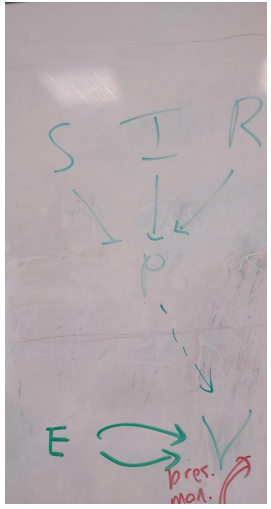


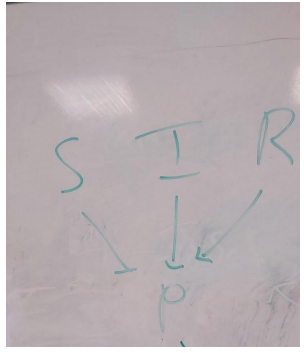
Sigma Migration Gives “Free” Migration

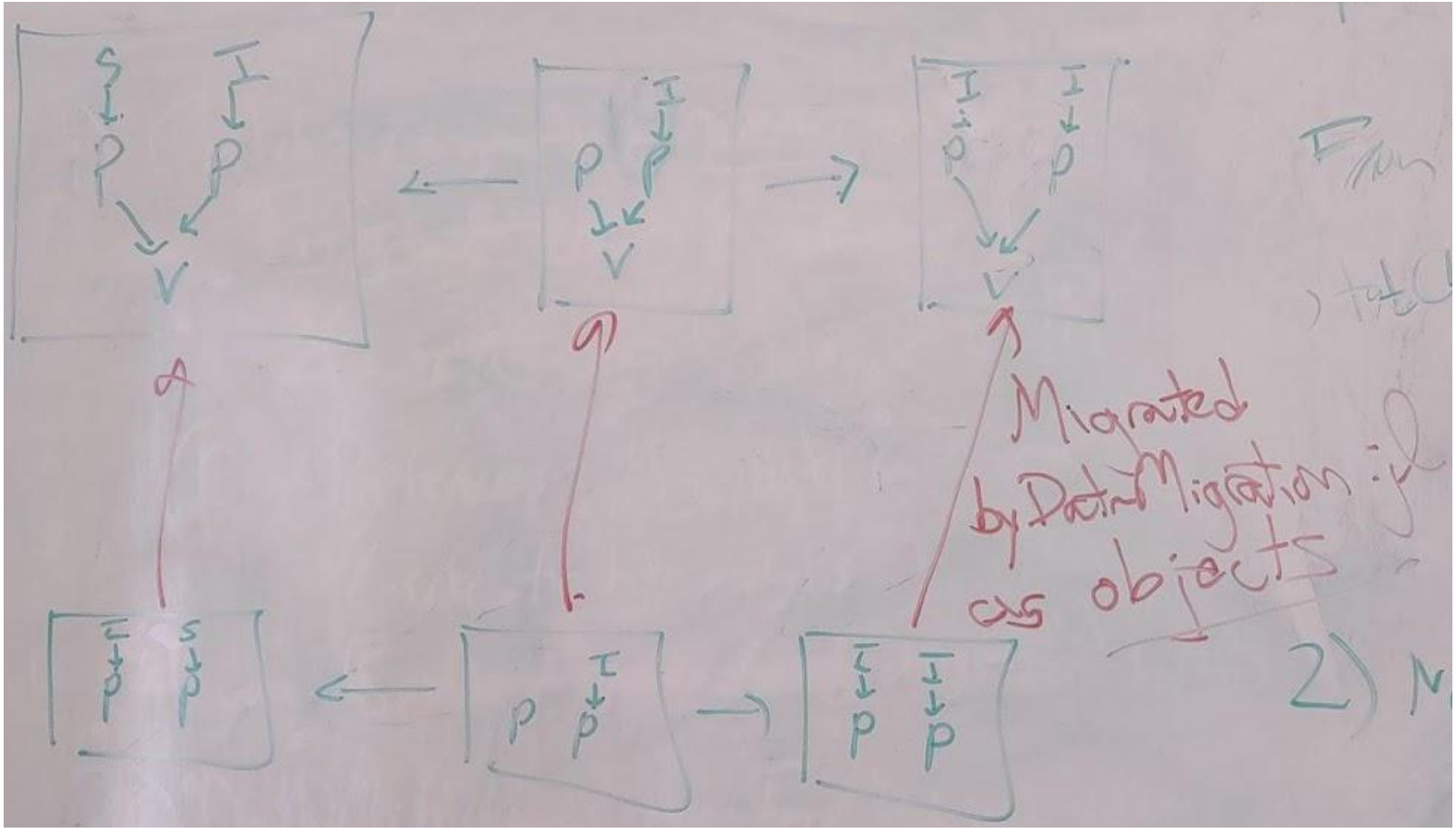


Migrating to Default Network Behaviour

Schema

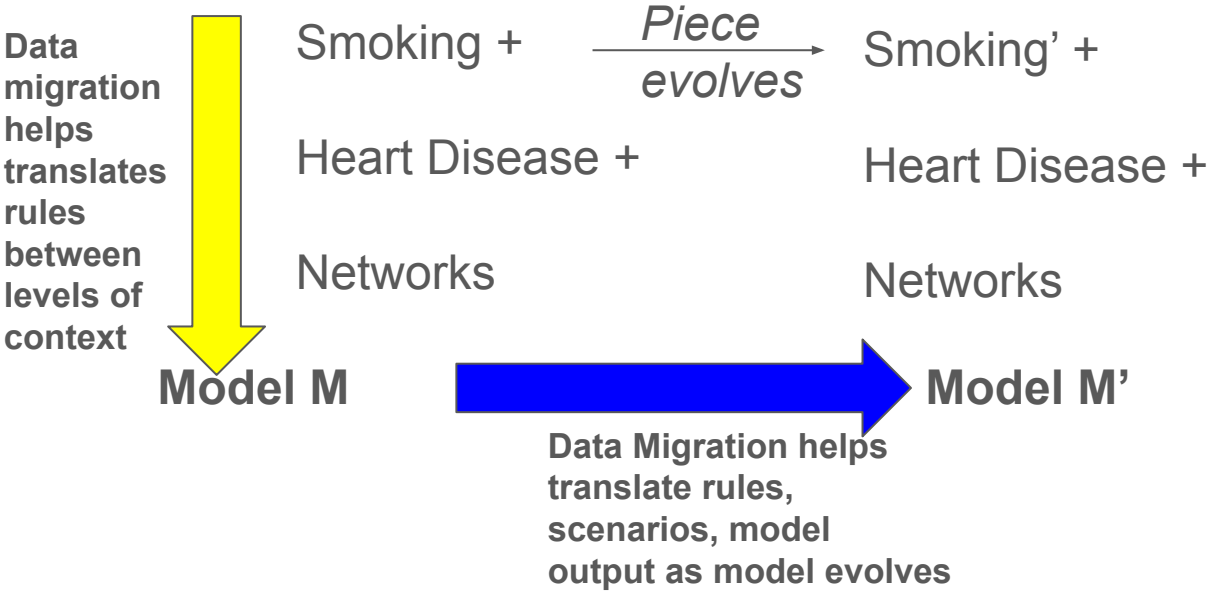






Data Migration Supports Translation Across Level of Model & Evolution of Model

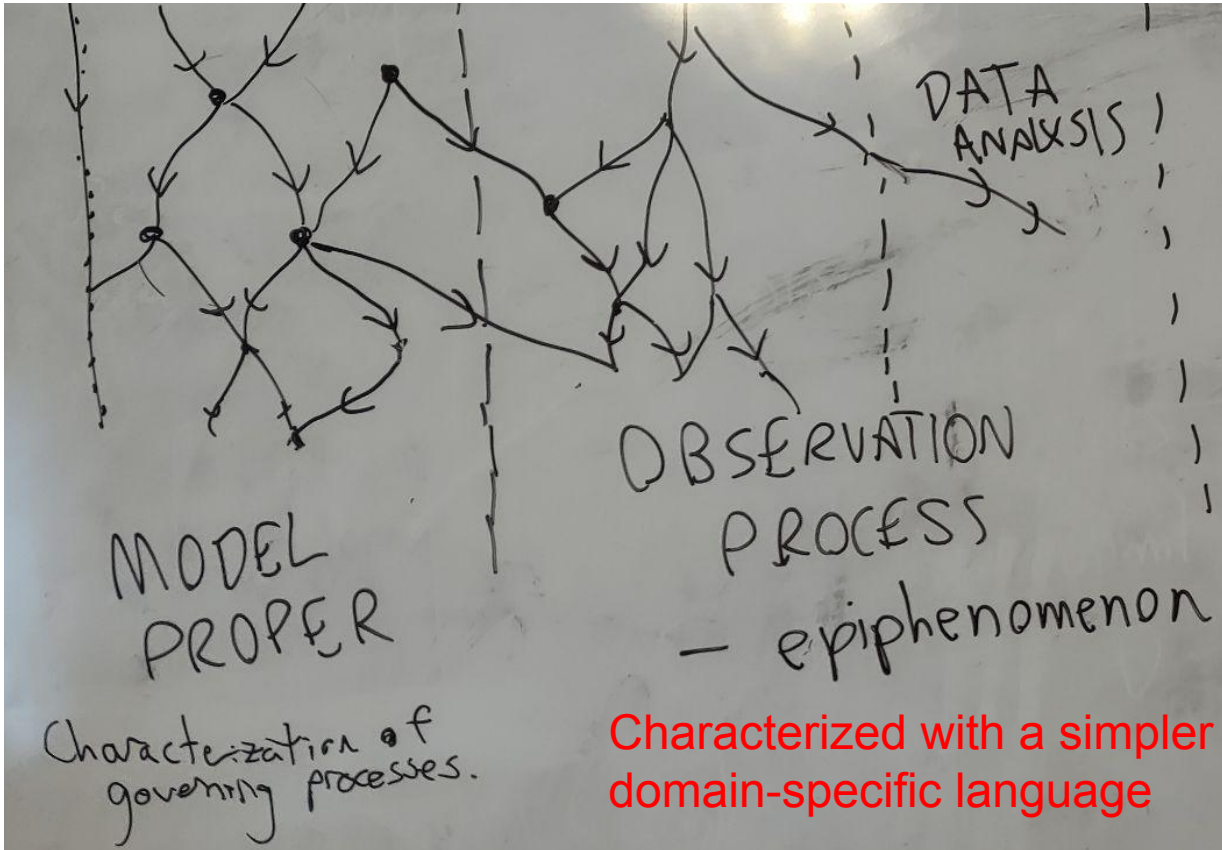
Data Migration Helps Translate with Model Context & Evolution

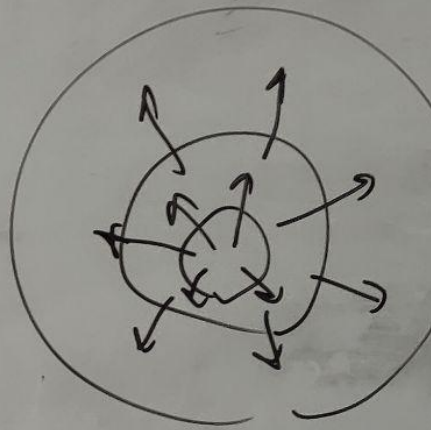


Observation Processes

Observation Processes: Aggregation over the Population & Current State

Observation Processes Separated from Governing Processes





Characterized with a simpler domain-specific language

Characterizing Observation Processes -- Desiderata

- Mathematically transparent
- Declarative
- Modular definition
- Modular use
- Per-run, user-specific enablement without model structure change
- Supports different levels of aggregation
- Supports summaries over time
- Evolves w/model structure (where possible, automatically)
- Affords staged computation
- Computationally efficient
- Accessible to non-modelers
- Exportable

Common Needs

- **Cross-sectional**: Summarizing model current state at varying levels of aggregation (including histogram by agent state in statechart)
- **Longitudinal**: Summarizing or more model events over time (potentially w/context), broken down by time interval

Common Structure to Capture

Both cross-sectional & longitudinal data

- Measurand
- Operator
  - **Predicate measurand**: count, proportion
  - **Continuous measurand**: max/min, average, median, mode
- Breakdown: support varying levels of aggregation
- Subpopulation to consider

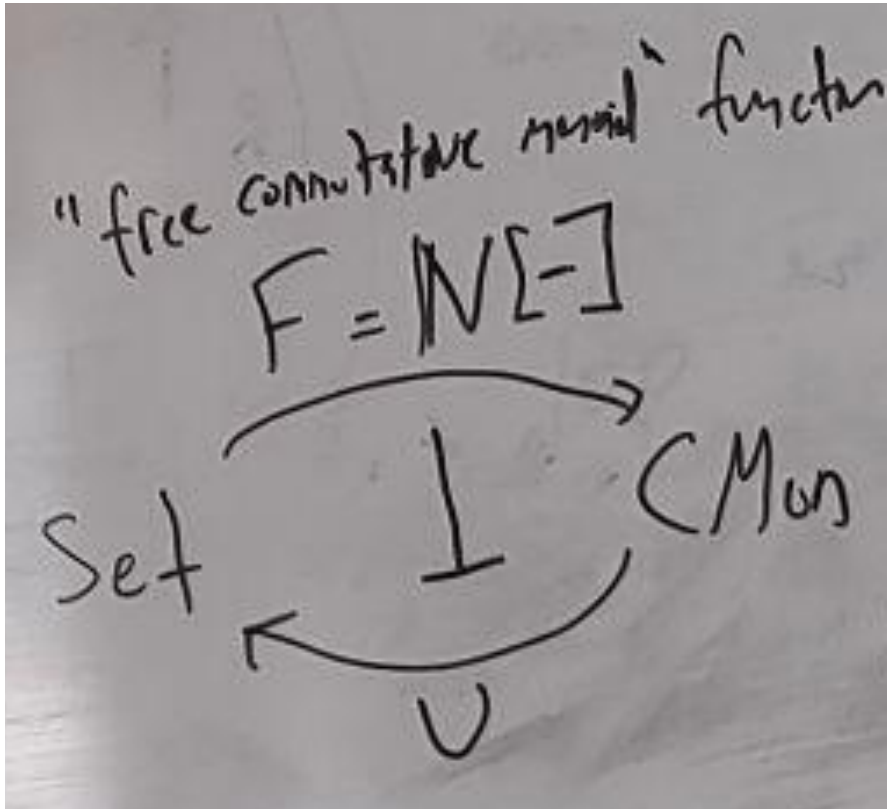
Longitudinal data

- Different time interval to consider

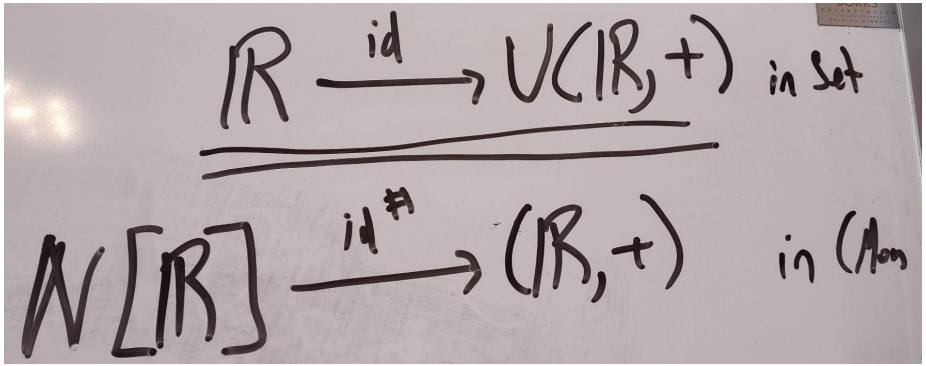
Optimizations

- Incremental computational of quantities
  - Incrementally counting number in state(s) by recording (discrete) in & out transitions (inflows and outflows) for such state(s)
- Hierarchical computation based on subpieces (exploiting topological structure)
  - subintervals of time (Computed per-day incidence contributes to computed per-week incidence)
  - subcategories of cross-tabulation breakdown

There is an adjunction between **Set** and Commutative Monoids



One example of the correspondence

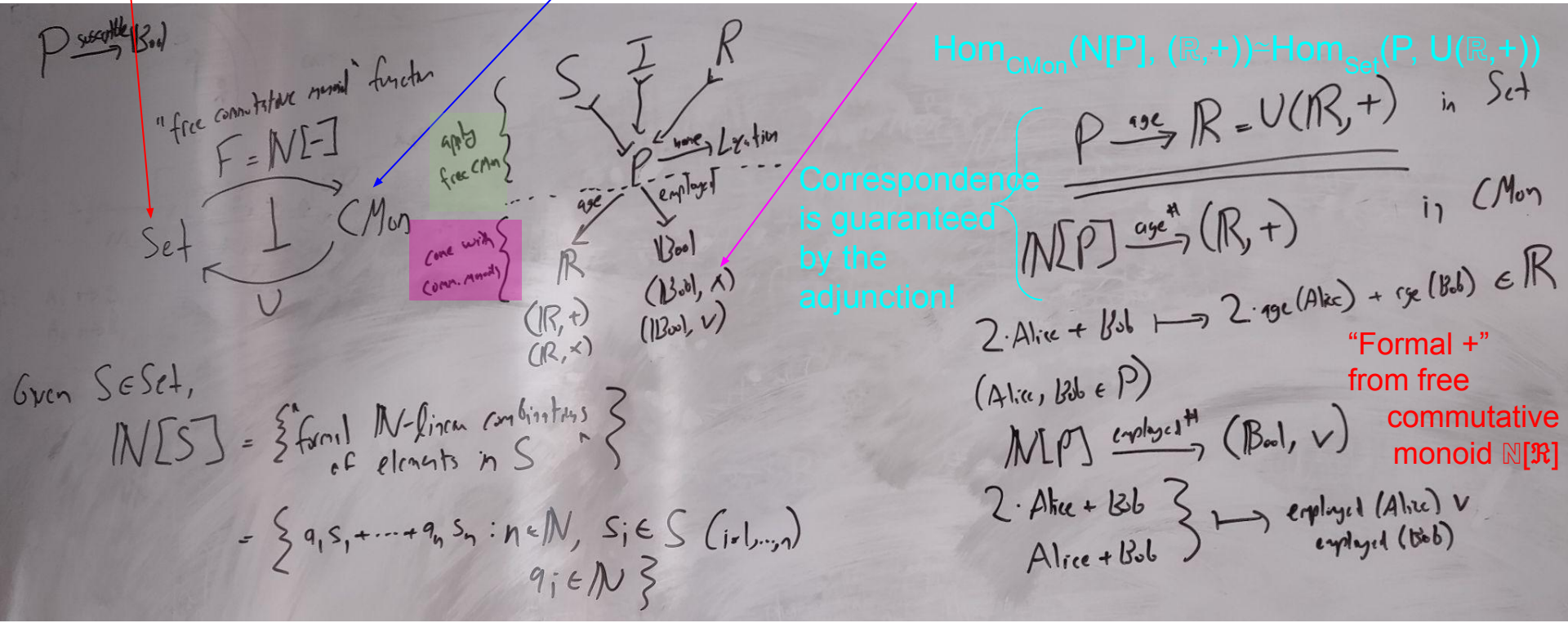


Aggregation 1

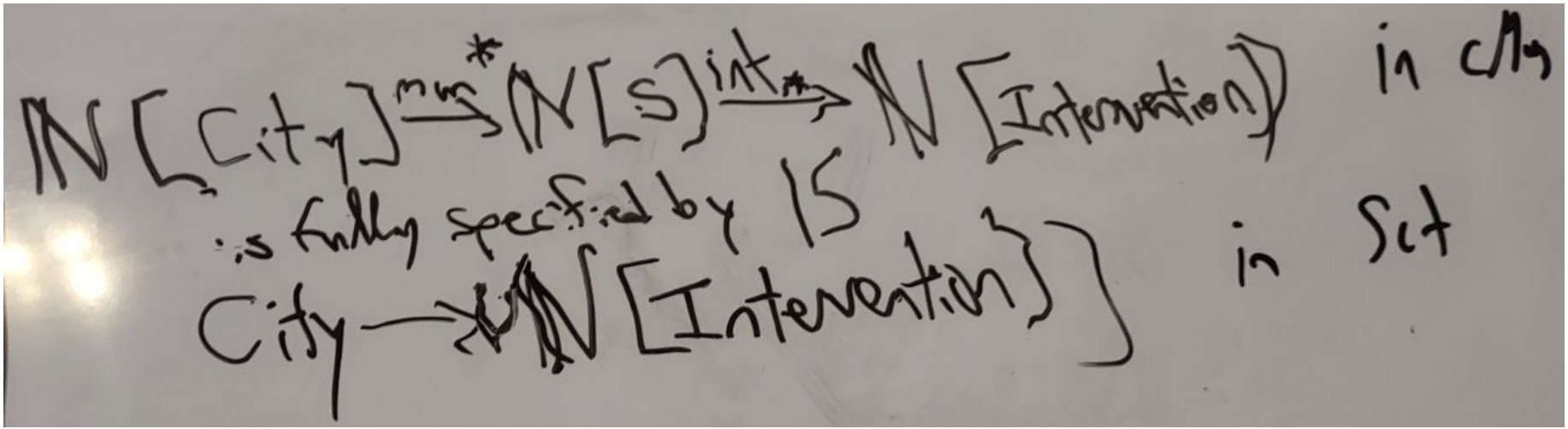
Objects are sets
Morphisms are functions

Objects are commutative monoids
Morphisms are monoid homomorphisms (preserving monoid structure)

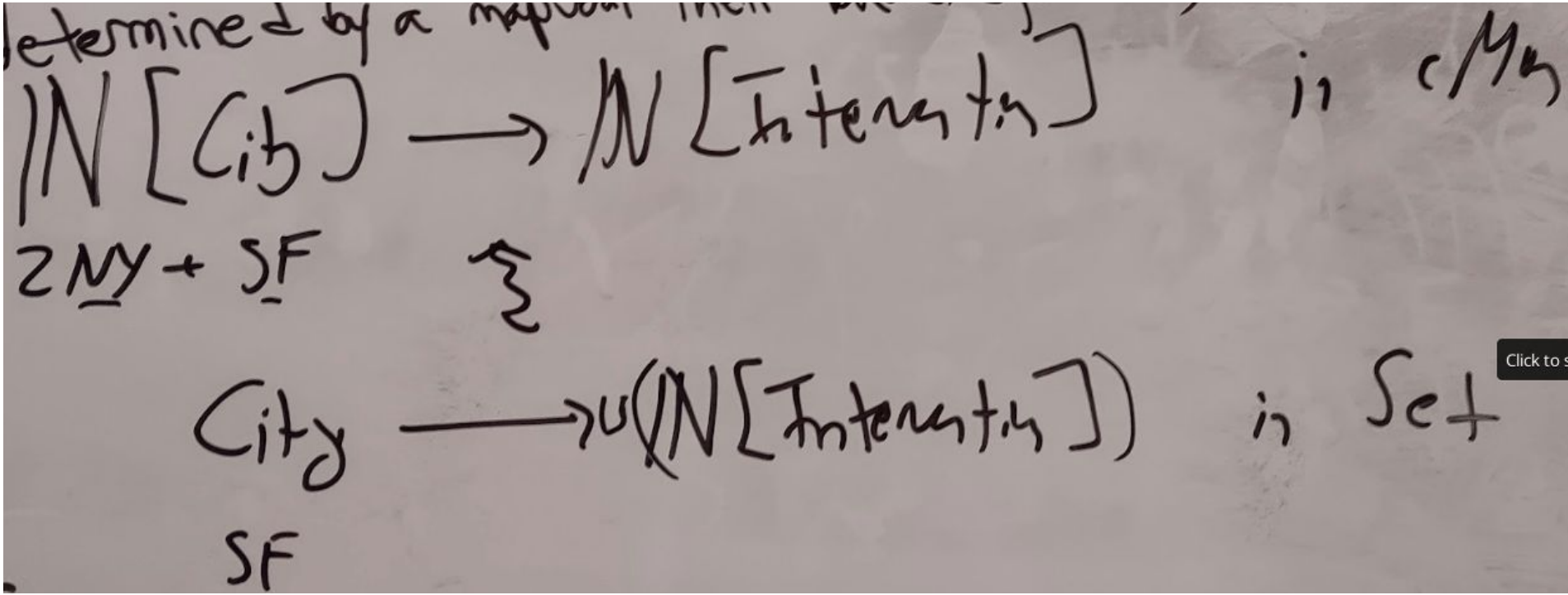
We can choose which of the possible monoids to apply here



Adjunction Guarantees Natural Isomorphism

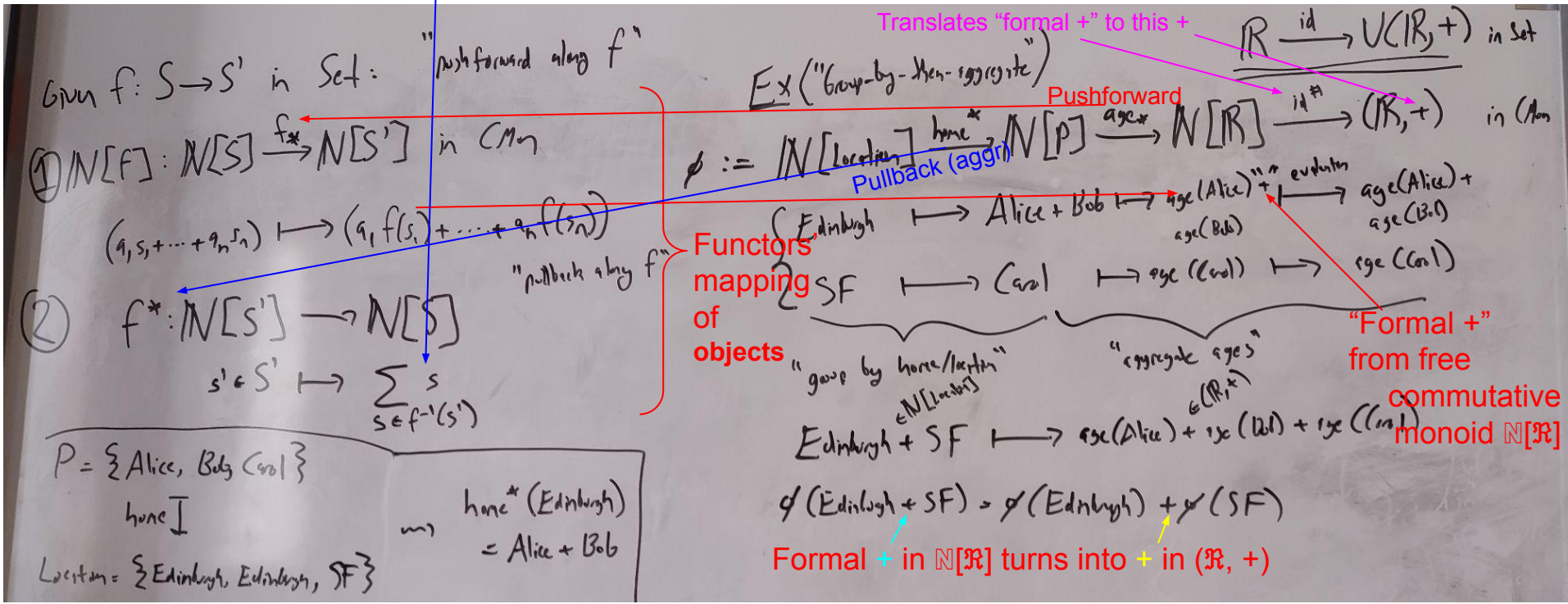


Because $\mathbb{N}[S]$ is a **Free** Monoidal Category, Maps out of it are Completely Determined by its Generators (Bases)

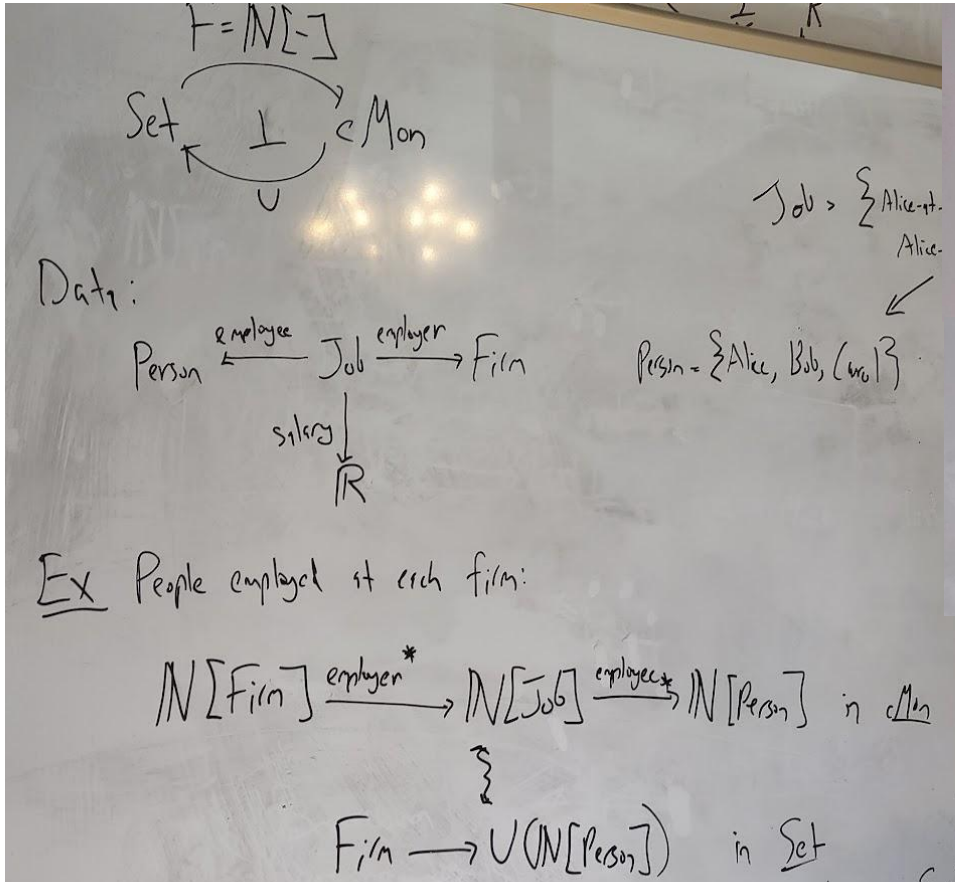


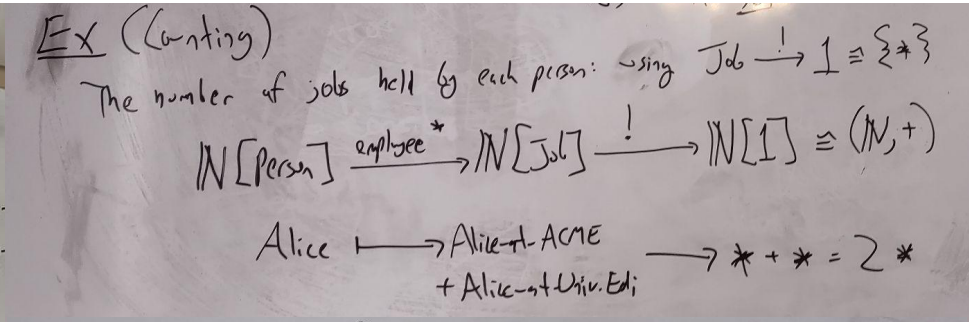
Aggregation 2: Functors from Morphisms

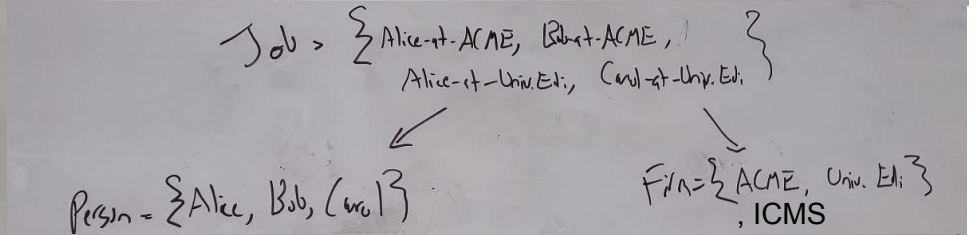
A key component of why this works is that, while in general, for element s' ∈ S' , $f^{-1}(s')$ gives an **entire set**, the order in which these are inserted into is irrelevant, as N[S] is a **commutative** monoid!

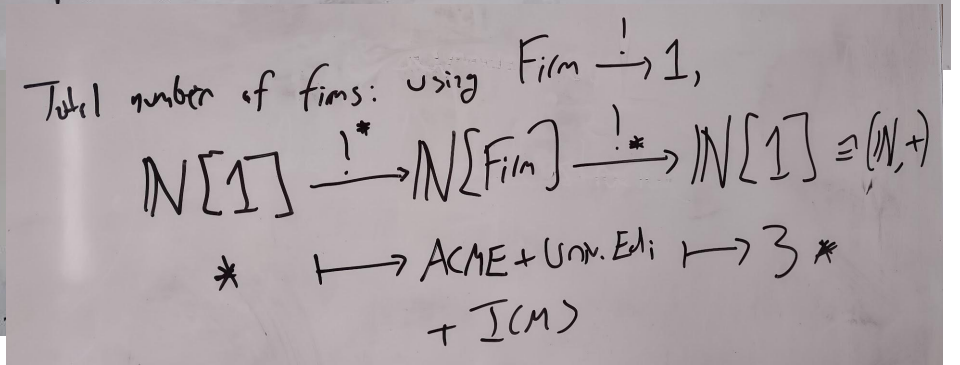


Another example: Counting via the Terminal Object

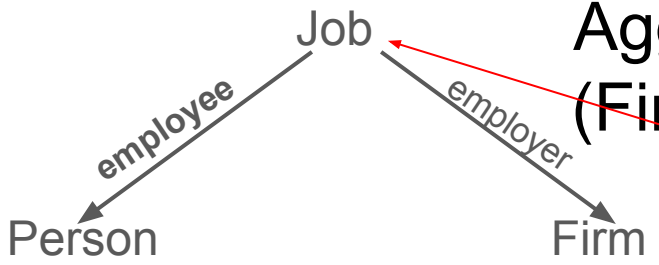








Aggregation of Pairs of (Firm, Person) for Jobs



A **span**

Pushforward

$f : Job \to Job^2$

$\mathbb{N}[Job] \xrightarrow{f_*} \mathbb{N}[Job^2]$

$j \longmapsto f(j)$

$q_1 j_1 + \cdots + q_n j_n \longmapsto q_1 f(j_1) + \cdots + q_n f(j_n)$

Chose $f = \Delta_{Job}$

$f(j) := (j, j)$

Ex (Aggregation of pairs)

Find all employee-employer pairs (with multiplicity):

$\mathbb{N}[1] \xrightarrow{!^*} \mathbb{N}[Job] \xrightarrow{\Delta_*} \mathbb{N}[Job \times Job] \xrightarrow{(employee \times employer)_*} \mathbb{N}[Person \times Firm]$

in **CMon**

(using $Job \xrightarrow{!} 1$)

(using diagonal map $\Delta : Job \to Job \times Job$, $j \mapsto (j, j)$)

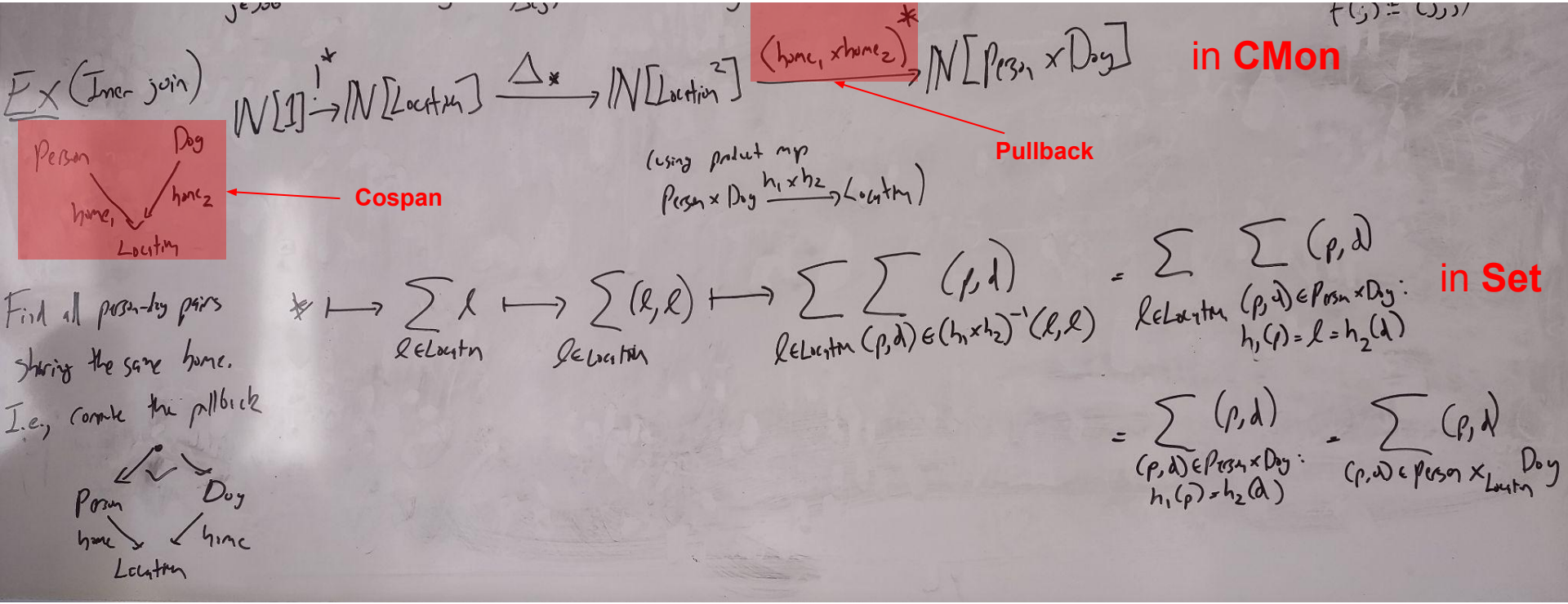
(using product $employee \times employer : Job^2 \to Person \times Firm$)

$* \longmapsto \sum_{j \in Job} j \longmapsto \sum_{j \in Job} \underbrace{(j, j)}_{\Delta(j)} \longmapsto \sum_{j \in Job} (employee(j), employer(j))$

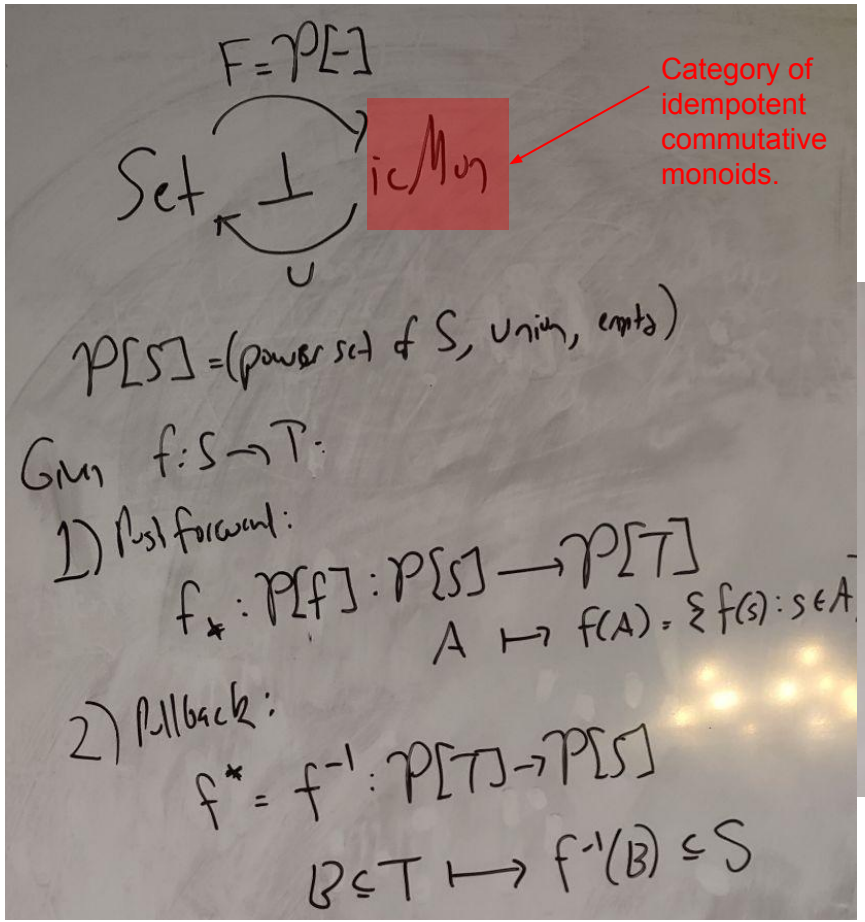
in **Set**

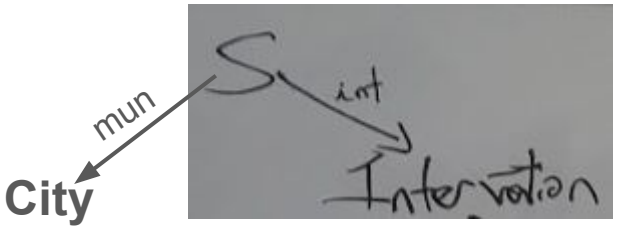
Inner Join Example: Persons & Dogs Sharing Same Home

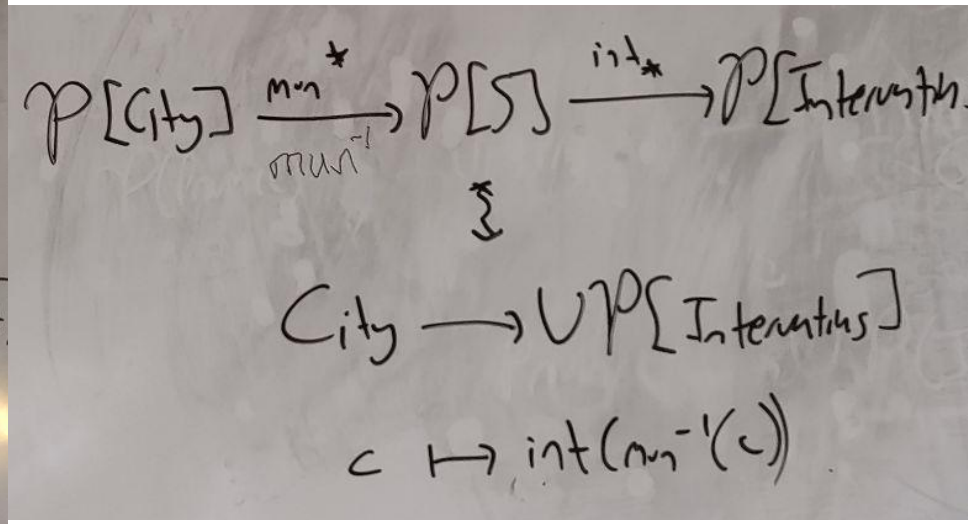
Previous example featured a **span**; this features a **cospan** & is similar to a **conjunctive query**



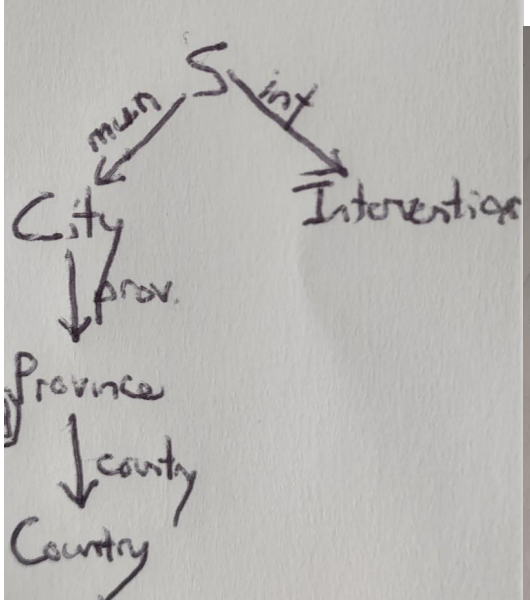
Handling Commutative **Idempotent** Monoids: Case of (Set, ∪)

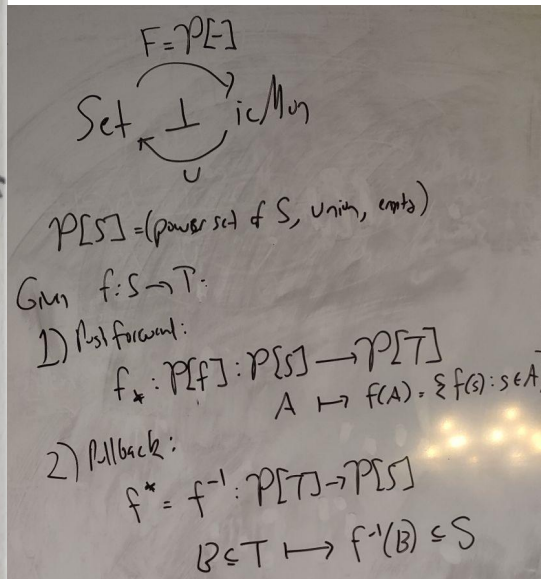


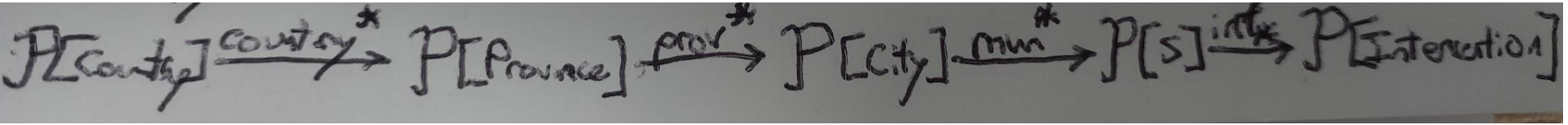


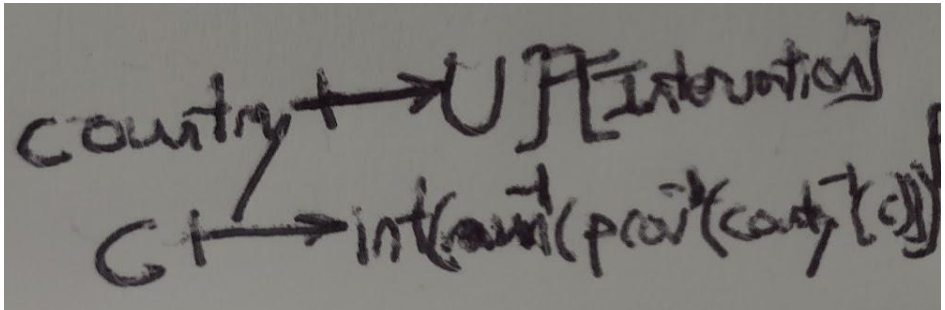


Multi-Level (Hierarchical) Aggregation

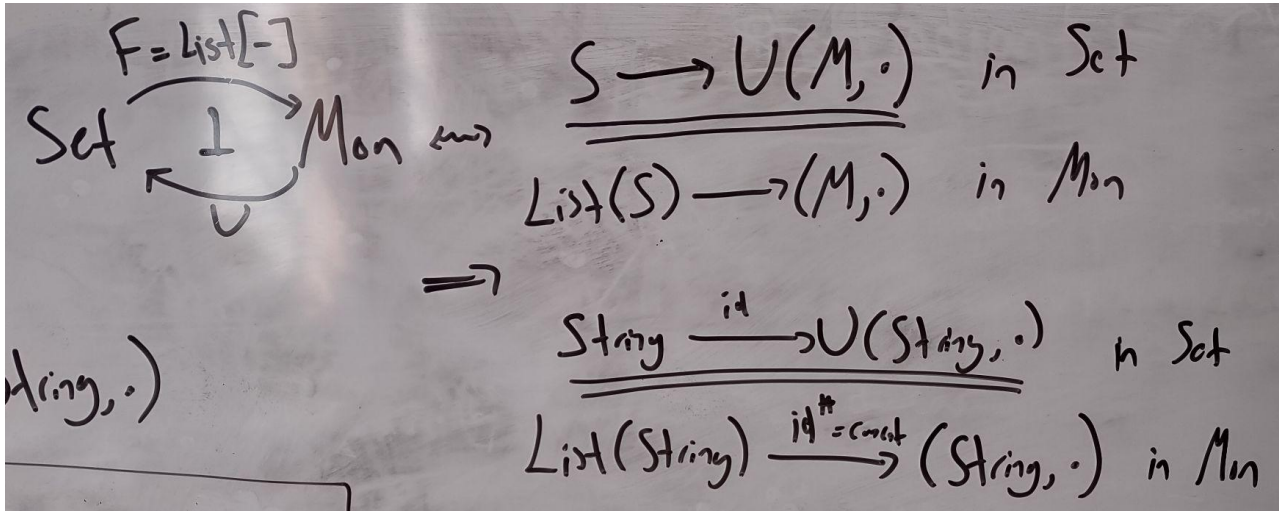




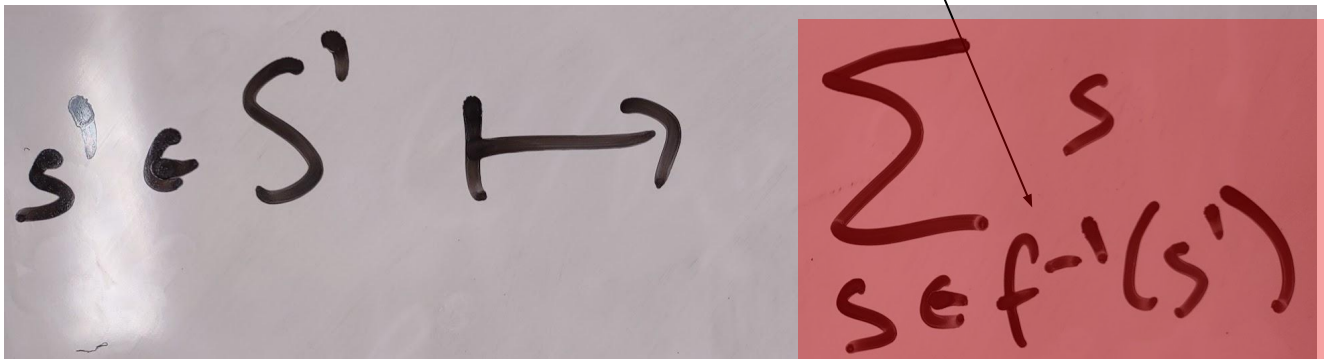




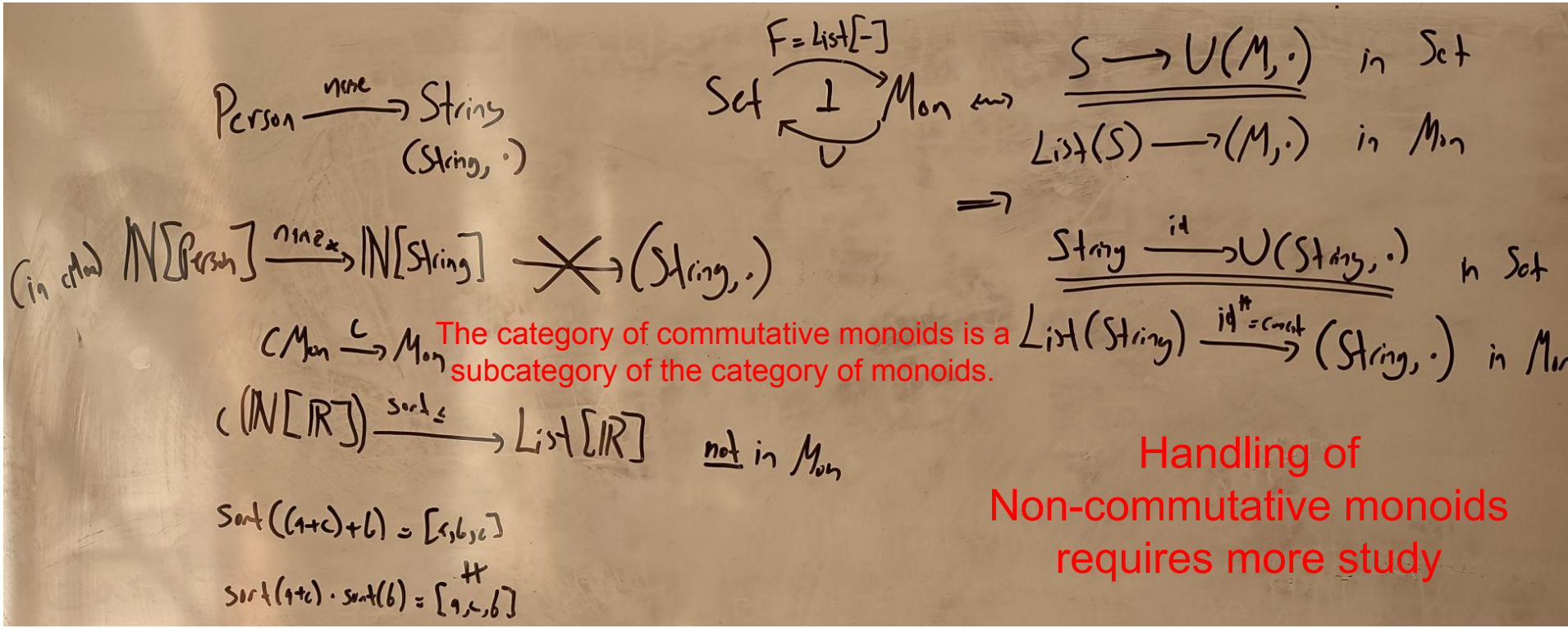
General Non-Commutative Monoids: **Mon**



A key problem is that, in general, for element s' ∈ S' , f^{-1}(s') gives an **entire set**, and there is **no canonical ordering for those** -- and yet order matters for a **non-commutative** monoid!



General Non-Commutative Monoids: **Mon**



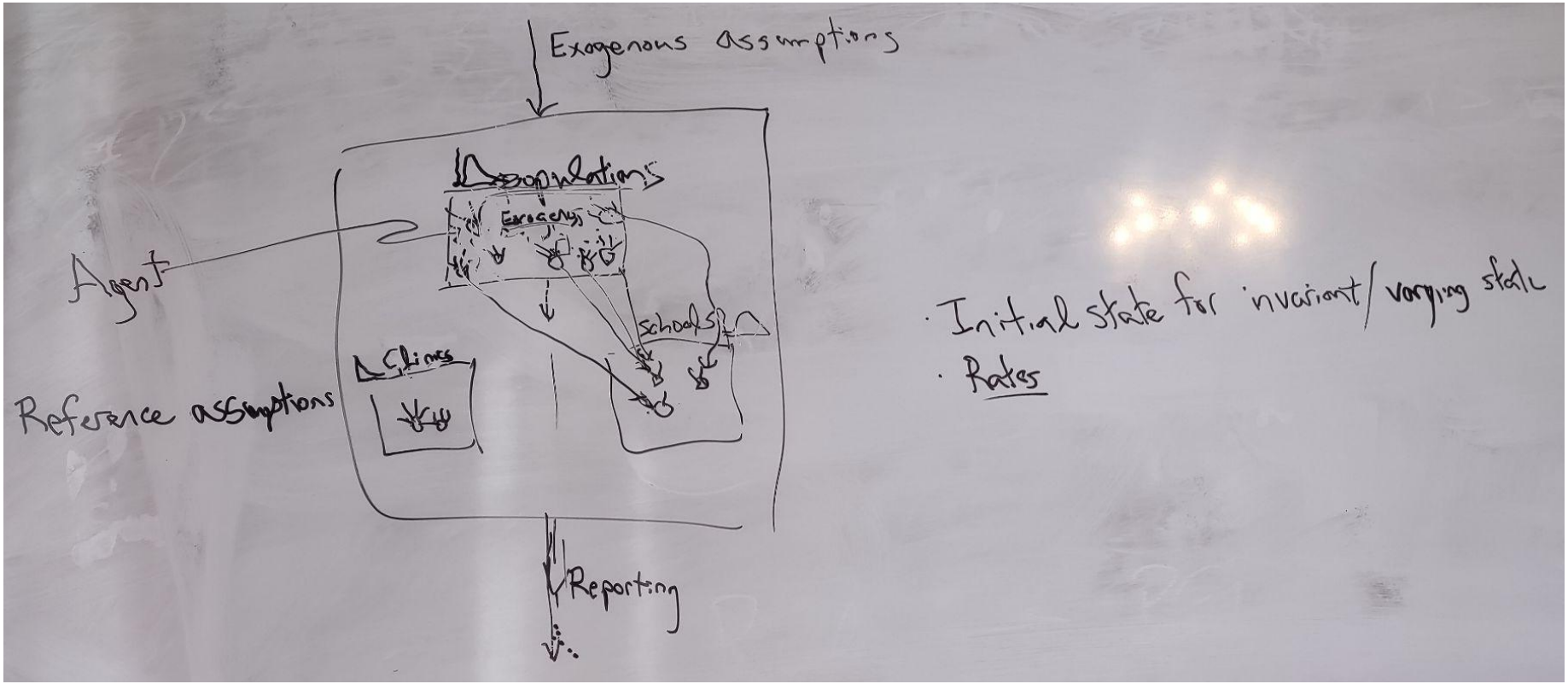
Observation & History Processes: Aggregation Over Time

Strategies for Time Aggregation

- Built-in knowledge of the nested nature of different timespans
- Instrumentation of rules via data migration functors to a schema augmented by
  - “counters” for the types of events
  - For non-counter events (e.g,. log messages), attributes as needed to accrete the information
- Use of temporal sheaves to aggregate up such data over broader time periods
  - Ability to compute values for broader timespans from those for smaller, without being forecedseparately

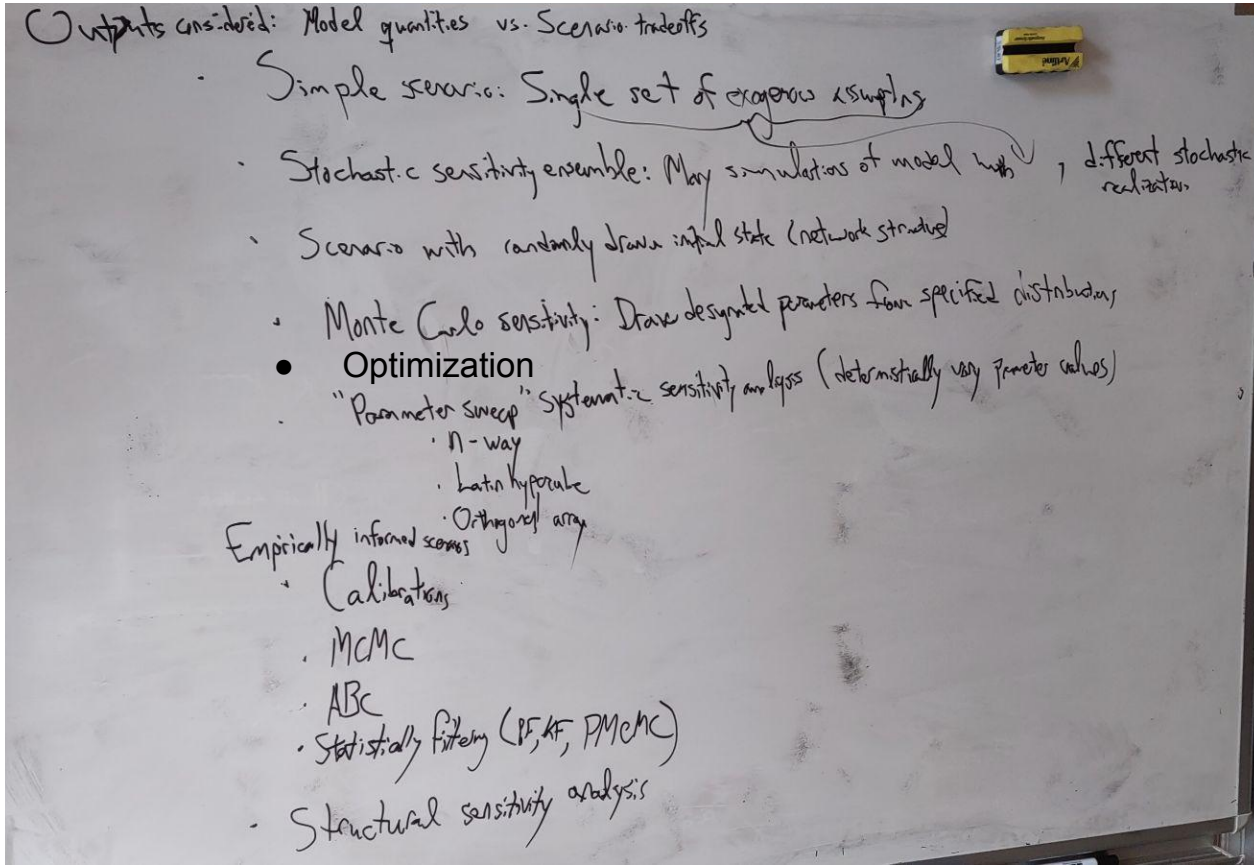
Exogenous Assumptions

Capturing Exogenous Assumptions



Scenarios

Variety in Scenarios



Distinguishing Two Distinct Notions of Hierarchy

- **In model specification**: Hierarchy use of exogenous assumptions: Scope & aggregation level at which model assumptions apply (across entire model, at level of population of agents, at level of a particular agent)
- **During model simulation**: Hierarchy in nesting of agents (e.g., family members are nested in a family context, nested in a neighbourhood, nested in a city, nested in a province, and in a country).
    - This may relate to run-time structures: There would seem to be a choice as to whether this is captured through nesting of structures at runtime vs. capturing inclusions via monomorphisms
    -

Duality Between (Model Assumptions & Behaviour) and (Model Behaviour & Model Observer Processes)

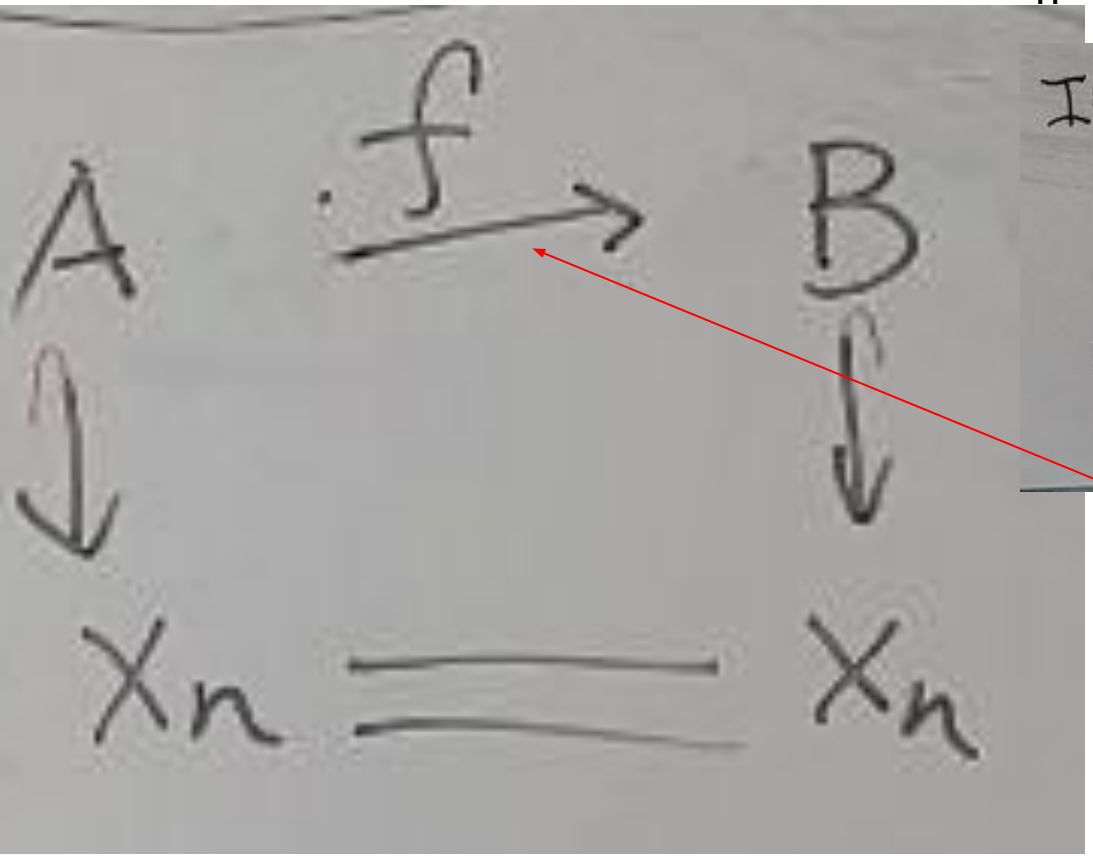
- Model exogenous assumptions affect but aren’t shaped by governing behaviour of a model
- Epiphenomenal observation processes observe but don’t modify governing behaviour of a model

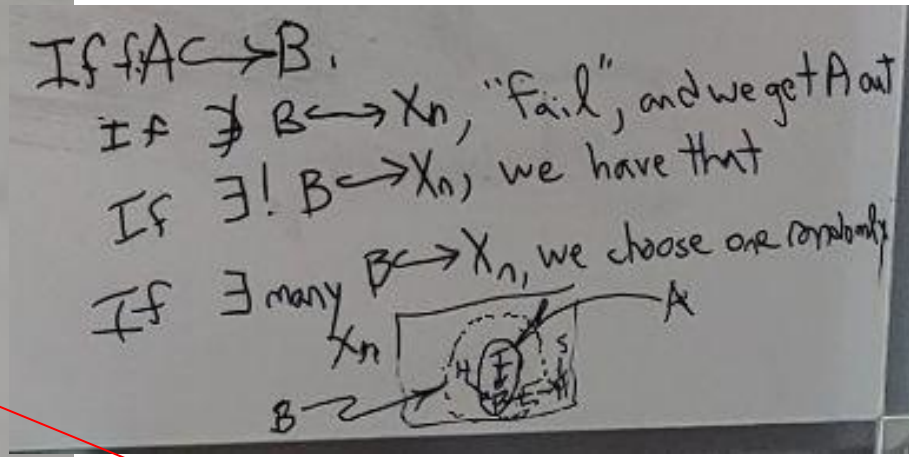
Questions

- Could this be used for non-commutative (and, indeed, non-symmetric) monoids? (e.g., for string concatenation)?
- What is the mechanism to get back the pre-image (elements that map to each value via f)?

Wiring Diagrams & their Operators

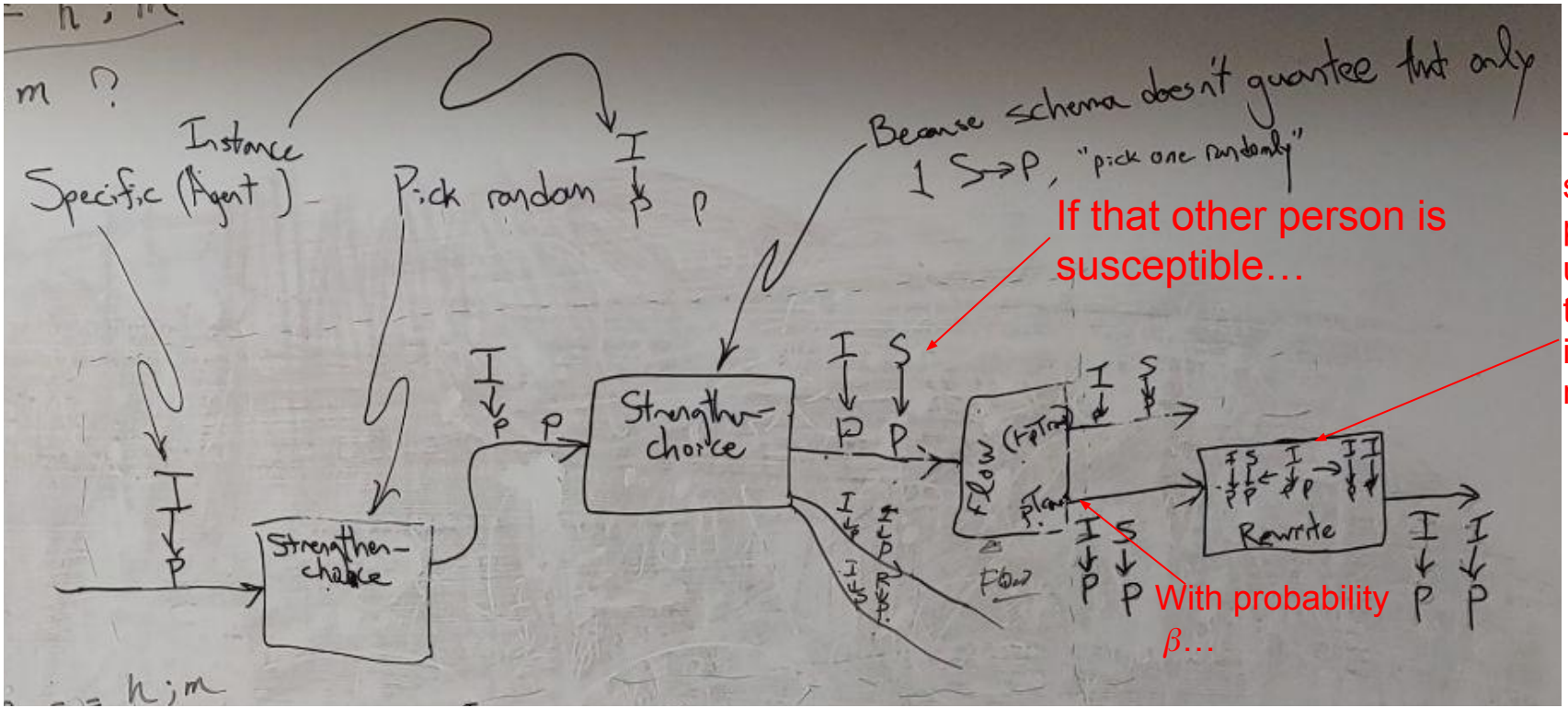
"Strengthen Choice" -- Get a (Randomly Chosen) Instance of this Pullback in X_n, Failing If Doesn't Exist



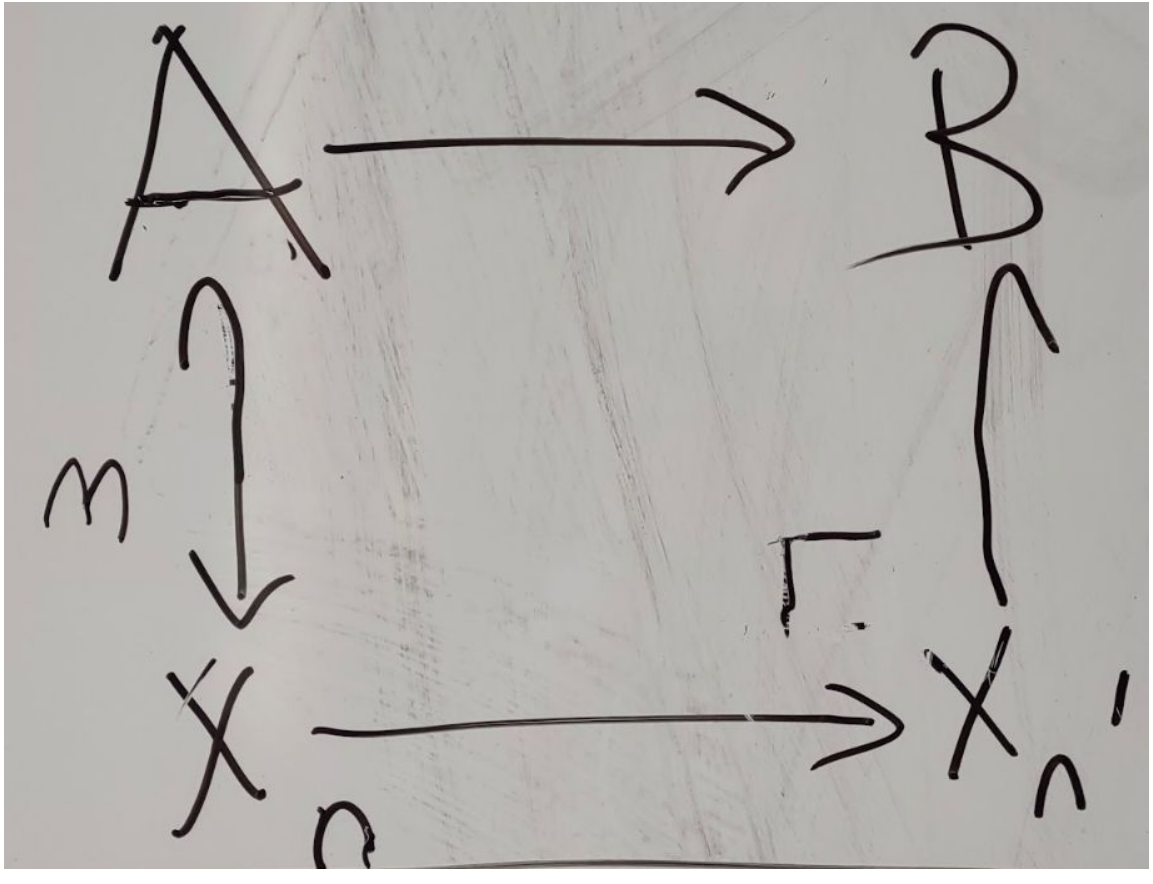


Remember that each morphism here is a **C-Set homomorphism**.

Capturing Random Mixing via Strengthen_Choice, Flow & Rewrite Blocks



“Strengthen Force” -- $X_{n'}$ is the Result of the Pushout

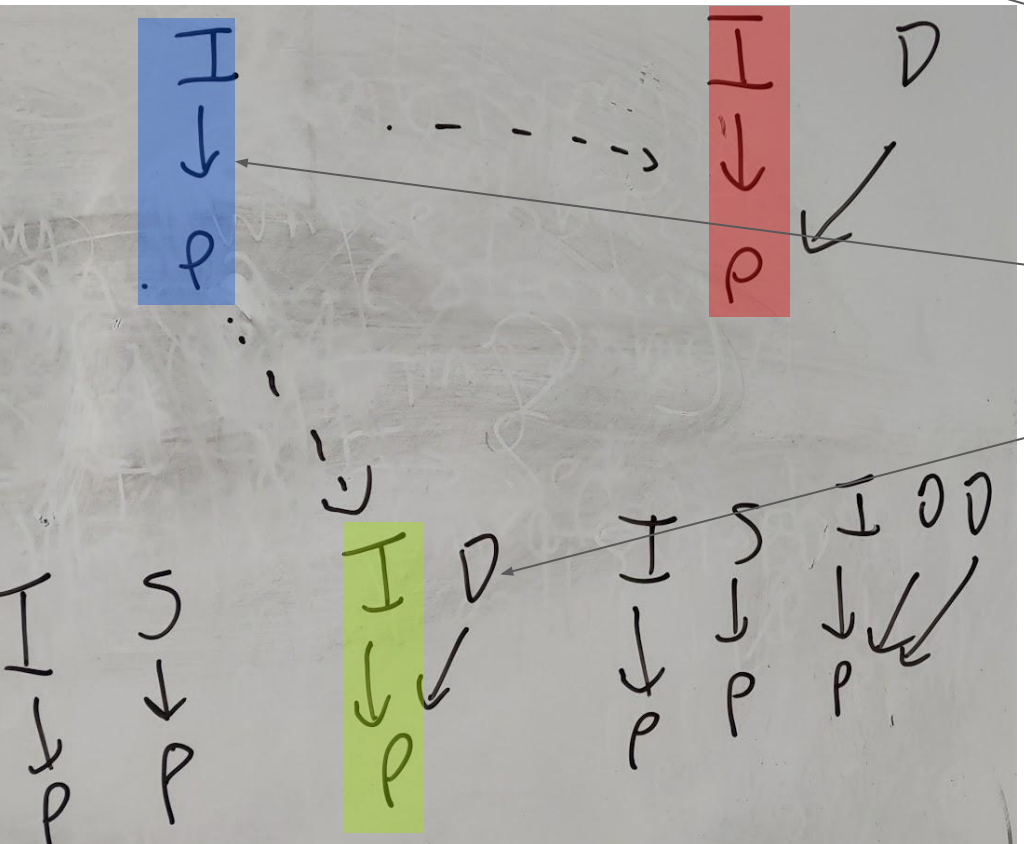


We glue X_n & B together at the image of A in each.

$X_n \rightarrow X_{n'}$ undergoes the same homomorphism seen in A → B;

B in $X_{n'}$ is put in the same context as is A in X_n

Understanding Pushouts of CSets -- Take Pushout of the **Image of the Match Hom** and State X_t Fibered by the Match



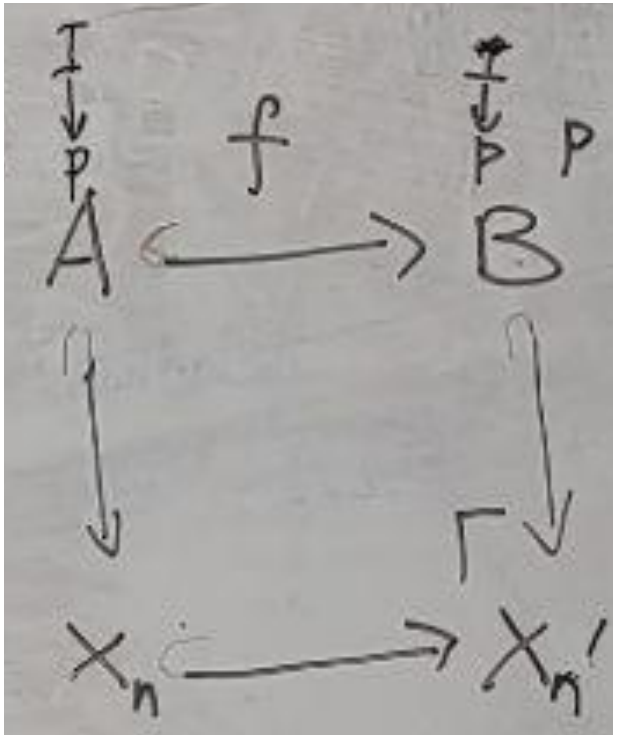
Even though have D in pattern B, and in X_n, because these are not in the homomorphism match I -> P , get an addition of two dogs

We glue X_n & B together at the image of A in each.

$X_n \rightarrow X_{n'}$ undergoes the same homomorphism seen in A → B;

B in $X_{n'}$ is put in the same context as is A in X_n

“Strengthen Force”

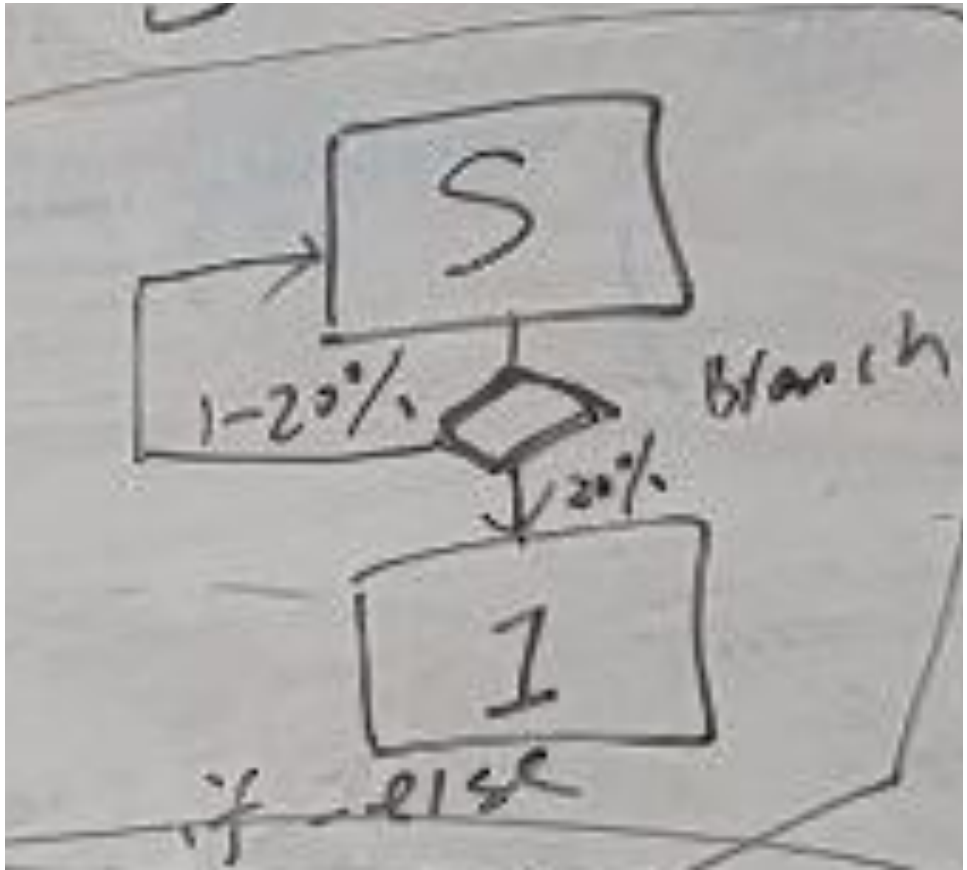


If $f: A \hookrightarrow B$, then X_{n+1} contains

$B + X_n$ glued around a common a ie.,

with a as a common subpiece

Capturing Branching Logic with a “Flow” Block



Forthcoming Theoretical Foci

Areas of Theory to be Further Elucidated

- Model composition
- Logically linked aspects of model inputs/outputs
  - Exogenous quantities/Parameters
  - Scenarios (single realizations, ensembles, sensitivity analyses, calibrations, optimizations, ABC, PF, PMCMC, etc.)
- Integration of numerical integration and occurrence of stochastic transitions
- Mapping to different semantics
- Handling stochastic differential equations (rather than strictly deterministic ODEs)
- Hierarchical models
- Clearer formulation of hybrid continuous/discrete interaction

Implementation:
Key Mechanisms for Running

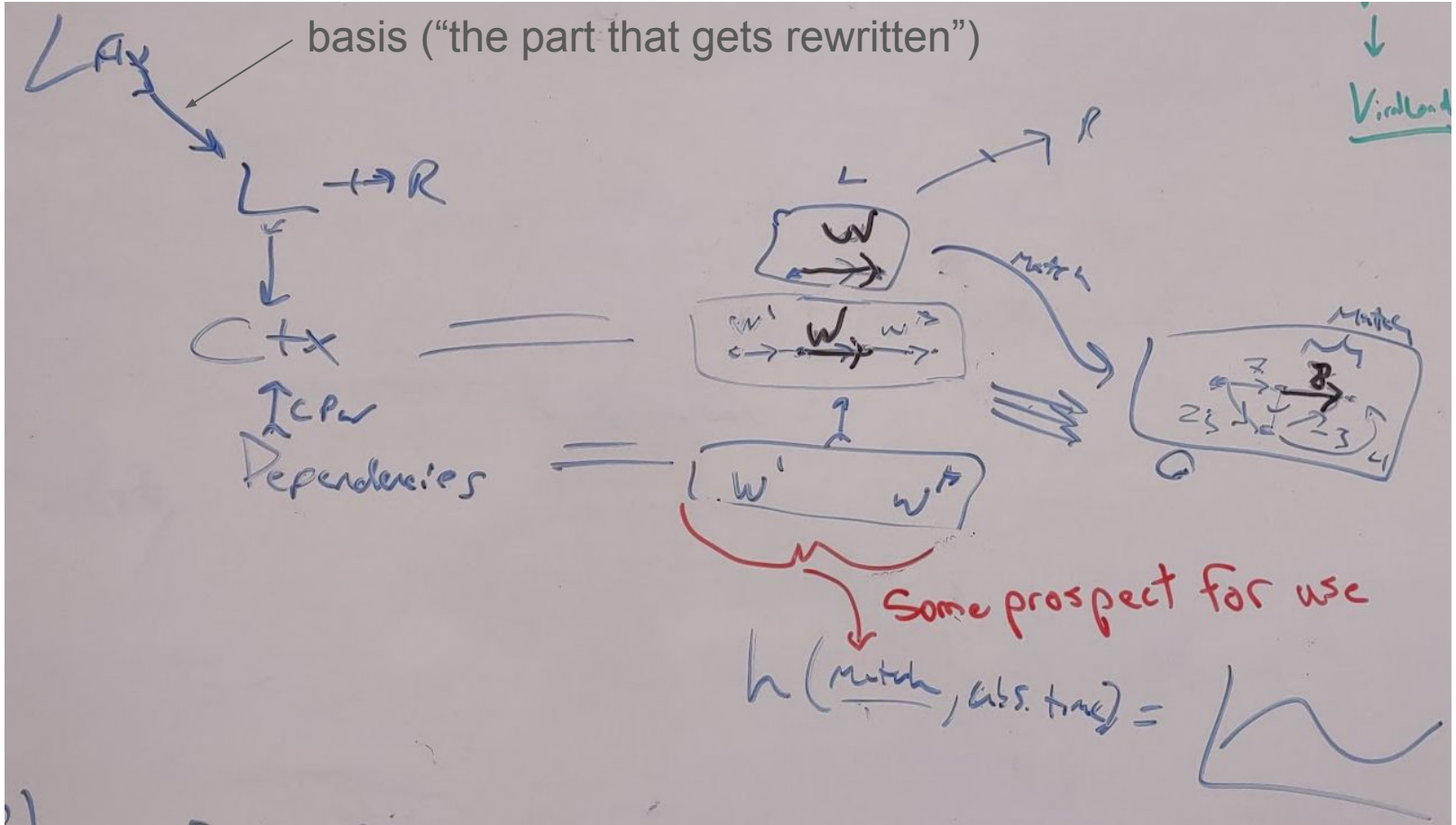
Central Implementation Abstractions

- ABM -- keeps rules (ABMRules), ABMFlows, index of names of ints
- ABMRule -- “A stochastic rewrite rule with a **dependent hazard rate**”
- AbsTimer & types thereof (including stateless, time-invariant AbsHazard, both continuous & discrete)
- AbsDynamics -- for ODEs
- ABMFlow
- Sampler
- PatternType (Regular, Empty, Representable)
- RuntimeABM -- the ABM runtime structures: current state & time, cum. events, sampler, rng, rule names=>int & ODE structures
- Traj

ABMRule -- Algebraic Rewriting Rule Plus Timer (Type)

```
"""
A stochastic rewrite rule with a dependent hazard rate

A basis is a subobject of the pattern of the rule for which we want a timer
per match. By default, the basis → pattern map is just id(pattern).

NB: A "basis" corresponds to the map from "L_fix" in the Edinburgh discussions
(See "Lfix limits what Changes Are of Concern in Pattern Matching" in
AlgebraicABMs slides)

"""

@struct_hash_equal struct ABMRule
  rule::Rule
  timer::AbsTimer
  basis::Maybe{ACSetTransformation}
  name::Maybe{Symbol}
  pattern_type::PatternType
  ABMRule(r::Rule, t::AbsTimer; basis=nothing, name=nothing) =
    new(r, t, basis, name, pattern_type(r, is_exp(t)))
end
```

Extracting Further Information on Selected Dependencies



ABM -- Static Features of the Agent-Based Model

```
"""
An agent-based model.
"""*
#*** Key structure for an ABM -- rules, continuous dynamics, names (what are these?)
#*** ASK: Are the names associated with the variables in the patterns in the rewrite rule?
@struct_hash_equal struct ABM
    rules::Vector{ABMRule}
    dyn::Vector{ABMFlow}
    #*** A map from the name of a rule to its index in "rules".
    names::Dict{Symbol, Int}
    function ABM(rules, dyn=[])
        names = Dict(n=>i for (i,n) in enumerate(nameof.(rules)) if !isnothing(n))
        new(rules, dyn, names)
    end
end
```

Key Summary

- **ABM** keeps track of key static data structures (e.g., rules with timers [hazard rate generators])
- **RuntimeABM** keeps track of dynamic data structures (current state, time, scheduler) and initializes the state & event queue
- **Timers** provide ways of generating hazard rates&thus potentially draw an event time
- The **run!** function actually performs successive handling of events, and accumulating a trajectory (via the Traj structure) of output, calling **get_match** with the pattern type
- The **sampler** basically keeps track of events (each at a time drawn from a hazardDistribution, and identified merely by a key, which commonly relates them to a rule and then necessary information about them to process the event).
- **Coproducts of representables** provide key optimizations, because Yoneda Lemma lets us know how many we have at any one time, and instead of enqueuing a n corresponding rules for drawing a hazard & thus events, if we have a stateless, memoryless time invariant, rule @rate λ, we successively sample the first at rate $n\lambda$, and consider as coming from a random tuple of values from each representable.

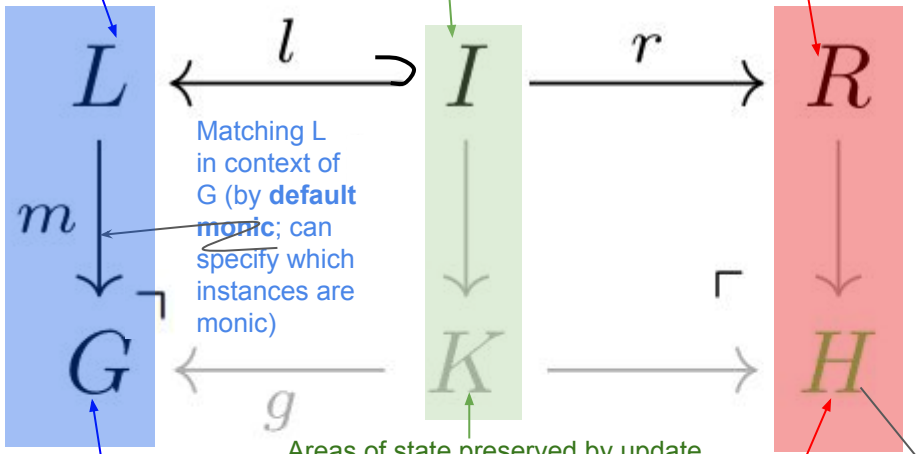
Rewriting Rules Triggered via Hazard Rates

Basic Rewriting Context

Match (everything here but not in I is deleted)
For attributes, *variables* can be included, so as to capture the value of that attribute & use in R

Features in L retained in R

Replacement (everything here but not in I is added)



Matching L in context of G (by **default monic**; can specify which instances are monic)

Areas of state preserved by update

All objects are C-Sets, all arrows are C-Set homomorphisms

xmap

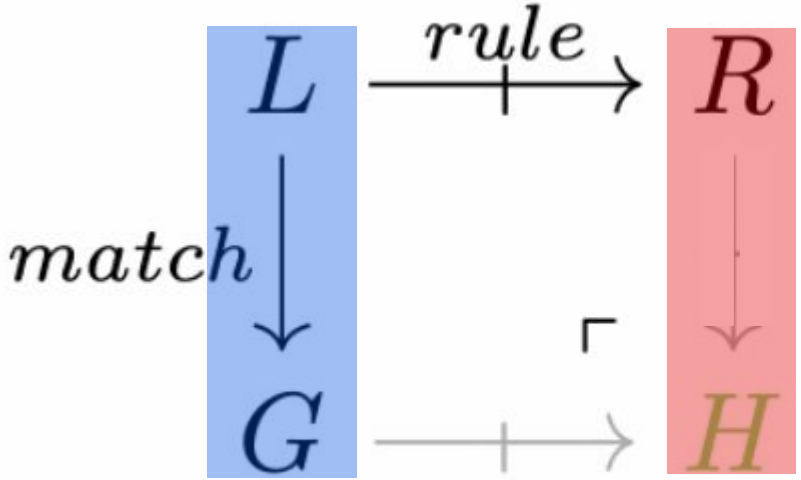
H'

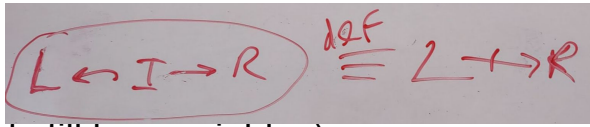
Original state (X_t)

Updated state (X_{t+1}; incorporates R but still has variables)

Updated state (X_{t+1}'; incorporates R but and all variables replaced w/values)

Considered as Partial Maps





Images adapted from Brown, K., Patterson, E., Hanks, T. and Fairbanks, J., 2022. Computational Category-Theoretic Rewriting. In International Conference on Graph Transformation (pp. 155-172). Springer, Cham. https://arxiv.org/pdf/2111.03784.pdf

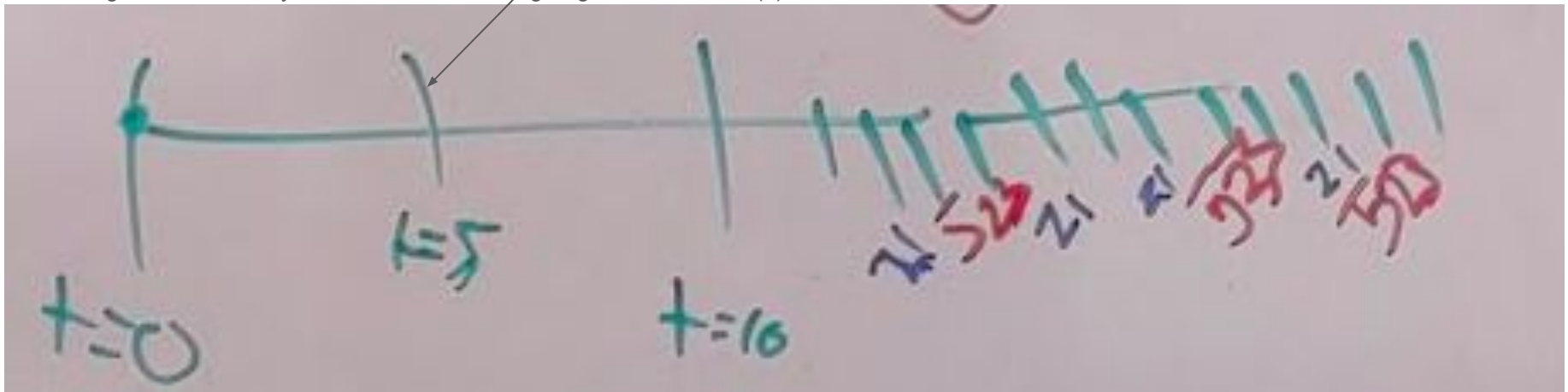
Underlying Rewriting Mechanisms -- Match & Schedule

- Especially because there is no notion of “each agent” needing updates, we need to **match** pattern to find situations that can trigger dynamics via matching (e.g., that an agent is in a given state)
- These situations do not generally need to fire immediately -- for example, if an agent has entered a state; it could exhibits a hazard rate or timeout transition for leaving
- Using hazard rates (not PDF) lets us adjust firing rate as state evolves
- [General case] For *state-dependent* events (hazard rates), we avoid **ongoing** need to reschedule by incrementally computing Bernoulli draws for the probability of firing, for a certain timestep
- For *state-independent* hazard, we **schedule** trigger, Check if match still applies when fires
- By default, if match still remains after rewriting, create new event in the schedule
    - A key exception -- when leaving state in a statechart (know that won’t match again)

Hybrid (Static-Dynamic) Scheduling
The **Supporting** Role of an Event Schedule (Queue)

- **Model state independent timing:** Pre-scheduled (both memoryless & memoryful)
- **Timing dependent on Model state:** Events whose scheduling is state dependent
- Regardless of type, scheduled events can be descheduled if they are preempted eg
  - Agent dies
  - Agent leaves because of matching a different pattern

Event is associated with a “key” uniquely identifying it (e.g., rule id and any required information on the particular morphism m with which it is associated). The timing is determined by a draw from a function giving the hazard rate (?).



Functions

- **pattern_type** -- determine type of a pattern (currently, EmptyP, RegularP, RepresentableP)
- **get_hazard** -- gets the hazard for a pattern, returning a UnivariateDistribution. Note that this includes taking into account the faster sampling for state-independent, memoryless, time-invariant coproducts of representables
- **get_match** -- gets a match for the given pattern type and event information (schedule key). This handles any basis (L_{fix}), and can draw randomly from matches for representables
- **pops!**(rt::RuntimeABM) -- “Pop the next random event, advance the clock”
- **run!**

Key Competing Clocks Functions

- For a sampler, competing clocks maintains a set of events
- **enable!**(sampler, key, hazardDistr, ?, ?, rng): Creates an event with a draw from that key
- **next(**sampler, tnow, rng): Gets the next event (pure function -- no change to queue)
- **disable!(**sampler, key, time): simply delete something from the queue
- **transition_entry** -- dictionary (maps from key to an Int used in a queue); useful for asking if this haskey of a key

Helper functions in Upstream.jl

- **pops!**(sampler, rng, time): collates all events from the queue occurring at the next time, deletes such events (via pop!) and returns pair of next time & vector of the keys at that time
- **pop!**(sampler, rng, time): gets & deletes first event from the queue via calling next & disable! in competing clocks, and returns pair of next time and that event's key (defined in Upstream.jl)

Rewrites

- Multiple events (and thus rewrites) at specific points in time have to superimpose their effects, which they do so through pullback mechanisms together with composition
-

Major Concepts

```
"""
An agent-based model.
"""*
#*** Key structure for an ABM -- rules, continuous dynamics, names (what are these?)
#*** ASK: Are the names associated with the variables in the patterns in the rewrite rule?
@struct_hash_equal struct ABM
  rules::Vector{ABMRule}
  dyn::Vector{ABMFlow}
  #*** A map from the name of a rule to its index in "rules".
  names::Dict{Symbol, Int}
  function ABM(rules, dyn=[])
    names = Dict(n=>i for (i,n) in enumerate(nameof.(rules)) if !isnothing(n))
    new(rules, dyn, names)
  end
end
```

Representation of Current State of ABM

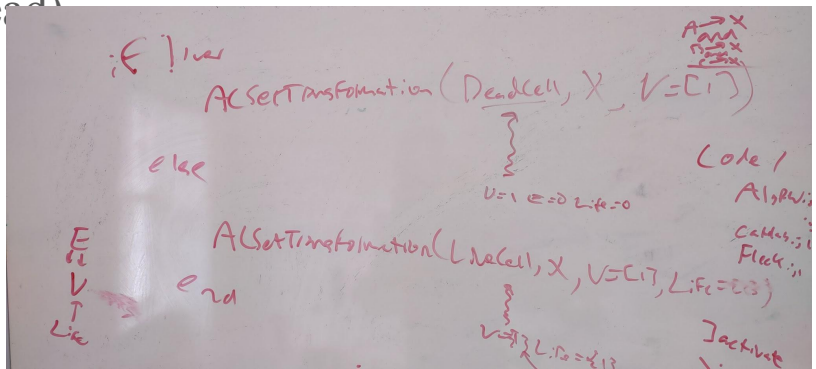
```
@struct_hash_equal struct ABM
  rules::Vector{ABMRule}
  dyn::Vector{ABMFlow}
  ABM(rules, dyn=[]) = new(rules, dyn)
end
```

NB: *clocks* is always in 1-to-1 correspondence with abm.rules.

```
"""
Data @struct_hash_equal structure for maintaining simulation information while running an ABM
"""
mutable struct RuntimeABM
  state::ACSet
  const clocks::Vector{AbsHomSet}
  tnow::Float64
  nevent::Int
  const sampler::SSA # stochastic simulation algorithm
  const rng::Distributions.AbstractRNG
  function RuntimeABM(abm::ABM, init::T; sampler=default_sampler) where T<:ACSet
    rt = new(init, init_homset.(abm.rules, Ref(init), Ref(additions(abm))),
             0., 0, sampler(), Random.RandomDevice())
    for (i, homset) in enumerate(rt.clocks)
      kv = homset isa ExplicitHomSet ? pairs(homset) : [nothing => create(init)]
      for (key, val) in kv
        haz = get_hazard(val, 0., abm.rules[i].timer)
        enable!(rt.sampler, i => key, haz, 0., 0., rt.rng)
      end
    end
    rt
  end
end
```

```
"""Create a context of n living neighbors for either a dead or alive cell"""
function living_neighbors(n::Int; alive=true)::ACSetTransformation
  X = LifeState(1)
  alive && add_part!(X, :Life, live=1)
  for _ in 1:n
    v = add_part!(X, :V)
    add_part!(X, :Life, live=v)
    add_edge!(X, v, 1)
  end
  homomorphism(alive ? LiveCell : DeadCell, X; initial=(V=[1],))
end;
```

- we have a central cell (as created by LifeState(1))
- We create n living neighbours of this central cell, each marked live and with an edge going to it from the central cell
- Because this is used in a morphism L <- I, we have to return an ACSetTranformation from I (given as live or dead)
- The last line is essentially equivalent to this: *initial* basically constrains the map to ensure that we are making the morphism to the central cell



Trajectories of Event-Vectors and the X_t + $X_{t'}$ ACSets

The apex is the span the K in the match, but rather unimportant

```
"""
A trajectory of an ABM: each event time and result of `save`.
"""
@struct_hash_equal struct Traj
  init::ACSet
  events::Vector{Tuple{Float64, Int, Any}}
  hist::Vector{Span{<:ACSet}}
end

Traj(x::ACSet) = Traj(x, Pair{Float64, Any}[], Span{ACSet}[])

function Base.push!(t::Traj, τ::Float64,rule::Int, v::Any, sp::Span{<:ACSet})
  push!(t.events, (τ, rule, v))
  isempty(t.hist) || codom(left(sp)) == codom(right(last(t.hist))) || error(
    "Bad history \n$(codom(left(sp))) \n!= \n$(right(last(t.hist)))"
  )
  push!(t.hist, sp)
end
```

Running the ABM

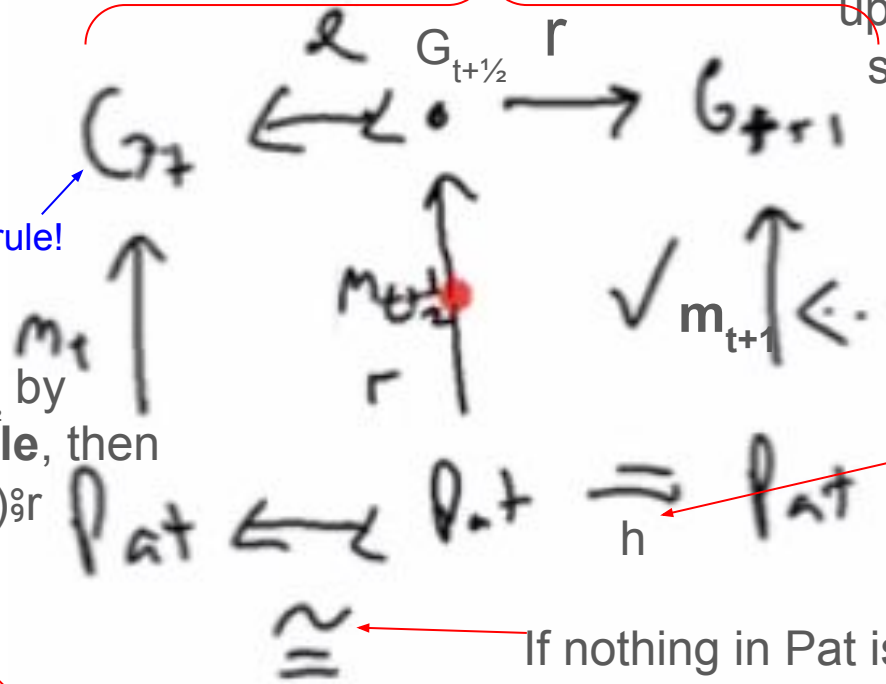
```
function run!(abm::ABM, rt::RuntimeABM, output::Traj;
              save=deepcopy, maxevent=MAXEVENT, maxtime=Inf)

  # Helper functions that automatically incorporate the runtime `rt`
  log!(rule::Int, sp::Span) = push!(output, rt.tnow, rule, save(rt.state), sp)
  disable!'(key::Pair) = disable!(rt.sampler, key, rt.tnow)
  disable!'(i::Int) = disable!'(i => nothing)
  function enable!'(m::ACSetTransformation, rule::Int, key=nothing)
    haz = get_hazard(m, rt.tnow, abm.rules[rule].timer)
    enable!(rt.sampler, rule => key, haz, rt.tnow, rt.tnow, rt.rng)
  end

  # Main loop
  while rt.nevent < maxevent && rt.tnow < maxtime
    # get next event + update clock time
    which = next(rt)

    if isnothing(which)
      @info "Stochastic scheduling algorithm ran out of events"
      return output
    end

    # Unpack data associated with the current event
    event::Int, key::Maybe{KeyType} = which
    rule::ABMRule, clocks::AbsHomSet = abm.rules[event], rt.clocks[event]
    rule_type::Symbol = ruletype(rule) # DPO, SPO, etc.
```

Updating after each update while handling simultaneous updates

```
# bring the match 'up to speed' given the previous (simultaneous) updates
for (l, r) in first.(update_data)
  m = pull_back(l, m) · r
end
```

Span captures partial map $G_t \nrightarrow G_{t+1}$ after updates by series of simultaneous rules

NB: Despite superficial similarity, this is not itself depicting a rewrite rule!



If nothing in Pat is deleted when transitioning $G_t \rightarrowtail G_{t+½}$ by **previous simultaneous rule**, then have $m_{t+½}$ = pull_back(ℓ, m_t)⨾r

Otherwise, there there is no hom of Pat into the apex

This is the match that we seek to the match AFTER this previous executed rule (ℓ,r). B/c h is id & square commutes, if Pat still ∃ after (ℓ,r), m_{t+1}=pull_back(ℓ, m_t)⨾r

If nothing in Pat is deleted when transitioning $G_t \rightarrowtail G_{t+1}$, then this monic arrow will be an isomorphism, otherwise, it will not be

pull_back(ℓ, m_t) = $m_{t+½}$ or hom.m

Recall: For Representables, we Count the “Parts”

What is the notion of the “parts” here is a general notion for CSets (not just representables); this is the same notion as add_part! for CSets; each element of the Set mapped to y an object in the schema is termed a “representable”. For representables, there is one main “Driver” for the representable; the others follow through from that -- for example, might have the representable for “E”, where object “E” maps to a set of size 1; others then follow of necessity from this (due to naturality). KB notes that for representables, often the Set for the “driver” of the representable only has a single element, but not always -- consider the case of reflexive graphs, where the “driver” has 3 elements.

```
"""
A pattern match from a coproduct of representables is just a choice of parts
in the codomain. E.g. matching L = •→• • •  is just a random choice of edge and
two random vertices.

The vector of ints refers to parts of L which are the counits of the left kan
extensions that define the representables (usually this is just wherever the
colimit leg sends 1, as there is often just one X part in the representable X).

WARNING: this is only viable if the timer associated with the rewrite rule is
symmteric with respect to the disjoint representables and has a simple
exponential timer.
"""
@struct_hash_equal struct RepresentableP <: PatternType
  parts::Dict{Symbol, Vector{Int}}
end
```

```
m = get_match(pattern_type(rule), pattern(rule), rt.state, clocks, key;
                basis=basis(rule))
```

Ideas for Future Work

Ideas for Dimensional Quantities

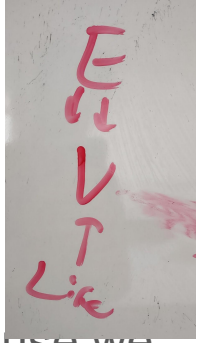
- Derive units when calculating aggregate totals
- Ensure dimensional consistency when calculating
  - Stock & flow models
  - Observables (when totalling up, has to be consistent)
  - Hazard rates (here, know needs to be dimension 1/Time)
  - Conditionals (e.g., for branches in statecharts)
- For Petri Nets, rates
- For observables: When know units, could ask for automatic conversion between units (e.g., report in per month values)
- When comparing output
- Dimensional consistency supported for parameters

Questions

Questions

- This doesn’t have {SchLifeGraph} in part because in terms of syntax, it is a macro being applied, not a type being declared directly with this line `@acset_type LifeState(SchLifeGraph) <: AbstractSymmetricGraph;`

Questions



- The game of life has the following schema, specifying cell information
- This creates a dead cell: `const DeadCell = LifeState(1) # a single dead cell` because we just have a cell where we have only a vertex (as specified by the argument “1”, due to a constructor for graphs that uses the only integer argument for the size of V).
- We thus get V=[1], Life=∅, and Hom(V, Life) = ∅
- By contrast, we have

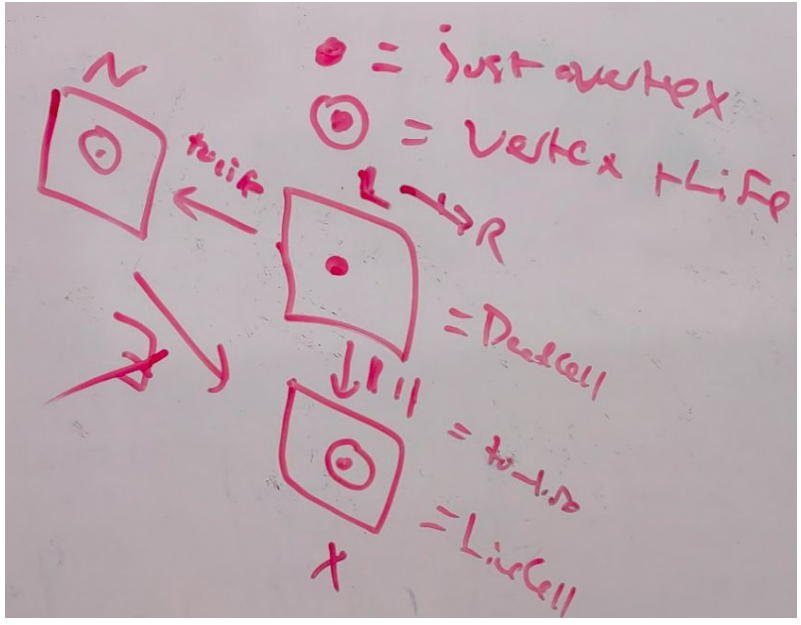
```
const LiveCell = @acset LifeState begin V=1; Life=1; live=1 end # a single living cell
```

and thus V=[1], Life=[1], and Hom(V, Life) = 1 ↦ 1

A Required Application Condition

- This needs the last application condition:
- To ensure that “DeadCell” doesn’t simply match the “empty” part of a “LiveCell”, that is, we want to ensure that:

```
# A cell is born iff it has three living neighbors
birth = TickRule(id(DeadCell), to_life;
                 ac=[PAC(living_neighbors(3; alive=false)),
                     NAC(living_neighbors(4; alive=false)),
                     NAC(to_life)]);
```



Questions

- this create a dead cell, because for graphs (from whose schema this schema is derived), we have a special constructor that can take just an integer, and create a graph with a single vertex. This creates a cell with just a vertex, and empty "Life" `const DeadCell = LifeState(1) # a single dead cell`
- Meaning of "initial" here: `homomorphism(alive ? LiveCell : DeadCell, X; initial=(V=[1],))`
- Why does this have the last application condition:

```
129
130 # Create an initial state
131 G = make_grid([1 0 1 0 1;
132                0 1 0 1 0;
133                0 1 0 1 0;
134                1 0 1 0 1;
135                1 0 1 0 1])
136 view_life(G);
137 G = make_grid(ones(1,1))
138 # Run the model
139 res = run!(AddCoords(GoL), G; maxevent=2);
```

```
# A cell is born iff it has three living neighbors
birth = TickRule(id(DeadCell), to_life;
                 ac=[PAC(living_neighbors(3; alive=false)),
                     NAC(living_neighbors(4; alive=false)),
                     NAC(to_life)]);
```

Questions

- this create a dead cell, because for graphs (from whose schema this schema is derived), we have a special constructor that can take just an integer, and create a graph with a single vertex. This creates a cell with just a vertex, and empty "Life" `const DeadCell = LifeState(1) # a single dead cell`
- Meaning of "initial" here: `homomorphism(alive ? LiveCell : DeadCell, X; initial=(V=[1],))`
- Why does this have the last application condition:

```
129
130 # Create an initial state
131 G = make_grid([1 0 1 0 1;
132                0 1 0 1 0;
133                0 1 0 1 0;
134                1 0 1 0 1;
135                1 0 1 0 1])
136 view_life(G);
137 G = make_grid(ones(1,1))
138 # Run the model
139 res = run!(AddCoords(GoL), G; maxevent=2);
```

```
# A cell is born iff it has three living neighbors
birth = TickRule(id(DeadCell), to_life;
                 ac=[PAC(living_neighbors(3; alive=false)),
                     NAC(living_neighbors(4; alive=false)),
                     NAC(to_life)]);
```

Questions

- this create a dead cell, because for graphs (from whose schema this schema is derived), we have a special constructor that can take just an integer, and create a graph with a single vertex. This creates a cell with just a vertex, and empty "Life" `const DeadCell = LifeState(1) # a single dead cell`
- Meaning of "initial" here: `homomorphism(alive ? LiveCell : DeadCell, X; initial=(V=[1],))`
- Why does this have the last application condition:

```
129
130 # Create an initial state
131 G = make_grid([1 0 1 0 1;
132                0 1 0 1 0;
133                0 1 0 1 0;
134                1 0 1 0 1;
135                1 0 1 0 1])
136 view_life(G);
137 G = make_grid(ones(1,1))
138 # Run the model
139 res = run!(AddCoords(GoL), G; maxevent=2);
```

```
# A cell is born iff it has three living neighbors
birth = TickRule(id(DeadCell), to_life;
                 ac=[PAC(living_neighbors(3; alive=false)),
                     NAC(living_neighbors(4; alive=false)),
                     NAC(to_life)]);
```

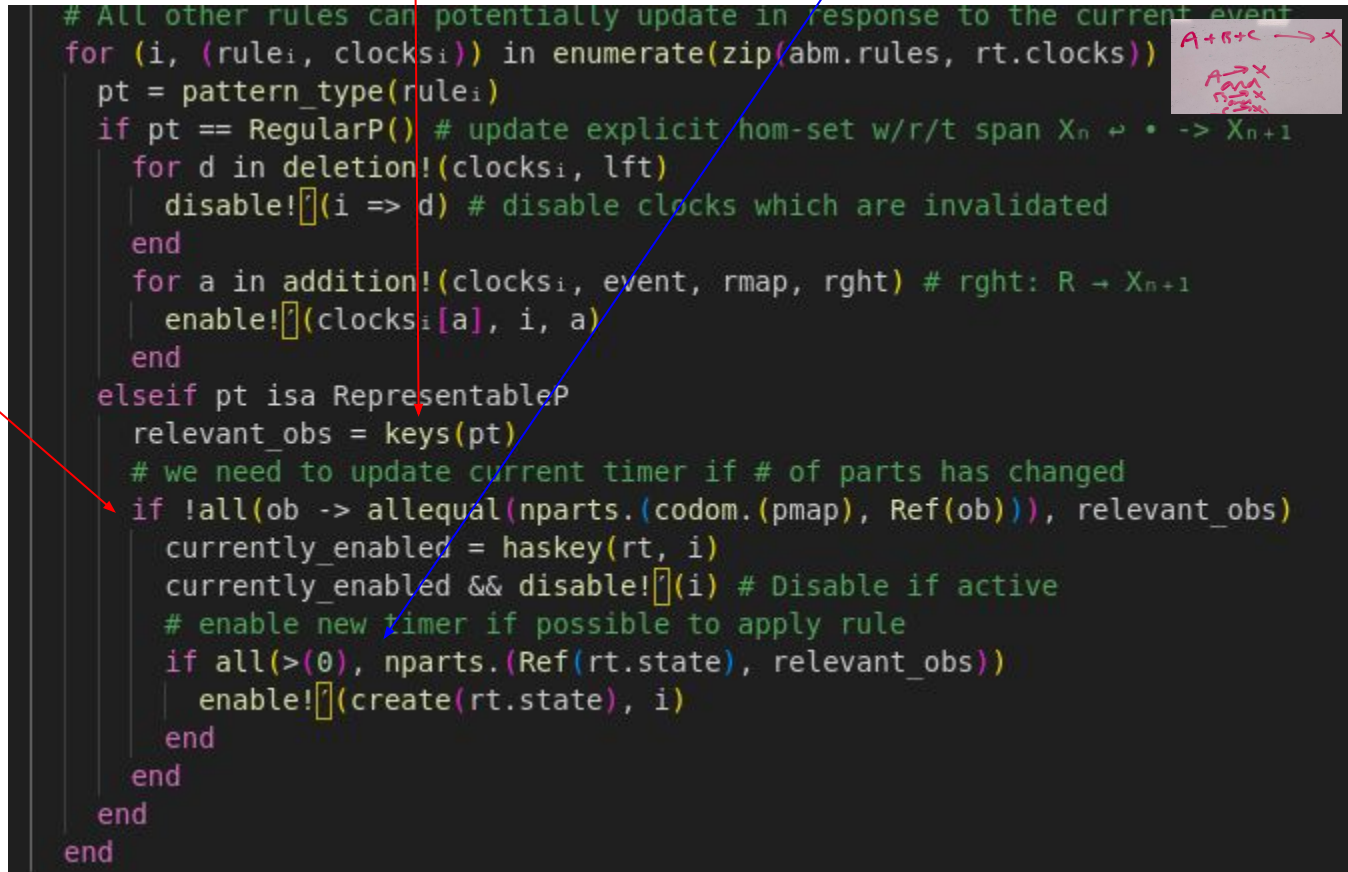
Recall: for each of the “parts”, we have a Symbol count associated with the RepresentableP.

Don’t bother enabling if any are 0 -- for this case, there will be 0 pairs; see:

Question: Seeking Understanding on this Code

Here, we are checking if the size of the set mapped to by any of the objects have changed. (Here, we quantify over symbols ob, and compare the size of each; Ref(ob) just duplicates to be of equal size as the vector of parts.)

Question: Is it possible that the # remains invariant but identity changes?



```
# All other rules can potentially update in response to the current event
for (i, (rule_i, clocks_i)) in enumerate(zip(abm.rules, rt.clocks))
  pt = pattern_type(rule_i)
  if pt == RegularP() # update explicit hom-set w/r/t span X_n ↩ • -> X_{n+1}
    for d in deletion!(clocks_i, lft)
      disable!'(i => d) # disable clocks which are invalidated
    end
    for a in addition!(clocks_i, event, rmap, rght) # rght: R → X_{n+1}
      enable!'(clocks_i[a], i, a)
    end
  elseif pt isa RepresentableP
    relevant_obs = keys(pt)
    # we need to update current timer if # of parts has changed
    if !all(ob -> allequal(nparts.(codom.(pmap), Ref(ob))), relevant_obs)
      currently_enabled = haskey(rt, i)
      currently_enabled && disable!'(i) # Disable if active
      # enable new timer if possible to apply rule
      if all(>(0), nparts.(Ref(rt.state), relevant_obs))
        enable!'(create(rt.state), i)
      end
    end
  end
end
```

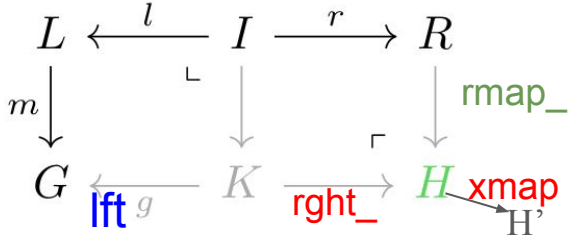
Question: Confirm Understanding for the Below

rmap_ is R->H, and rmap is R->H' (thus that rt.state is set to codom(rmap))

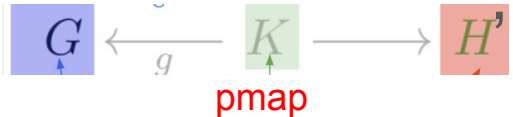
xmap is
a variable binding
morphism?

```
441     # If RegularPattern, we have an explicit match, otherwise randomly pick one
442     m = get_match(pattern_type(rule), pattern(rule), rt.state, clocks, key)
443
444     # Excute rewrite rule and unpack results
445     rw_result = (rule_type, rewrite_match_maps(getrule(rule), m))
446     rmap_ = get_rmap(rw_result...)
447     xmap_ = get_expr_binding_map(getrule(rule), m, rw_result[2])
448     (lft, rght_) = get_pmap(rw_result...)
449     rmap, rght = compose.([rmap_,rght_],Ref(xmap_))
450     pmap = Span(lft, rght)
451     rt.state = codom(rmap) # update runtime state
452     log!(event, pmap)      # record event result
```

lft is and rght are as follows:
rght is rght_
with variables
substituted in



pMap is the span below



Questions 2

- For single pushout, is R -> H a partial map? No
- What is the thinking on optimizing use of discrete cases?

Optimizing Memoryless Processes

Here, when scheduling we don't need to enumerate explicit homs for representables when we have a memoryless transition -- we can just use counts.

```
"""A collection of timers associated at runtime w/ an ABMRule"""
abstract type AbsHomSet end

@struct_hash_equal struct EmptyHomSet <: AbsHomSet end

@struct_hash_equal struct DiscreteHomSet <: AbsHomSet end

@struct_hash_equal struct ExplicitHomSet <: AbsHomSet val::IncHomSet end

Base.keys(h::ExplicitHomSet) = keys(h.val)

Base.pairs(h::ExplicitHomSet) = pairs(h.val)

Base.getindex(h::ExplicitHomSet, i) = h.val[i]

deletion!(h::ExplicitHomSet, m) =  deletion!(h.val, m)

addition!(h::ExplicitHomSet, k, r, u) = addition!(h.val, k, r, u)

"""Initialize runtime hom-set given the rule and the initial state"""
function init_homset(rule::ABMRule, state::ACSet, additions::Vector{<:ACSetTransformation})
  p, sd = pattern_type(rule), state_dep(rule.timer)
  p == EmptyP() && return EmptyHomSet()
  (sd || p == RegularP()) && return ExplicitHomSet(IncHomSet(pattern(rule), additions, state))
  @assert p isa RepresentableP  "$(typeof(p))"
  return DiscreteHomSet()
end
```

Pleasing Symmetry in Game of Life

```
# A previously empty cell is colonized iff it has three living neighbors
birth = TickRule(id(DeadCell), to_life;
                 ac=[PAC(living_neighbors(3; alive=false)),
                     NAC(living_neighbors(4; alive=false)),
                     NAC(to_life)]);

# A living cell remains alive iff it has ≥ 2 living neighbors but < 4 living neighbors
death = TickRule(to_life, id(DeadCell);
                 ac=[PAC(living_neighbors(2)),
                     NAC(living_neighbors(4))]);
```