Structured Handling of Scoped Effects

Zhixuan Yang 🗣

Marco Paviotti

Nicolas Wu

@ESOP 2022

Birthe van den Berg

Tom Schrijvers

Imperial College
London

KU LEUVEN

Structured Handling of Scoped Effects

Zhixuan Yang 🗣
Marco Paviotti
Nicolas Wu

@ESOP 2022

Birthe van den Berg
Tom Schrijvers

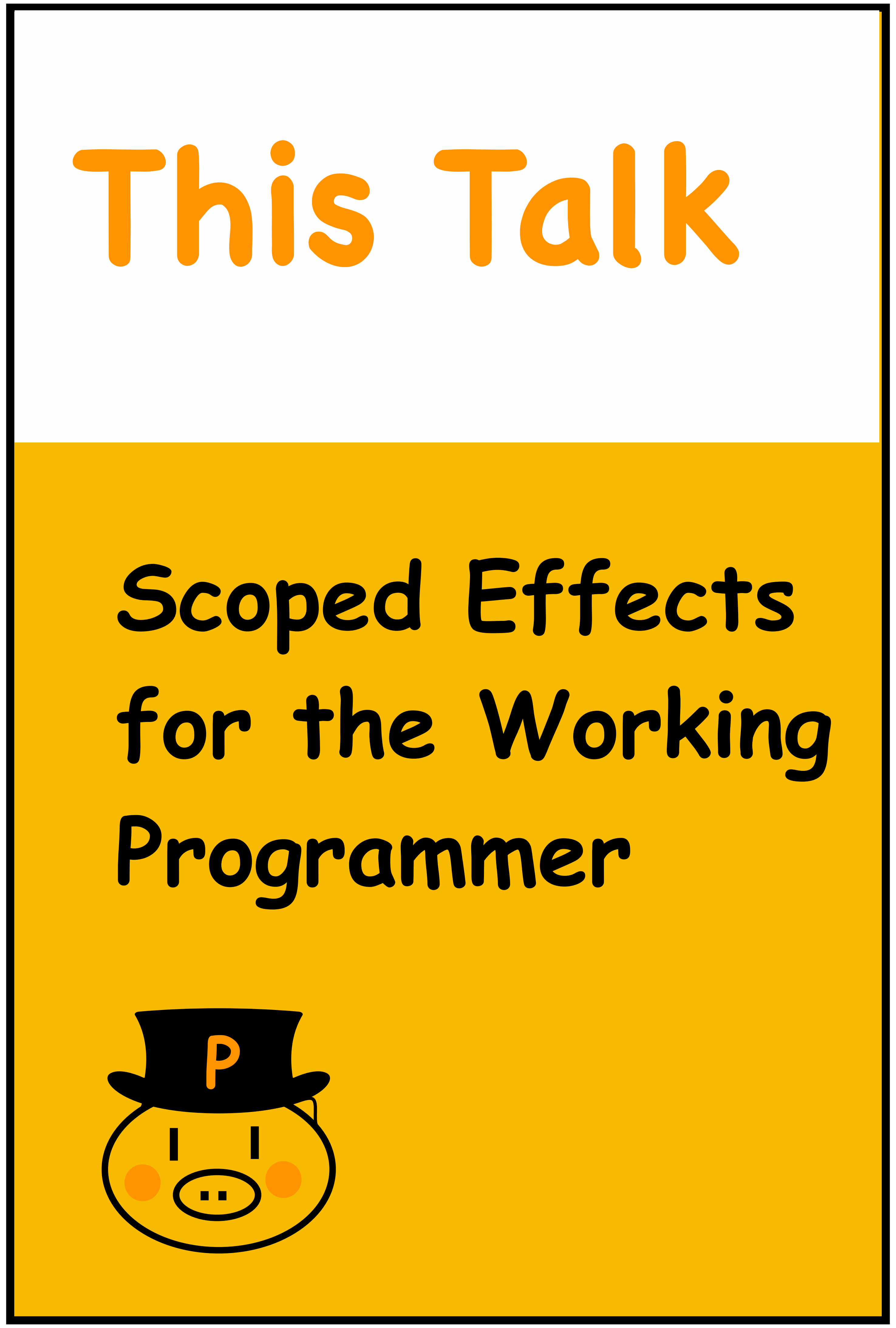
Imperial College London

KU LEUVEN

This Talk

Scoped Effects for the Working Programmer

This Talk



Scoped Effects
for the Working
Programmer

This Talk

Scoped Effects for the Working Programmer

More in the Paper

A Categorical Analysis of Our Approach

Algebraic Effects

A *computational effect* is modelled as an *algebraic theory.*

Example The effect of *mutable* **s**–*state* is modelled by

- two ***operations*** $\{\ \texttt{put : s} \rightsquigarrow \texttt{()},$
  $\texttt{get : ()} \rightsquigarrow \texttt{s}\ \}$

Algebraic Effects

A *computational effect* is modelled as an *algebraic theory.*

Example The effect of *mutable* **`s`**–*state* is modelled by

- two ***operations***
  ```
  { put : s  ⇝ (),
    get : () ⇝ s }
  ```
- several ***equations*** (pairs of *terms*) characterising **`put`** and **`get`**, such as
  ```
  do {put s; x ← get; k x} = do {put s; k s}
  ```
  …

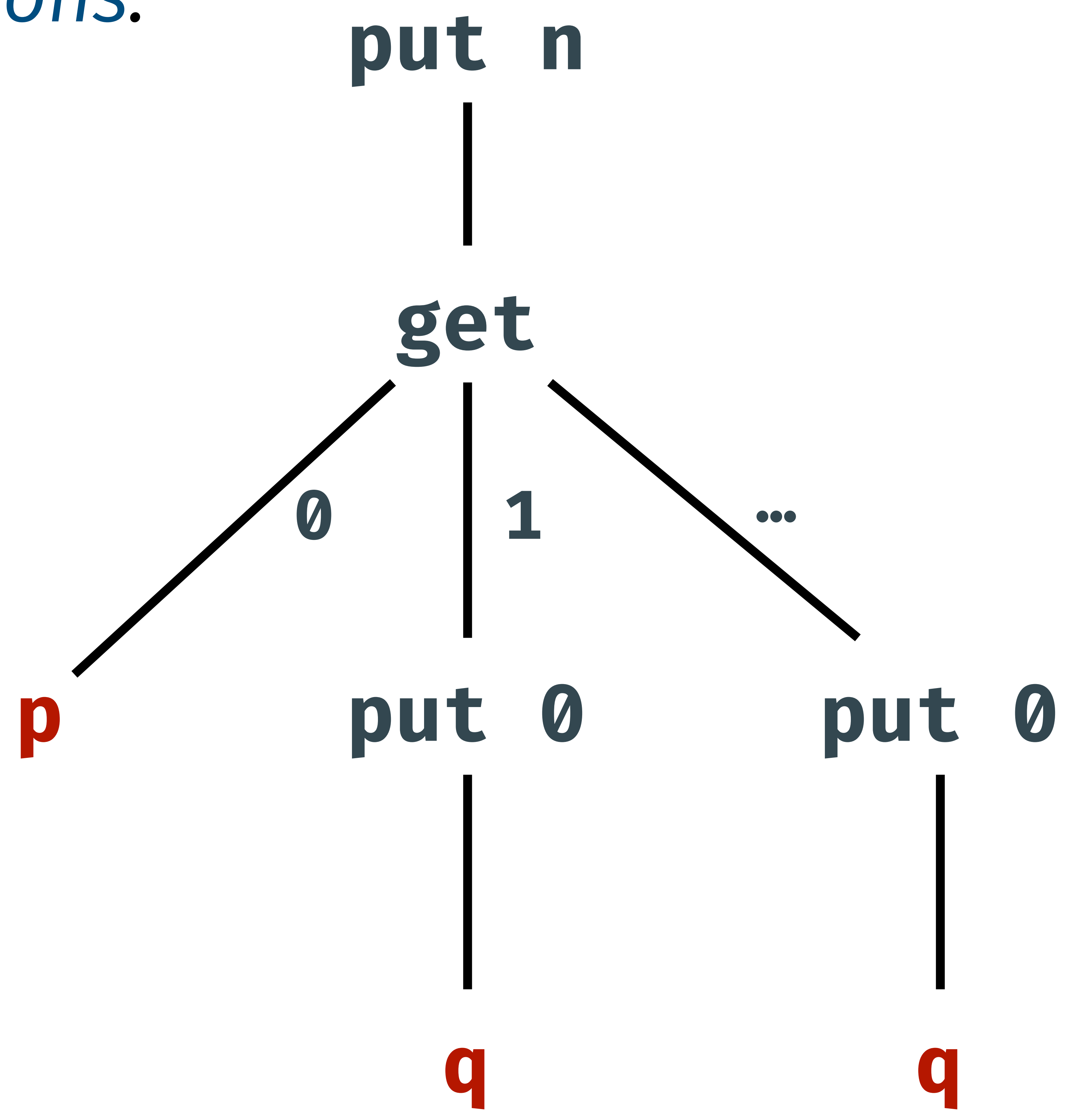
Terms of Operations

Terms of a theory are conceptually *trees of operations*.

Example *A term for a mutable* **`Int`***-state:*

```
do put n
   x ← get
   if x == 0
     then p
     else do put 0; q
```

$\simeq$



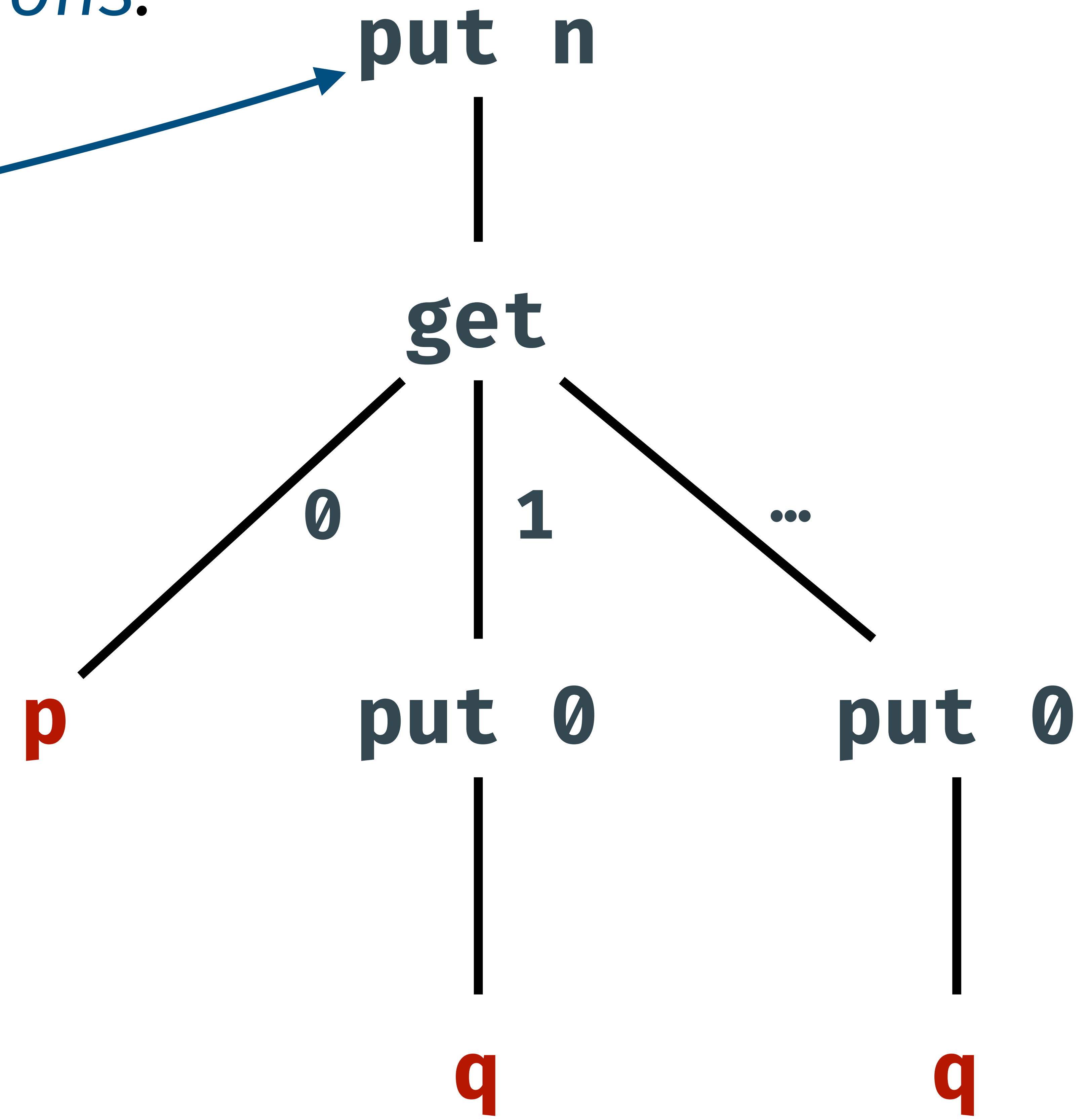
Terms of Operations

Terms of a theory are conceptually *trees of operations*.

Example *A term for a mutable* **`Int`***-state:*

```
do put n
   x ← get
   if x == 0
     then p
     else do put 0; q
```

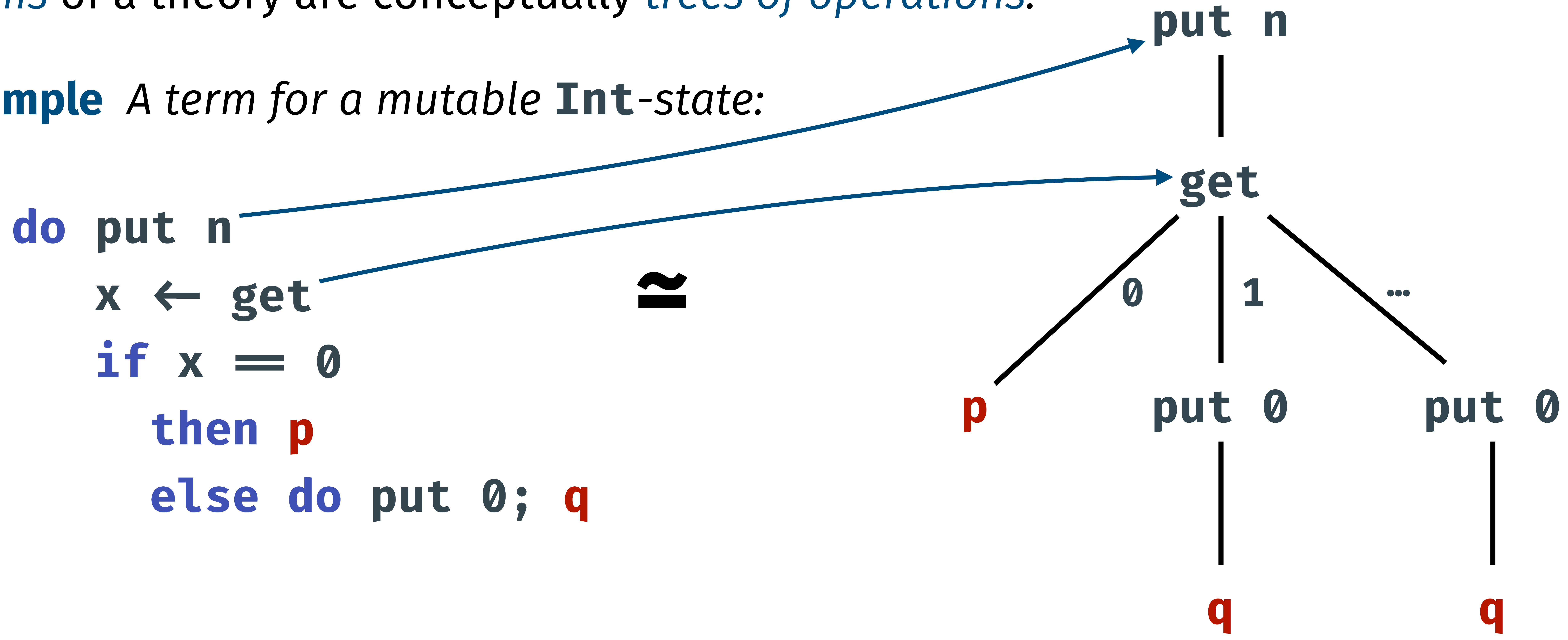
$\simeq$



Terms of Operations

Terms of a theory are conceptually *trees of operations*.

Example *A term for a mutable* **`Int`***-state:*



```
do put n
   x ← get
   if x == 0
     then p
     else do put 0; q
```

≃

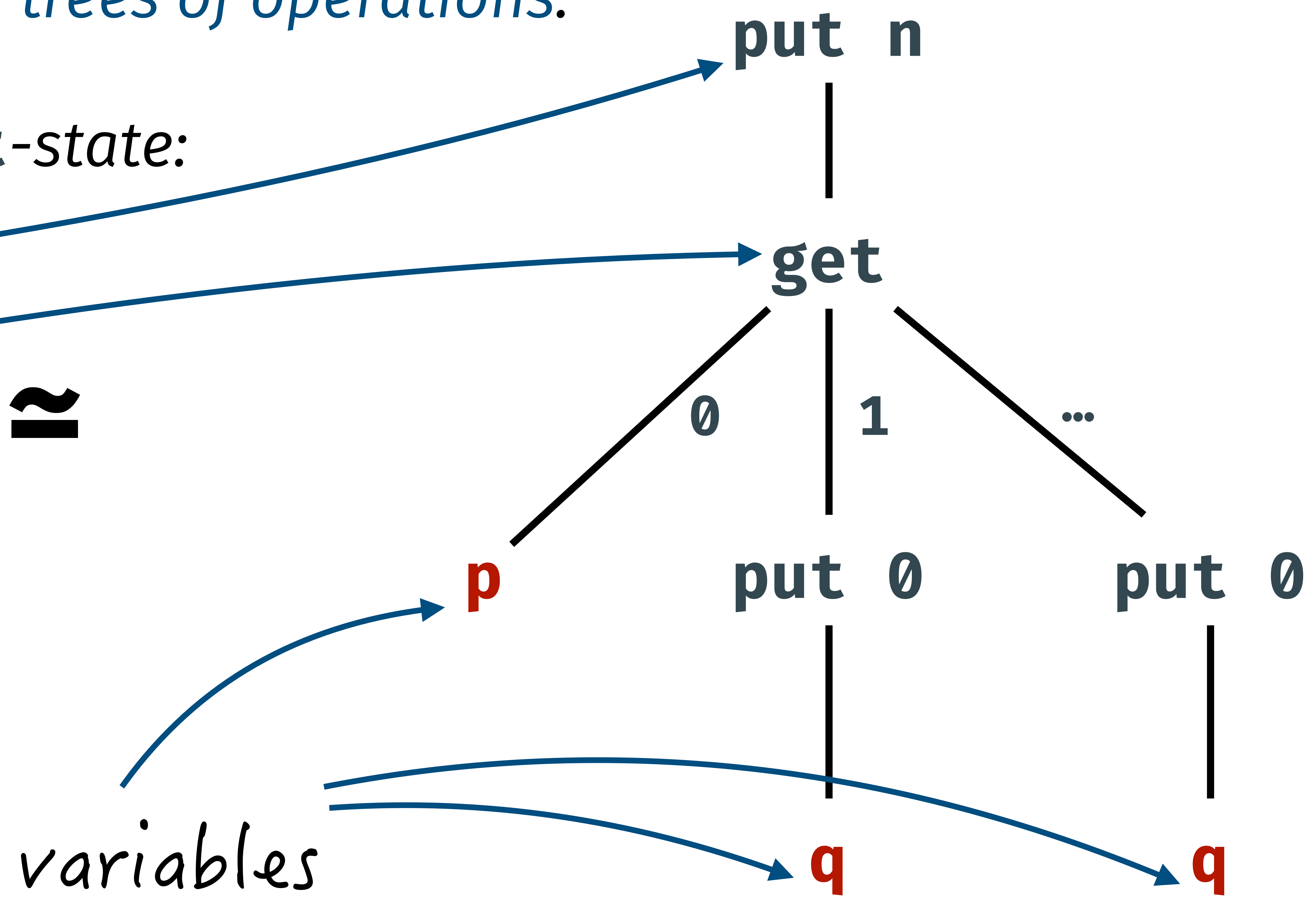
Terms of Operations

Terms of a theory are conceptually *trees of operations*.

Example *A term for a mutable* `Int`*-state:*

```
do put n
   x ← get
   if x == 0
     then p
     else do put 0; q
```

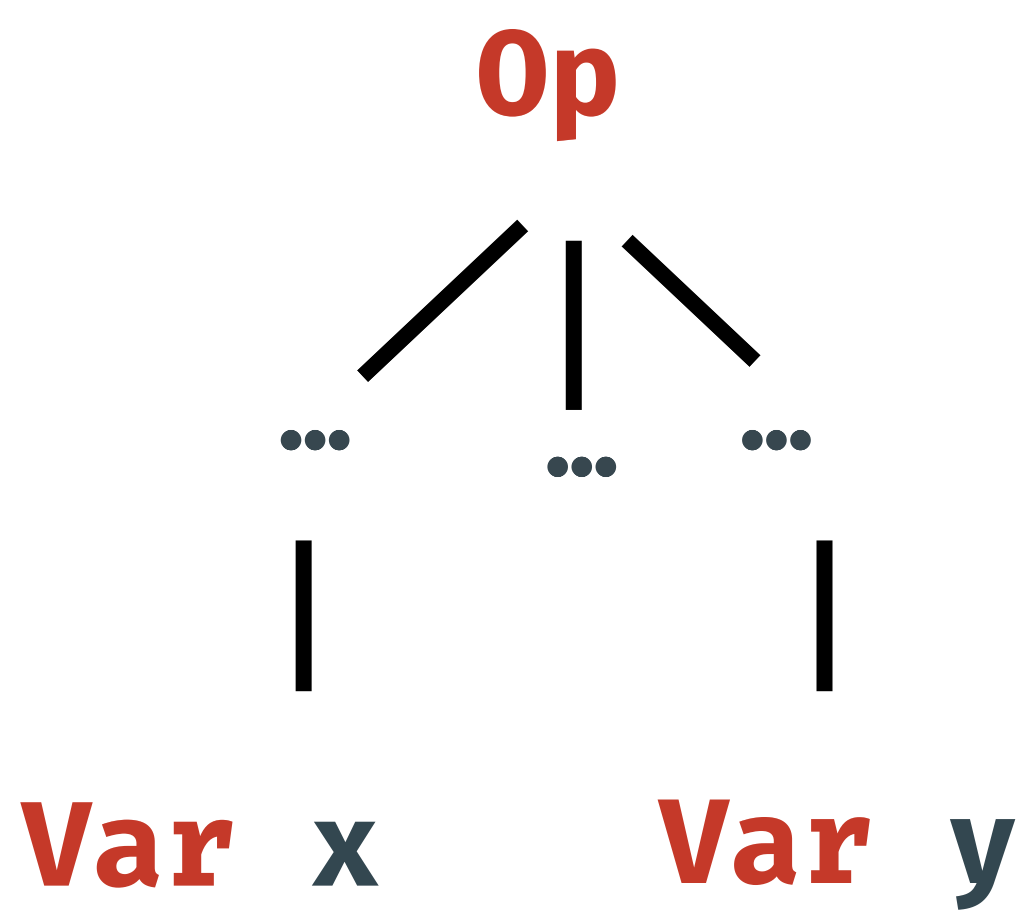
$\simeq$



Terms of Operations

Generally, *terms* of an operation signature **sig** :: * → * and variables of type **a** are

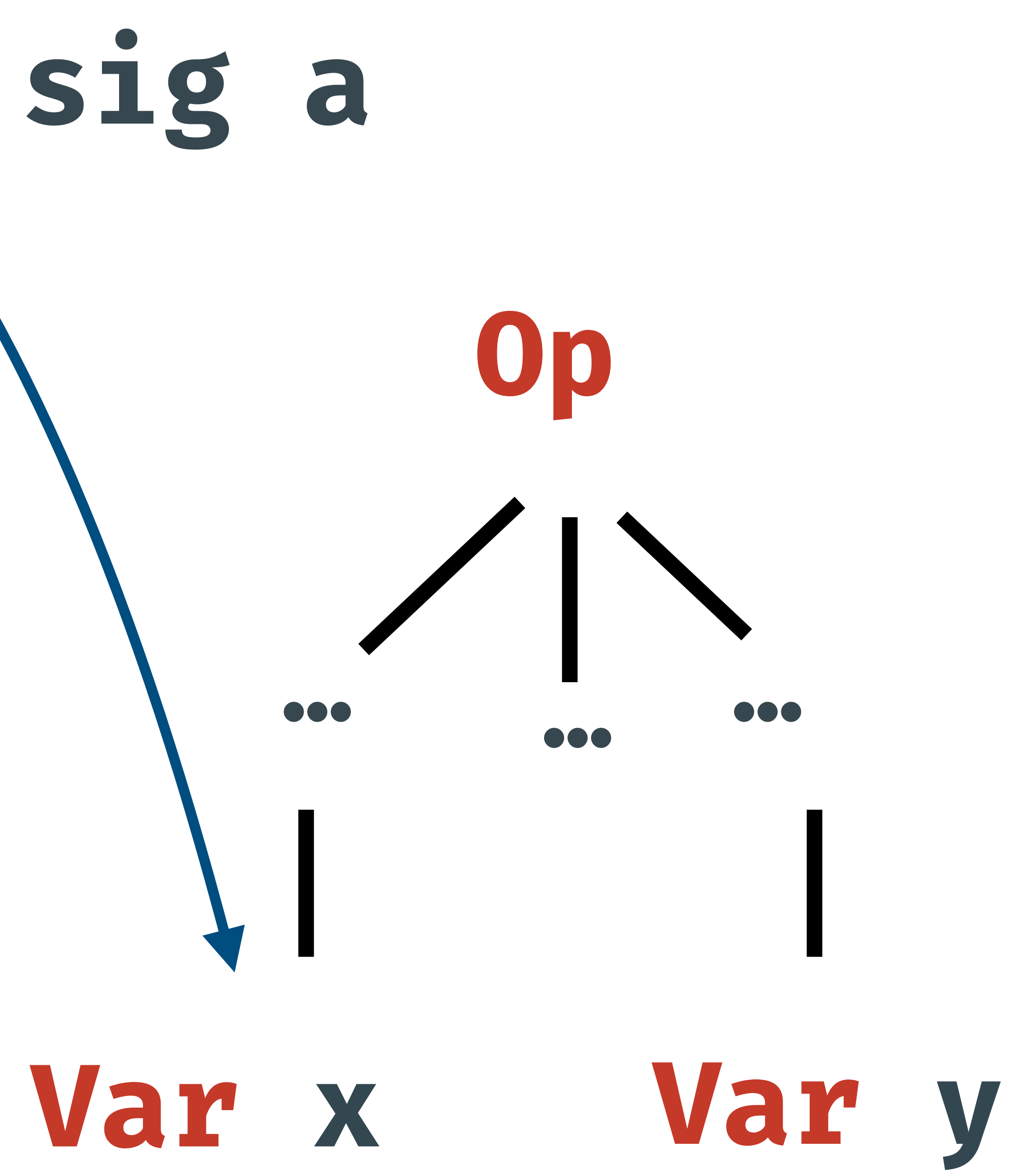
```
data Free sig a :: * where
  Var :: a → Free sig a
  Op  :: sig (Free sig a) → Free sig a
```



Terms of Operations

Generally, *terms* of an operation signature **sig** :: * → * and variables of type **a** are

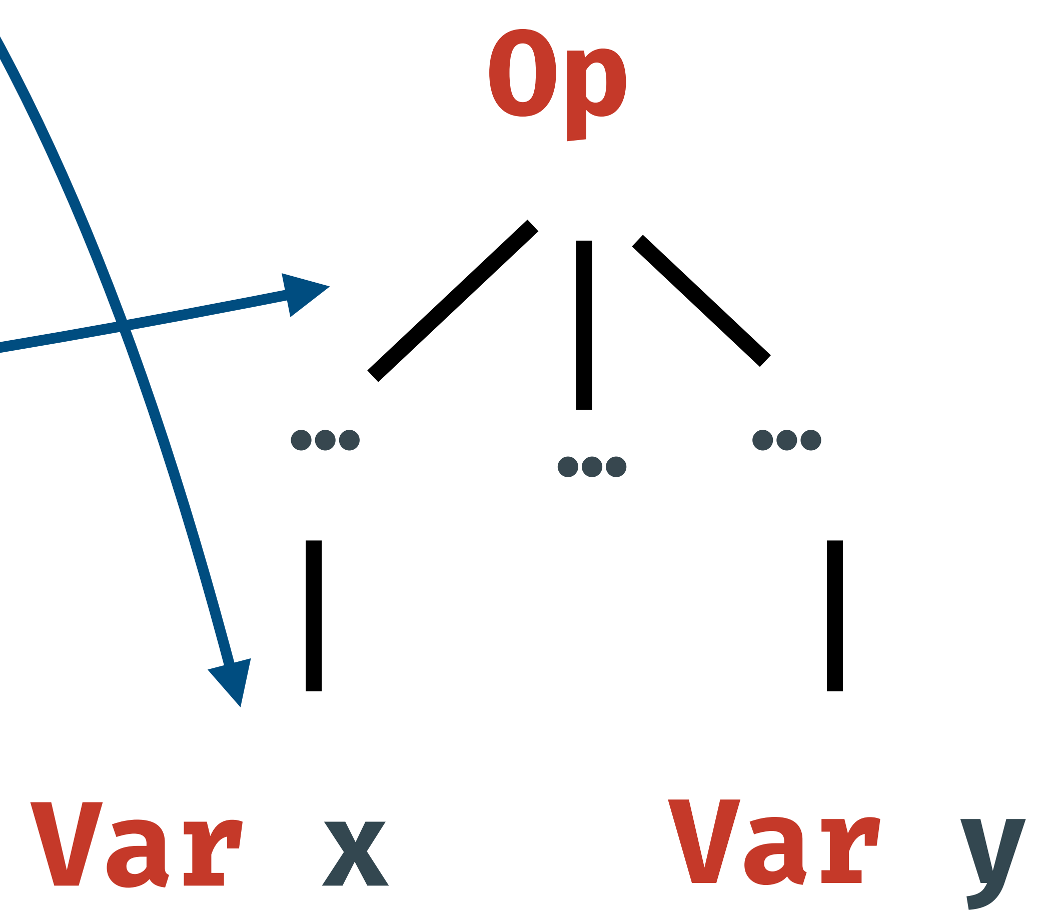
```
data Free sig a :: * where
  Var :: a → Free sig a
  Op  :: sig (Free sig a) → Free sig a
```



Terms of Operations

Generally, *terms* of an operation signature `sig :: * → *` and variables of type `a` are

```
data Free sig a :: * where
  Var :: a → Free sig a
  Op  :: sig (Free sig a) → Free sig a
```



Operations are `sig`-branching internal nodes

Signature Functors

Signature of operations can be packaged into a datatype.

Example The signature for the effect of **`Int`**-*state* and *exception throw* is

```
data ES :: * → * where
  Put   :: Int → (()   → x) → ES x
  Get   :: ()  → (Int  → x) → ES x
  Throw :: ()  → (Void → x) → ES x
```

parameter type

result type

`Void` is the type with no constructors

Signature Functors

Signature of operations can be packaged into a datatype.

Example The signature for the effect of **`Int`**-*state* and *exception throw* is

```
data ES :: * → * where
  Put   :: Int → x → ES x
  Get   :: (Int → x) → ES x
  Throw :: ES x
```

Term Model of Effectful Programs

Terms are a *syntactic model* of effectful computations.

Example A program involving **`Int`**-*state* and *exception* throwing:

```
safeDiv :: Int → Free ES Int
safeDiv n = Op (Get (λ s →
  if s ≡ 0
    then Op Throw
    else Op (Put (n / s)  (Var (n / s)))))
```

Variables are understood as return values

The Monad of Terms

We'd like to have *sequential composition* of (the term model of) computations, so we equip `Free sig` with a monad structure:

```
return :: a → Free sig a

(>>=) :: Free sig a → (a → Free sig b) → Free sig b
```

The Monad of Terms

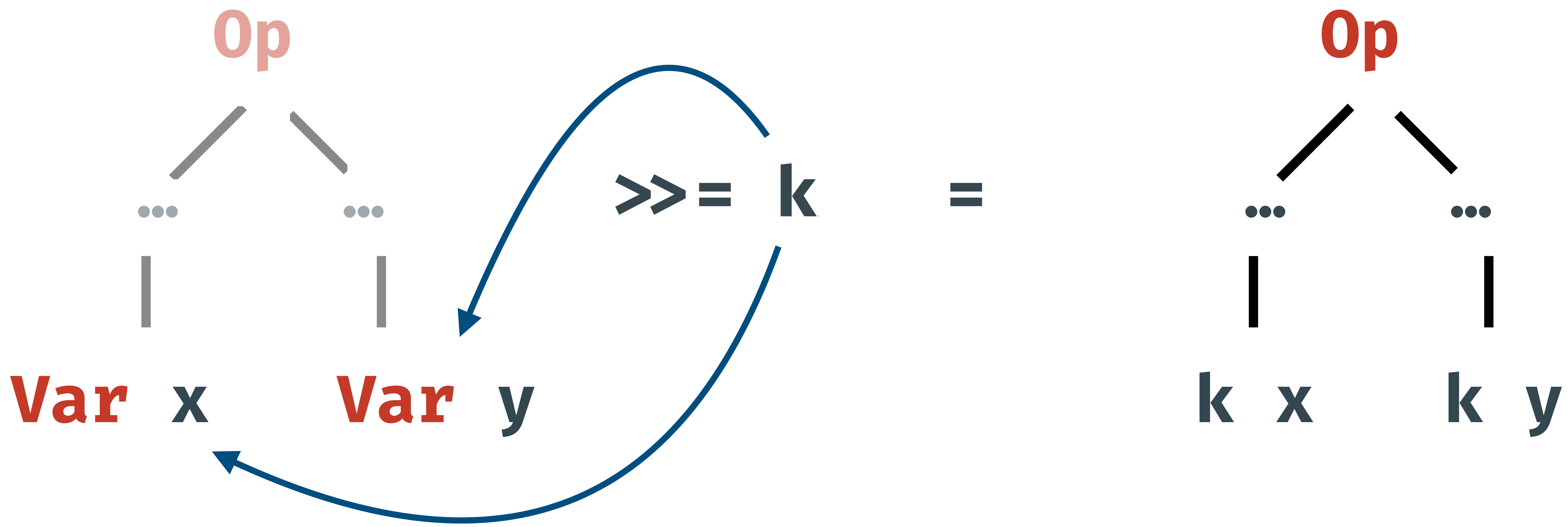
We'd like to have *sequential composition* of (the term model of) computations, so we equip `Free sig` with a monad structure:

```
return :: a → Free sig a
return = Var

(>>=) :: Free sig a → (a → Free sig b) → Free sig b
```

The Monad of Terms

We'd like to have *sequential composition* of (the term model of) computations, so we equip **Free sig** with a monad structure:

```
return :: a → Free sig a
return = Var

(>>=) :: Free sig a → (a → Free sig b) → Free sig b
```



Effectful Programs with Free Monads

Example `safeDiv` is also sequential composition of smaller programs:

```
safeDiv :: Int → Free ES Int
safeDiv n = Op (Get (s →
  if s ≡ 0
    then Op Throw
    else Op (Put (n / s)  (Var (n / s)))))
```

Effectful Programs with Free Monads

Example `safeDiv` is also sequential composition of smaller programs:

```
safeDiv :: Int → Free ES Int
safeDiv n = get >>= λ s →
              if s ≡ 0
                then throw
                else put (n / s) >>= λ _ → return (n / s)

  where get     = Op (Get Var)
        put s   = Op (Put s (Var ()))
        throw   = Op Throw
```

Effectful Programs with Free Monads

Example `safeDiv` is also sequential composition of smaller programs:

```
safeDiv :: Int → Free ES Int
safeDiv n = do s ← get
               if s ≡ 0
                 then throw
                 else do put (n / s); return (n / s)

  where get     = Op (Get Var)
        put s = Op (Put s (Var ()))
        throw = Op Throw
```

Effectful Programs with Free Monads

Example `safeDiv` is also sequential composition of smaller programs:

```
safeDiv :: Int → Free ES Int
safeDiv n = do s ← get
               if s ≡ 0
                 then throw
                 else do put (n / s); return (n / s)
```

`Free sig a` is just a *syntactic model* of effectful programs!

Handlers of Effects

Semantic models ("handlers") `<b :: *, f :: sig b→b>` *interpret ("handle")* programs with **`sig`**-operations:

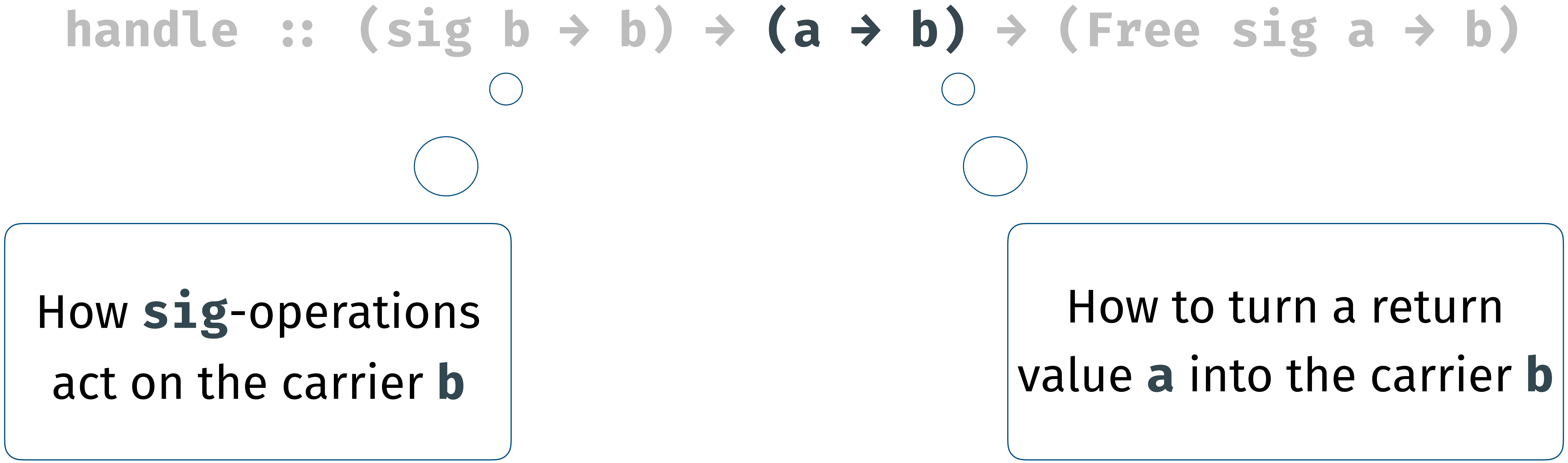
```
handle :: (sig b → b) → (a → b) → (Free sig a → b)
```

How **`sig`**-operations act on the carrier **`b`**

How to turn a return value **`a`** into the carrier **`b`**

Handlers of Effects

Semantic models (“handlers”) `<b :: *, f :: sig b→b>` *interpret (“handle”)* programs with **`sig`**-operations:



```
handle :: (sig b → b) → (a → b) → (Free sig a → b)
```

How **`sig`**-operations act on the carrier **`b`**

How to turn a return value **`a`** into the carrier **`b`**

Handlers of Effects

Semantic models (“handlers”) `<b :: *, f :: sig b→b>` *interpret (“handle”)* programs with **`sig`**-operations:

```
handle :: (sig b → b) → (a → b) → (Free sig a → b)
```

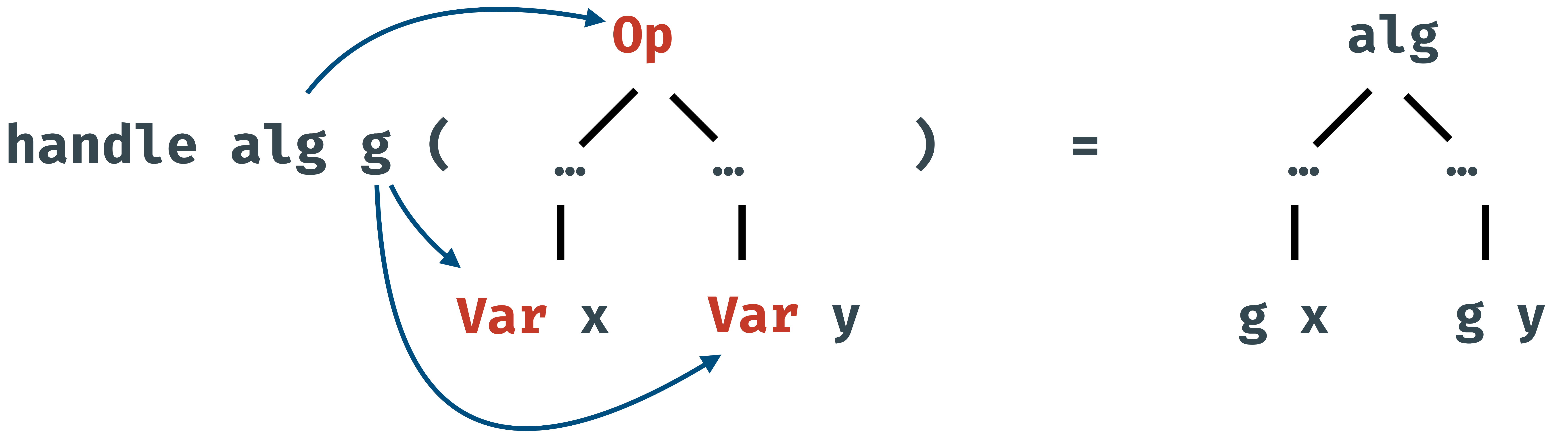
How **`sig`**-operations act on the carrier **`b`**

How to turn a return value **`a`** into the carrier **`b`**

Handlers of Effects

Semantic models ("handlers") `<b :: *, f :: sig b→b>` *interpret ("handle")* programs with **`sig`**-operations:

```
handle :: (sig b → b) → (a → b) → (Free sig a → b)
```



```
                         Op                                  alg
                        /  \                                /  \
handle alg g (        …      …        )    =              …      …
                      |      |                            |      |
                  Var x     Var y                        g x    g y
```

Handlers of Effects

Example Given a program `r :: Free ES a`, a handler `catchHdl r` that

- gives the ‘standard’ semantics to `Throw`, and
- leaves other operations unchanged:

```
catchHdl :: Free ES a
         → ES (Free ES a) → Free ES a
catchHdl r Throw = r
catchHdl r op    = Op op
```

Modularity of Handlers

Separating syntax from semantics allows different handlers of the same effect:

Example A non-standard handler of exception that *ignores* the recovery code `r`

```
catchHdl' :: Free ES a
          → ES (Free ES (Maybe a)) → Free ES (Maybe a)
catchHdl' r Throw = return Nothing
catchHdl' r op    = Call op
```

Non-Algebraic Operations

Why is exception throwing an operation but catching a handler?

Non-Algebraic Operations

Why is exception throwing an operation but catching a handler?

If we model **`catch`** as an operation with **`Free`**, then

```
(catch p r) >>= k     ═     catch (p >>= k) (r >>= k)
```

by the definition of **`>>=`** for **`Free`**, but this equality is ***undesirable:***

Non-Algebraic Operations

Why is exception throwing an operation but catching a handler?

If we model `catch` as an operation with `Free`, then

```
(catch p r) >>= k      =      catch (p >>= k) (r >>= k)
```

by the definition of `>>=` for `Free`, but this equality is ***undesirable:***

The *scopes* for catching exceptions are different!

Non-Algebraic Operations

Although `catch` can be modelled as handlers, we ***lose*** the separation of syntax and semantics for `catch`:

Suppose we want a program that *morally* means

“
```
do x ← catch (safeDiv 5) (return 42)
   put (x + 1)
```
”

Non-Algebraic Operations

With different handlers, we write for `catchHdl`

```
do x ← handle (catchHdl (return 42)) return
                (safeDiv 5)
   put (x + 1)
```

Non-Algebraic Operations

With different handlers, we write for **`catchHdl`**

```
do x ← handle (catchHdl (return 42)) return
                (safeDiv 5)
   put (x + 1)
```

but for **`catchHdl’`** we write

```
do xMb ← handle (catchHdl' (return 42)) (return · Just)
                (safeDiv 5)
   case xMb of
     Nothing   → return Nothing
     (Just x ) → do r ← put (x + 1); return (Just r )
```

Non-Algebraic Operations

With different handlers, we write for **`catchHdl`**

```
do x ← handle (catchHdl (return 42)) return     :: Free ES a
              (safeDiv 5)
   put (x + 1)
```

but for **`catchHdl'`** we write

```
do xMb ← handle (catchHdl' (return 42)) (return · Just) :: Free ES (Maybe a)
                (safeDiv 5)
   case xMb of
     Nothing   → return Nothing
     (Just x ) → do r ← put (x + 1); return (Just r))
```

Scoped Effects

We want to write *syntactic non-algebraic operations* and interpret them differently.

“
```
do x ← catch (safeDiv 5) (return 42)
   put (x + 1)
```
”

Scoped Effects

We want to write *syntactic non-algebraic operations* and interpret them differently.

“
```
do x ← catch (safeDiv 5) (return 42)
   put (x + 1)
```
”

Cause Handlers model the syntax and semantics of `catch` at the same time!

Scoped Effects

We want to write *syntactic non-algebraic operations* and interpret them differently.

“
```
do x ← catch (safeDiv 5) (return 42)
   put (x + 1)
```
”

Cause Handlers model the syntax and semantics of `catch` at the same time!

Solution Separate syntax and semantics.

Scoped Effects

We want to write *syntactic non-algebraic operations* and interpret them differently.

“
```
do x ← catch (safeDiv 5) (return 42)
   put (x + 1)
```
”

Cause Handlers model the syntax and semantics of `catch` at the same time!

Solution

- Generalising `Free` to non-algebraic (“*scoped*”) operations [Wu et al. 2014];
- Finding nice ways to handle them (*contribution of this paper*).

Syntax of Scoped Effects

Extending **`Free`** to accommodate scoped operations:

```
data Free f a :: * where
  Var :: a → Free f a
  Op  :: f (Free f a) → Free f a
```

Syntax of Scoped Effects

Extending `Free` to accommodate scoped operations:

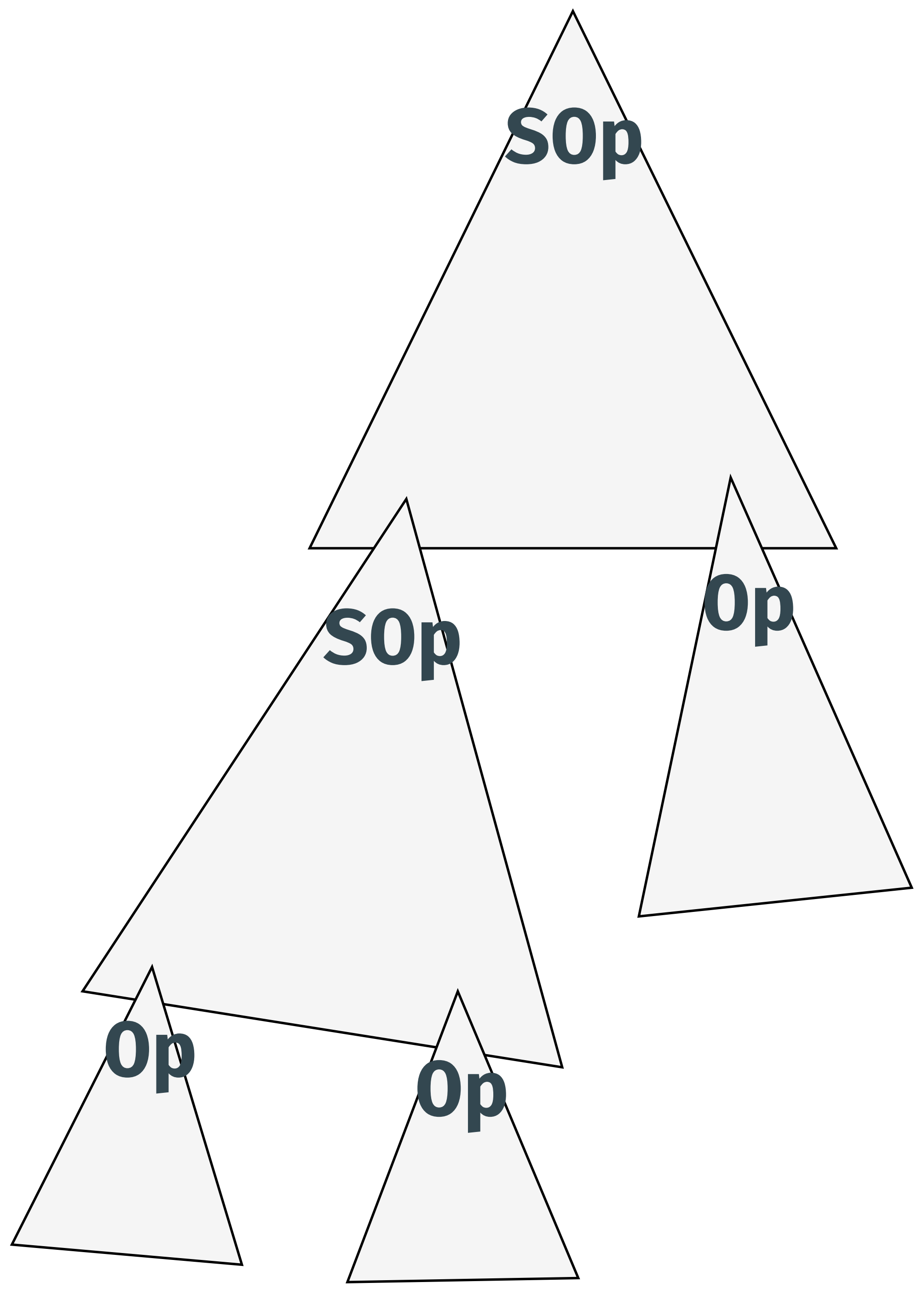
```
data FreeS f g a :: * where
  Var :: a → FreeS f g a
  Op  :: f (FreeS f g a) → FreeS f g a
  SOp :: g (FreeS f g (FreeS f g a)) → FreeS f g a
```

`f`: *signature of algebraic operations*
`g`: *signature of scoped operations*

Syntax of Scoped Effects

Intuition `Free f` are trees, while `FreeS f g` are *nested trees*:

- Boundary of a tree is the scope of an scoped operation



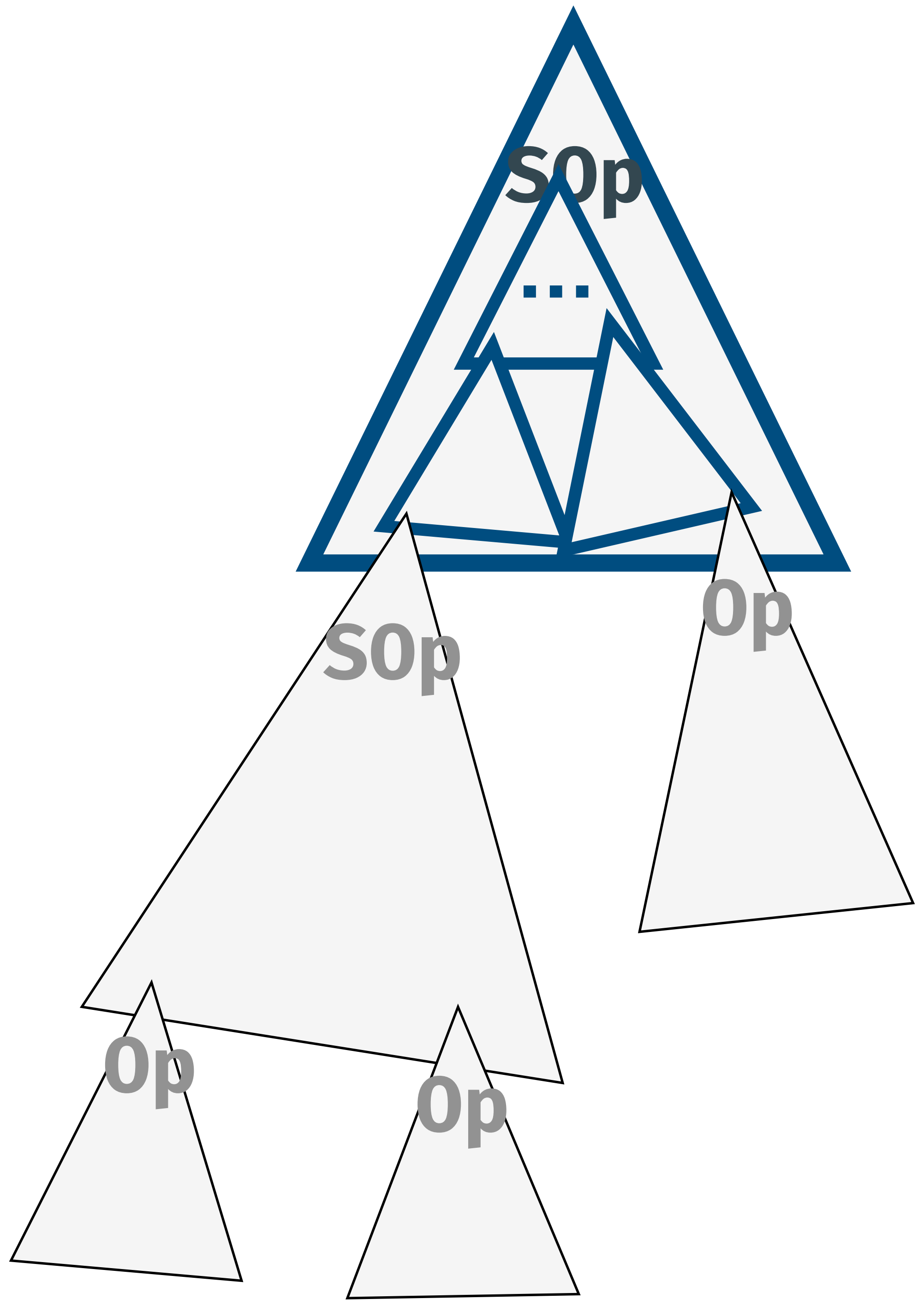
Syntax of Scoped Effects

Intuition **`Free f`** are trees, while **`FreeS f g`** are ***recursively*** *nested trees:*

- Boundary of a tree is the scope of an scoped operation

- Trees themselves can be nested trees, i.e. scoped operations can be nested.

```
catch (catch p h >>= k) h'
  >>= k'
```



Handlers of Scoped Effects

What are the handlers of scoped operations?

Proposal 1 Treating them as algebraic effects with recursion

```
data FreeS f g a :: * where
  Var :: a → FreeS f g a
  Op  :: f (FreeS f g a) → FreeS f g a
  SOp :: g (FreeS f g (FreeS f g a)) → FreeS f g a
```

Handlers of Scoped Effects

What are the handlers of scoped operations?

Proposal 1 Treating them as algebraic effects with recursion

```
data FreeS f g a :: * where
  Var :: a → FreeS f g a
  Op  :: (f + g ∘ FreeS f g) (FreeS f g a) → FreeS f g a
```

Scoped operations are treated as algebraic operations whose signature is *recursively* defined

Handlers of Scoped Effects

What are the handlers of scoped operations?

Proposal 1 Treating them as algebraic effects with recursion, thus a handler for signatures `f` and `g` is a type `c` equipped with

```
opB   :: f c → c          sopB  :: g (FreeS f g c) → c
```

Handlers of Scoped Effects

What are the handlers of scoped operations?

Proposal 1 Treating them as algebraic effects with recursion, thus a handler for signatures `f` and `g` is a type `c` equipped with

```
opB   ::  f  c → c          sopB  ::  g  (FreeS  f  g  c) → c
```

Problem `sopB` has *too much* freedom on how to use `FreeS f g`

Proposal of This Paper

A *functorial algebra* for algebraic signature **f** and scoped signature **g** has

- and a type **c :: *** equipped with

```
opB  ::  f c → c
sopB ::  g (h c) → c
```

- A functor **h :: *→*** equipped with

```
varE :: ∀x. x → h x
opE  :: ∀x. f (h x) → h x
sopE :: ∀x. g (h (h x)) → h x
```

Proposal of This Paper

A *functorial algebra* for algebraic signature **f** and scoped signature **g** has

- and a type **c** :: * equipped with

```
opB   ::  f c → c
sopB  ::  g (h c) → c
```

- A functor **h** :: *→* equipped with

```
varE  ::  ∀x. x → h x
opE   ::  ∀x. f (h x) → h x
sopE  ::  ∀x. g (h (h x)) → h x
```

which gives rise to a handling function:

```
handle  ::  FunctorialAlg h c
        →  (a → c) → FreeS f g a → c
```

Some Examples

- *Exception throwing and catching* handled by `<Maybe, Maybe a, …>`
- *Explicit nondeterminism with scoped search strategies* like

```
    bfs (or (dfs (or  ... ))
             (or x y))
```

  handled by `<x ⟼ ([x], [[x]]),  [a], …>`

- *Parallel composition handled by a resumption monad.*

What Else in the Paper

THM There is an adjunction between functorial algebras and the category $\mathbb{C}$ (for pure values)

$$\begin{tikzcd}
\mathit{Fn}\text{-}\mathit{Alg} \arrow[r, shift right, "U_{\mathit{Fn}}"'] \arrow[r, phantom, "\perp"] & \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, shift right, "\mathit{Free}_{\mathit{Fn}}"'] \arrow[r, shift right, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right, "\uparrow"'] \arrow[loop right, "T"]
\end{tikzcd}$$

whose induced monad T is isomorphic to **`FreeS f g`**.

What Else in the Paper

Functorial algebras are compared with two other adjunctions for handling scoped effects: *indexed algebras* and *Eilenberg-Moore algebras*.

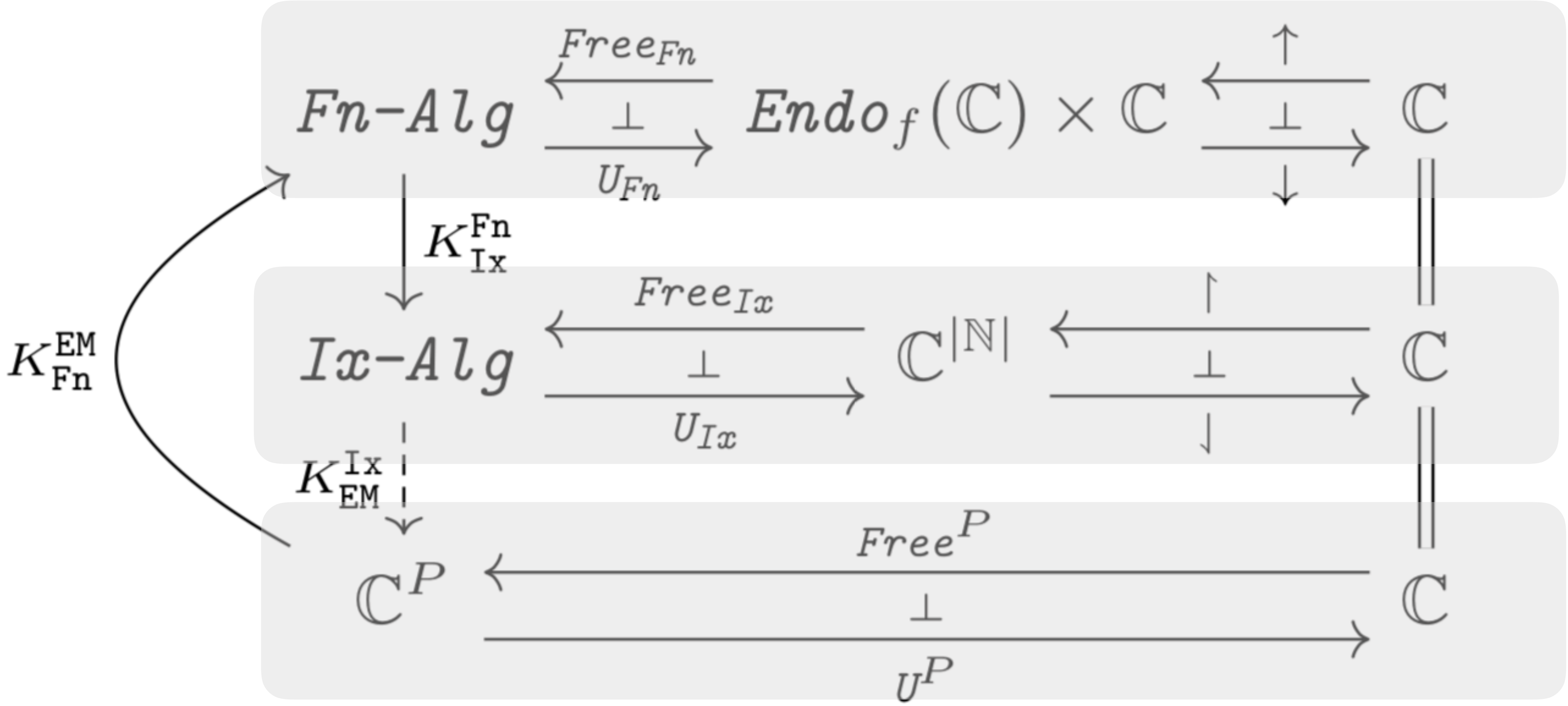
THM There are comparison functors between these adjunctions. Thus all these models have equal expressivity.

$$\begin{tikzcd}
\mathit{Fn}\text{-}\mathit{Alg} \arrow[r, shift right=1.5ex, "U_{\mathit{Fn}}"'] \arrow[r, phantom, "\perp"] \arrow[d, "K^{\mathtt{Fn}}_{\mathtt{Ix}}"] & \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, shift right=1.5ex, "\mathit{Free}_{\mathit{Fn}}"'] \arrow[r, shift right=1.5ex, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right=1.5ex, "\uparrow"'] \arrow[d, equal] \\
\mathit{Ix}\text{-}\mathit{Alg} \arrow[r, shift right=1.5ex, "U_{\mathit{Ix}}"'] \arrow[r, phantom, "\perp"] \arrow[d, dashed, "K^{\mathtt{Ix}}_{\mathtt{EM}}"'] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, shift right=1.5ex, "\mathit{Free}_{\mathit{Ix}}"'] \arrow[r, shift right=1.5ex, "\downharpoonleft"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right=1.5ex, "\upharpoonright"'] \arrow[d, equal] \\
\mathbb{C}^P \arrow[rr, shift right=1.5ex, "U^P"'] \arrow[rr, phantom, "\perp"] \arrow[uu, bend left=60, "K^{\mathtt{EM}}_{\mathtt{Fn}}"] & & \mathbb{C} \arrow[ll, shift right=1.5ex, "\mathit{Free}^P"']
\end{tikzcd}$$

What Else in the Paper

Functorial algebras are compared with two other adjunctions for handling scoped effects: *indexed algebras* and *Eilenberg-Moore algebras*.

THM There are comparison functors between these adjunctions. Thus all these models have equal expressivity.



$$\begin{tikzcd}
\mathit{Fn}\text{-}\mathit{Alg} \arrow[r, shift right=2, "U_{\mathit{Fn}}"'] \arrow[r, phantom, "\perp"] \arrow[d, "K^{\mathrm{Fn}}_{\mathrm{Ix}}"] & \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, shift right=2, "\mathit{Free}_{\mathit{Fn}}"'] \arrow[r, shift right=2, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right=2, "\uparrow"'] \arrow[d, equal] \\
\mathit{Ix}\text{-}\mathit{Alg} \arrow[r, shift right=2, "U_{\mathit{Ix}}"'] \arrow[r, phantom, "\perp"] \arrow[d, dashed, "K^{\mathrm{Ix}}_{\mathrm{EM}}"'] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, shift right=2, "\mathit{Free}_{\mathit{Ix}}"'] \arrow[r, shift right=2, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right=2, "\uparrow"'] \arrow[d, equal] \\
\mathbb{C}^P \arrow[rr, shift right=2, "U^P"'] \arrow[rr, phantom, "\perp"] \arrow[uu, bend left=60, "K^{\mathrm{EM}}_{\mathrm{Fn}}"] & & \mathbb{C} \arrow[ll, shift right=2, "\mathit{Free}^P"']
\end{tikzcd}$$

What Else in the Paper

Functorial algebras are compared with two other adjunctions for handling scoped effects: *indexed algebras* and *Eilenberg-Moore algebras*.

THM There are comparison functors between these adjunctions. Thus all these models have equal expressivity.

$$\begin{tikzcd}
\mathit{Fn\text{-}Alg} \arrow[r, shift right, "U_{Fn}"'] \arrow[r, phantom, "\perp"] \arrow[d, "K^{\mathrm{Fn}}_{\mathrm{Ix}}"] & \mathit{Endo}_f(\mathbb{C}) \times \mathbb{C} \arrow[l, shift right, "Free_{Fn}"'] \arrow[r, shift right, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right, "\uparrow"'] \arrow[d, equal] \\
\mathit{Ix\text{-}Alg} \arrow[r, shift right, "U_{Ix}"'] \arrow[r, phantom, "\perp"] \arrow[d, dashed, "K^{\mathrm{Ix}}_{\mathrm{EM}}"'] & \mathbb{C}^{|\mathbb{N}|} \arrow[l, shift right, "Free_{Ix}"'] \arrow[r, shift right, "\downarrow"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, shift right, "\uparrow"'] \arrow[d, equal] \\
\mathbb{C}^P \arrow[rr, shift right, "U^P"'] \arrow[rr, phantom, "\perp"] \arrow[uu, bend left=60, "K^{\mathrm{EM}}_{\mathrm{Fn}}"] & & \mathbb{C} \arrow[ll, shift right, "Free^P"']
\end{tikzcd}$$

Take-Home Messages

📬🏠

- Non-algebraic operations need not to be handlers.
- They can be operations and handled in a structural way.

Back up slides

Scoped Scoped Operations?

We indeed can make a further generalisation:

```
SSOp :: g (FreeS f g (FreeS f g (FreeS f g a)))
     → FreeS f g a
```

corresponding to operations that look like

```
op ({ P  }{x. Q },
    { P' }{y. Q’}, … )
```

Example: explicit substitution `subst(P)(x. Q)`, but not too many.

Connections to Delimited Control

Can we implement scoped operations with shift/reset?

- Sounds plausible.

Are shift/reset scoped operations?

- Interesting direction. We need to develop scoped operations on *parameterised monads*, since `shift` and `reset` are not operations on ordinary monad `Cont r` but on parametric monad `Cont`.

```
shift :: ((a → r) → Cont r r) → Cont r a
reset :: Cont r r → Cont w r
```