Open games: the long road to practical applications

Jules Hedges[1]
joint work with loads of people

[1]Max Planck Institute for Mathematics in the Sciences, Leipzig

ACT@UCR, 15th April 2020, via quarantine

Lenses

The category **Lens** has:

- Objects: Pairs of sets (X^+, X^-)
- Morphisms $(X^+, X^-) \to (Y^+, Y^-)$: pairs of functions $v : X^+ \to Y^+$ and $u : X^+ \times Y^- \to X^-$
- Composition

$$(X^+, X^-) \xrightarrow{(v_1, u_1)} (Y^+, Y^-) \xrightarrow{(v_2, u_2)} (Z^+, Z^-)$$

on the top $v(x) = v_2(v_1(x))$, on the bottom

$$u(x, z) = u_1(x, u_2(v_1(x), z))$$

- Non-obvious fact: This composition is associative, so **Lens** is a category
- It's a monoidal category with pointwise cartesian product

The chain rule

Central observation:

$$\text{if } y = f(x) \text{ then } \mathrm{d}x = \mathrm{d}y / f'(x)$$

If $z = g(f(x))$ then the lens composition law tells us how to get $\mathrm{d}x$ from x and $\mathrm{d}z$

Normally known as morphisms of cotangent bundles, aka. the chain rule

Resulting slogan: destructive updates compose by a chain rule

(cf. existing "chain rules" for conditional probability & conditional entropy)

Dubious claim: $\mathrm{d}x = \mathrm{d}y/f'(x)$ is the central trick of machine learning

From lenses to optics

We can construct **Lens**($\mathcal{C}$) over any category $\mathcal{C}$ with finite products

There's a non-obvious generalisation to any monoidal category $\mathcal{C}$: same objects, $\hom_{\mathbf{Lens}(\mathcal{C})}((X^+, X^-), (Y^+, Y^-)) =$

$$\int^{A \in \mathcal{C}} \hom_{\mathcal{C}}(X^+, A \otimes Y^+) \times \hom_{\mathcal{C}}(A \otimes Y^-, X^-)$$

That's a coend in **Set**, = a certain equivalence class of triples

$$(A \in \mathcal{C}, v : X^+ \to A \otimes Y^+, u : A \otimes Y^- \to X^-)$$

If $\mathcal{C}$ is cartesian monoidal then this reduces to lenses (proof requires the Ninja Yoneda Lemma)

Historical claim: This is a huge generalisation of the chain rule

Bayesian inversion[1]

Pick your favourite probability monad $\mathcal{D}$ (distribution monad on **Set**, Giry monad on **Meas**, Radon monad on **Top**, ...)

A Kleisli map $f : X \to \mathcal{D}(Y)$ is a conditional distribution $\mathbb{P}(Y|X)$

Given a prior distribution $\alpha \in \mathcal{D}(X)$ and an observation $y \in Y$, Bayes gives us a posterior $\mathbb{P}(\alpha = x | f(\alpha) = y)$

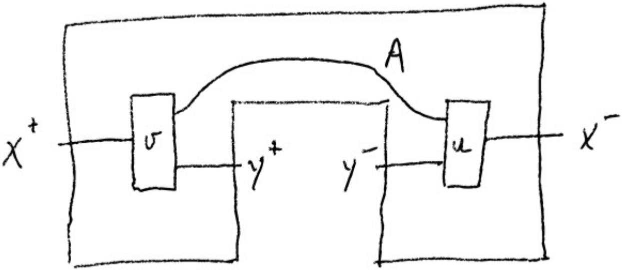
If we have conditional distributions $Y|X$ and $Z|Y$, Bayesian updates compose by lens composition

In summary: Bayes defines a functor $\mathrm{Kl}(\mathcal{D}) \to$ **Lens** (handwaving /0)

[1] Joint with work Toby Smythe

Comb diagrams[2]

An optic $(A, v, u) : (X^+, X^-) \to (Y^+, Y^-)$ is naturally drawn as a comb:

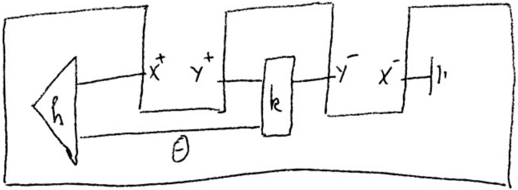


[2]Sort of due to Mitchell Riley, work in progress with... Mario Román, Davidad Dalrymple, Bruno Gavranović, who did I forget?

Open games in one slide

An open game $\mathcal{G} : (X^+, X^-) \to (Y^+, Y^-)$ consists of:

- A set $\Sigma_{\mathcal{G}}$ of strategy profiles
- A $\Sigma_{\mathcal{G}}$-indexed family of optics[3] $(X^+, X^-) \to (Y^+, Y^-)$
- For every $\sigma \in \Sigma_{\mathcal{G}}$ and every context



a best response subset of $\Sigma_{\mathcal{G}}$

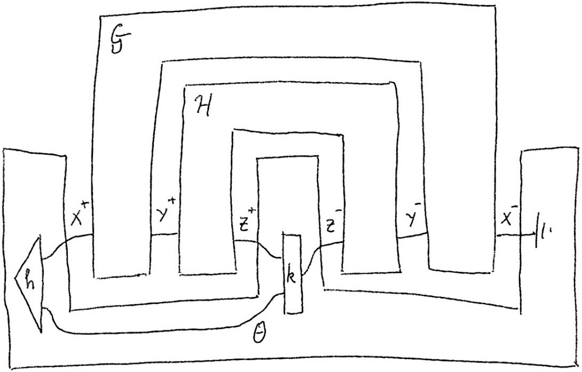
[3]Probably over the (symmetric monoidal) kleisli category of a distribution monad

Composing open games (1/3)

$$(X^+, X^-) \xrightarrow{\mathcal{G}} (Y^+, Y^-) \xrightarrow{\mathcal{H}} (Z^+, Z^-)$$

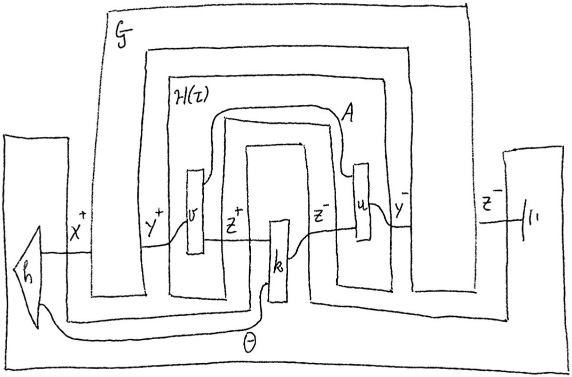
$$\Sigma_{\mathcal{H}\circ\mathcal{G}} := \Sigma_{\mathcal{G}} \times \Sigma_{\mathcal{H}}$$

To evaluate $\mathcal{H} \circ \mathcal{G}$ with strategy profile (σ, τ) in context...



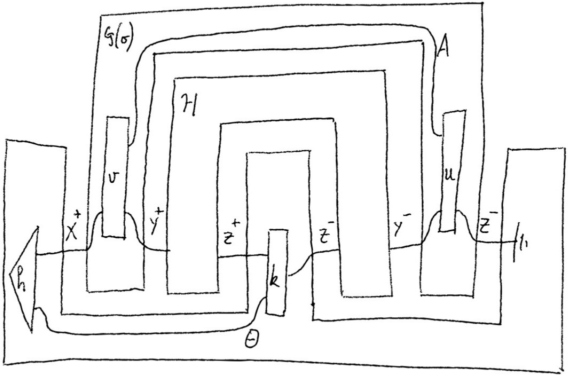
Composing open games (2/3)

...we 1. evaluate $\mathcal{G}$ with strategy profile σ in context



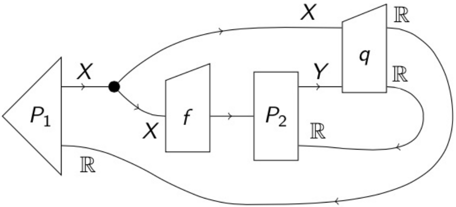
Composing open games (3/3)

...and 2. evaluate $\mathcal{H}$ with strategy profile τ in context



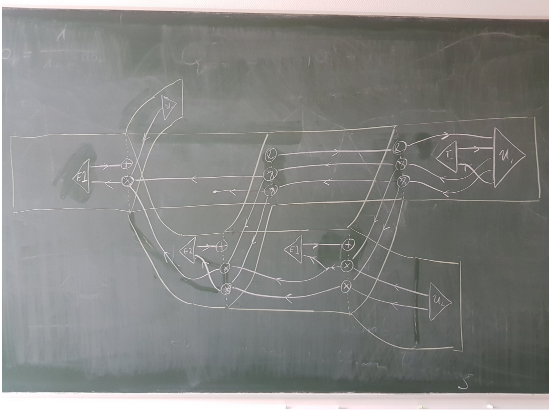
Pros of open games

- Open games are compositional, duh
- Explicit representation of system-context interaction
- Handles Bayesian Nash equilibrium, a very expressive solution concept
- Flexible in some ways
- String diagrams are
  1. exponentially more compact than extensive form
  2. game-theoretically very intuitive:



Cons of open games

- Inflexible in many ways
- Very steep learning curve
- "Games with irregular structure" require overkill:



- Say nothing about standard problems, eg. equilibrium selection, computing equilibria

Open learners[4]

An open learner $X \to Y$ is *just* an open game $(X, X) \to (Y, Y)$ where:

- best response depends only on h and $k(v_\sigma(h))$
- every best response is a singleton (ie. deterministic)

(It's a monoidal subcategory)

Current working hypothesis: open games are a better setting to do machine learning anyway

[4] Due to Brendan Fong, David Spivak & Rémy Tuyéras

Surprising application 1: variational inference

Idea: Given a conditional distribution $f : X \to \mathcal{D}(Y)$, computing the posterior $\mathbb{P}[\alpha = x \mid f(\alpha) = y]$ is hard[5], so we approximate it

Want a parameterised family of lenses, with v fixed and u tending to the Bayesian inverse

An open learner is a parameterised family of lenses + update operation

(Previously in Backprop As Functor, v was the thing to be learned)

---

[5] Because integration is hard

Surprising application 2: Reinforcement learning[6]

Jointly controlled Markov process (aka. Markov game)

- State space Q, action space A
- Transition function $Q \times A \to \mathcal{D}(Q)$, state payoffs $Q \times A \to \mathbb{R}^n$
- Strategy profiles $Q \to A$
- Individual goal: maximise discounted sum of payoffs

Value function iteration (a method used to compute Markov equilibria):

$$\cdots \to (Q, \mathbb{R}^n) \to (Q, \mathbb{R}^n) \to (Q, \mathbb{R}^n) \to I$$

The value function iterator is a representable functor

[6] Joint work with Viktor Winschel

An ecological collapse game[7]

2 states, 'prosperous' and 'collapsed'

In 'prosperous' state, play a social dilemma to invest in environment

In 'collapsed' state get punished, lose control, wait to transition back

Tool computes value of a strategy successfully on a real model, checked against existing Matlab implemenation

Iterating coend optics 200 times almost melted my laptop, coend bound variable grew to ~5Gb

[7] Joint work with Wolfram Barfuss

The need for tool support

Some people can reason purely graphically (eg. ZX calculus), open games don't have this luxury

Working with real examples on paper is impossible in practice

Monoidal category of (even Bayesian) open games is easy to implement in Haskell

But working with real examples is still very impractical

No time to wait for a string diagrams compiler[8]

[8] I.e. put in a string diagram, get out a term in the logical language of $\circ$ and $\otimes$

A domain specific language

```
lemonMarket = Block □ □
                [Line □ □ "nature (fromFreqs [(Good, 1), (Bad, 4)])" ["quality"] □,
                 Line ["quality"] □ "decision \"seller\" [Low, High]" ["price"] ["lemonUtilitySeller quality price buy"],
                 Line ["price"] □ "decision \"buyer\" [Buy, NotBuy]" ["buy"] ["lemonUtilityBuyer quality price buy"]]
                □ □
```

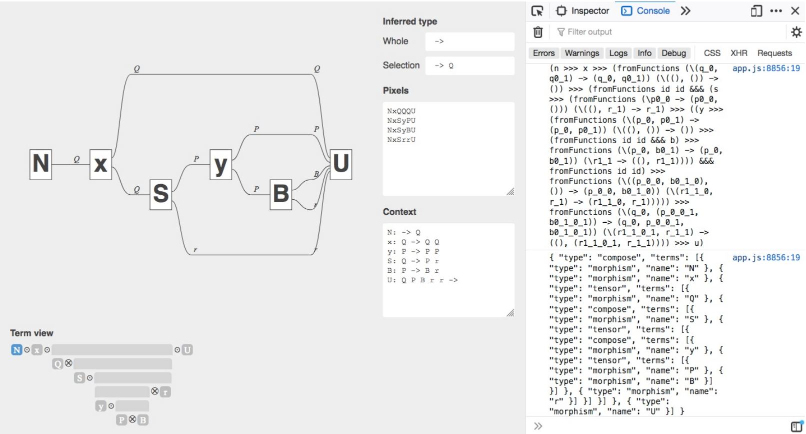
This is abstract syntax (I can't write a parser)

Refers to Haskell datatypes and functions supplied by the user

Compiles to Haskell:

```
lemonMarket = reindex (\x -> (x, ())) ((reindex (\x -> ((), x)) ((fromFunctions (\x -> x) (\(quality, price, buy) -> ())) >>> (reindex (\(a1, a2, a3) -> ((a1, a2), a3)) (((reindex (\x -> (x, ())) ((rei
ndex (\x -> ((), x)) ((fromFunctions (\() -> ((), ())) (\((quality, price, buy), ()) -> (quality, price, buy))) >>> (reindex (\x -> ((), x)) ((fromFunctions (\x -> x) (\x -> x)) &&& ((nature (fromFreqs
 [(Good, 1), (Bad, 4)]))))))) >>> (fromFunctions (\((), quality) -> quality) (\(quality, price, buy) -> ((quality, price, buy), ())))))) >>> (reindex (\x -> (x, ())) ((reindex (\x -> ((), x)) ((fromFunc
tions (\quality -> (quality, quality)) (\((quality, price, buy), ()) -> (quality, price, buy))) >>> (reindex (\x -> ((), x)) ((fromFunctions (\x -> x) (\x -> x)) &&& ((decision "seller" [Low, High]))))
)) >>> (fromFunctions (\(quality, price) -> (quality, price)) (\(quality, price, buy) -> ((quality, price, buy), lemonUtilitySeller quality price buy)))))) >>> (reindex (\x -> (x, ())) ((reindex (\x ->
 ((), x)) ((fromFunctions (\(quality, price) -> ((quality, price), price)) (\((quality, price, buy), ()) -> (quality, price, buy))) >>> (reindex (\x -> ((), x)) ((fromFunctions (\x -> x) (\x -> x)) &&&
 ((decision "buyer" [Buy, NotBuy])))))) >>> (fromFunctions (\((quality, price), buy) -> (quality, price, buy)) (\(quality, price, buy) -> ((quality, price, buy), lemonUtilityBuyer quality price buy))))
))))) >>> (fromLens (\(quality, price, buy) -> ()) (curry (\((quality, price, buy), ()) -> (quality, price, buy)))))
```

Sideline: the Statebox editor



What the tool does[9]

The tool is a model checker, aka. counterexample finding tool

User puts in a strategy profile, tool finds all failures of equilibrium

Things currently not supported:

- Computing equilibria
- Infinitely repeated games
- Continuous action spaces
- Infinitely supported distributions
- Parameterised families of games

[9] https://github.com/jules-hedges/open-games-hs

What makes a good application?

The big constraint: We can't compute equilibria. The tool supports a guess and check methodology

- Not too small: compositionality is overkill[10]
- Not too big: can't guess an equilibrium
- (it may be there's nothing in between those!!)
- Abstractions that can be exploited
- Constraints coming from the implementation, eg. discrete
- Standard constraints from game theory, eg. Nash equilibria are meaningful

[10] "Hello World, Enterprise Edition"

A failed application[11]

- Players bid for a good (e.g. a token for access to a resource), followed by a resale market
- Could bid high because you believe the good is valuable, or because you believe you can sell for a profit
- Question: How do the auction and market structure affect strategies?
- A bit too big to comfortably draw in extensive form
- Too big to guess equilibria, they have to be computed (by the "method of pain")
- Standard model is continuous, uses calculus
- The real killer: Philipp is interested in parameterised families – beyond my programming ability

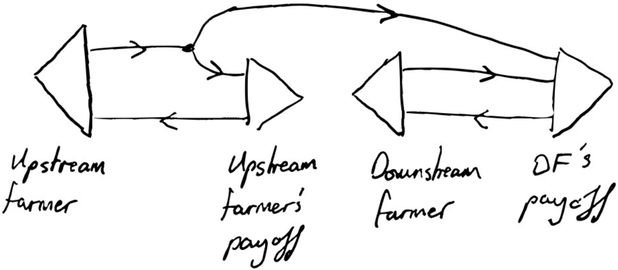
[11] Joint work with Philipp Zahn

Common pool resource situations[12]

- People face a social dilemma (aka prisoner's dilemma) to either follow the rules, or break the rules
- Breaking the rules is individually better but globally worse
- Example: Farmers on an irrigation channel, how much water to take?
- Real people will take too much, described by Nash equilibrium
- Add a monitor and penalties for discovered rule-breaking
- How to incentivise the monitor? One idea: Give them the bottom plot
- Question: We want to explore which institutional configurations lead to a 'good' equilibrium

[12] Joint work with Seth Frey and Joshua Tan, based on famous work of Elinor Ostrom

Step 1: Social dilemma



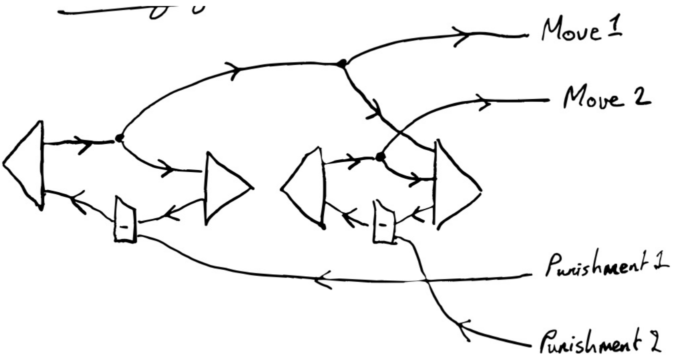
Step 1 code

```
monitoringGame1Src   = Block □ □
                       [Line □ □ "reindex const (decision \"upstreamFarmer\" [Crack, Flood])" ["x"] ["payoff1 x"],
                        Line □ □ "reindex const (decision \"downstreamFarmer\" [Crack, Flood])" ["y"] ["payoff2 x y"]]
                       □ □
```

```
data FarmerMove = Crack | Flood deriving (Eq, Ord, Show)

payoff1 :: FarmerMove -> Rational
payoff1 Crack = 2
payoff1 Flood = 4

payoff2 :: FarmerMove -> FarmerMove -> Rational
payoff2 _ Crack = 2
payoff2 Flood Flood = 2
payoff2 Crack Flood = 4

monitoringGame1Eq = equilibrium monitoringGame1 trivialContext
```

```
> monitoringGameEq (certainly Flood, certainly Crack)
□
> monitoringGameEq (certainly Flood, fromFreqs [(Flood, 2), (Crack, 1)])
□
> monitoringGameEq (uniform [Flood, Crack], fromFreqs [(Flood, 2), (Crack, 1)])
[DiagnosticInfo {player = "upstreamFarmer", observedState = "()", unobservedState = "((),())", strategy = "
fromFreqs [(Crack,1 % 2),(Flood,1 % 2)]", payoff = "3 % 1", optimalMove = "Flood", optimalPayoff = "4 % 1"}
,DiagnosticInfo {player = "downstreamFarmer", observedState = "()", unobservedState = "((),Flood)", strateg
y = "fromFreqs [(Flood,2 % 3),(Crack,1 % 3)]", payoff = "8 % 3", optimalMove = "Flood", optimalPayoff = "3
% 1"},DiagnosticInfo {player = "downstreamFarmer", observedState = "()", unobservedState = "((),Crack)", st
rategy = "fromFreqs [(Flood,2 % 3),(Crack,1 % 3)]", payoff = "8 % 3", optimalMove = "Flood", optimalPayoff
= "3 % 1"}]
```

Step 2: Opening the boundary



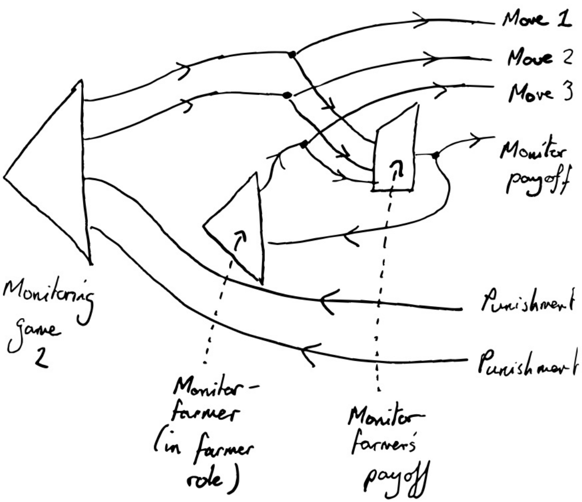
Lesson: We had to think ahead to later in order to design a good abstraction (subtracting punishments) now

Step 2 code[13]

```
monitoringGame2Src    = Block [] []
                        [Line [] [] "reindex const (decision \"upstreamFarmer\" [Crack, Flood])" ["x"] [
"payoff1 x - punishment1"],
                         Line [] [] "reindex const (decision \"downstreamFarmer\" [Crack, Flood])" ["y"]
 ["payoff2 x y - punishment2"]]
                        ["x", "y"] ["punishment1", "punishment2"]
```

[13] Yes, I'm going to show all the code, this is *applied*

Step 5: farmer is monitor

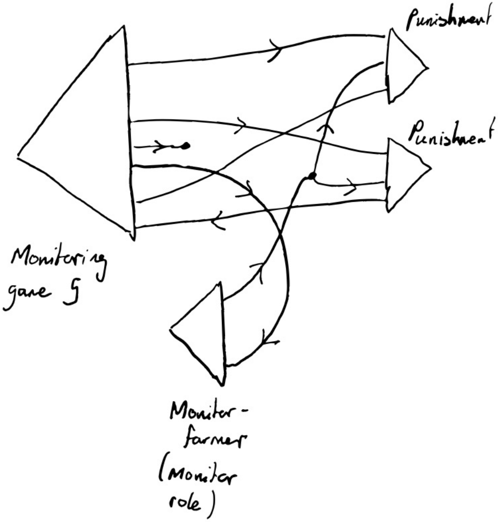


Step 5 code

```
monitoringGame5Src = Block [] []
                             [Line [] [] "monitoringGame2" ["farmerMove1", "farmerMove2"] ["punishment1", "
punishment2"],
                              Line [] [] "reindex const (decision \"starvedFarmer\" [Flood])" ["farmerMove3
"] ["monitorPayoff"],
                              Line ["(farmerMove1, farmerMove2, farmerMove3)"] [] "fromFunctions (\\(x,y,z)
 -> payoff3 x y z) (id)" ["monitorPayoff"] []]
                             ["farmerMove1", "farmerMove2", "farmerMove3", "monitorPayoff"] ["punishment1",
 "punishment2"]
```

```
data MonitorMove = Work | Shirk deriving (Eq, Ord, Show)

monitorPayoff :: MonitorMove -> Rational
monitorPayoff Work = -1
monitorPayoff Shirk = 0

punisher :: FarmerMove -> MonitorMove -> Rational
punisher _ Shirk = 0
punisher Crack Work = 0
punisher Flood Work = 3

monitoringGame3Eq = equilibrium monitoringGame3 trivialContext
```

Step 6: adding monitor back

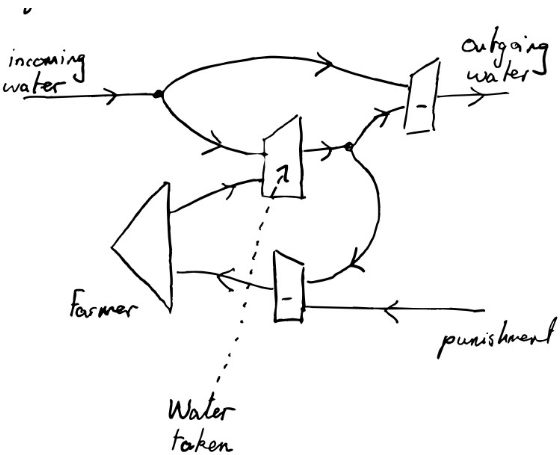


Step 6 code

```
monitoringGame6Src = Block □ □
                         [Line □ □ "monitoringGame5" ["farmerMove1", "farmerMove2", "farmerMove3", "m
onitorPayoff"] ["punisher farmerMove1 monitorMove", "punisher farmerMove2 monitorMove"],
                          Line □ □ "reindex const (decision \"monitor\" [Work, Shirk])" ["monitorMove
"] ["monitorPayoff"]]
                         □ □
```

```
> monitoringGame6Eq (certainly Flood, certainly Flood, certainly Flood, certainly Work)
[DiagnosticInfo {player = "upstreamFarmer", observedState = "()", unobservedState = "((((),()),()),())",
strategy = "fromFreqs [(Flood,1 % 1)]", payoff = 1 % 1, optimalMove = "Crack", optimalPayoff = 2 % 1},Dia
gnosticInfo {player = "downstreamFarmer", observedState = "()", unobservedState = "((((),()),()),Flood)",
 strategy = "fromFreqs [(Flood,1 % 1)]", payoff = (-1) % 1, optimalMove = "Crack", optimalPayoff = 2 % 1}
]
> monitoringGame6Eq (certainly Flood, certainly Flood, certainly Flood, certainly Shirk)
□
> monitoringGame6Eq (certainly Crack, certainly Crack, certainly Flood, certainly Shirk)
[DiagnosticInfo {player = "upstreamFarmer", observedState = "()", unobservedState = "((((),()),()),())",
strategy = "fromFreqs [(Crack,1 % 1)]", payoff = 2 % 1, optimalMove = "Flood", optimalPayoff = 4 % 1},Dia
gnosticInfo {player = "downstreamFarmer", observedState = "()", unobservedState = "((((),()),()),Crack)",
 strategy = "fromFreqs [(Crack,1 % 1)]", payoff = 2 % 1, optimalMove = "Flood", optimalPayoff = 4 % 1}]
> monitoringGame6Eq (certainly Crack, certainly Crack, certainly Flood, certainly Work)
□
```

Step 7: a better abstraction

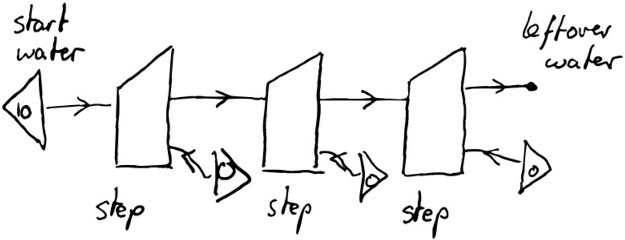


Step 7 code

```
irrigationStepSrc = Block ["startLevel"] []
                          [Line [] [] "reindex const (decision \"farmer\" [Crack, Flood])" ["farmerMove"]
 ["farmerWater startLevel farmerMove - punishment"]]
                          ["startLevel - farmerWater startLevel farmerMove"] ["punishment"]
```

```
farmerWater :: Rational -> FarmerMove -> Rational
farmerWater startLevel Crack = if startLevel >= 2 then 2 else startLevel
farmerWater startLevel Flood = if startLevel >= 5 then 5 else startLevel
```

Step 8: Following the water



Step 8 code

```
monitoringGame7Src = Block □ □
                     [Line ["10"] □ "irrigationStep" ["levelAfter1"] ["0"],
                      Line ["levelAfter1"] □ "irrigationStep" ["levelAfter2"] ["0"],
                      Line ["levelAfter2"] □ "irrigationStep" ["levelAfter3"] ["0"]]
                     □ □
```

```
> monitoringGame7Eq (certainly Flood, certainly Flood, certainly Flood)
□
> monitoringGame7Eq (certainly Flood, certainly Flood, certainly Crack)
□
> monitoringGame7Eq (certainly Flood, certainly Crack, certainly Flood)
[DiagnosticInfo {player = "farmer", observedState = "()", unobservedState = "(((),5 % 1),5 % 1)", strateg
y = "fromFreqs [(Crack,1 % 1)]", payoff = 2 % 1, optimalMove = "Flood", optimalPayoff = 5 % 1}]
> monitoringGame7Eq (certainly Crack, certainly Flood, certainly Flood)
[DiagnosticInfo {player = "farmer", observedState = "()", unobservedState = "(((),()),10 % 1)", strategy
= "fromFreqs [(Crack,1 % 1)]", payoff = 2 % 1, optimalMove = "Flood", optimalPayoff = 5 % 1}]
```

General conclusions for ACT

- Number 1 conclusion:

  Realising the promised benefits of ACT is still hard

- Need detailed and equal dialogue between theory & domain experts
- Interdisciplinary work is very costly
- Designing good abstractions will always be an art form
- Software is necessary, string diagrams software not necessary
- String diagrams may not even be the best representation!

Economics as worst case scenario

- This is not publishable in economics venues
- Project has not paid off for my collaborators
- Hard to get new economists in, can't do without them[14]
- Need publications to signal for interdisciplinary funding
- Maybe a place in business, but risky

[14] They only get more important over time!

Ending on good news

- Evidence that ACT can support a modelling workflow
- Many types of learning share a common categorical foundation
- $\Longrightarrow$ towards categorical cybernetics (aka CyberCat)