Effectful Programming across Heterogeneous Computations

Work in Progress: Usefulness and Novelty are not Guaranteed!

Jean Abou Samra
Martin Bodin
Yannick Zakowski

JFLA'23

Monadic Computations

The Free Monad

Heterogeneous Monadic Programming

Mixing Up the Free Monad

Payoff and Perspectives

Monadic Computations

Monadic Computations

In a statically typed functional language

A way to encode and program with effectful computations

One monad = One class of computations

Pure Computations

Terms of type X are pure computations returning values of type X

`(λx ⇒ x * x) 2` $\rightsquigarrow$ 4

Pure Computations

Terms of type X are pure computations returning values of type X

```
(λx ⇒ x * x) 2 ⇝ 4
```

Computations can be sequenced: it’s the let-binding operation

```
         t ⇝ v
    u[v/x] ⇝ v’
----------------------------------
 let x := t in u ⇝ v’
```

Failing Computations

Terms of type X are failing computations returning values of type X

Computations can be sequenced

Failing Computations

`None | Some (v : X)`

Terms of type option X are failing computations returning values of type X

Computations can be sequenced

Failing Computations

`None | Some (v : X)`

Terms of type option X are failing computations returning values of type X

The family of types option is the failure monad

Computations can be sequenced

Failing Computations

`None | Some (v : X)`

Terms of type option X are failing computations returning values of type X

The family of types option is the failure monad

Computations can be sequenced: it's the bind operation

```
    t ⇝ Some v
  u[v/x] ⇝ v'
--------------------
 x ← t;; u ⇝ v'
```

```
     t ⇝ None
--------------------
 x ← t;; u ⇝ None
```

Failing Computations

`None | Some (v : X)`

Terms of type option X are failing computations returning values of type X

Pure computations can be embedded: it's the ret operation (`ret ≜ Some`)

Computations can be sequenced: it's the bind operation

$$\frac{\begin{array}{c} \mathtt{t} \rightsquigarrow \mathtt{Some\ v} \\ \mathtt{u[v/x]} \rightsquigarrow \mathtt{v'} \end{array}}{\mathtt{x} \leftarrow \mathtt{t;;\ u} \rightsquigarrow \mathtt{v'}} \qquad \frac{\mathtt{t} \rightsquigarrow \mathtt{None}}{\mathtt{x} \leftarrow \mathtt{t;;\ u} \rightsquigarrow \mathtt{None}}$$

Stateful Computations

Terms of type $S \rightarrow S*X$ are stateful computations returning values of type X

The family of types $X \longmapsto (S \rightarrow S*X)$ is the stateful monad

$$\mathtt{ret}\ x \triangleq \lambda\sigma \Rightarrow (\sigma, x)$$

$$\frac{\begin{array}{c} t\ \sigma \rightsquigarrow (\sigma', v) \\ u[v/x]\ \sigma' \rightsquigarrow (\sigma'', v') \end{array}}{(x \leftarrow t;;\ u)\ \sigma \rightsquigarrow (\sigma'', v')}$$

Monadic Computations: A Convenient Abstraction

A type family `M : Type → Type` is a monad if it comes equipped with:

Monadic Computations: A Convenient Abstraction

A type family `M : Type → Type` is a monad if it comes equipped with:

```
ret  : ∀ X, X → M X
```

Monadic Computations: A Convenient Abstraction

A type family `M : Type → Type` is a monad if it comes equipped with:

```
ret  : ∀ X, X → M X

bind : ∀ X Y, M X → (X → M Y) → M Y
```

Monadic Computations: A Convenient Abstraction

A type family `M : Type → Type` is a monad if it comes equipped with:

```
ret  : ∀ X, X → M X

bind : ∀ X Y, M X → (X → M Y) → M Y

equ  : ∀ X, M X → M X → Prop
```

Monadic Computations: a Convenient Abstraction

A type family `M : Type → Type` is a monad if it comes equipped with:

```
ret  : ∀ X, X → M X

bind : ∀ X Y, M X → (X → M Y) → M Y
```

Monad laws:

```
x ← ret x;; k                = k x
x ← c;; ret x                = c
x ← c;; (λ x ⇒ y ← k x;; g) = y ← (x ← c;;k);; g
```

Used in Proof Assistants in Particular

Gallina

A pure functional language… So pure every function must terminate!

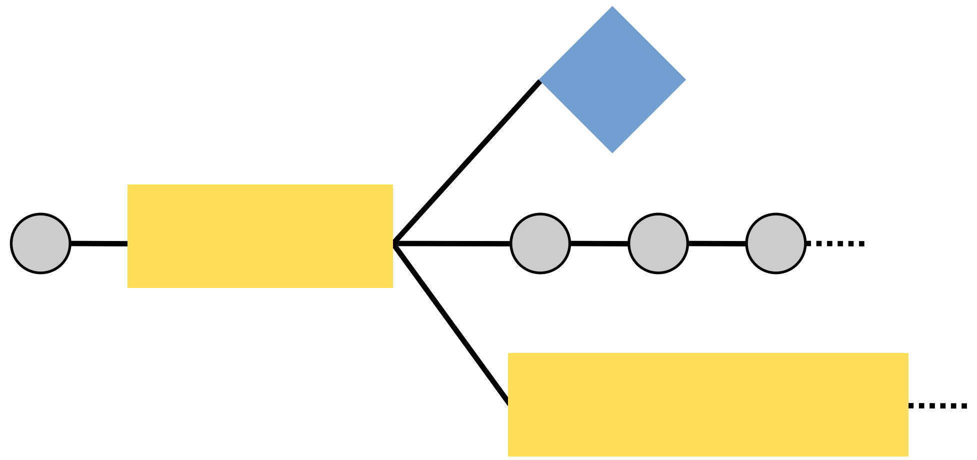
Monads are a convenient abstraction to represent effecful computations in Gallina as well as to reason about these computations

We discussed failure and state, but **divergence** can be represented as well!

One Specific Library

Gallina

Interaction Trees



A *generic toolkit* to define **and** reason about the semantics of interactive systems

Semantics: Compositional, Modular, Executable

Reasoning: Equational, termination sensitive

The Free Monad

The Free Monad: an Extensible Syntax

Effectful computations arise from their signature of operations

```
free E X
```

My computation is a glorified piece of syntax

able to perform operations specified in E

in order to compute a value of type X

Modelling Imp as Stateful Computations

Imperative programs are stateful computations

Syntax `p ≜ x := n; y := x`

Model `S → S * unit`

Modelling Imp as Stateful Computations

Imperative programs are stateful computations

Syntax `p ≜ x := n; y := x`

Model `S → S * unit`

$$[\![p]\!] = \lambda\sigma \Rightarrow \sigma[x \leftarrow n][y \leftarrow \sigma[x]]$$

Modelling Imp as Stateful Computations

Imperative programs are stateful computations

<u>Syntax</u> `p ≜ x := n; y := x`

<u>Model</u> `S → S * unit`

$$[\![p]\!] = \lambda\sigma \Rightarrow \sigma[x \leftarrow n][y \leftarrow \sigma[x]]$$

And hence one can **prove**

$$[\![p]\!] = \lambda\sigma \Rightarrow \sigma[x \leftarrow n,\ y \leftarrow n]$$

The Free Monad: an Extensible Syntax

Imperative programs are computations performing reads and writes

<u>Syntax</u> `p ≜ x := n; y := x`

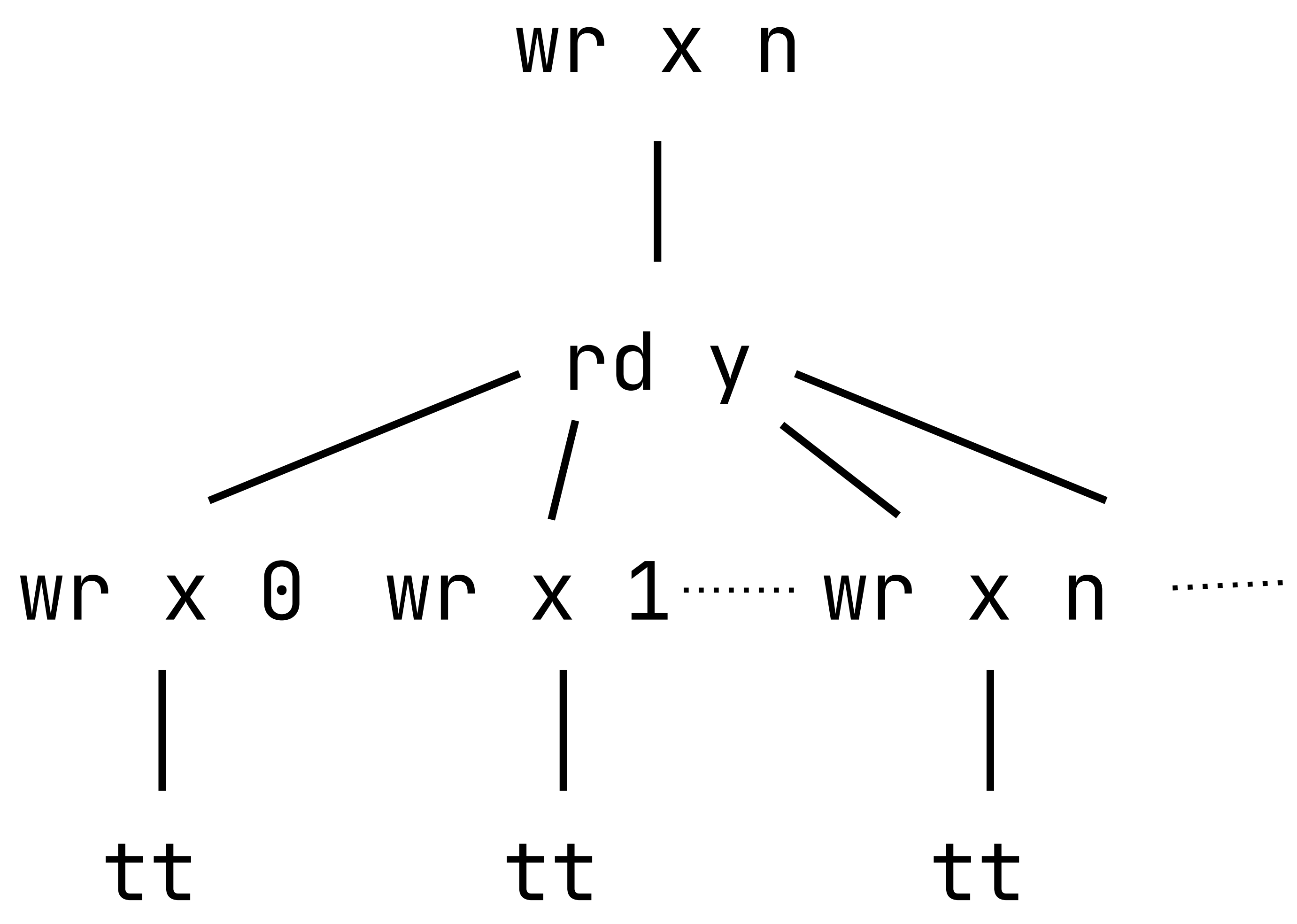
<u>Model</u> `free RW unit`

The Free Monad: an Extensible Syntax

Imperative programs are computations performing reads and writes

Syntax `p ≜ x := n; y := x`

Model `free RW unit`



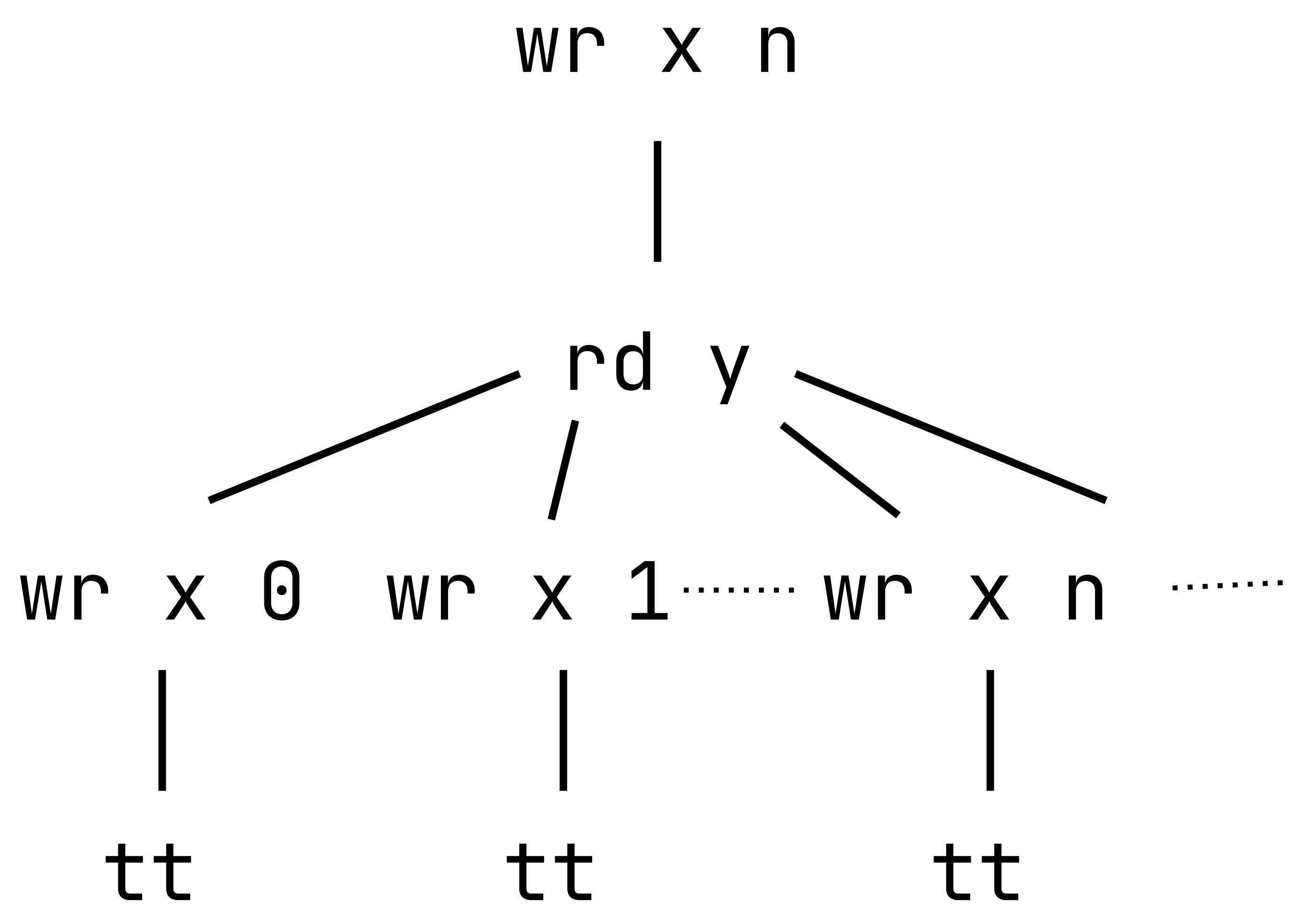
The Free Monad: an Extensible Syntax

Imperative programs are computations performing reads and writes

Syntax `p ≜ x := n; y := x`

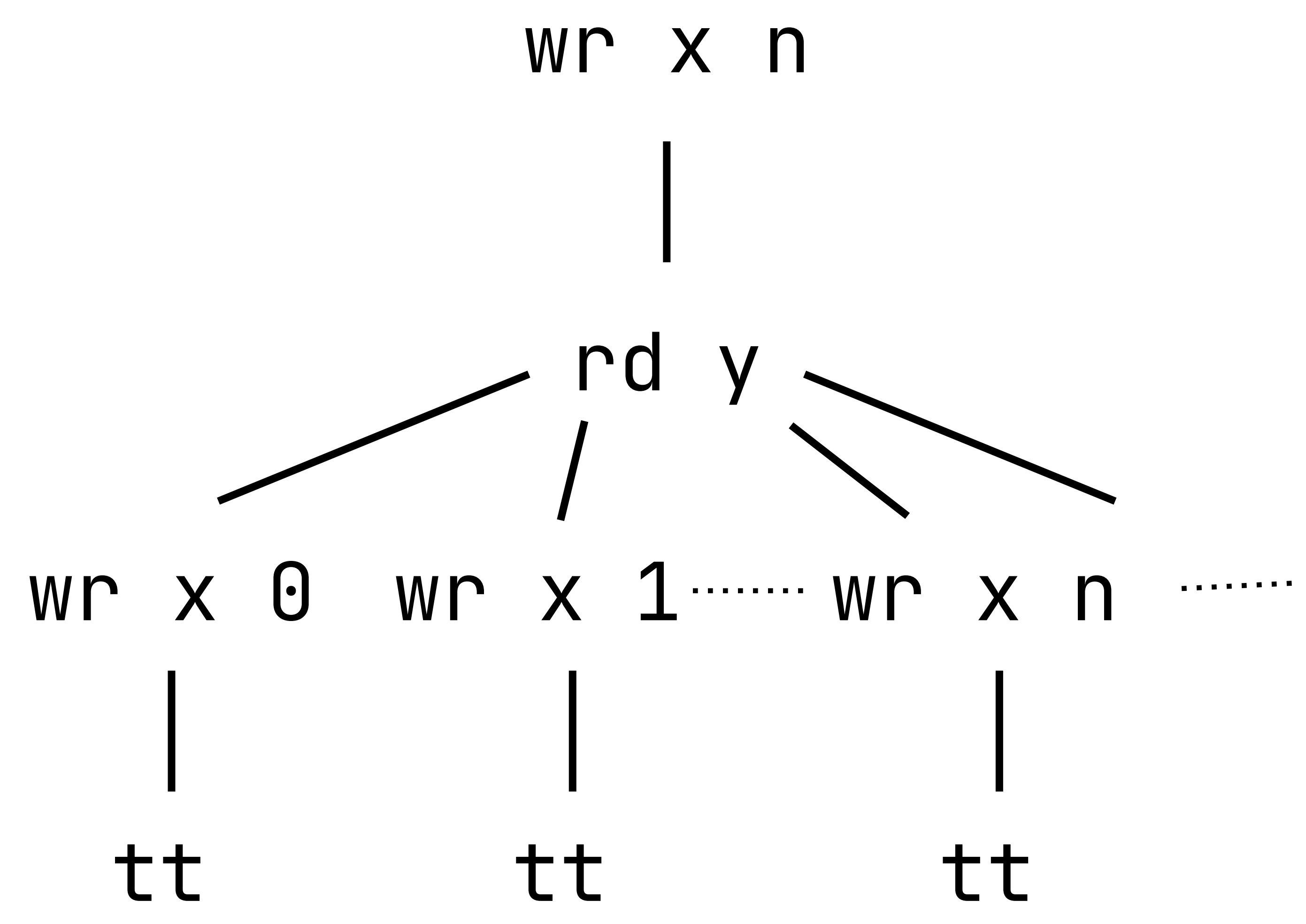
Model `free RW unit`

One **cannot prove** that we write n to y



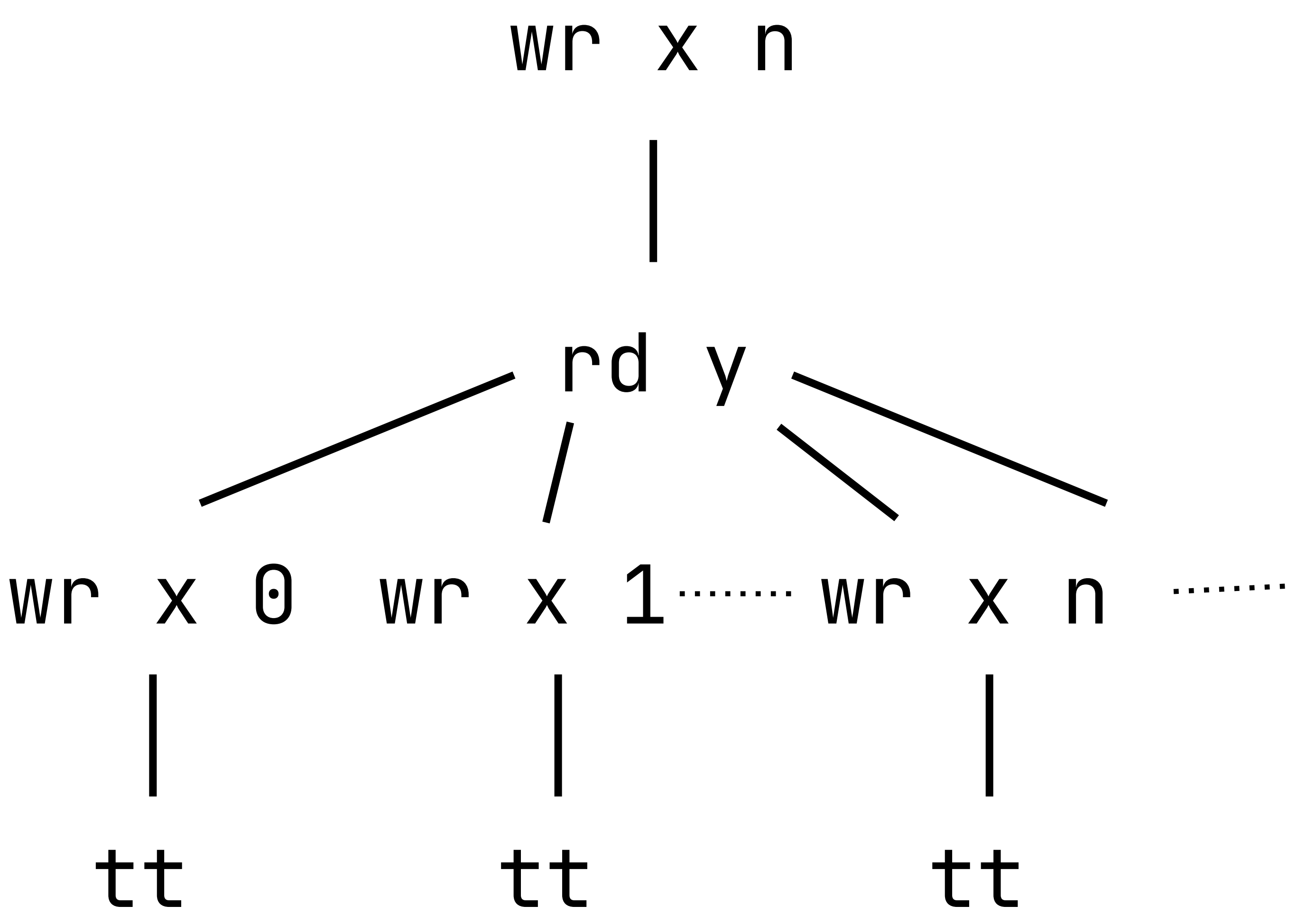
The Free Monad: Interpretation

Imperative programs are computations performing reads and writes implemented as stateful computations



The Free Monad: Interpretation

Imperative programs are computations performing reads and writes implemented as stateful computations



$$\xmapsto{\text{Interpretation}} \quad \lambda\sigma \Rightarrow \sigma[x\leftarrow n][y\leftarrow\sigma[x]]$$

The state monad is a possible implementation of the operations

Used at Scale to Model Languages: LLVM IR

Local state L_E

Global state G_E

Memory M_E

Calls $Call_E$

Stack of local frames SF_E

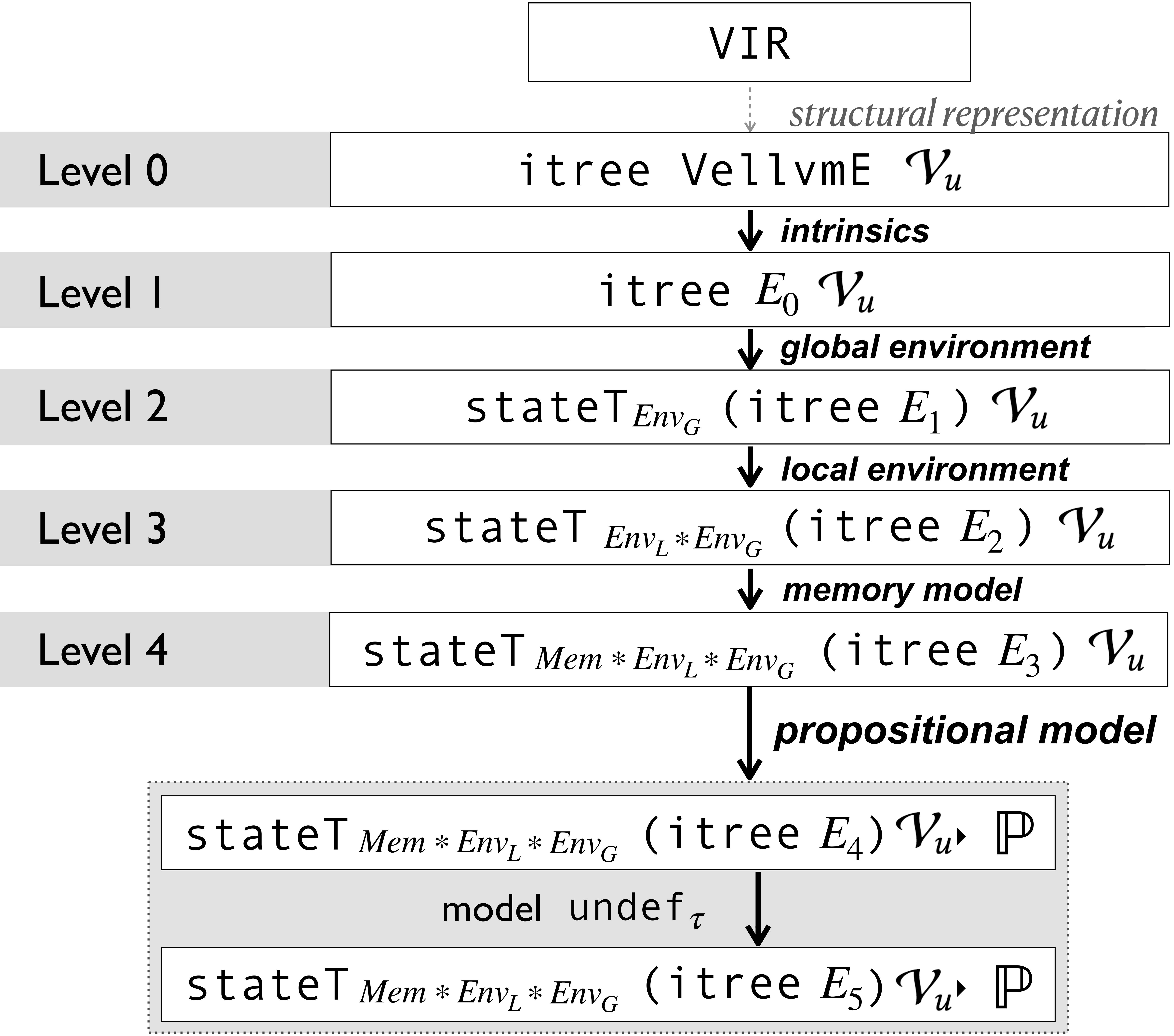
Non-determinism $Pick_E$

Undefined Behavior UB_E

Debugging $Debug_E$

Failure $FailE$

LLVM IR programs are computations performing
reads and writes to the local register;
read an writes to the global state;
interacting with the memory;
…



VIR

structural representation

Level 0 `itree VellvmE` $\mathcal{V}_u$

intrinsics

Level 1 `itree` E_0 $\mathcal{V}_u$

global environment

Level 2 `stateT`$_{Env_G}$ `(itree` E_1 `)` $\mathcal{V}_u$

local environment

Level 3 `stateT` $_{Env_L * Env_G}$ `(itree` E_2 `)` $\mathcal{V}_u$

memory model

Level 4 `stateT` $_{Mem * Env_L * Env_G}$ `(itree` E_3 `)` $\mathcal{V}_u$

propositional model

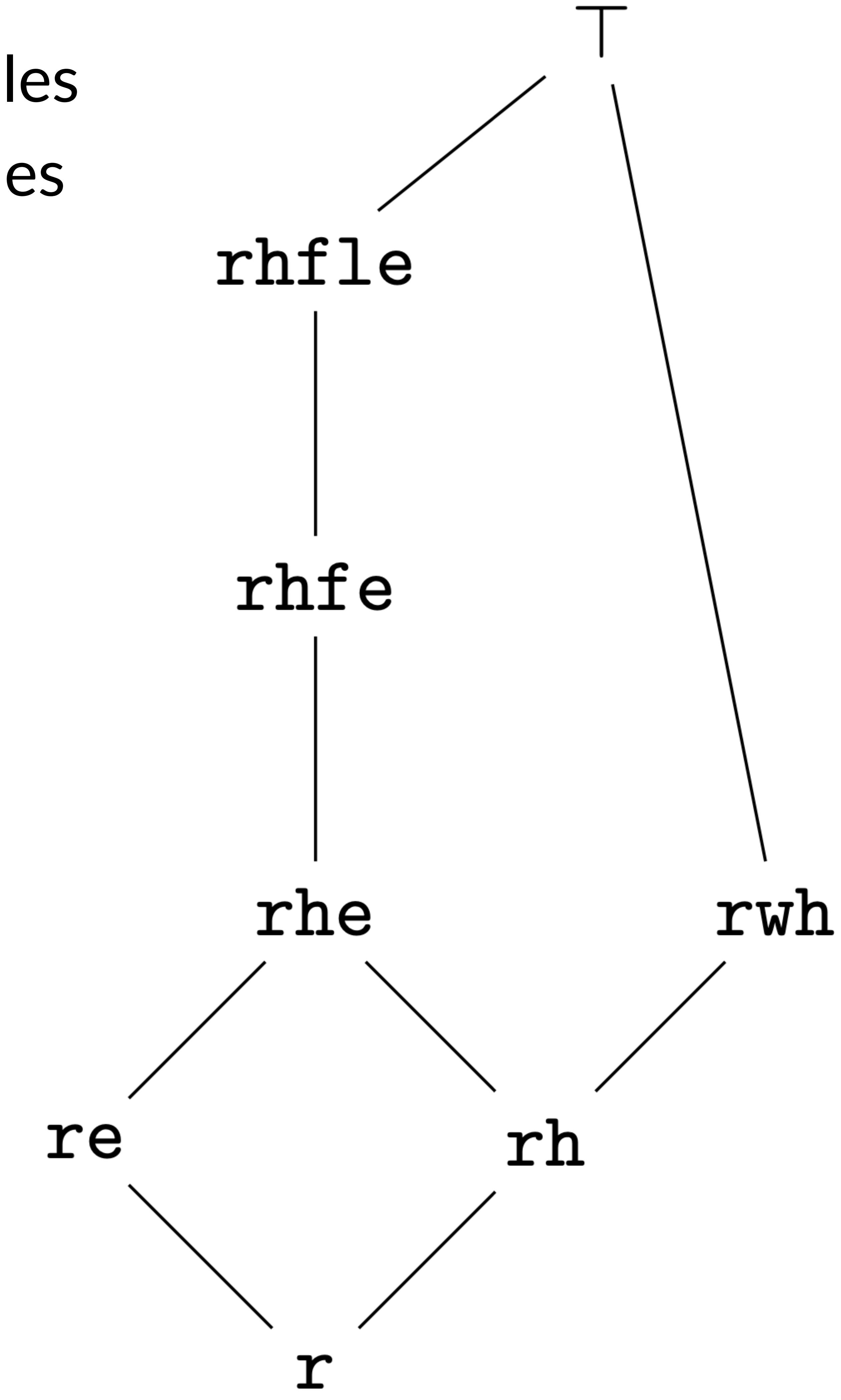
`stateT` $_{Mem * Env_L * Env_G}$ `(itree` E_4`)` $\mathcal{V}_u \blacktriangleright \mathbb{P}$

model `undef`$_\tau$

`stateT` $_{Mem * Env_L * Env_G}$ `(itree` E_5`)` $\mathcal{V}_u \blacktriangleright \mathbb{P}$

Used at Scale to Model Languages: R

r : reading global variables
w : writing global variables
h : heap operations
e : throwing errors
f : function calls
l : low level operations

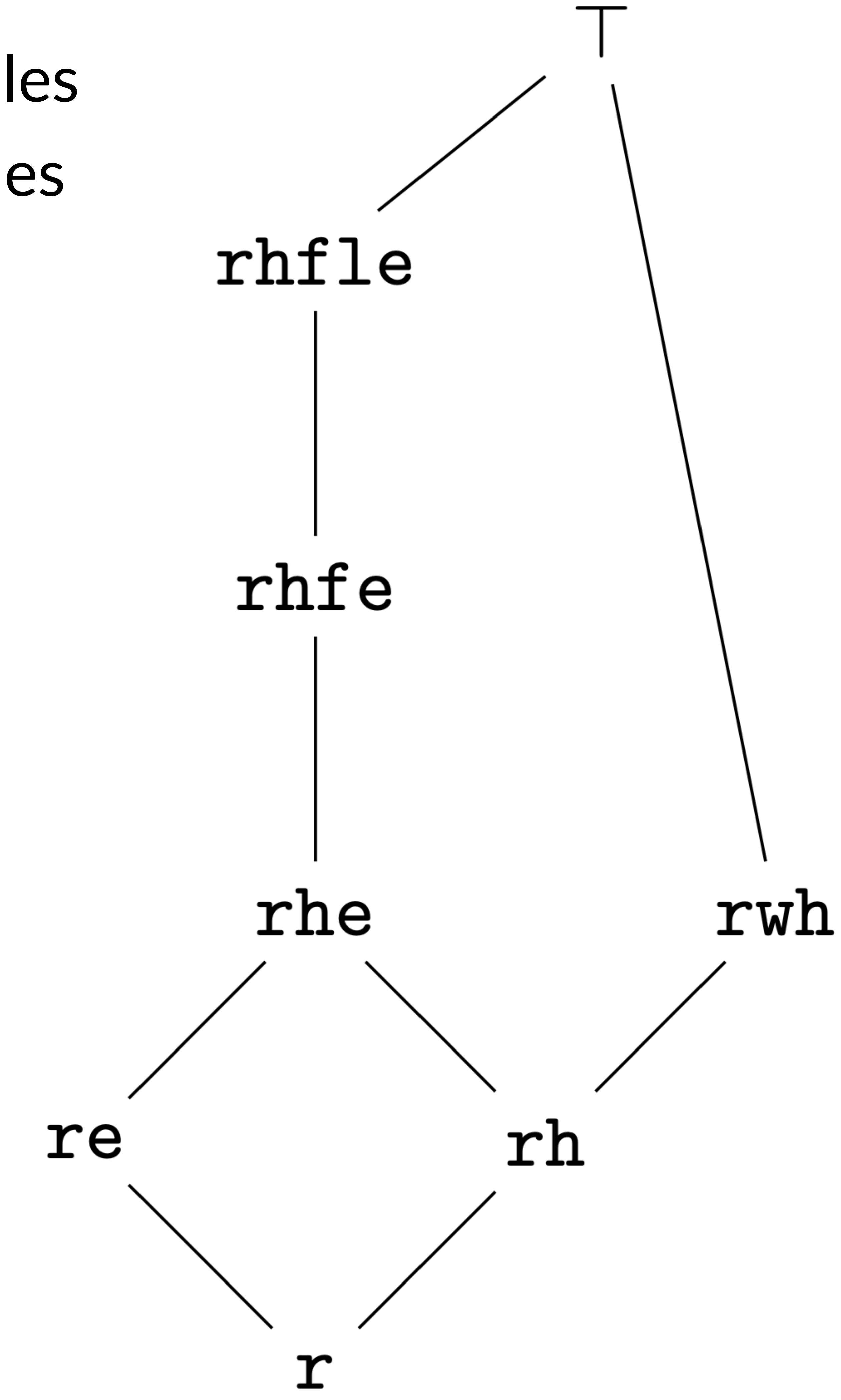


18kloc of monadic interpreter

but **not** uniformly complex at closer inspection

Used at Scale to Model Languages: R

r : reading global variables
w : writing global variables
h : heap operations
e : throwing errors
f : function calls
l : low level operations



18kloc of monadic interpreter

but **not** uniformly complex at closer inspection

We **should not** have to reason in a structure implementing all effects

Heterogeneous Monadic Programming

Can we write monadic interpreters at scale

on top of the free monad

combining computations at different types

and get invariants for free and proofs in simpler structures?

Heterogeneous Stateful Programming

Let's consider a single cell containing a natural number

Operations	Structure

Heterogeneous Stateful Programming

Let’s consider a single cell containing a natural number

Operations	Structure	
reads and writes	`state X`	`c → c * X`

Heterogeneous Stateful Programming

Let’s consider a single cell containing a natural number

Operations	Structure	
reads and writes	state X	`c → c * X`
reads	read X	`c → X`

Heterogeneous Stateful Programming

Let’s consider a single cell containing a natural number

Operations	Structure	
reads and writes	state X	`c → c * X`
reads	read X	`c → X`
writes	write X	`c * X`

Heterogeneous Stateful Programming

Let’s consider a single cell containing a natural number

Operations	Structure	
reads and writes	state X	`c → c * X`
reads	read X	`c → X`
writes	write X	`c * X`
nothing	pure X	`X`

Heterogeneous Monadic Programming

Heterogeneous Monadic Programming

Via Monad Morphisms

Transport via Monad Morphism

Computations in a given monad can be sequenced

How could I sequence computations in two distinct monads?

Transport via Monad Morphism

Computations in a given monad can be sequenced

How could I sequence computations in two distinct monads?

Have them agree to meet at a common place!

Transport via Monad Morphism

Monad morphism: a structure preserving map

```
Class Morph M T := { morph : ∀ X, M X → T X }
```

```
morph (c : M X) : T X ≜ c ▷ T
```

Transport via Monad Morphism

Monad morphism: a structure preserving map

```
Class Morph M T := { morph : ∀ X, M X → T X }
```

```
morph (c : M X) : T X ≜ c ▷ T
```

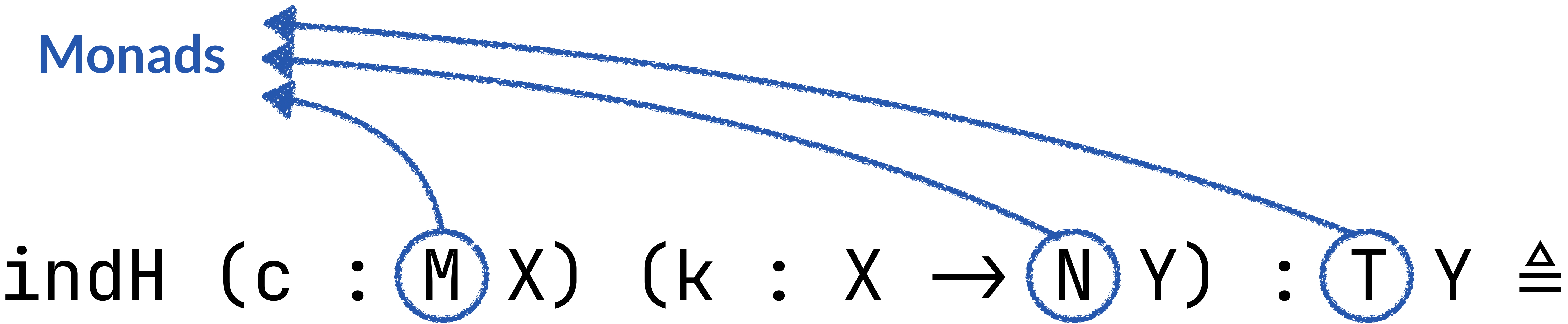
```
bindH (c : M X) (k : X → N Y) : T Y ≜
    x ← (c ▷ T) ((k x) ▷T)
```

Transport via Monad Morphism

Monad morphism: a structure preserving map

```
Class Morph M T := { morph : ∀ X, M X → T X }
```

```
morph (c : M X) : T X ≜ c ▷ T
```



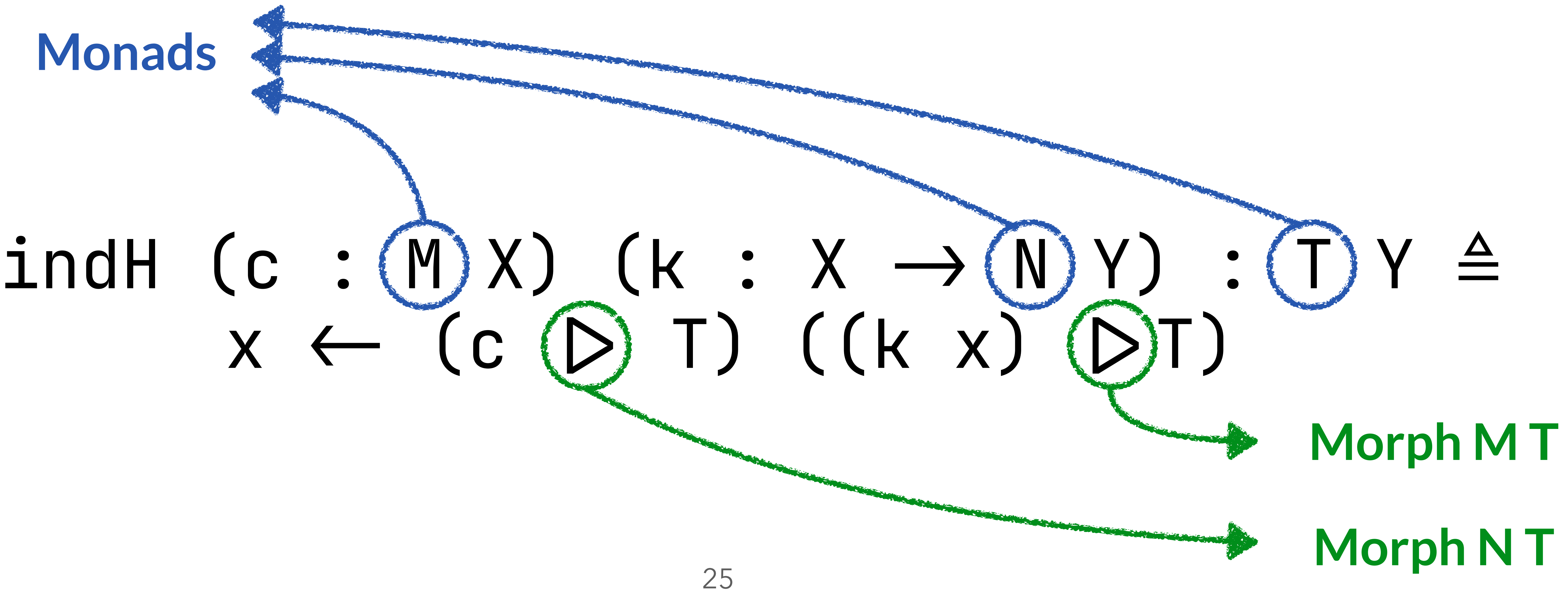
```
bindH (c : M X) (k : X → N Y) : T Y ≜
    x ← (c ▷ T) ((k x) ▷T)
```

Transport via Monad Morphism

Monad morphism: a structure preserving map

```
Class Morph M T := { morph : ∀ X, M X → T X }
```

```
morph (c : M X) : T X ≜ c ▷ T
```



```
bindH (c : M X) (k : X → N Y) : T Y ≜
      x ← (c ▷ T) ((k x) ▷ T)
```

Transport via Monad Morphism

Monad morphism: a structure preserving map

```
Class Morph M T := { morph : ∀ X, M X → T X }
```

```
morph (c : M X) : T X ≜ c ▷ T
```

Problem: how should we help Coq infer T?

```
bindH (c : M X) (k : X → N Y) : T Y ≜
    x ← (c ▷ T) ((k x) ▷T)
```

Transport via Monad Morphism

The monad laws generalize

$$\texttt{x} \leftarrow \underset{M}{\texttt{ret x}} \overset{T}{\texttt{;;}} \underset{N}{\texttt{k}} = \texttt{(k x)} \triangleright \texttt{T}$$

Transport via Monad Morphism

The monad laws generalize

$$\mathtt{x} \leftarrow \mathtt{ret}\ \underset{M}{\mathtt{x}}\ \overset{T}{\mathtt{;;}}\ \underset{N}{\mathtt{k}} = (\mathtt{k}\ \mathtt{x}) \triangleright \mathtt{T}$$

$$\mathtt{x} \leftarrow \underset{M}{\mathtt{c}}\ \overset{T}{\mathtt{;;}}\ \mathtt{ret}\ \underset{N}{\mathtt{x}} = \mathtt{c} \triangleright \mathtt{T}$$

Transport via Monad Morphism

The monad laws generalize

$$x \leftarrow \underset{M}{\mathtt{ret}\ x} \overset{T}{;;} \underset{N}{k} = (k\ x) \triangleright T$$

$$x \leftarrow \underset{M}{c} \overset{T}{;;} \underset{N}{\mathtt{ret}\ x} = c \triangleright T$$

$$x \leftarrow \underset{A}{c} \overset{T}{;;} (\lambda\ x \Rightarrow y \leftarrow \underset{B}{k\ x} \overset{BC}{;;} \underset{C}{g}) = y \leftarrow (x \leftarrow c \overset{AB}{;;} k) \overset{T}{;;} g$$

Transport via Monad Morphism

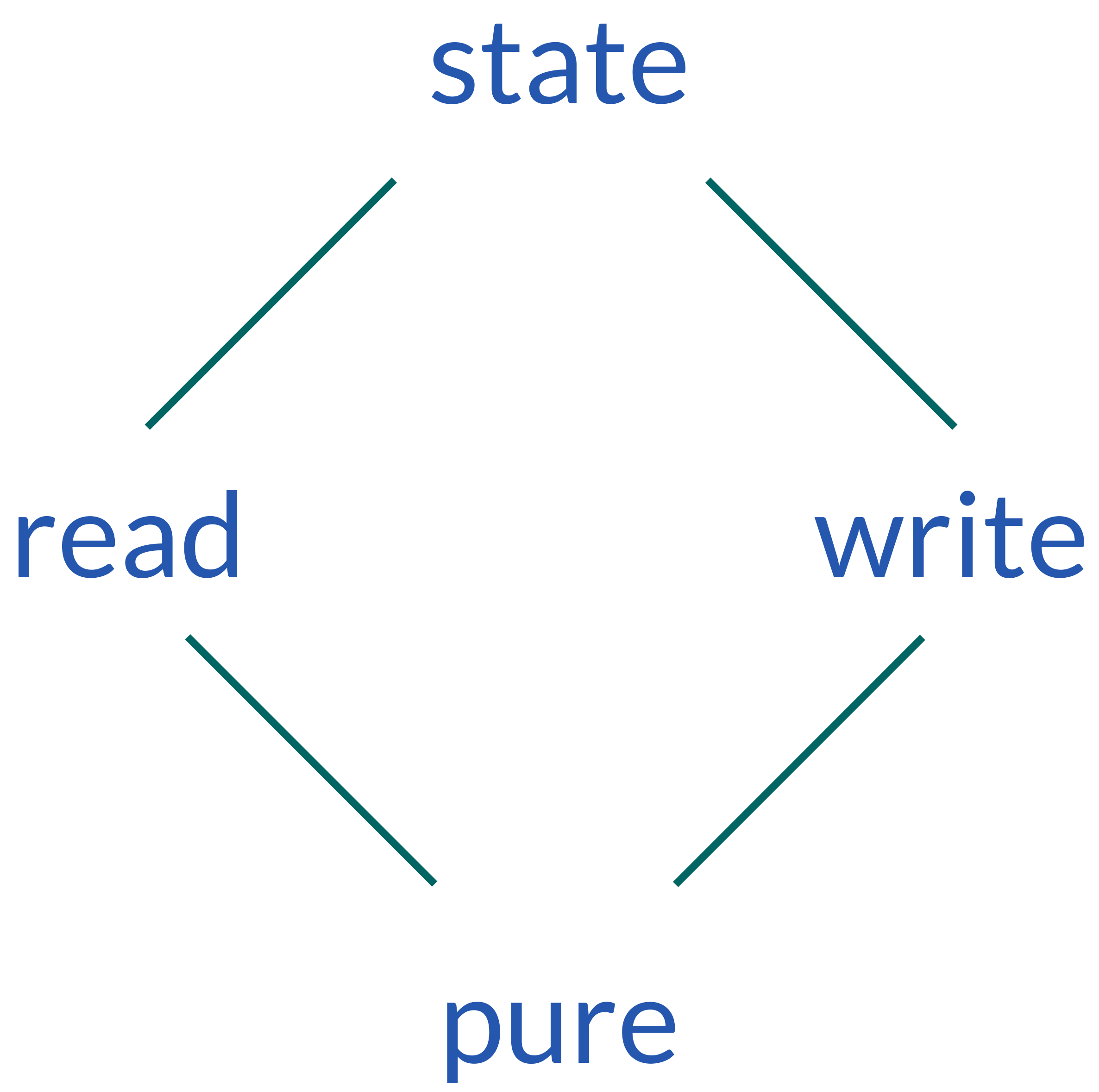
The monad laws generalize

Given the right coherence properties!

$$\begin{tikzcd}
 & & T & & \\
\circlearrowleft & AB \arrow[ur] & \circlearrowleft & BC \arrow[ul] & \circlearrowleft \\
A \arrow[ur] \arrow[uurr, bend left] & & B \arrow[ul] \arrow[ur] & & C \arrow[ul] \arrow[uull, bend right]
\end{tikzcd}$$

(More or Less) Stateful Computations

At the cost of unacceptable annotations, we can mix up computations once our morphisms are defined, and coherence properties proved



Heterogeneous Monadic Programming

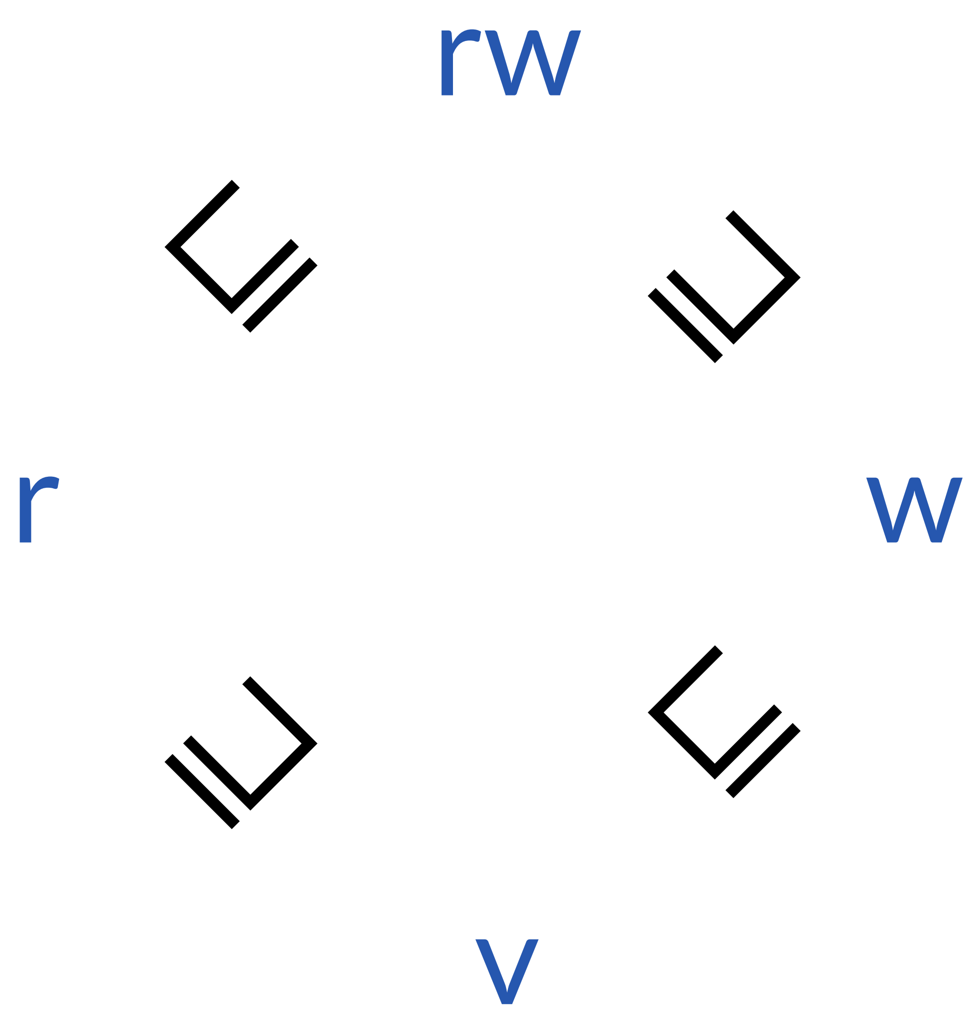
Indexation

A Partial Order to Index

$$\begin{tikzcd}
 & \text{state } X & \\
\text{read } X \arrow[ur] & & \text{write } X \arrow[ul] \\
 & \text{pure } X \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

A Partial Order to Index

`(I, ⊑) partial order`

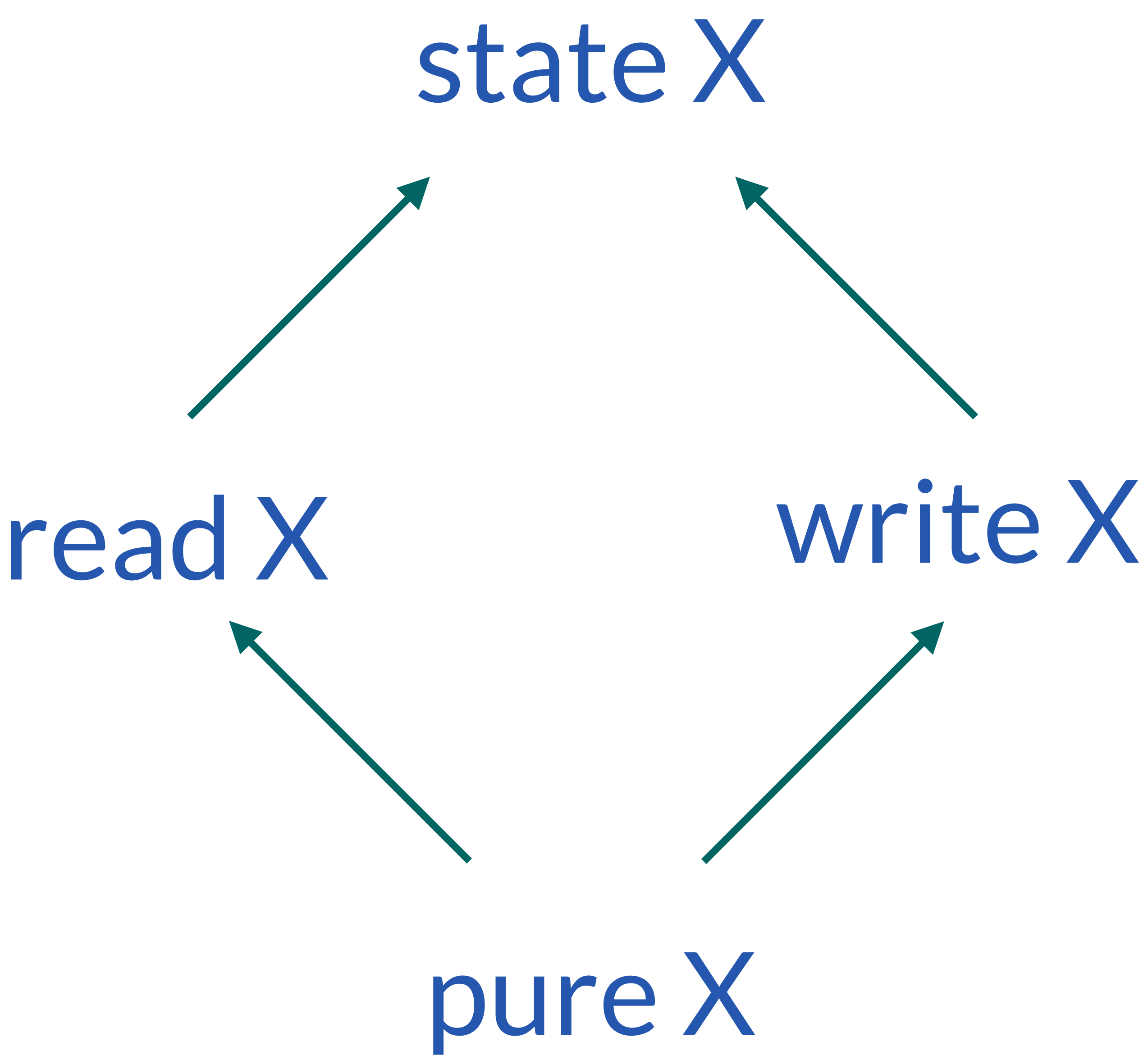


rw

r

w

v



state X

read X

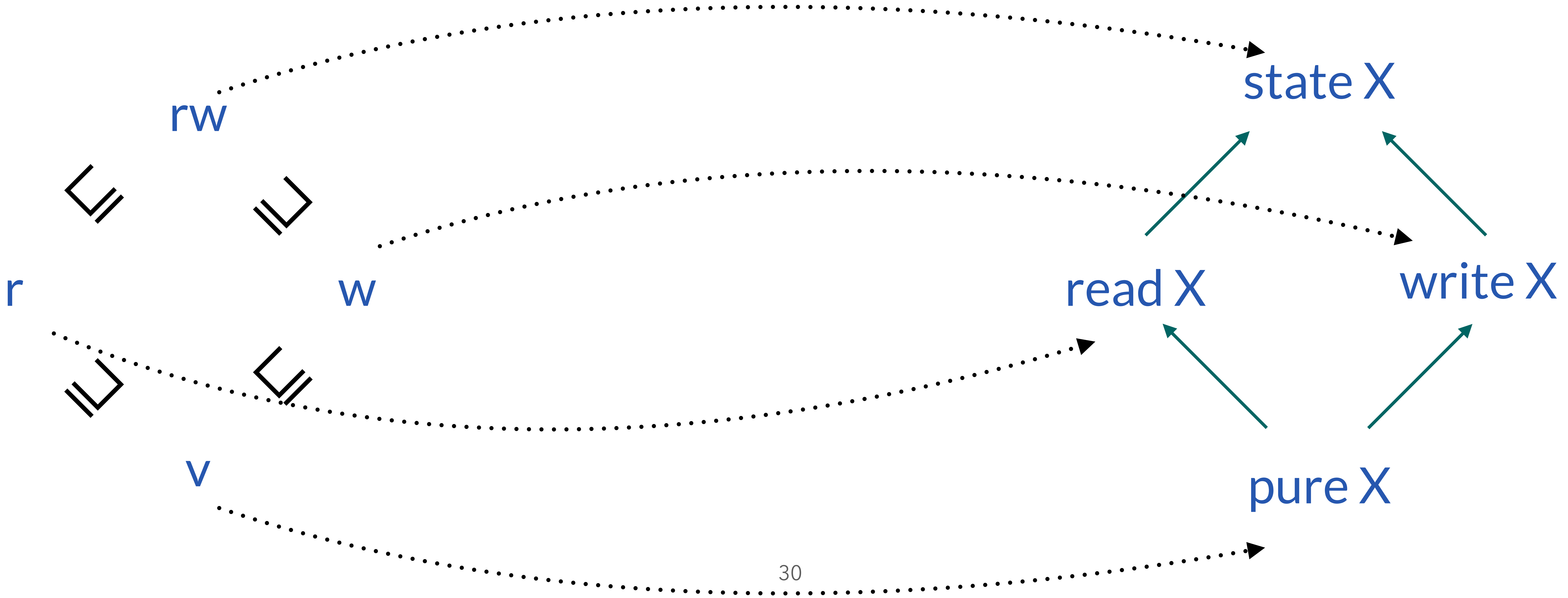
write X

pure X

A Partial Order to Index

`[|.|] : I → Monad`

`(I, ⊑) partial order`



rw

⊑ ⊒

r

w

⊒ ⊑

v

state X

read X

write X

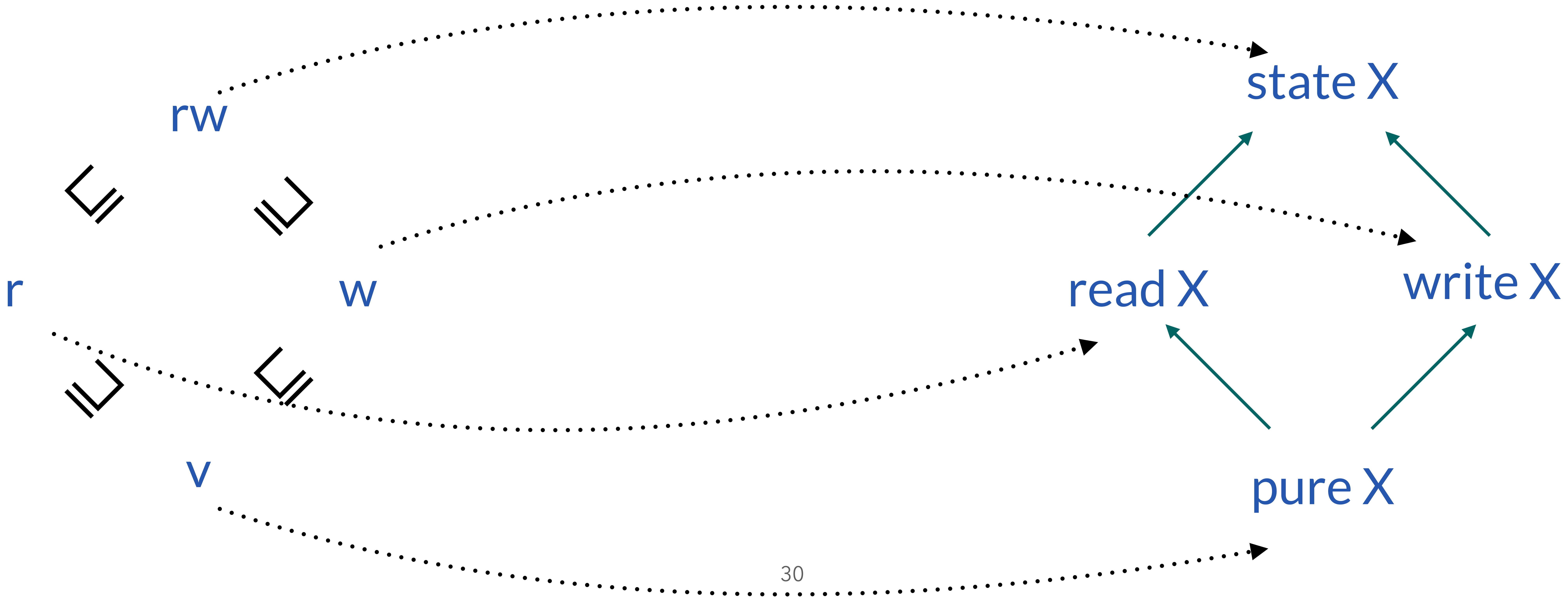
pure X

A Partial Order to Index

`[|.|] : I → Monad`

`(I, ⊑) partial order`

`reify_le : ∀ i j, i ⊑ j → Morph [|i|] [|j|]`



A Partial Order to Index

We sequence computations *across indices*

```
bindH (c : [|i|] X) (k : X → [|j|] Y) : [|t|] Y ≜
      x ← (c ▷ [|t|]) ((k x) ▷ [|t|])
```

if we can find two proofs: `i ⊑ t` and `j ⊑ t`

A Partial Order to Index

We sequence computations *across indices*

```
bindH (c : [|i|] X) (k : X → [|j|] Y) : [|t|] Y ≜
      x ← (c ▷ [|t|]) ((k x) ▷[|t|])
```

if we can find two proofs: `i ⊑ t` and `j ⊑ t`

We guarantee the coherence requirements once and for all:

```
(c ▷ (i ⊑ j)) ▷(j ⊑ k) = c ▷ (i ⊑ k)
```

Heterogeneous Monadic Programming

Directed Set

A Join to Settle on our Destination

`(I, ⊑) ordered set`

We sequence computations across indices

```
bindH (c : [|i|] X) (k : X → [|j|] Y) : [|t|] Y ≜
    x ← (c ▷ [|t|]) ((k x) ▷[|t|])
```

if we can find two proofs: `i ⊑ t` and `j ⊑ t`

A Join to Settle on our Destination

`(I, ⊑, ⊔) directed set`

We sequence computations across indices

```
bindH (c : [|i|] X) (k : X → [|j|] Y) : [|i ⊔ j|] Y ≜
      x ← (c ▷ [|i ⊔ j|]) ((k x) ▷ [|i ⊔ j|])
```

and we always have: `i ⊑ [|i ⊔ j|]` and `j ⊑ [|i ⊔ j|]`

A Join to Settle on our Destination

No annotation needed anymore!

But some dependent programming sneaks in

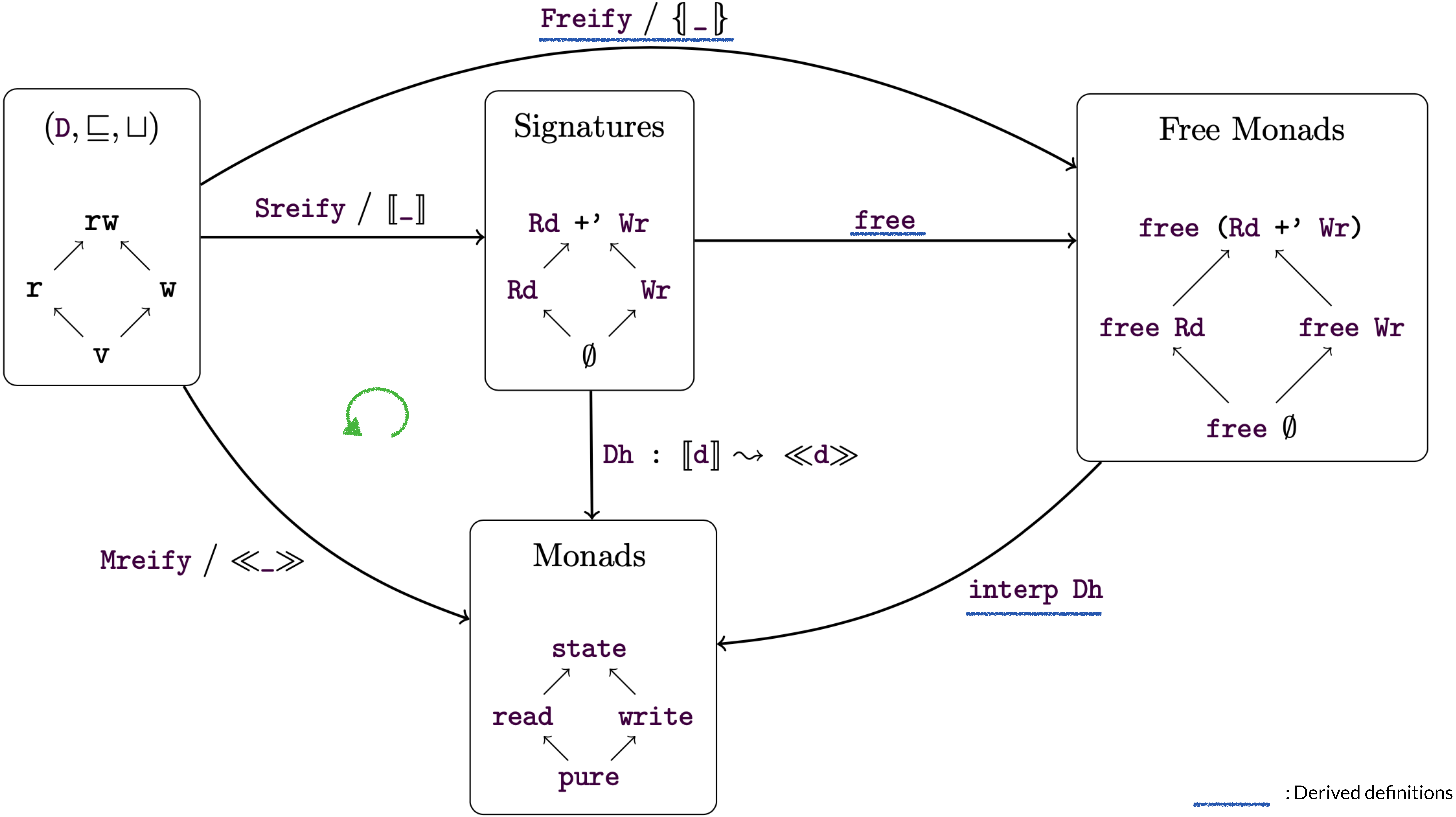
`(I, ⊑, ⊔) directed set`

We sequence computations across indices

```
bindH (c : [|i|] X) (k : X → [|j|] Y) : [|i ⊔ j|] Y ≜
      x ← (c ▷ [|i ⊔ j|]) ((k x) ▷ [|i ⊔ j|])
```

and we always have: `i ⊑ [|i ⊔ j|]` and `j ⊑ [|i ⊔ j|]`

Mixing Up the Free Monad



Freify / $\{\!|\_|\!\}$

$(\mathtt{D}, \sqsubseteq, \sqcup)$

rw

r

w

v

Sreify / $[\![\_]\!]$

Signatures

Rd +' Wr

Rd

Wr

∅

free

Free Monads

free (Rd +' Wr)

free Rd

free Wr

free ∅

Dh : $[\![\mathtt{d}]\!] \rightsquigarrow \langle\!\langle \mathtt{d} \rangle\!\rangle$

Mreify / $\langle\!\langle \_ \rangle\!\rangle$

interp Dh

Monads

state

read

write

pure

: Derived definitions

Commutation Property

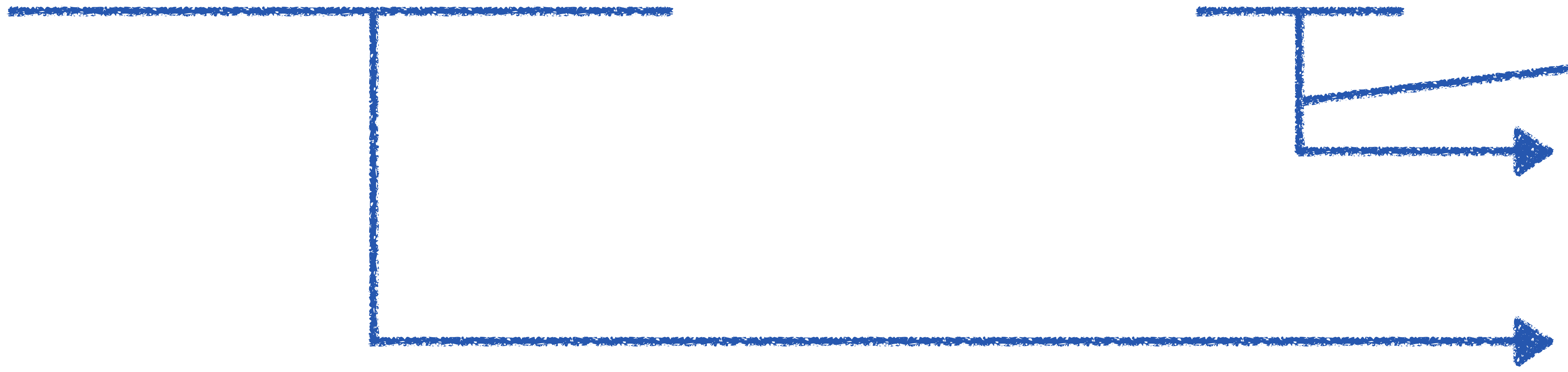
Heterogenous computations can be broken back down into their simpler components

```
∀ (c : free [|i|] X) (k : X → free [|j|] Y)

I (Fbind m k) = Mbind (I m) (λx ⇒ I (k x))
```

Commutation Property

Heterogenous computations can be broken back down into their simpler components

```
∀ (c : free [|i|] X) (k : X → free [|j|] Y)

I (Fbind m k) = Mbind (I m) (λx ⇒ I (k x))
```



Simpler monadic structures

Complex monadic structure

Payoff and Perspectives

Toy Example and Illustration

```
Variable init  : cell → free Wr unit
Variable fetch : free Rd cell
Definition main (n : cell) : free (Rd + Wr) bool =
  init n;;
  v1 ← fetch;;
  v2 ← fetch;;
  ret (v1 =? v2)
```

```
{ λb ⇒ b = true }
```

Toy Example and Illustration

```
Variable init  : cell → free Wr unit
Variable fetch : free Rd cell
Definition main (n : cell) : free (Rd + Wr) bool =
  init n;;
  v1 ← fetch;;
  v2 ← fetch;;
  ret (v1 =? v2)

{ λb ⇒ b = true }
```

➡ Instantiating our interface allows us to write the program above

Toy Example and Illustration

```
Variable init  : cell → free Wr unit
Variable fetch : free Rd cell
Definition main (n : cell) : free (Rd + Wr) bool =
  init n;;
  v1 ← fetch;;
  v2 ← fetch;;
  ret (v1 =? v2)
{ λb ⇒ b = true }
```

➡ Instantiating our interface allows us to write the program above

➡We can leverage the invariant inherent to read-only computations in the proof

Perspectives

Perspectives

- ➡ Better robustness
  - ⇝ Tactics to handle indices in types

Perspectives

- ➡ Better robustness
  - ⇝ Tactics to handle indices in types
- ➡ Clarifying how to leverage the approach
  - ⇝ Should the monads come with their specification monad à la Maillard?

Perspectives

- ➡ Better robustness
  - ⇝ Tactics to handle indices in types
- ➡ Clarifying how to leverage the approach
  - ⇝ Should the monads come with their specification monad à la Maillard?
- ➡ Scaling up
  - ⇝ Can it deliver and help in Martin’s R project?

Perspectives

- ➡ Better robustness
  - ⇝ Tactics to handle indices in types
- ➡ Clarifying how to leverage the approach
  - ⇝ Should the monads come with their specification monad à la Maillard?
- ➡ Scaling up
  - ⇝ Can it deliver and help in Martin’s R project?
- ➡ Beyond subsets of operations
  - ⇝ Increment operation interpreted into monotone functions of the cell

Perspectives

- ➡ Better robustness
  - ⇝ Tactics to handle indices in types
- ➡ Clarifying how to leverage the approach
  - ⇝ Should the monads come with their specification monad à la Maillard?
- ➡ Scaling up
  - ⇝ Can it deliver and help in Martin's R project?
- ➡ Beyond subsets of operations
  - ⇝ Increment operation interpreted into monotone functions of the cell
- ➡ Related Work
  - ⇝ Isn't it just Katsumata's graded monads?
Didn't Swamy's lightweight monadic programming already address this?

Effectful Programming across Heterogeneous Computations

Thanks!

https://gitlab.inria.fr/yzakowsk/ordered-signatures/-/tree/jfla23/