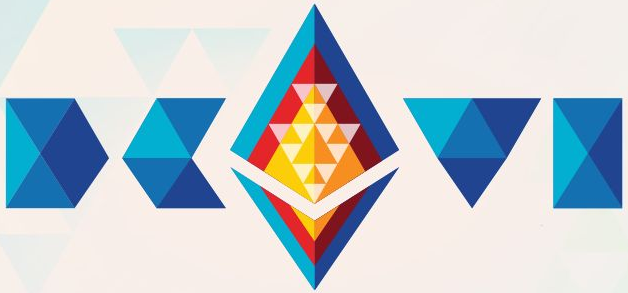


Compositionality

The 10x Engineer Secret Sauce
Part 1 - Theory

Fabrizio Genovese

20squares

Section 1

Section I: Systems and Processes

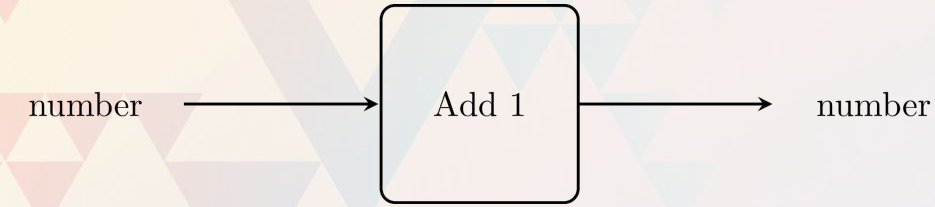
Compositionality is the study of how systems compose to give life to more complex systems.

- **Systems** are things we can transform and act upon
- **Processes** are things that act on systems, transforming them.

Big insight from modern mathematics: often the best way to understand systems is by studying their processes: describing things in terms of how they interact.

Systems and Processes

Example: Numbers



Here we see a process:

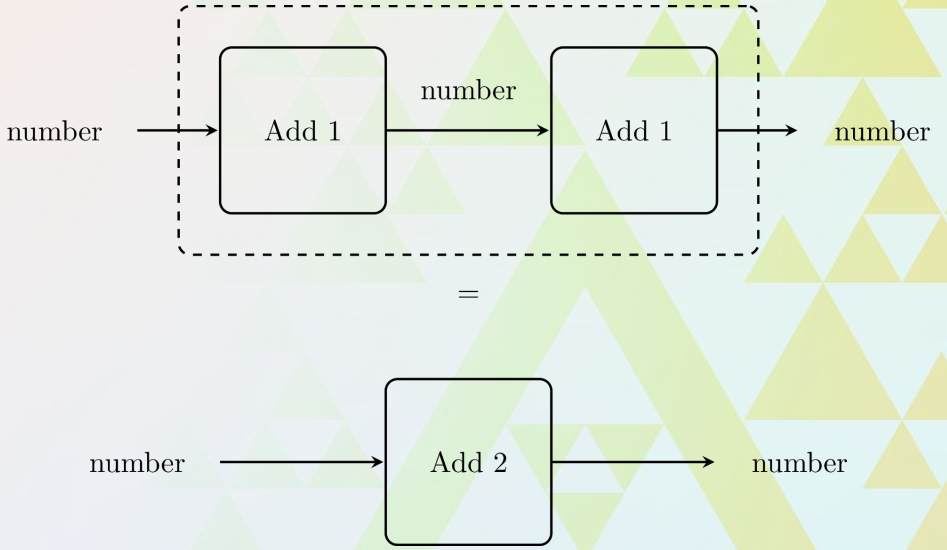
- Taking a number as input
- Spitting out a number as output
- The process sums 1 to the input.

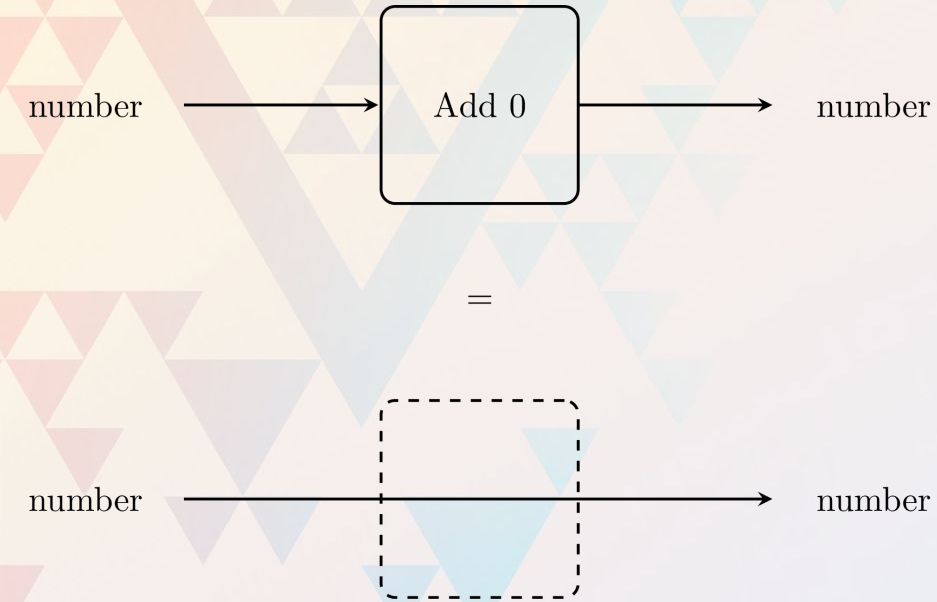
Systems and processes
Composition

Processes can be composed!

Composition is obtained by connecting wires with **matching types**.

On the right, we are composing the process in the previous slide with itself.





Systems and processes

Identity process

For each system, there is a “**do nothing process**”:
…it just outputs the input as it is!

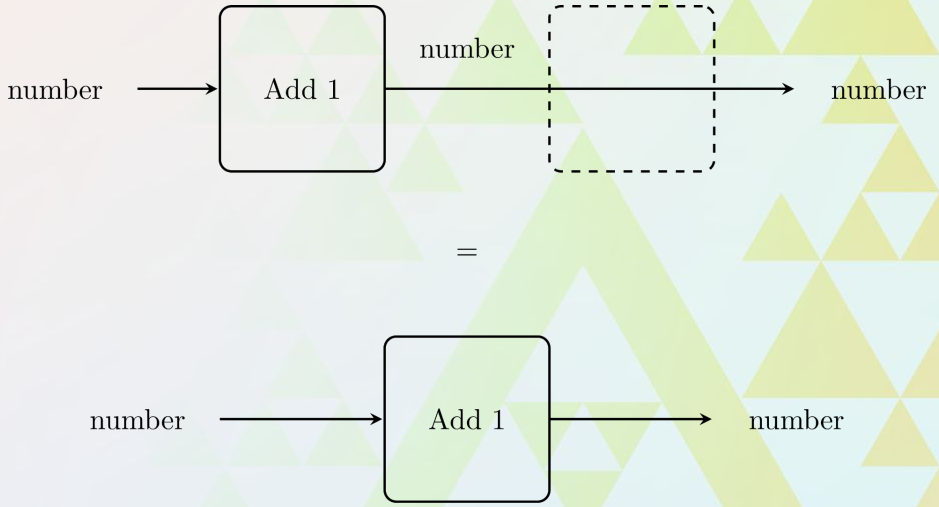
In our numbers example, the identity process is the one adding 0 to the input.

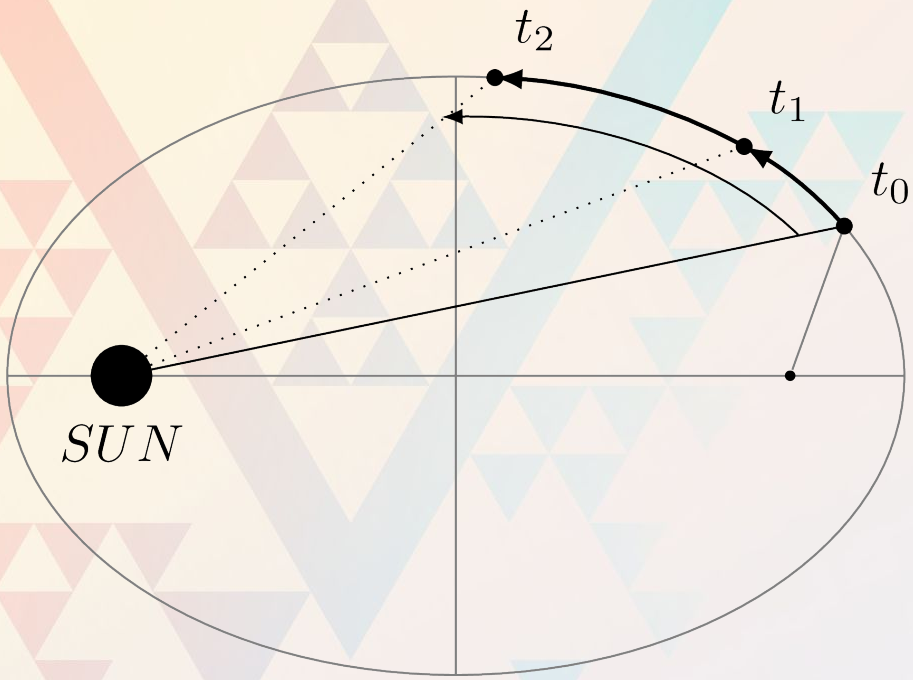
As identity processes do nothing, **we resolve to just not drawing them**.

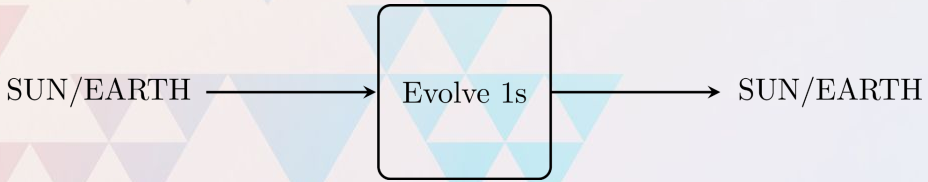
Systems and processes
Identity laws

As identity processes do nothing, if we compose them with any other process the result will be the process we started with.

Graphically, this is obvious!







Systems and Processes

Example: Gravitation

Here on the left you see a typical “Sun - Earth” gravitational system.

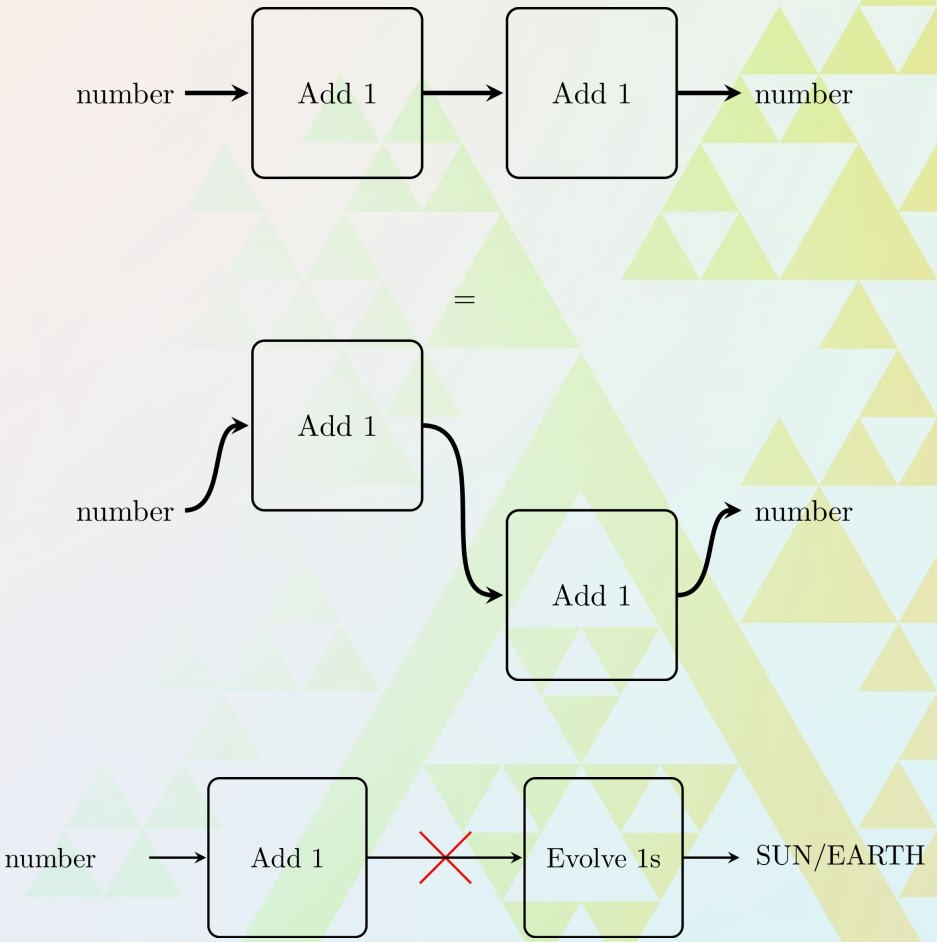
- We have a process called **evolve**:
  - You feed to it some initial positions and velocities
  - It returns the position and velocity some time later (e.g. 1 second later)
- Identity is “evolve for 0 seconds”
- Composition is sum: “evolve for 1 second” composed with “evolve for 2 seconds” equals “evolve for 3 seconds”.

Systems and Processes

Connectivity Matters!

It does not matter how we arrange processes spatially. It only matters how they connect to each other. **Pictures can be topologically deformed!**

...But a composition of processes is **valid only if their types match!**



Systems and processes

Types are important!

Imagine you have a car: a car is designed to need gas to run, but you are physically able to pour any liquid in the gas tank.

This is because the gas tank opening has type **liquid** and not type **gas**. And yet, filling up the tank with water will have (very) undesirable consequences.

Take home message: Types are important! We can type things more strictly (which is safer but reduces our ability to compose) or more liberally (which makes prototipation easier but may incur in more emerging behavior later on). Finding the right balance is an art and not a science!

Compositionality is based on **Category Theory**.

Systems and processes

Where do systems and processes live?

It is evident by now that defining which systems and processes are admissible in your setup matters.

Compositionality is based on **category theory**, which defines the universe in which we act: It is a mathematical field concerned with the formal study of recurring patterns.

Formally, we are always working in a **category** of systems and processes that defines our interests.

Section 2

Section II: Category Theory

Category Theory
Defining categories

To formally define a **category**, you must specify:

Objects: these are the systems you want to study;

Morphisms: these are your processes;

Identities: for each object (a system) we require a particular morphism (a process) representing the 'process that doesn't do anything'.

Composition law: you have to say how processes compose. This rule must obey some laws (associativity of composition, identity laws).

Category Theory

Examples of categories

Here are two interesting examples of categories:

- There is a category **Set** of sets and functions. Sets are the objects, functions are morphisms, composition is function composition. For each set there is an identity function doing nothing.
- There is a category **Data** of data types and programs. Data types are the objects, programs are processes turning data types into other data types. Composition is program piping. Identity returns the input without doing anything.

Category Theory

Intentional vs. Extensional perspective

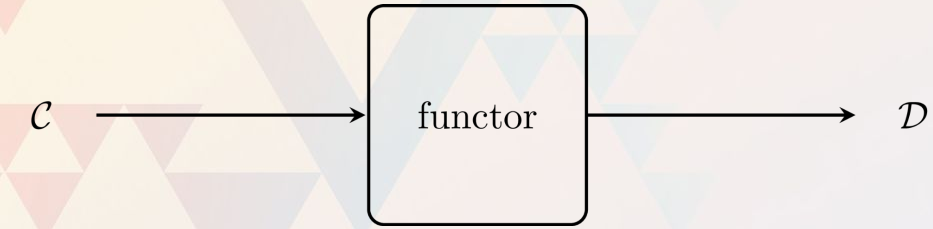
In **Set**, consider the set **Z** of integer numbers. There is exactly one function **Z** → **Z** That takes a number and adds a number n to it.

On the contrary, in **Data** there are multiple data types implementing the integer numbers. Choose one, say **Int**. There are many different programs **Int** → **Int** summing n to your term. Here go a couple:

```
return x + n ;
```

```
for ( i = 0 , i <= n , i ++)
{
x = x + 1;
}
return x ;
```

Set is extensional: it only cares about how things behave. **Data** is intensional: it cares about how things are implemented!



Category Theory
Functors

Functors are morphisms between categories. That is, processes that allow to change our underlying universe. If **C**, **D** are categories, A functor **C** → **D** must:

- Send objects of **C** to objects of **D**;
- Send morphisms of **C** to morphisms of **D**;
- Send identities to identities;
- Respect composition.

Functors are themselves processes! But in which universe do they live? In **Cat**, the category of categories and functors between them!

Category Theory
Example of functor

We can define a functor **Data** → **Set** that:

- Sends every datatype to the set it implements, e.g. **Int** is sent to **Z**;
- Sends every program to the function it implements.

We see that the program that doesn't do anything is sent to the identity function, and that piping programs amounts to compose the functions they are mapped to.

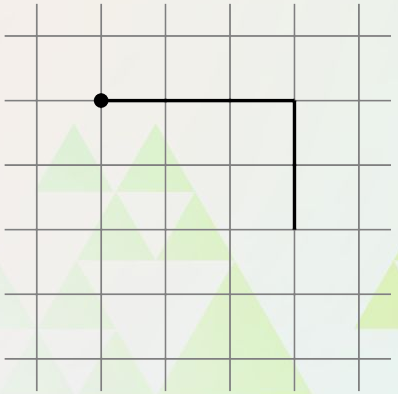
That is, there is a functor allowing us to go from a "intentional perspective" where we do care about implementation details to an "extensional perspective" where we are only interested in behavior.

Section 3

Section III: Process Design

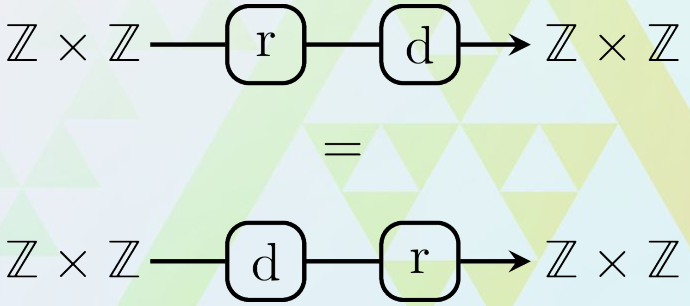
Process Design

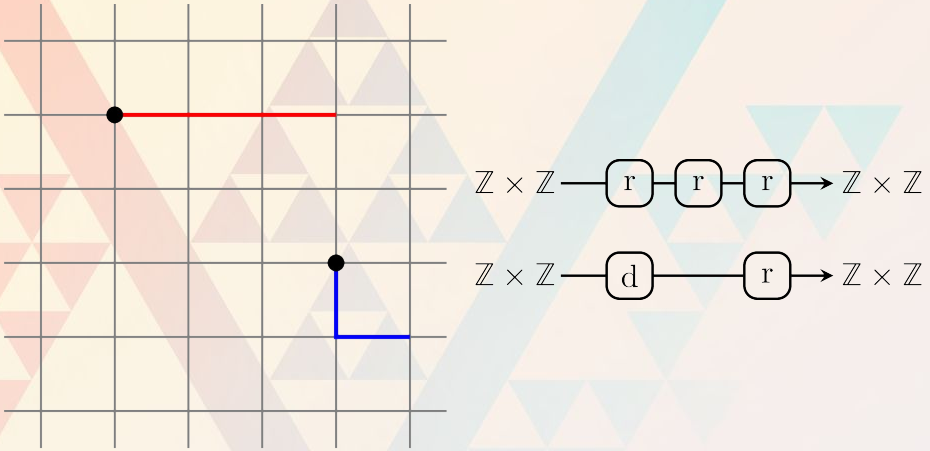
Example: Moving on a grid

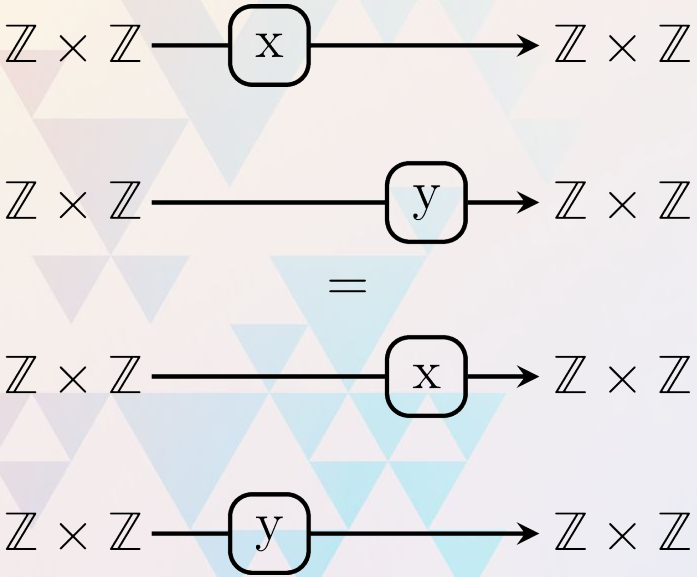


Consider someone moving on a grid. In this case, the state of our system is a couple of integers representing coordinates. Processes are expressed as compositions of four fundamental processes, “go up/down/left/right 1 step”.

Notice moreover how some **equations** hold. For instance the fundamental processes ‘commute’.







Process Design

Example: Moving on a grid, concurrently

Suppose you have multiple agents moving on the grid. We now want to represent multiple systems at the same time.

We have the same equations as before, but as a consequence of “only connectivity matters” we also have this equation on the left, where x, y are any processes, modelling concurrency.

The underlying universe of a concurrent system like this one is a particular type of category, called **monoidal category**.

Process Design

Example: accessing and rewriting records

Consider the record on the right. We want to define processes to read/replace subfields (e.g. **name**).

Moreover, notice that in this example **person** is just a couple of terms of type **name** and **surname**, respectively.

We also want to be able to replace a subfield of a given type with a subfield of a different type, e.g. replacing **surname** with **age**. In this case, the overall type of **person** will have to change, as it contains now a couple of terms of type **name** and **age**.

```
person : {
    name : John
    surname : Doe
}
```

```
person : {
  name : John
  address : {
    street : Calle 24
             No.38-71
    city : Bogotá
  }
}
```

Process Design

Example: accessing and rewriting records, compositionally.

Moreover, we want our processes to be compositional. For example, consider the record on the left.

Address is itself a record. To view/rewrite, say, **street** from **person** we would like to be able to compose setters/getters of **person** and **address**.

Section 4

Section IV: Lenses and Optics

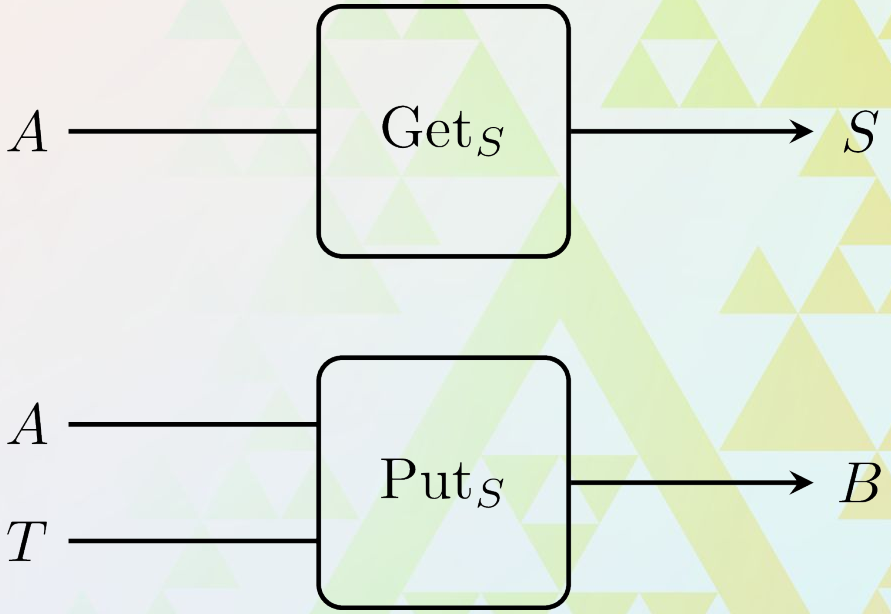
Lenses and Optics

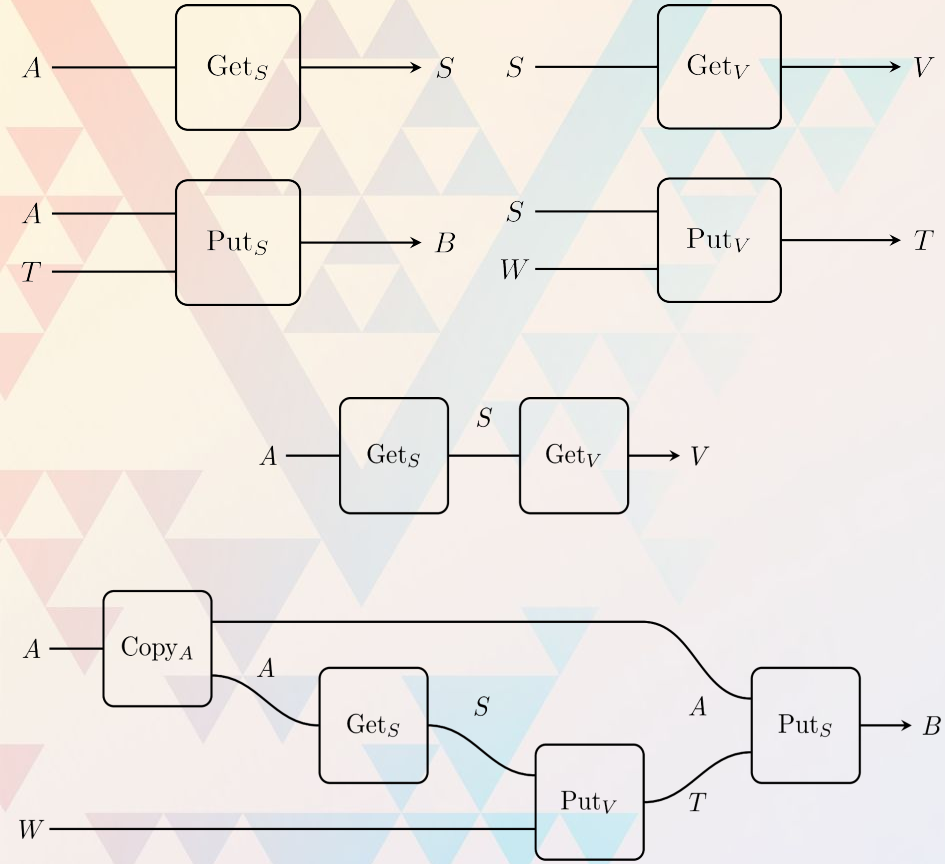
The solution to our problem: lenses

Consider a record of type **A** (e.g. **person**) and a subfield of type **S** (**surname**). Consider moreover another type **T** (e.g. **age**). A *lens* is a couple of processes called **Get** and **Put**, respectively.

- **Get** models the function returning the value of subfield **S** of **A** (read name from **person**).
- **Put** models the function replacing the subfield **S** of **A** with a value of type **T** (replace **surname** with **age** in **person**). It returns a new record having type **B**.

Given **A**, any subfield **S** and any arbitrary type **T** , we can write software to **automatically** infer **B** and define the processes above.





Lenses and Optics

Lens composition

But how do these things compose?

Suppose we have record **A** with subfield **S**

- If **S** is itself a record, we also have the lens in the right column.
- In particular if we rewrite **V** with **W** the type of **S** changes to **T**.
- We use **A**, **S**, **T**, to instantiate the lens in the left column.

…Then we can compose them, as in the third and fourth row!

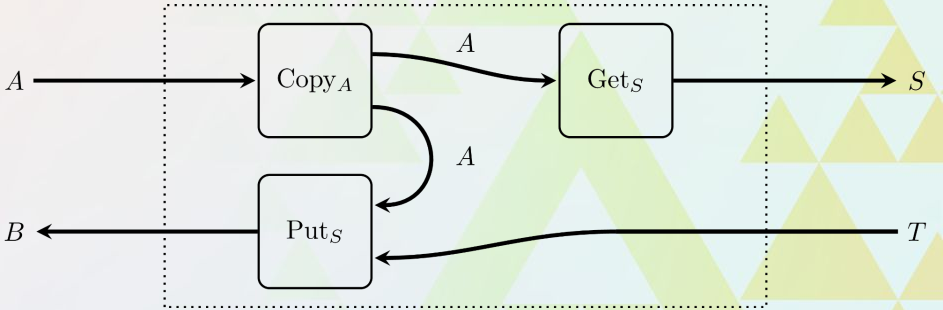
Lenses and Optics
Drawing lenses

This composition seems very cumbersome. It becomes much better if we represent a lens graphically as on the left.

Notice how we flipped **Put** , so now arrows go both ways.

We interpret this as a process saying:

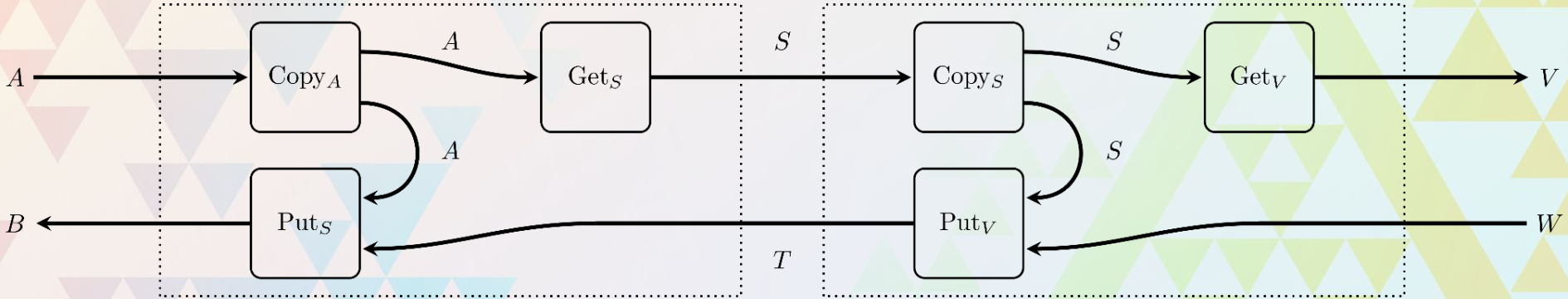
*"If you give me an **A**, I'll read field **S** and forward it. If someone returns something of type **T** , I'll replace field **S** in **A** with it and return a **B**."*

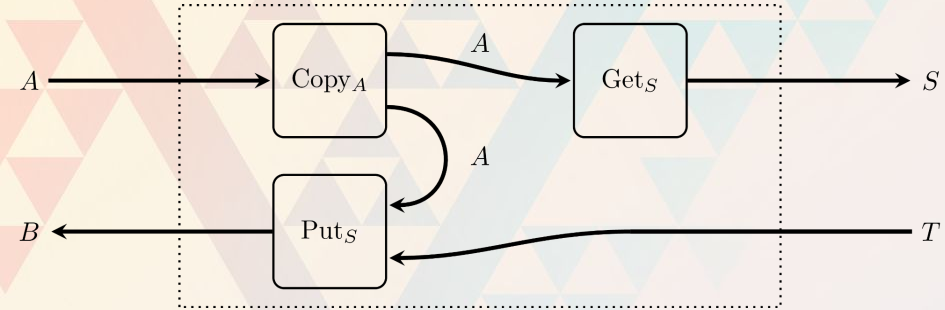


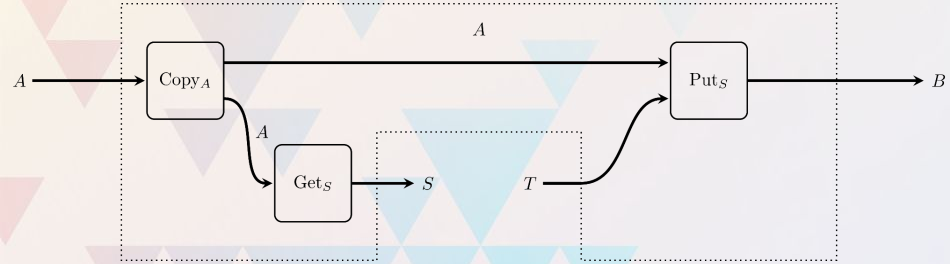
Lenses and Optics

Lens composition, graphically

Furthermore, composition now works graphically as usual!







Lenses and Optics
A different perspective

In the last slides we represented a lens as in the first row.

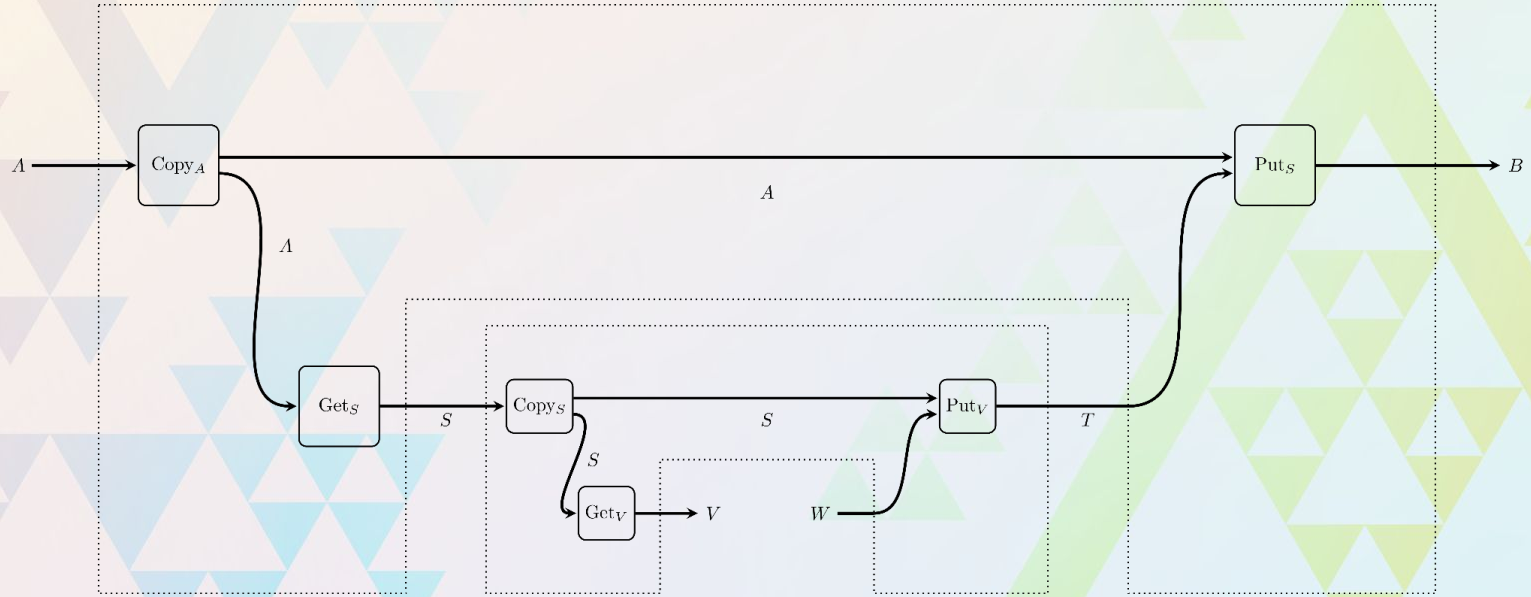
By topologically rearranging things around, we can represent the same lens as a sort of **comb**, like in the second row.

Here the bidirectional nature of lenses is even more evident: The comb receives an **A**, sends out a **request** of type **S**, waits for an answer of type **T** and produces an output of type **B**.

Lenses and Optics

Comb composition

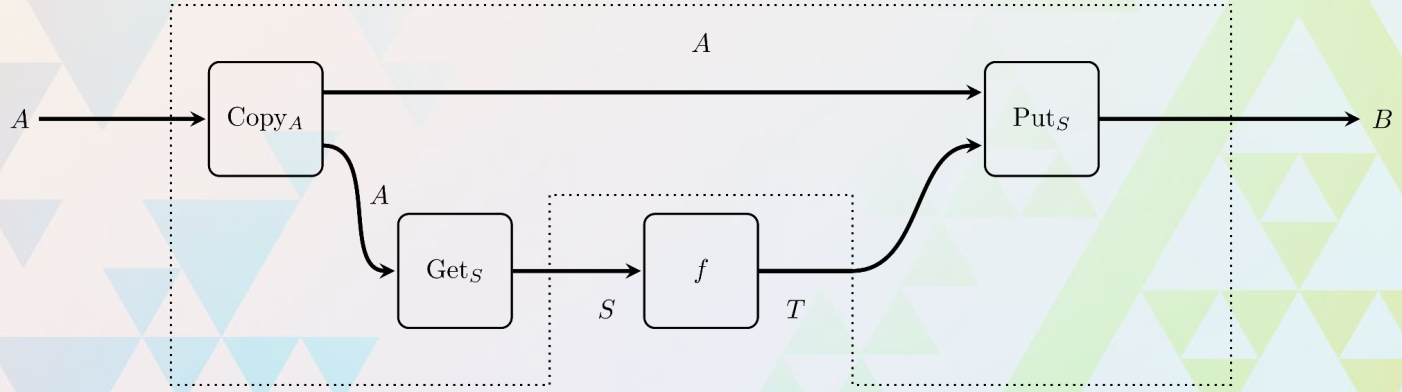
In this new framework, we compose combs by nesting.



Lenses and Optics

Combs as incomplete processes

The comb perspective makes us realize something: a comb looks like a traditional process such as the one we saw in the early slides, but it is **missing a piece**. Indeed, if we had some process **S** → **T** we could **fill** the comb:

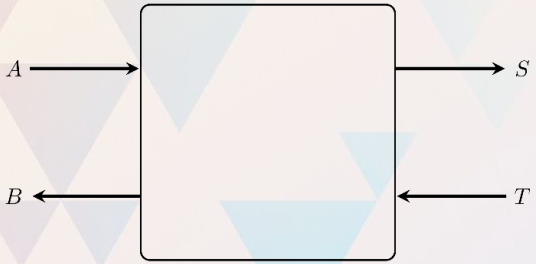


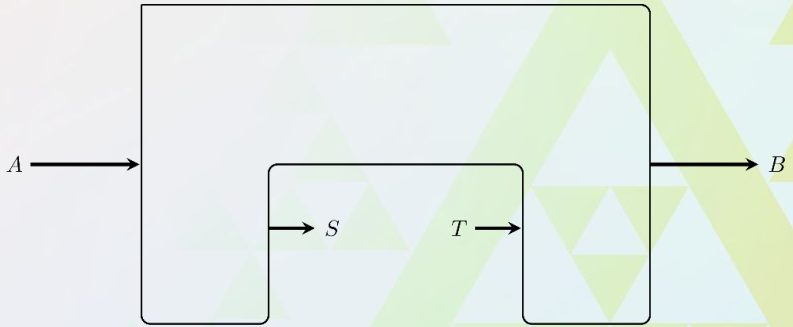
Hence you can view a comb as a process that kinda says: I am an incomplete process. Give me a way to turn an **S** into a **T** and I will behave as a standard process!'

Lenses and Optics

Abstracting things away

Now we realize that we didn't just model reading/accessing records. **We found a pattern.**
There are many things that compose this way! Let's abstract details for a second:





These processes represent two different points of view on the same thing: **bidirectional transformations.** We can easily turn one view into the other. Most importantly, we can 'fill' the insides of these processes with pretty much whatever we like.

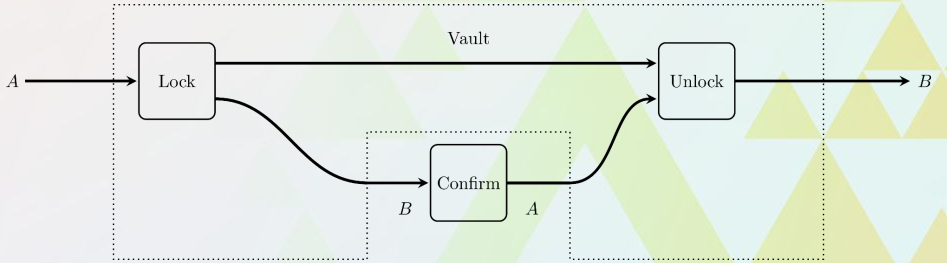
Lenses and Optics

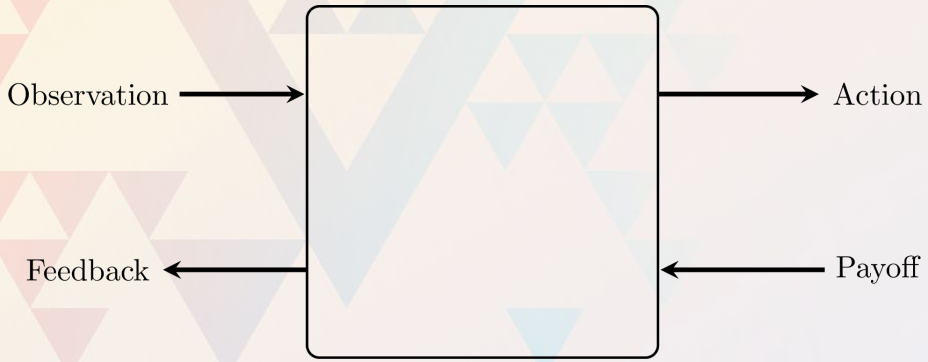
Example: escrows

We can represent an escrow trade between parties **A** and **B** as an optic.

- **A** locks some funds in a **vault**, and sends a 'waiting for delivery' message to **B**;
- After receiving this message **B** knows that funds are locked, and ships the goods;
- When delivery happens, **A** provides confirmation, that 'completes the comb' and allows **B** to unlock funds.

Comb composition in this setting models the idea that **B** can in turn sub-escrow the trade to some other party **C** : a dropshipping logic.





Lenses and Optics Example: open games

We can use bidirectional transformations to model a compositional version of **game theory**. This will be demoed in detail in Part II by Philipp.

We represent a game as a bidirectional transformation: It observes something from the **context** (this could be some other player' move), then takes an **action**. This action will have some impact on the context that will result in some **payoff**.

The 'mysterious' **feedback** wires is what enables composability: you have to think of it as **the portion of payoff you have to return to someone else** (e.g. repaying a debt you took to bet on a betting game). Inside the box live **strategies**, that determine actions from observations.

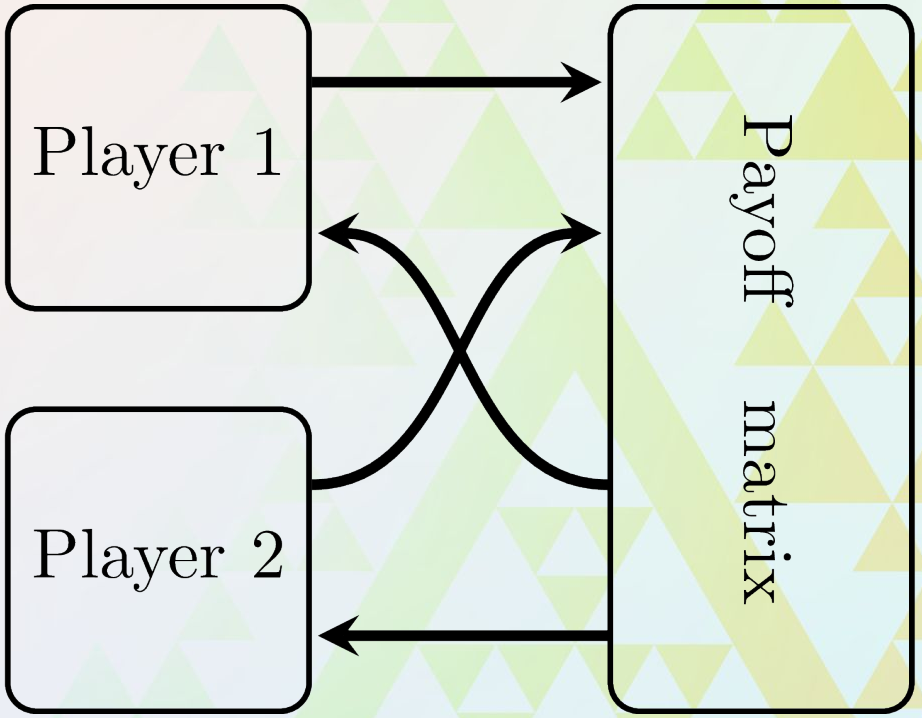
Lenses and Optics

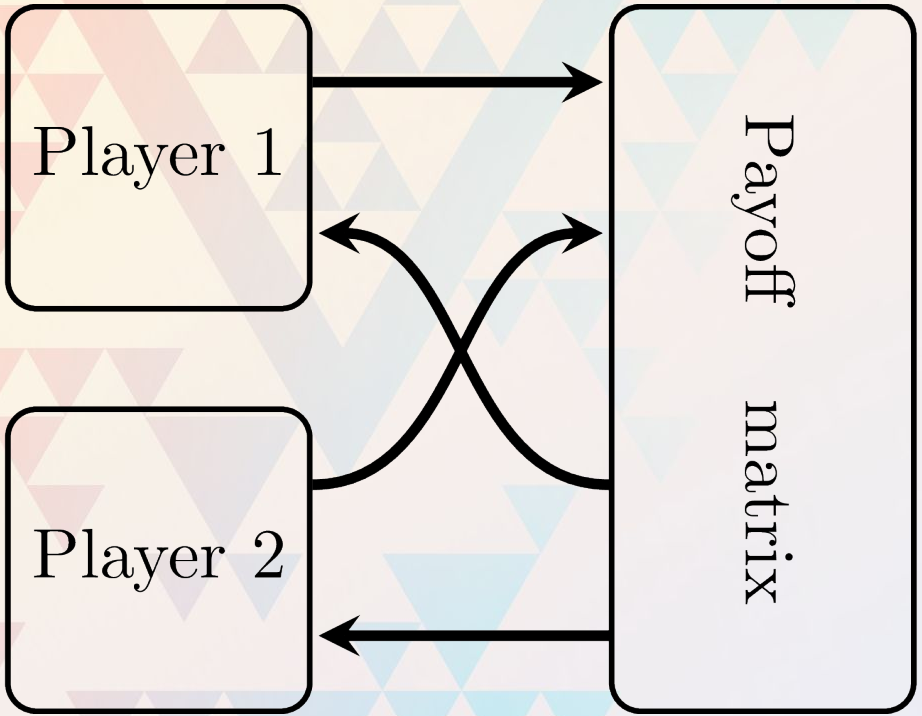
Example: prisoner dilemma [1/2]

As a last example, let us model prisoner dilemma. It is the composition of three open games, depicted on the right.

Two subgames represent the **players**. These are processes with no wires on the left, since a player in prisoner dilemma doesn't observe anything, just makes a choice. Similarly, the payoff does not produce any feedback.

Inside these two processes live players' strategies.





Lenses and Optics

Example: prisoner dilemma [2/2]

The subgame on the left is the **payoff matrix.**

Notice how players' actions are the matrix observations. Similarly, the matrix doesn't receive any payoff, it merely distributes payoffs to players in the form of feedback.

It has no strategic content.

Notice how a game actions (resp. payoffs) are some other game's observations (resp. feedbacks)!

The theory part is concluded!

The tutorial culminated with the introduction of bidirectional transformations. We generalized techniques invented to model database read/write to design completely different processes, such as escrow trades or game theory.

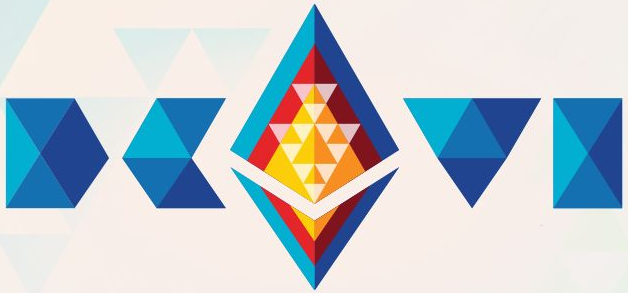
Bidirectional transformations have countless examples: they can be used to model server infrastructure; machine learning algorithms; backpropagation and gradient descent; dynamical systems, etc.

Compositional techniques in general have even more broader applications: programming language theory; quantum mechanics, computing and programming; concurrency, etc.

I hope this tutorial catpilled you enough to start thinking compositionally!

In this tutorial we saw how we can use compositionality to guide our intuition in process design.

We built notions from the ground up, starting from simple examples.



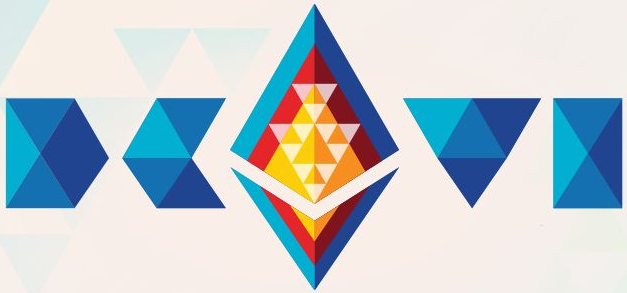
Thank you!

Fabrizio Genovese

20squares

fabrizio@20squares.xyz

@fabgenovese



Compositionality

The 10x Engineer Secret Sauce
Part 2 - Application: Game Theory

Philipp Zahn

20squares

Compositional Game Theory

A new formal language for game theoretic reasoning

Based on the categorical framework Fabrizio introduced

Software implementation for game-theoretic modelling

What the implementation provides

Model strategic interactions

Analyze models in various ways:

- Interactively
- As automatic tests running in the back
- Integrated with ML frameworks

Key innovation

Compositionality!

Seamless de-composition of model *and* code

- You can approach the model in a divide-and-conquer fashion
- You can modularize your code and it will maintain a proper game-theoretic meaning

How is this useful?

A way to deal with complex scenarios

Speeds up process of game-theoretic modelling

Can be part of a larger software stack

Plan for part 2

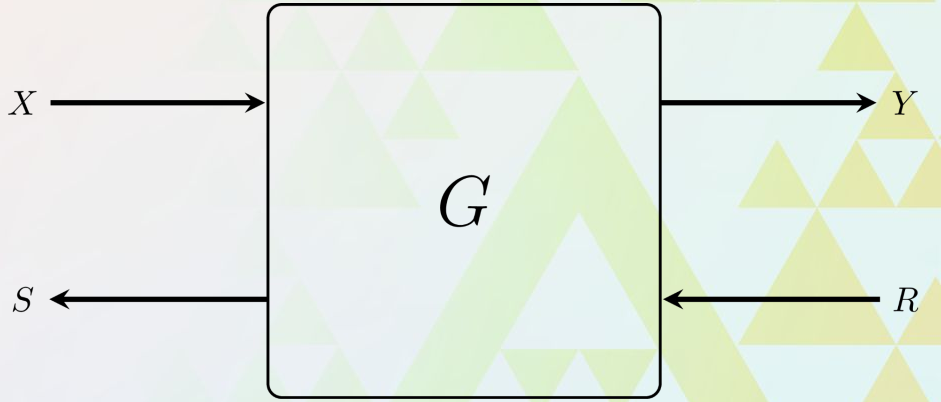
How to represent games in the engine?

How to analyze games in the engine?

Leading example: simplified staking model

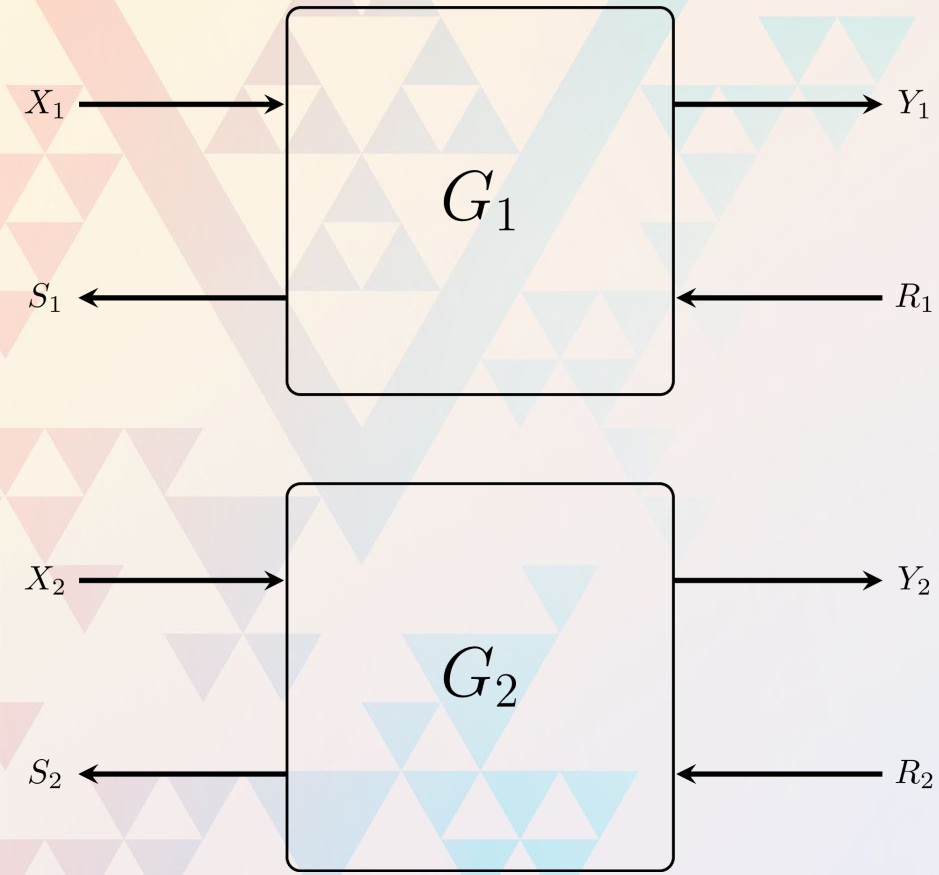
Background: Theory of Open Games

Basic Open Game



Building blocks and operations

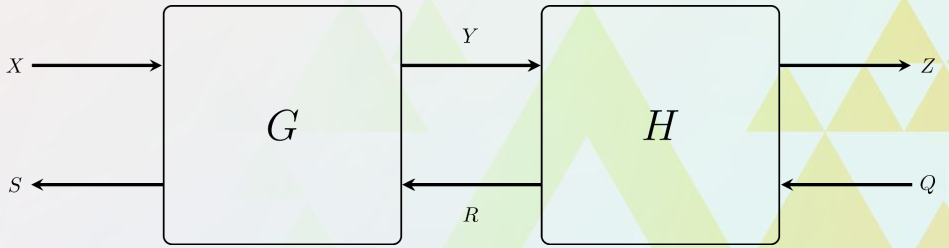
- Compose larger open games from simpler open games
- Atomic building blocks: *decision, computation*
- Composition operations: *sequential* and *parallel*



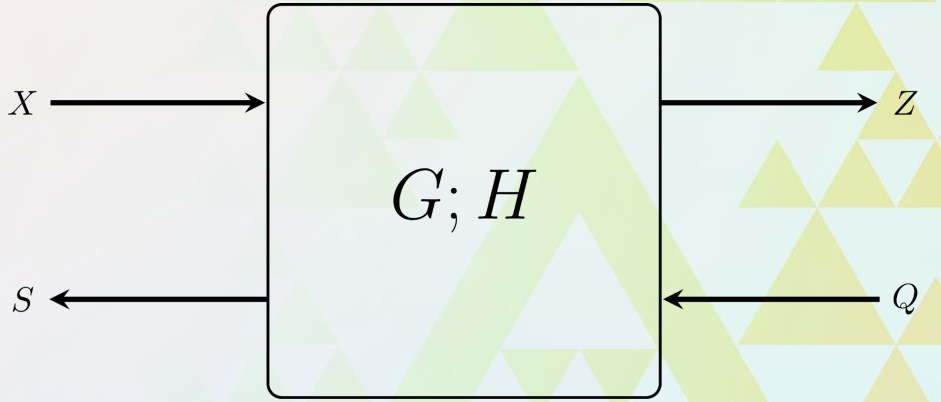
Parallel Composition

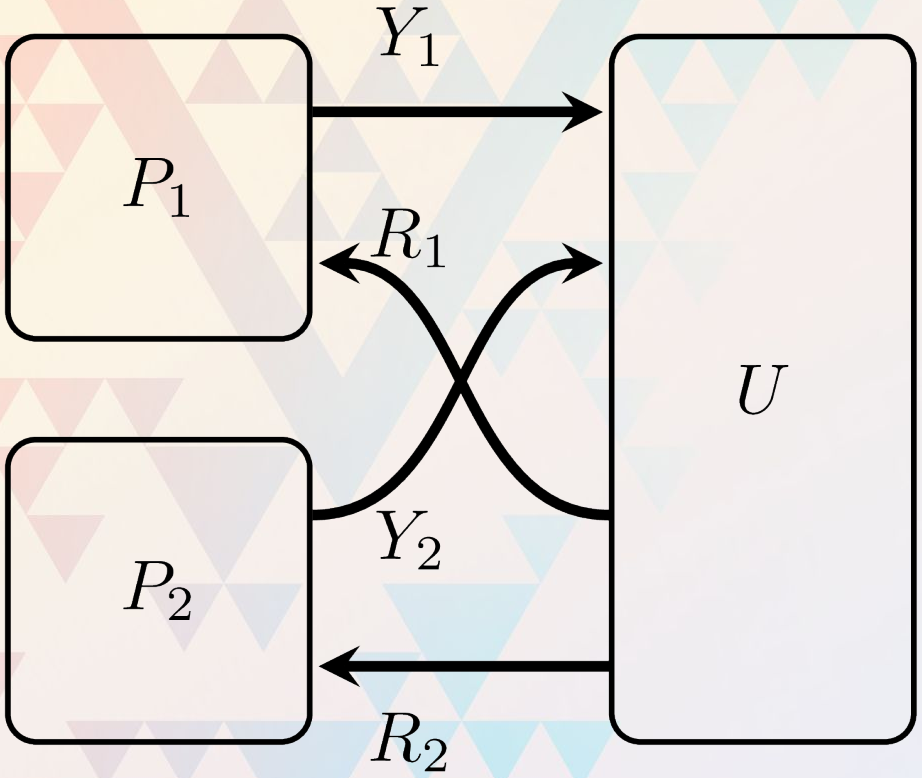
Parallel Composition

Sequential Composition



Sequential Composition





Prisoner Dilemma

Implementation

Implementation

DSL embedded in Haskell

Under development

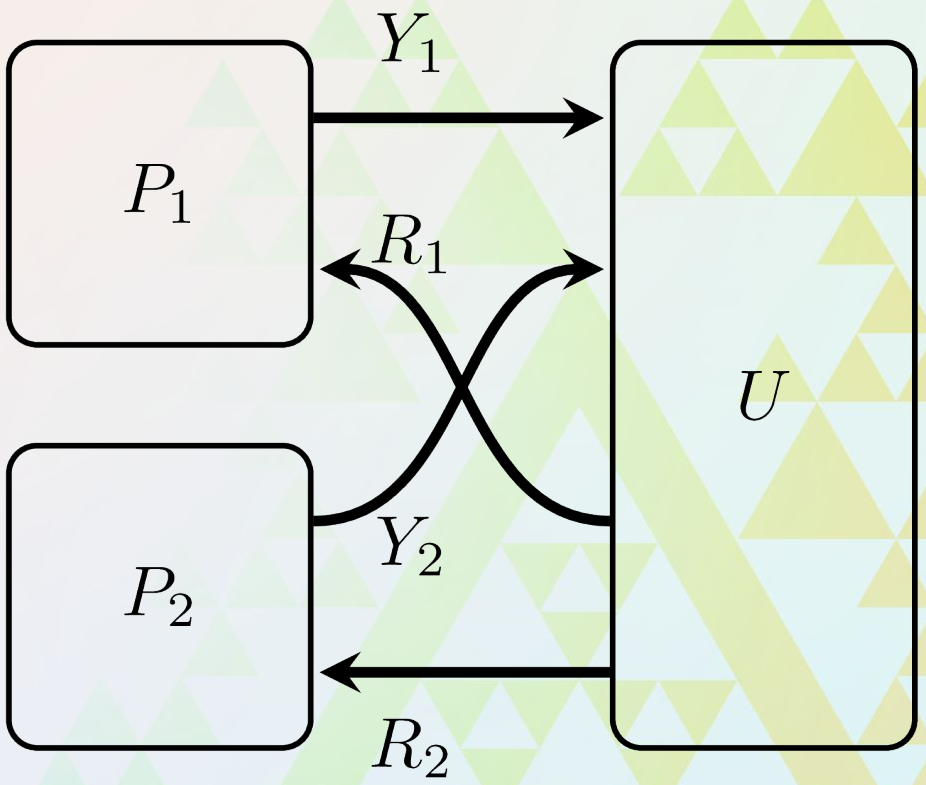
Used for staking protocols, token design, and applications outside of crypto

How to represent games?


```
_nameOfGame = [opengame|

inputs      :X;
feedback    :S;
:--------------------:

INTERNALS OF G

:--------------------:
output      :Y;
returns     :R;

|]
```


```
:----------------------:

inputs     :x;
feedback   :s;
operation  : dependentDecision _playerName_;
output     :y;
returns    :r;

. . .

:----------------------:
```

```
prisonersDilemma = [opengame|

inputs    :                                  ;
Feedback  :                                  ;
:--------------------------------------------:

inputs    :                                  ;
feedback  :                                  ;
operation : dependentDecision "player1" (const
                [Cooperate,Defect])          ;
outputs   : decisionPlayer1                  ;
returns   : payoffPlayer1                    ;

inputs    :                                  ;
feedback  :                                  ;
operation : dependentDecision "player2" (const
                [Cooperate,Defect])          ;
outputs   : decisionPlayer2                  ;
returns   : payoffPlayer2                    ;

inputs    : decisionPlayer1,decisionPlayer2;
Feedback  :                                  ;
Operation : forwardFunction payoffsPD        ;
outputs   : payoffPlayer1,payoffPlayer2      ;
Returns   :                                  ;

:--------------------------------------------:

outputs   :                                  ;
returns   :                                  ;
|]
```



Staking example

A simplified staking model

Focus on compositionality principle when representing games

Illustrate the “zooming in” for the analysis

(Based on a blog post – with more details)

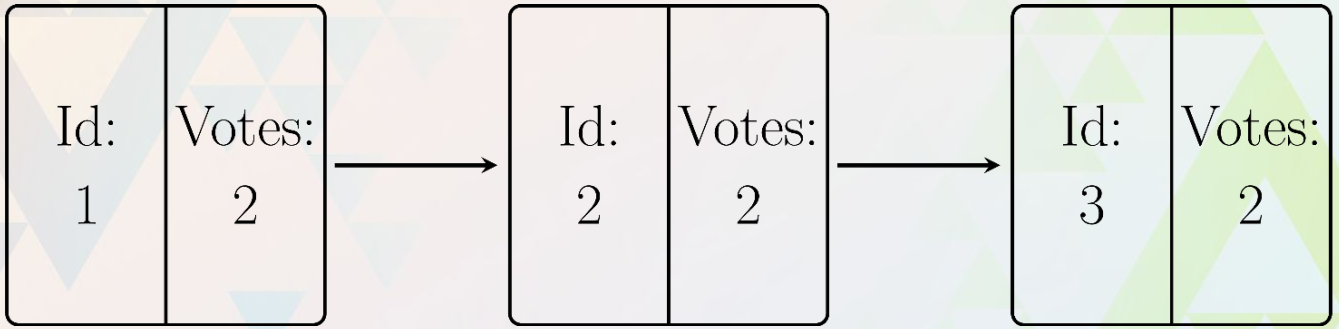
Setup

In each period: 1 proposer; 2 validators

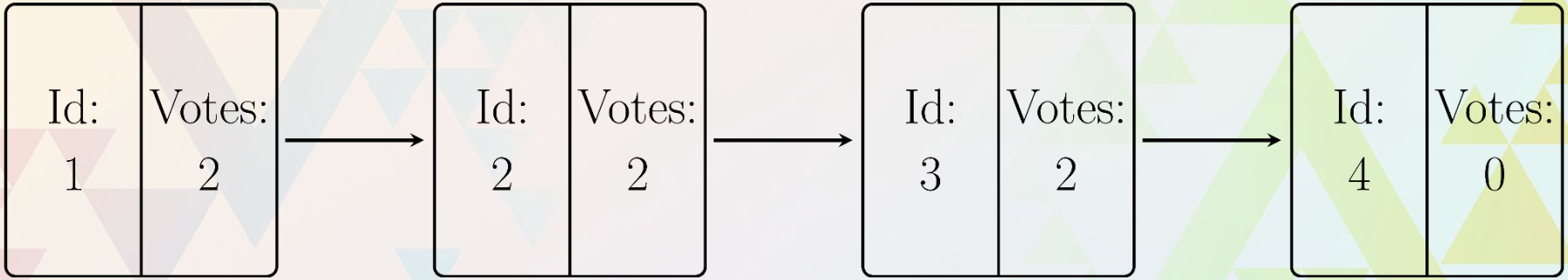
Proposer decides whether to extend the chain and if so on which block

The validators observe the proposed new head and the last chain; they then vote on the block which they view as the legitimate head

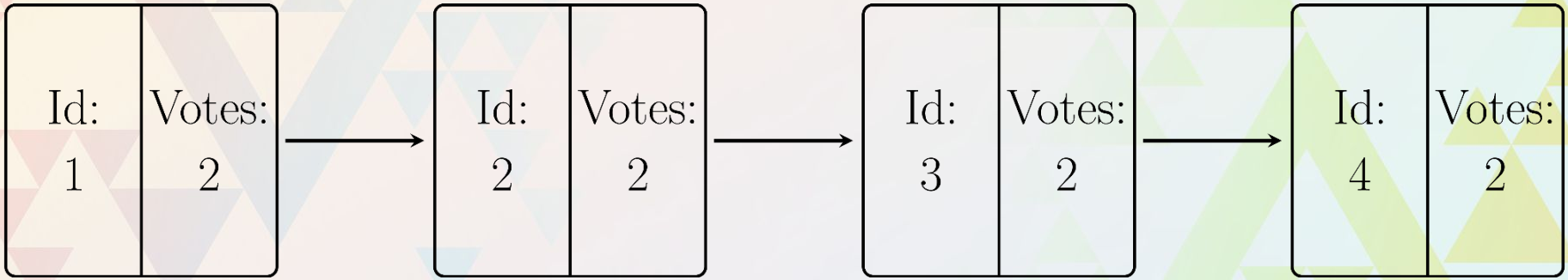
Chain example



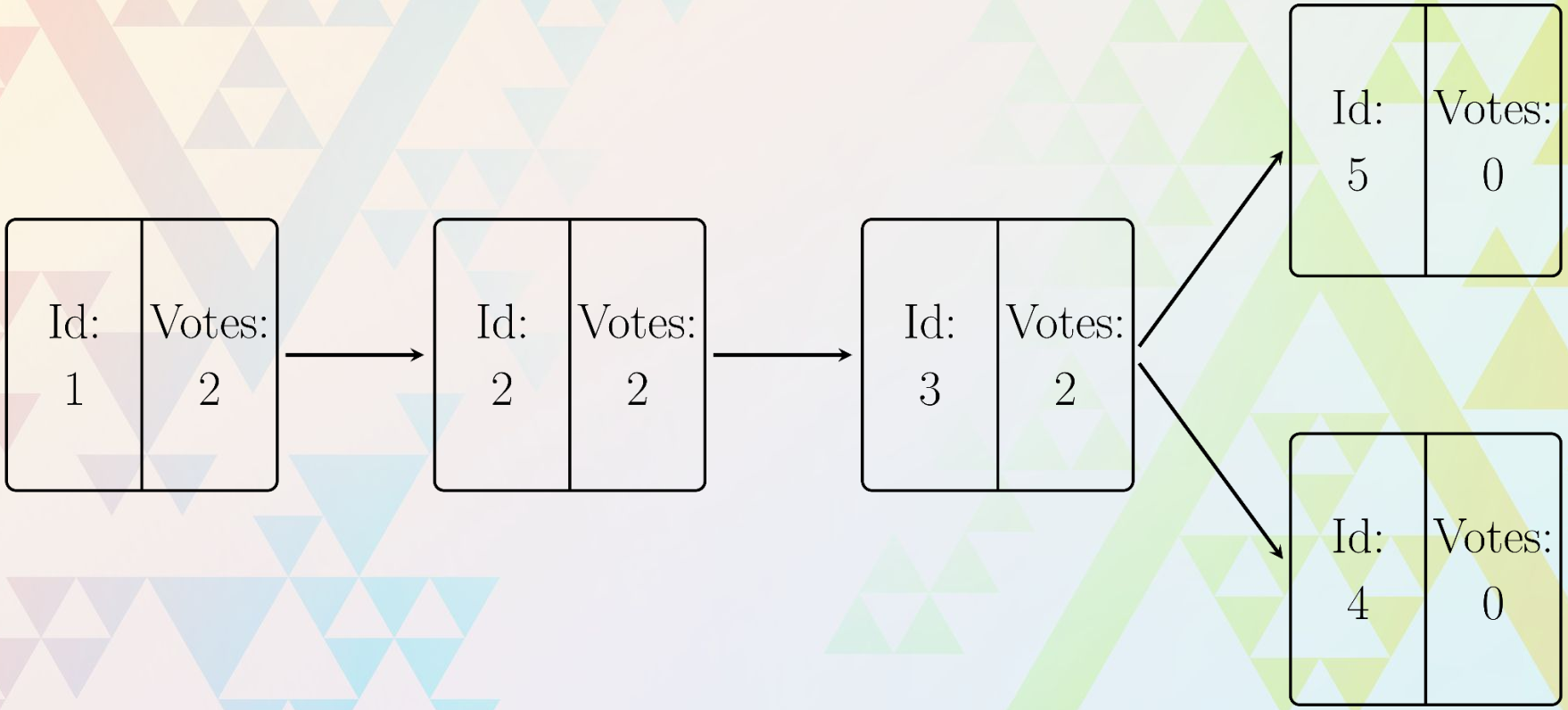
Proposer action



Validator actions



Fork after proposer actions



Conclusion

Future development of the engine

Towards “economic” verification: decompose EVM contracts into open games (current work with people from the EF Formal Verification Group)

Extensions of the available analytics (and model building block): E.g. AMMs related questions

Extensions of the engine itself: theory development

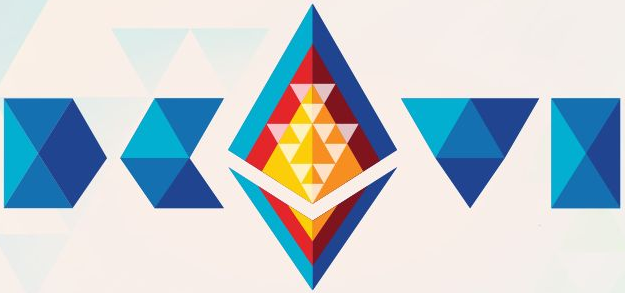
Compositionality beyond Open Games

Fabrizio introduced the quest: How do systems compose?

Current theory development of the theory behind open games further generalizes the framework beyond game theory

Control Theory, Reinforcement Learning, Active Inference, ...

What the abstraction enables: Recognize common patterns and be able to consider new blends of systems



Thank you!

Philipp Zahn
20squares
philipp@20squares.com