Monad Transformers



Modular Algebraic Effects

What binds them together

Tom Schrijvers — KU Leuven
with Maciej Pirog, Nicolas Wu and Mauro Jaskelioff

Modular Effect Models for Haskell

Modular
Purely Functional Effects

```
dec = do s ← get
         if s > 0
         then put (s - 1)
         else fail
```

State + Failure

Monads

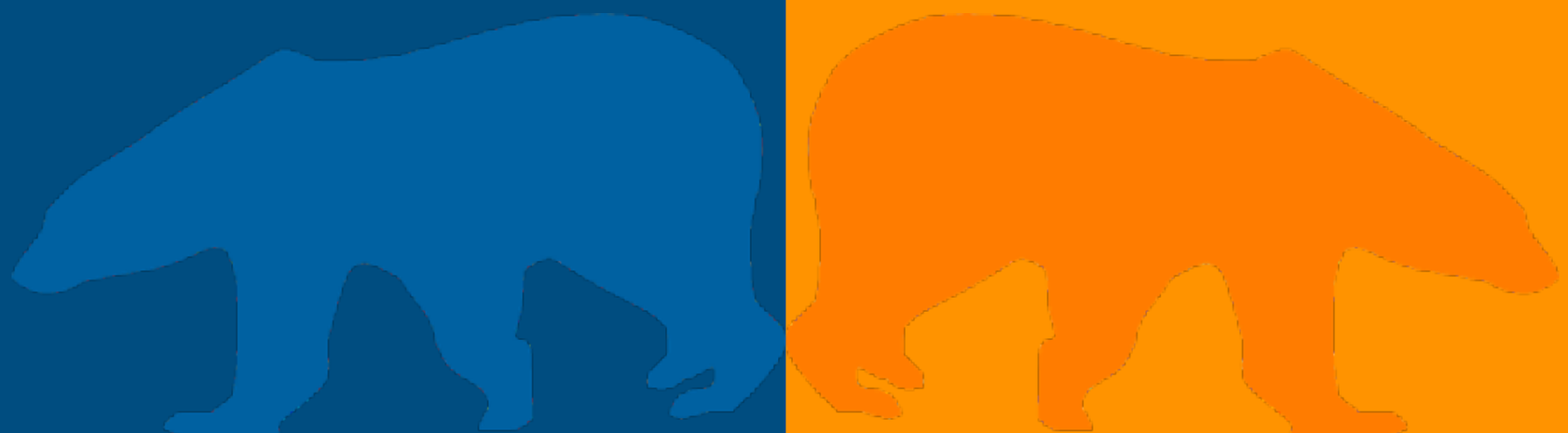
Moggi [1989]

Wadler [1990]

Algebraic Effects

Plotkin & Power [2002]

Plotkin & Pretnar [2013]



Monad Transformers

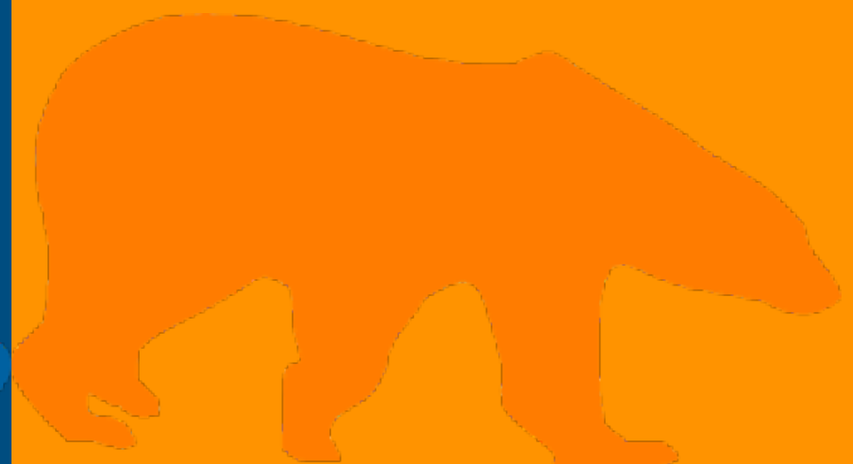
Liang et al. [1995]

mtl · transformers · Monatron · monads-tf · …

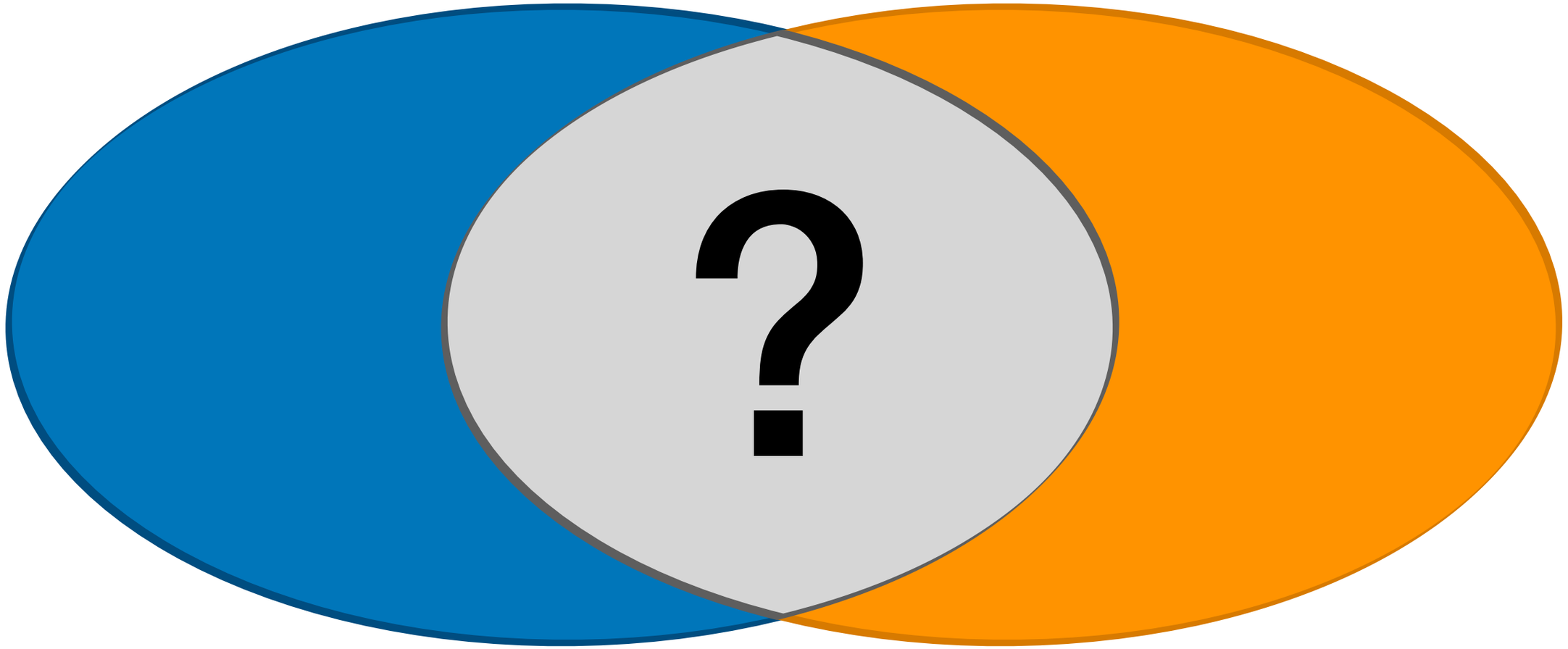


Modular Algebraic Effects

effect-handlers · freer · fused-effects · polysemy · …



Expressivity Comparison



monad transformers

algebraic effects

The Plan

The Plan



The Plan

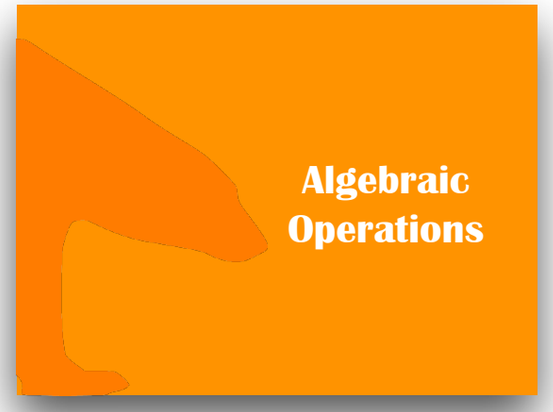




The Plan







The Plan

Monad Transformers



Algebraic Operations

Algebraic Effects & Handlers

The Plan

Monad Transformers



Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers

The Plan

Monad Transformers



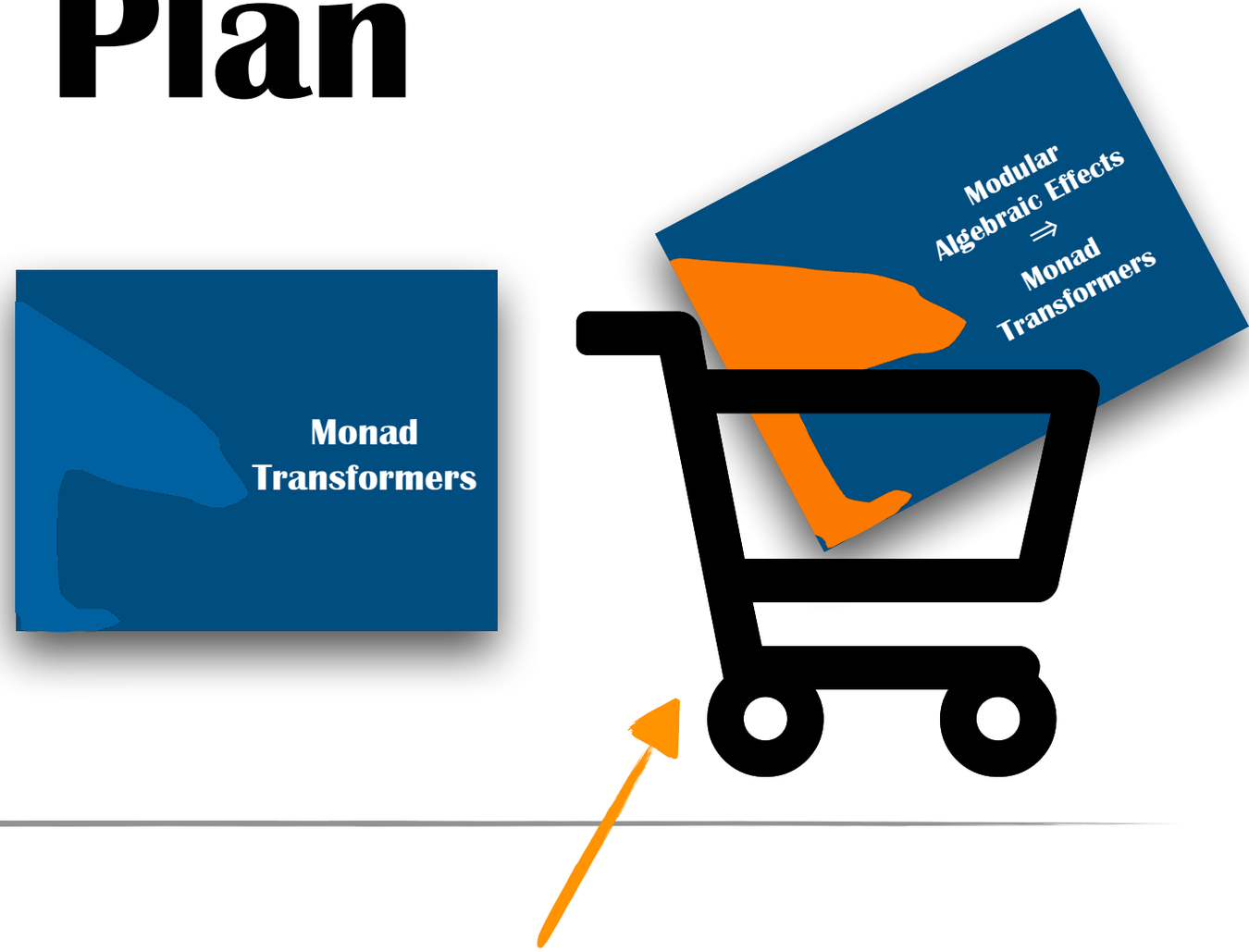
Algebraic Operations

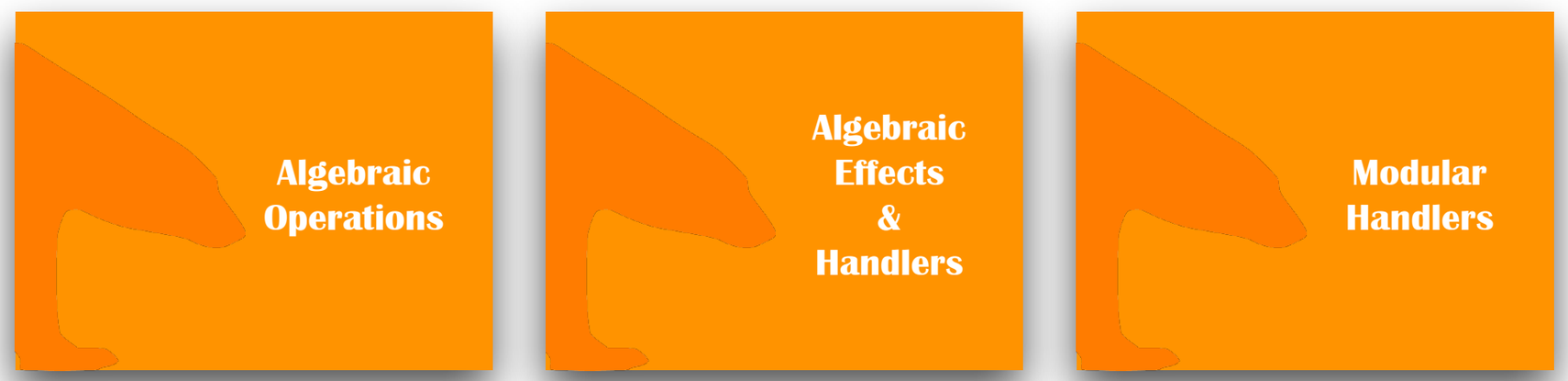
Algebraic Effects & Handlers

Modular Handlers



The Plan


Monad
Transformers
Modular
Algebraic Effects
⇒
Monad
Transformers


Algebraic
Operations
Algebraic
Effects
&
Handlers
Modular
Handlers



The Plan

Monad Transformers

Modular Algebraic Effects ⇒ Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers

Monad Transformers ⇒ Modular Algebraic Effects



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



Monad Transformers

Monad Transformer

Monad Transformer

1. Representation Type



```
newtype StateT s m a = StateT (s → m (a,s))
```

Monad Transformer

1. Representation Type

```
newtype StateT s m a = StateT (s → m (a,s))
```

2. Monad Instance

```
return :: a → StateT s m a
(>>=)  :: StateT s m a → (a → StateT s m b) → StateT s m b
```

Monad Transformer

1. Representation Type

```
newtype StateT s m a = StateT (s → m (a,s))
```

2. Monad Instance

```
return :: a → StateT s m a
(>>=)  :: StateT s m a → (a → StateT s m b) → StateT s m b
```

3. Trans Instance

```
lift :: Monad m ⇒ m a → StateT s m a
```

Monad Transformer

1. Representation Type

```
newtype StateT s m a = StateT (s → m (a,s))
```

2. Monad Instance

```
return :: a → StateT s m a
(>>=)  :: StateT s m a → (a → StateT s m b) → StateT s m b
```

3. Trans Instance

```
lift :: Monad m ⇒ m a → StateT s m a
```

4. Operations

```
get :: StateT s m s
put :: s → StateT s m ()
```

Monad Transformer

1. Representation Type

```
newtype StateT s m a = StateT (s → m (a,s))
```

2. Monad Instance

```
return :: a → StateT s m a
(>>=)  :: StateT s m a → (a → StateT s m b) → StateT s m b
```

3. Trans Instance

```
lift :: Monad m ⇒ m a → StateT s m a
```

4. Operations

```
get :: StateT s m s
put :: s → StateT s m ()
```

5. Run Function

```
runStateT :: StateT s m a → s → m (a,s)
```

Application Code with Overloaded Effects

```
inc :: MonadState Int m ⇒ m ()
inc = get >>= put (s + 1)
```

Application Code with Overloaded Effects

```
inc :: MonadState Int m ⇒ m ()
inc = get >>= put (s + 1)
```



4.a Monad Subclass Declaration

```
class Monad m ⇒ MonadState s m | m → s
  where get :: m s
        put :: s → m ()
```

Application Code with Overloaded Effects

```
inc :: MonadState Int m ⇒ m ()
inc = get >>= put (s + 1)
```

4.a Monad Subclass Declaration

```
class Monad m ⇒ MonadState s m | m → s
    where get :: m s
          put :: s → m ()
```

```
> runId (runStateT inc 41)
((),42)
```

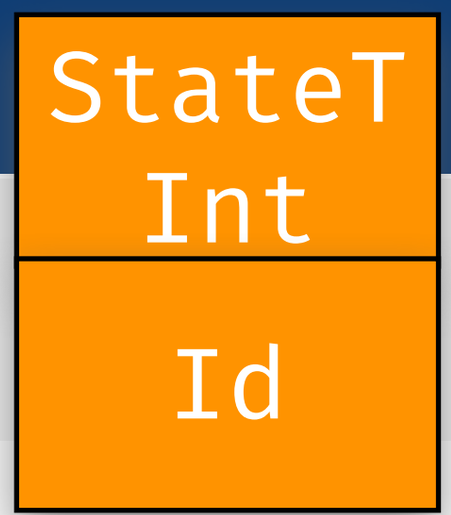
Application Code with Overloaded Effects

```
inc :: MonadState Int m ⇒ m ()
inc = get >>= put (s + 1)
```

4.a Monad Subclass Declaration

```
class Monad m ⇒ MonadState s m | m → s
  where get :: m s
        put :: s → m ()
```

```
> runId (runStateT inc 41)
((),42)
```



Application Code with Overloaded Effects

```
inc :: MonadState Int m ⇒ m ()
inc = get >>= put (s + 1)
```

4.a Monad Subclass Declaration

```
class Monad m ⇒ MonadState s m | m → s
    where get :: m s
          put :: s → m ()
```

```
> runId (runStateT inc 41)
((),42)
```



4.b Monad Subclass Instance

```
instance Monad m ⇒ MonadState s (StateT s m)
    where get = …
          put = …
```

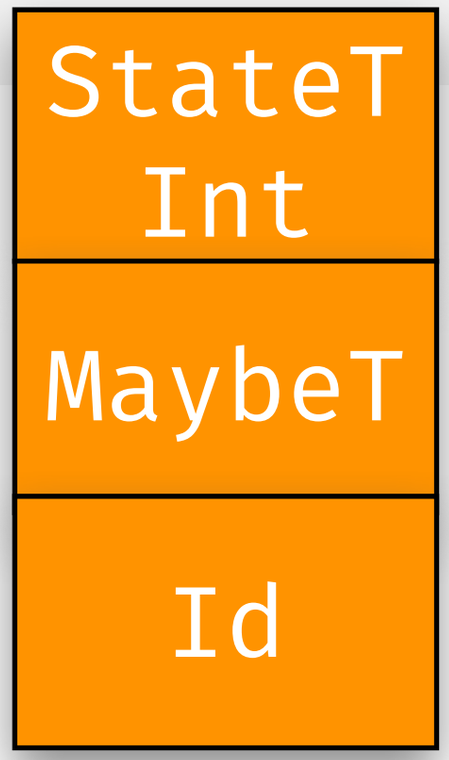
Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

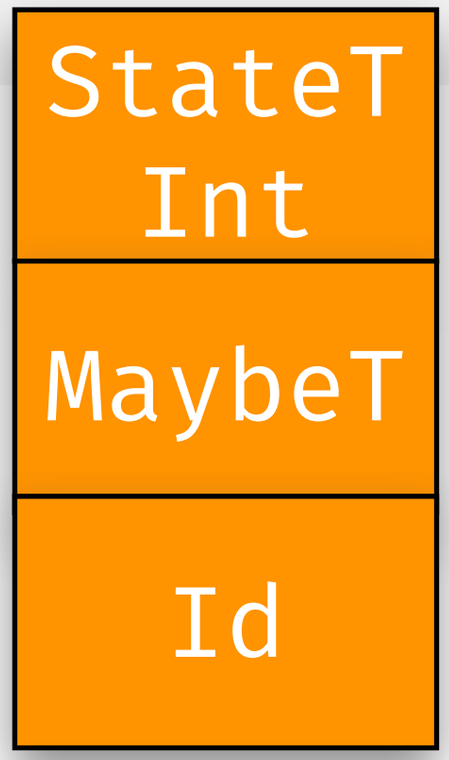
```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

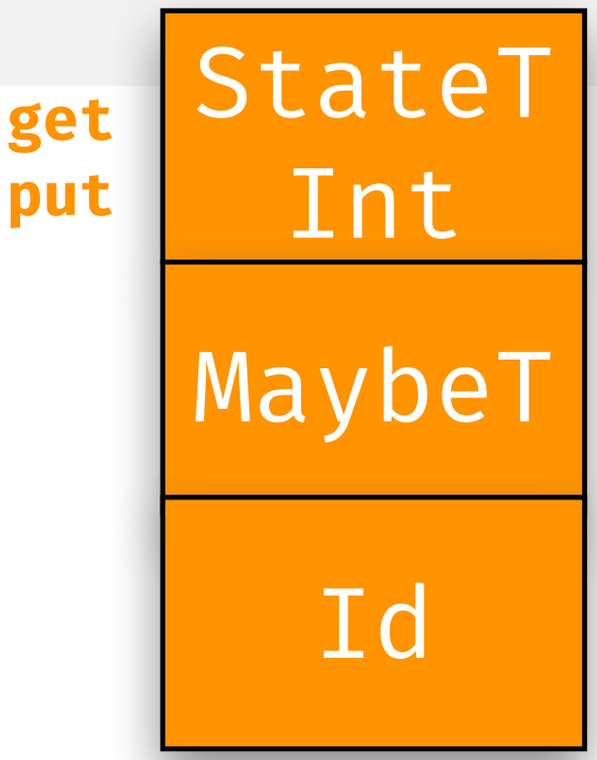
```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

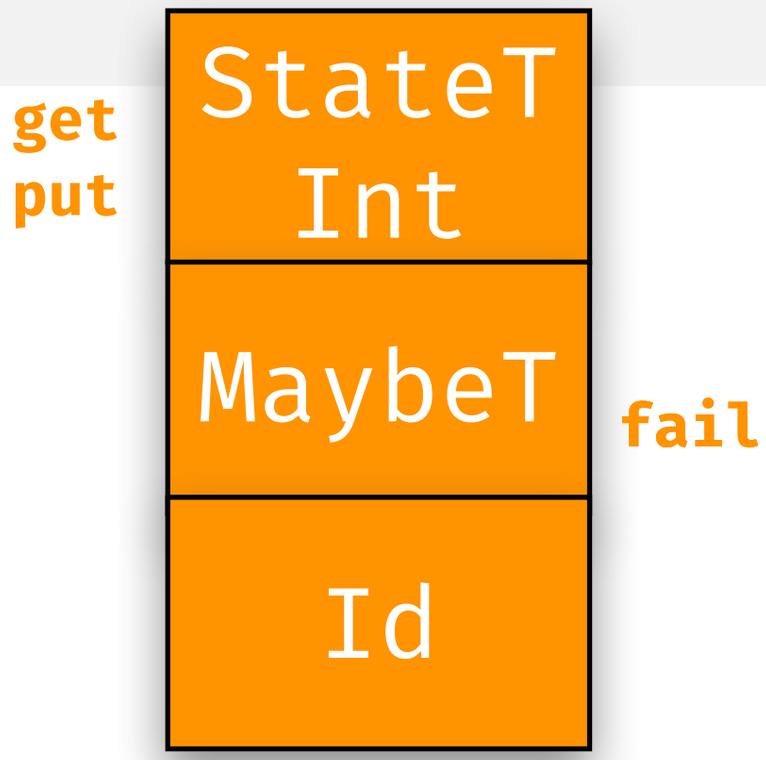
```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

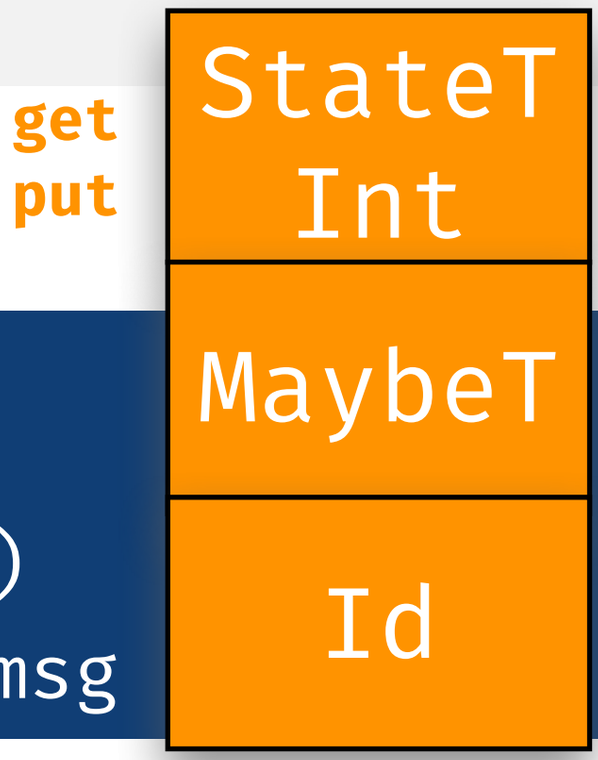
```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



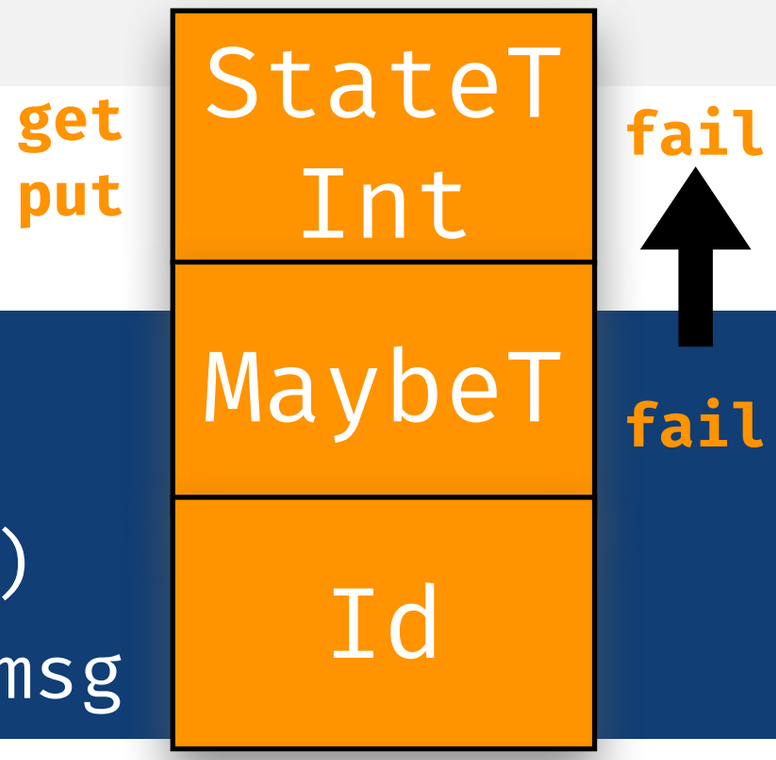
6. Lifting Subclass Instance

```
instance MonadFail m
      ⇒ MonadFail (StateT s m)
  where fail msg = lift fail msg
```

Combining Effects

```
prog :: (MonadState Int m, MonadFail m)
     ⇒ m ()
prog = inc >> fail “oops”
```

```
> runId (runMaybeT (runStateT prog 41))
Nothing
```



6. Lifting Subclass Instance

```
instance MonadFail m
      ⇒ MonadFail (StateT s m)
  where fail msg = lift fail msg
```

The Plan



Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers

The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



Algebraic Operations

General Form

```
op :: forall a. A → (B → M a) → M a
```

parameter — A

arity — B

continuations — (B → M a)

General Form

```
op :: forall a. A → (B → M a) → M a
```

parameter — arity — continuations

Example: State

```
put :: s → (() → M a) → M a
get :: () → (s → M a) → M a
```

Algebraicity Property

captures intuition of continuations

```
op p k >>= f
=
op p (\b -> k b >>= f)
```

Example: State

```
(put s  k)>>=f = put s  (\() -> k ()>>=f)
(get () k)>>=f = get () (\s  -> k s >>=f)
```

Combined Signature

```
op :: forall a. Sig (M a) → M a
```

Combined Signature

```
op :: forall a. Sig (M a) → M a
```

Example: State

1. Signature Functor


```
data STATE s k
    = Get (s → k)
    | Put s k
```

The Plan

Monad Transformers

Algebraic Effects & Handlers

Modular Handlers



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



Algebraic Effects & Handlers

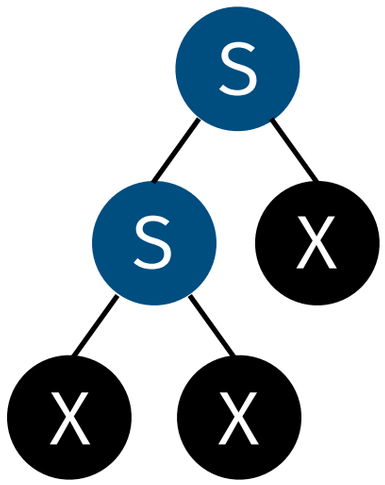
Approach for Signature S

cf. compositional denotational semantics

Approach for Signature S

cf. compositional denotational semantics

free monad



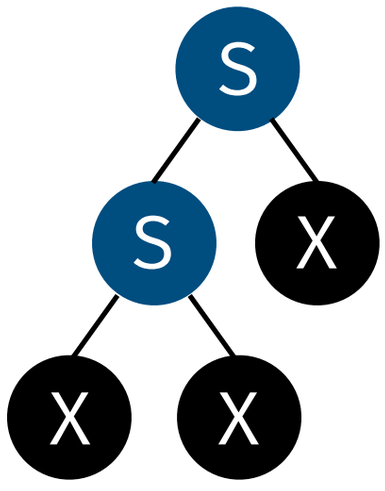
`Free S X`

cf.
abstract syntax tree

Approach for Signature S

cf. compositional denotational semantics

free monad



`Free S X`

cf.
abstract syntax tree

handler

h



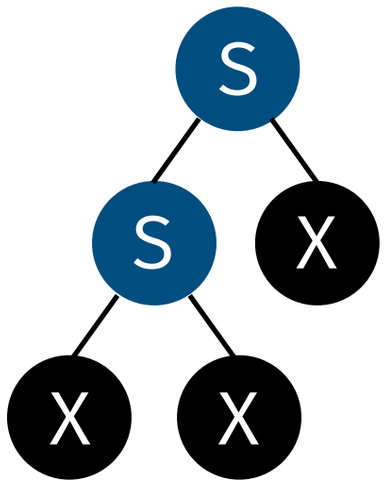
`Free S X → C`

cf.
semantic function

Approach for Signature S

cf. compositional denotational semantics

free monad



`Free S X`

cf.
abstract syntax tree

handler

h

`Free S X → C`

cf.
semantic function

meaning



`C`

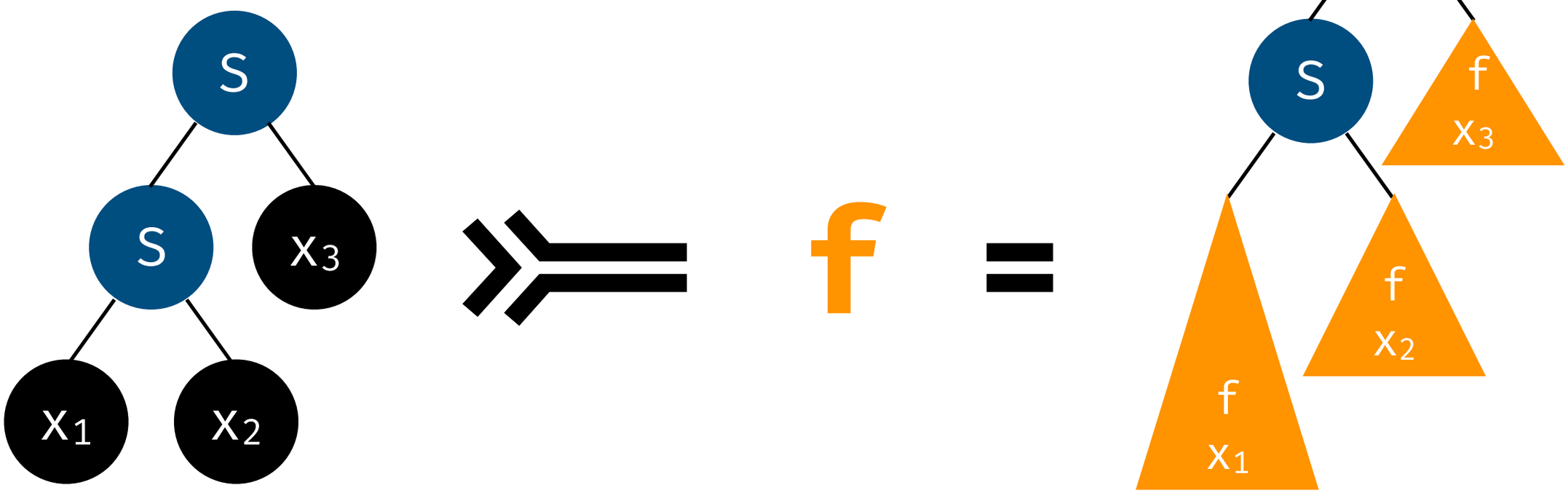
cf.
semantic domain

Free Monad

cf. abstract syntax tree

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

instance Functor sig
⇒ Monad (Free sig)
```



Handler = Fold

cf. compositional semantic function

```
h :: Free S X → C
```

Handler = Fold

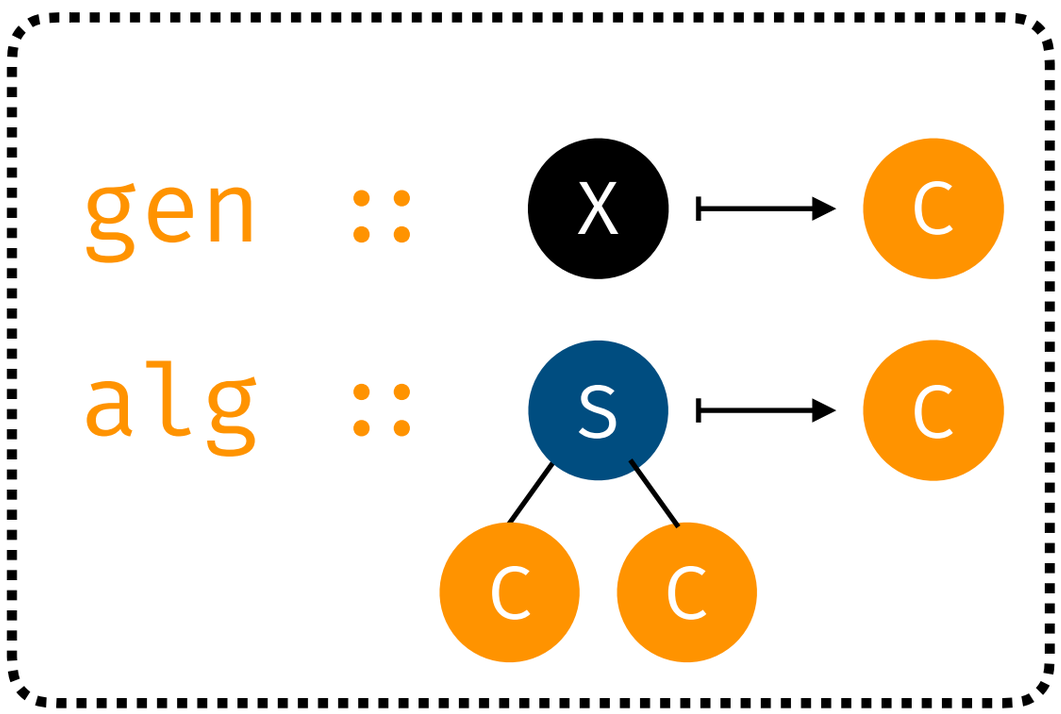
cf. compositional semantic function

```
h :: Free S X → C
h = fold gen alg
```

Handler = Fold

cf. compositional semantic function

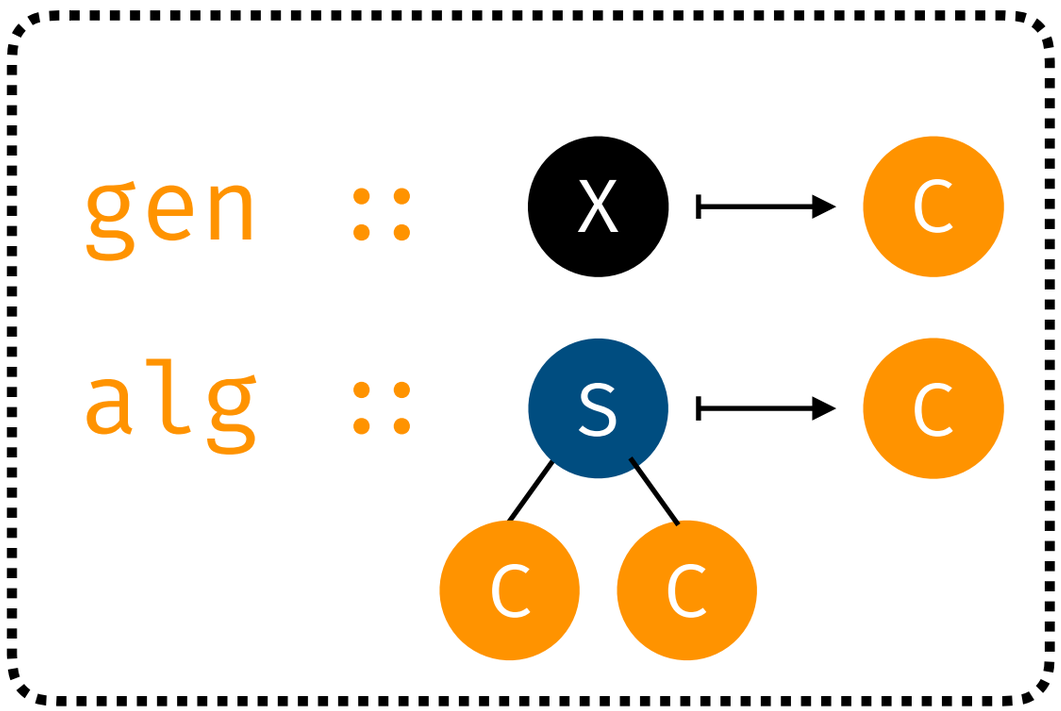
```
h :: Free S X → C
h = fold gen alg
```

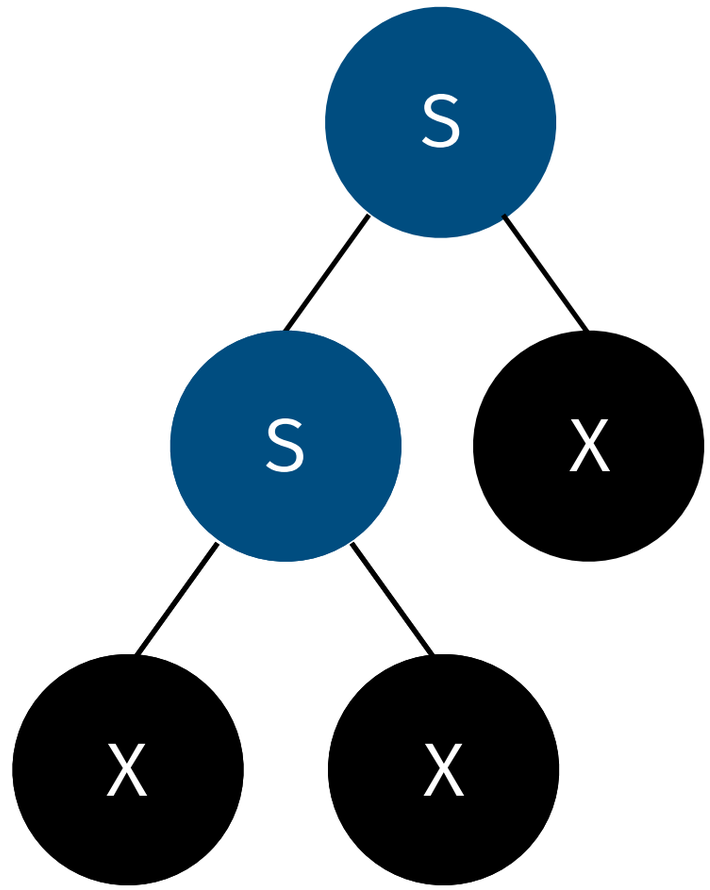


Handler = Fold

cf. compositional semantic function

```
h :: Free S X → C
h = fold gen alg
```

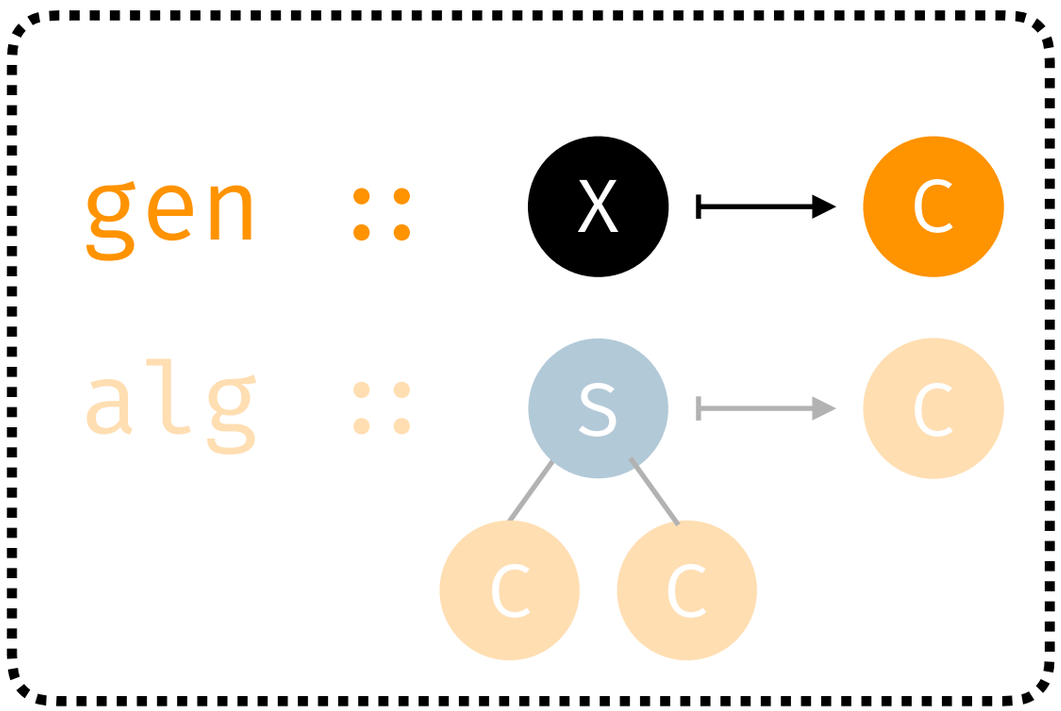


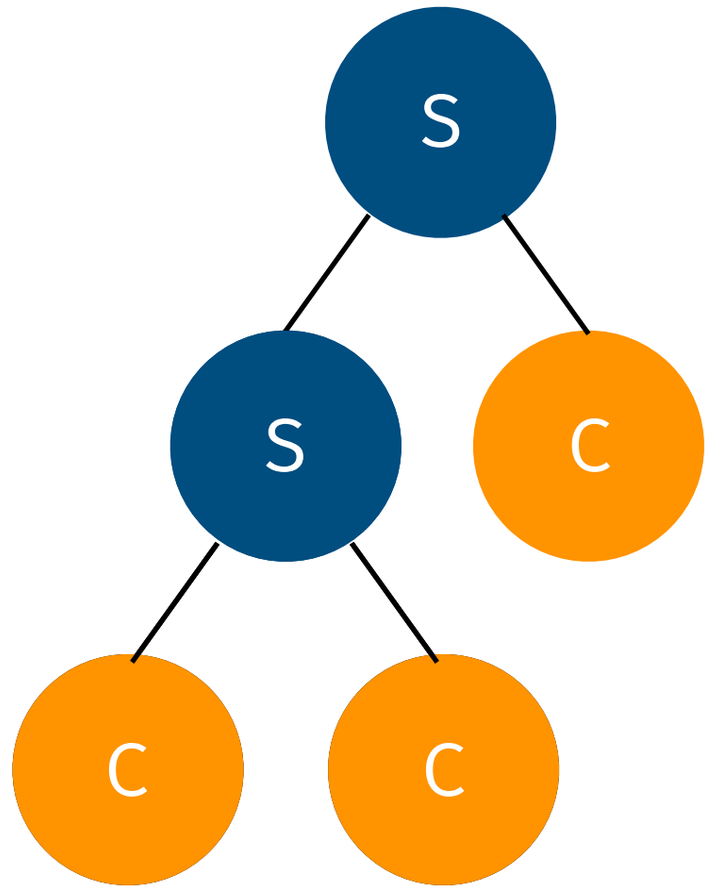


Handler = Fold

cf. compositional semantic function

```
h :: Free S X → C
h = fold gen alg
```

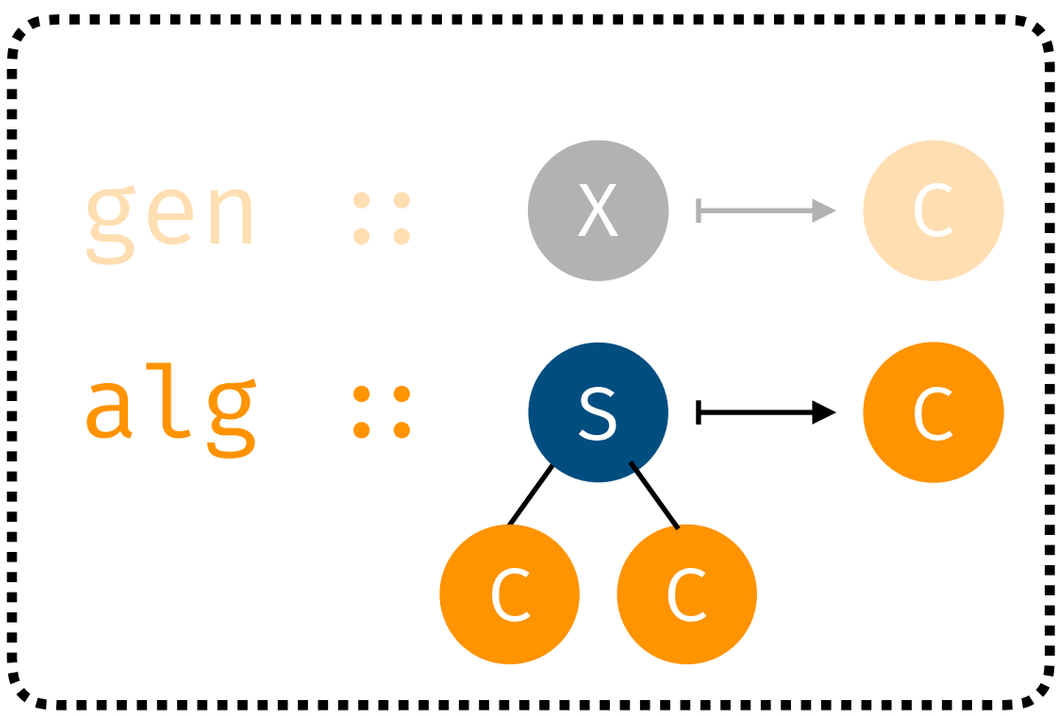


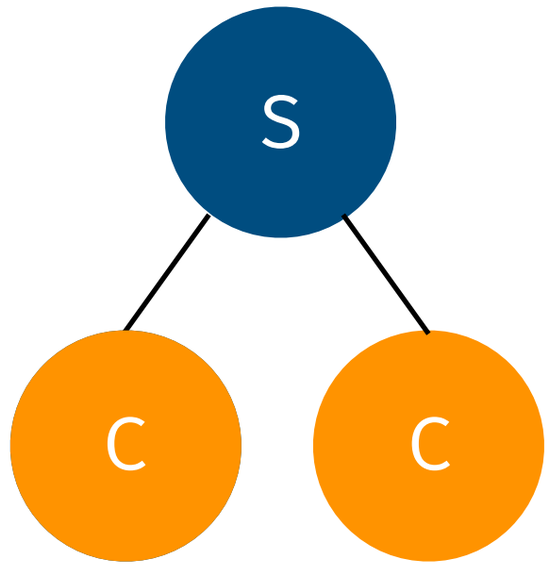


Handler = Fold

cf. compositional semantic function

```
h :: Free S X → C
h = fold gen alg
```



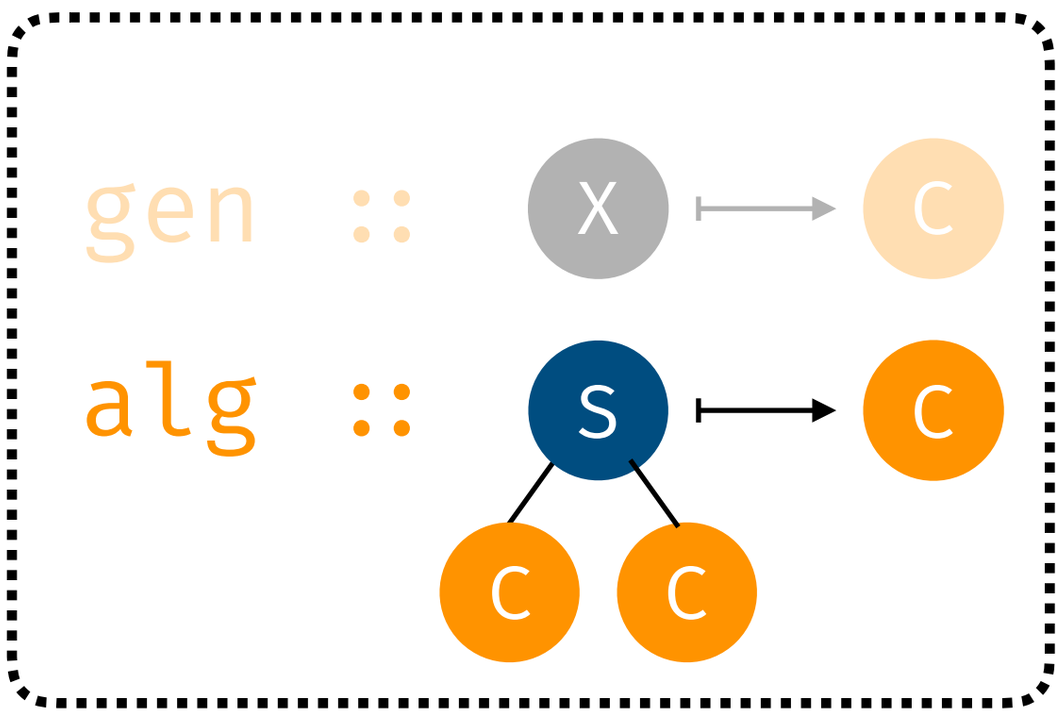


Handler = Fold

cf. compositional semantic function

```
h :: Free S X → C
h = fold gen alg
```





State Handler

```
(SC s, genS, algS)
```

State Handler

(SC s, genS, algS)

2. Carrier Type



```
type SC s x = s → x
```

State Handler

(SC s, genS, algS)

2. Carrier Type

```
type SC s x = s → x
```

3. Generator

```
genS :: x → SC s x
genS x = \s → x
```

State Handler

(SC s, genS, algS)

2. Carrier Type

```
type SC s x = s → x
```

3. Generator

```
genS :: x → SC s x
genS x = \s → x
```

4. Algebra

```
algS :: STATE s (SC s x) → (SC s x)
algS (Put s' k) = \s → k s'
algS (Get k)    = \s → k s s
```

The Plan

Monad Transformers

Algebraic Operations

Modular Handlers



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



Modular Handlers

Composing Algebraic Effects

composing signatures

S_1



S_2

easy

composing handlers

(C_1, gen_1, alg_1)



(C_2, gen_2, alg_2)

non-trivial

Composing Signatures

with Functor Co-Product

```
type S = STATE s + (FAIL + VOID)

data STATE s r = Get (s → r) | Put s r
data FAIL r    = Fail String
data VOID r    -- empty base case
```

Composing Signatures

with Functor Co-Product

```
type S = STATE s + (FAIL + VOID)
```

```
data STATE s r = Get (s → r) | Put s r
data FAIL r    = Fail String
data VOID r    -- empty base case
```

Functor Co-Product

```
data (s1 + s2) r = Inl (s1 r) | Inr (s2 r)
```

Composing Handlers

Sequential Handling

Free (S_1+(S_2+VOID)) X

Composing Handlers

Sequential Handling

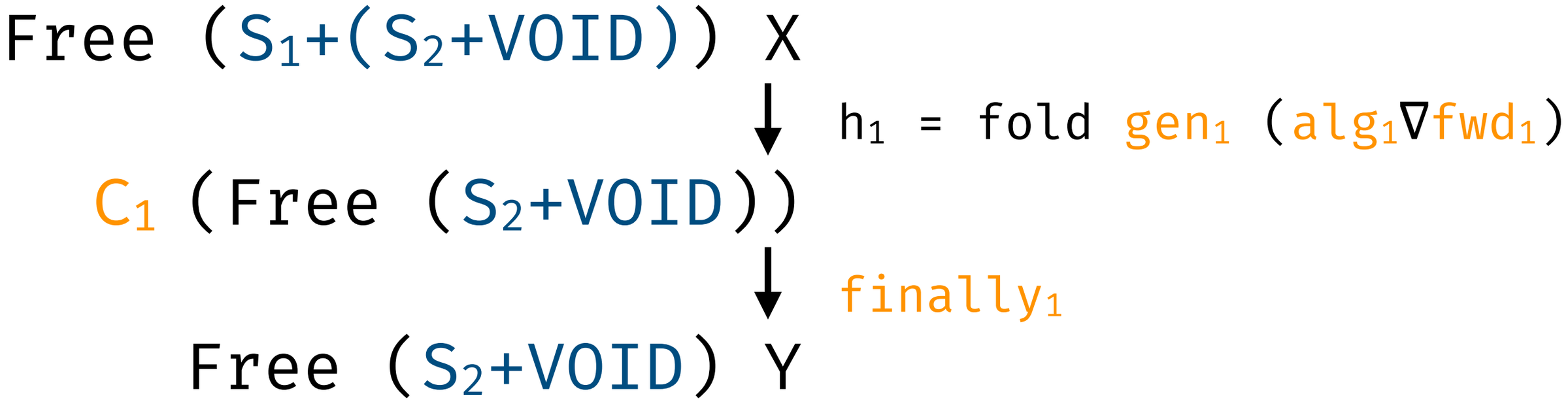
Free (S_1+(S_2+VOID)) X

h_1 = fold gen_1 ($alg_1 \nabla fwd_1$)

C_1 (Free (S_2+VOID))

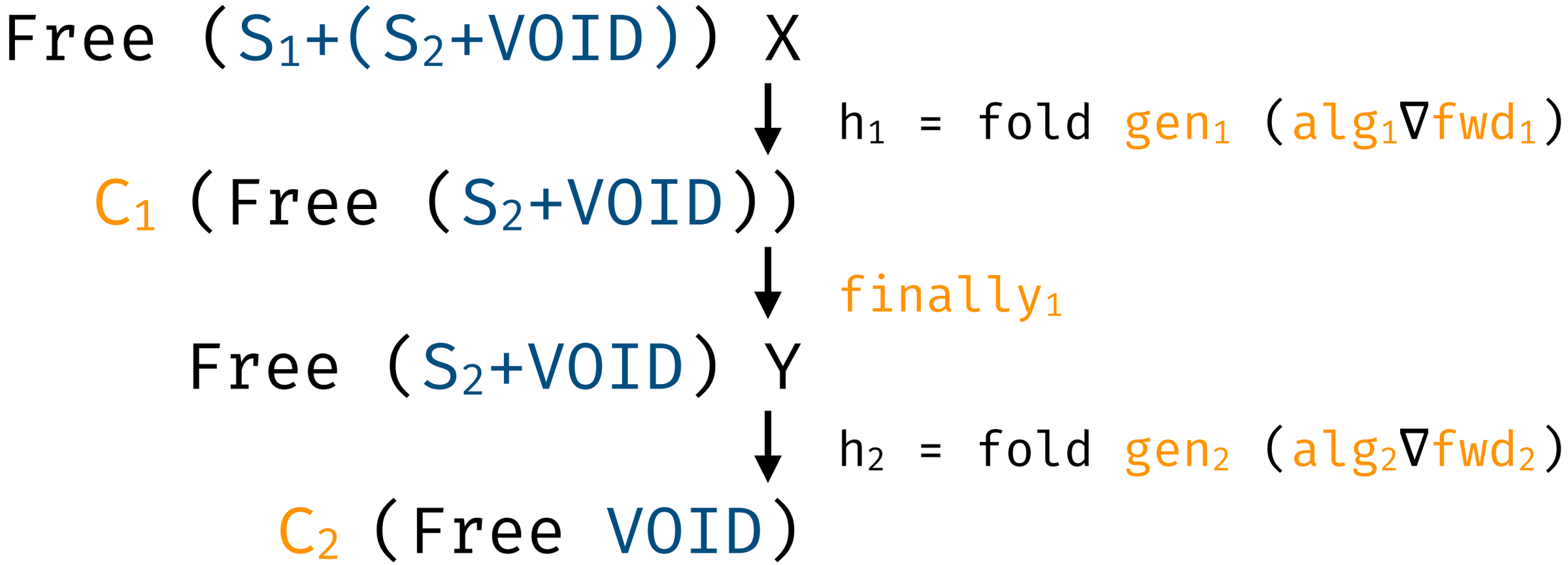
Composing Handlers

Sequential Handling



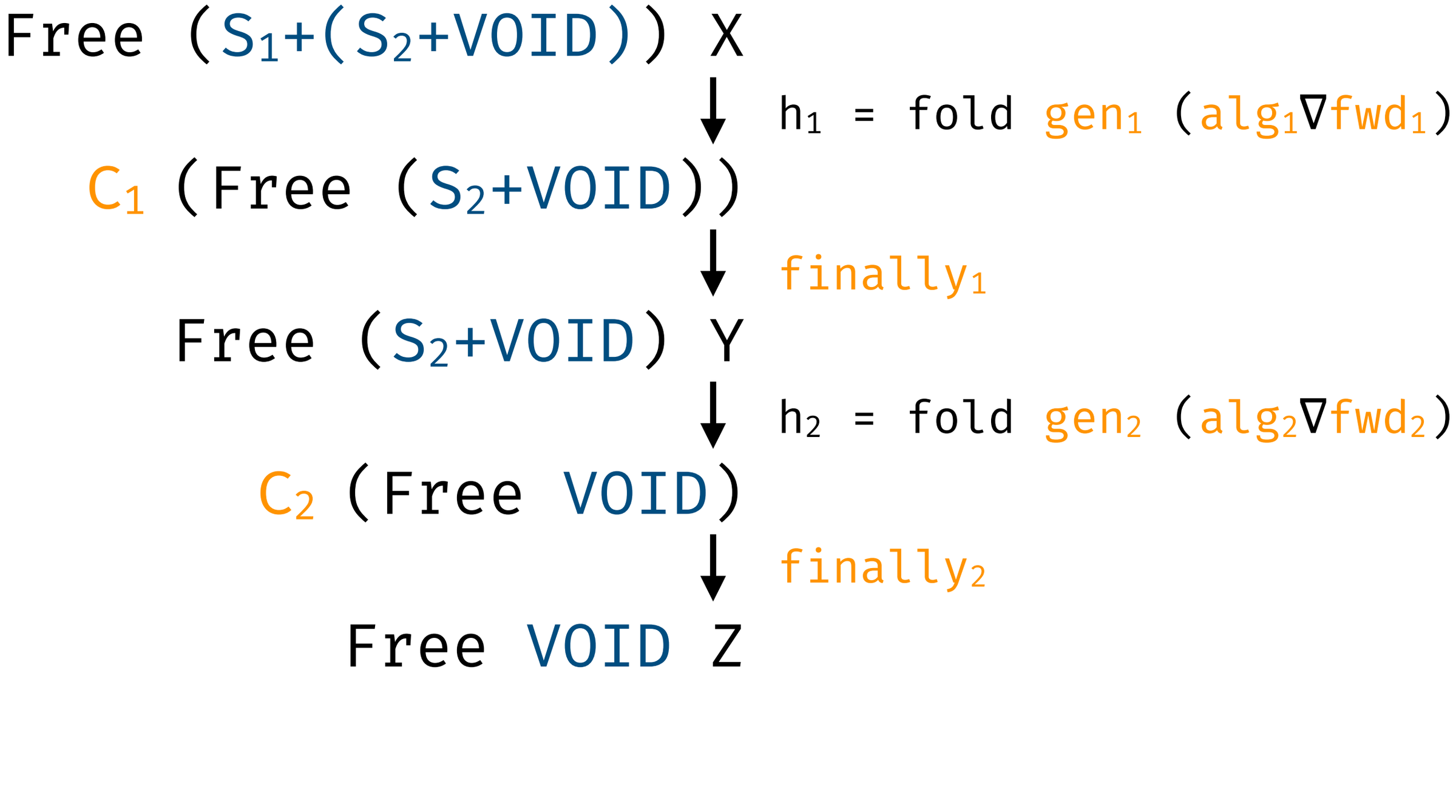
Composing Handlers

Sequential Handling



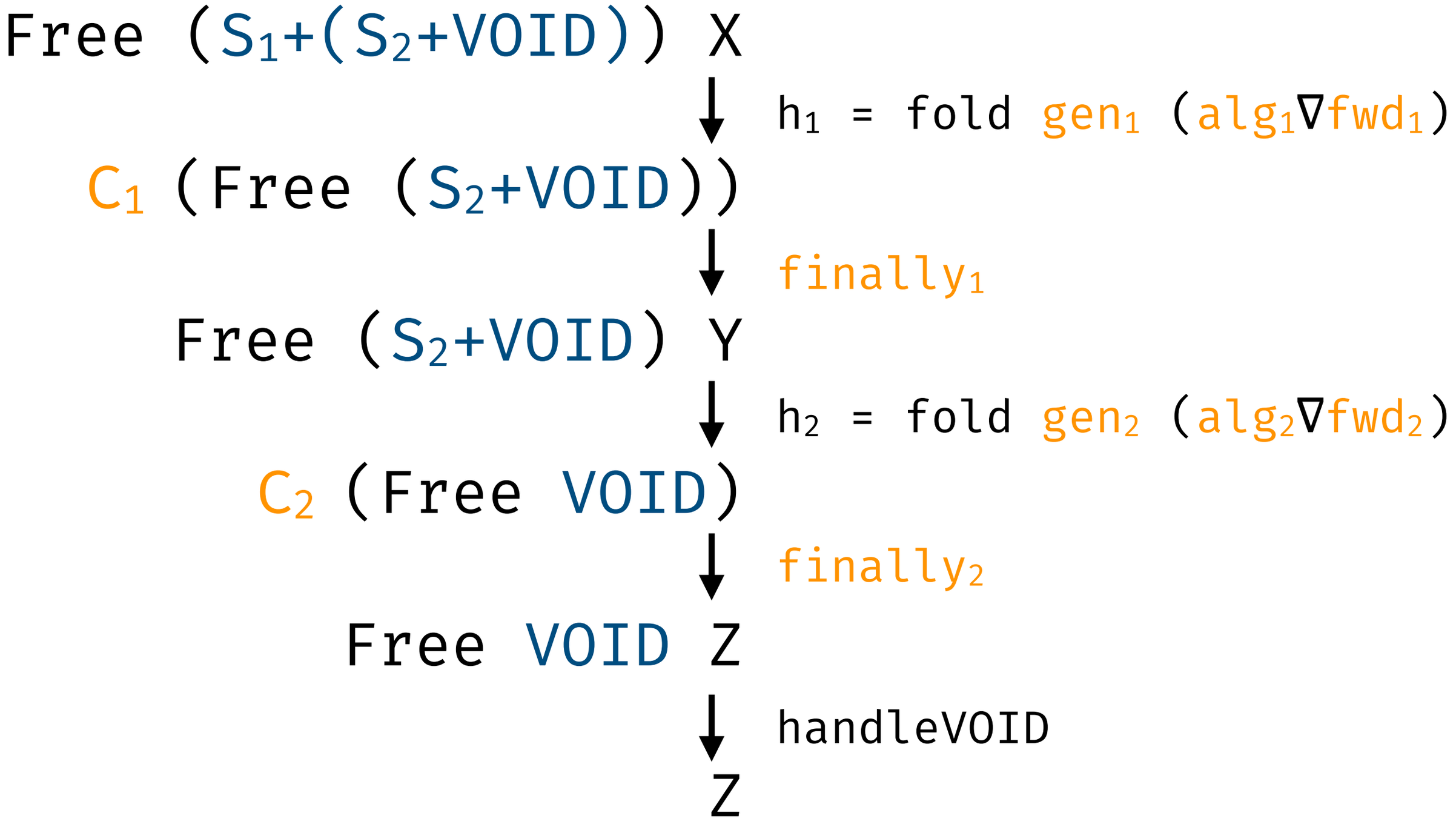
Composing Handlers

Sequential Handling



Composing Handlers

Sequential Handling



Composing Handlers

Sequential Handling

h_1 :: Free (S_1+S_2) X → C_1 (Free S_2)

h_1 = fold gen_1 ($alg_1 \nabla fwd_1$)

Composing Handlers

Sequential Handling

```
h₁ :: Free (S₁+S₂) X → C₁ (Free S₂)
h₁ = fold gen₁ (alg₁∇fwd₁)
```

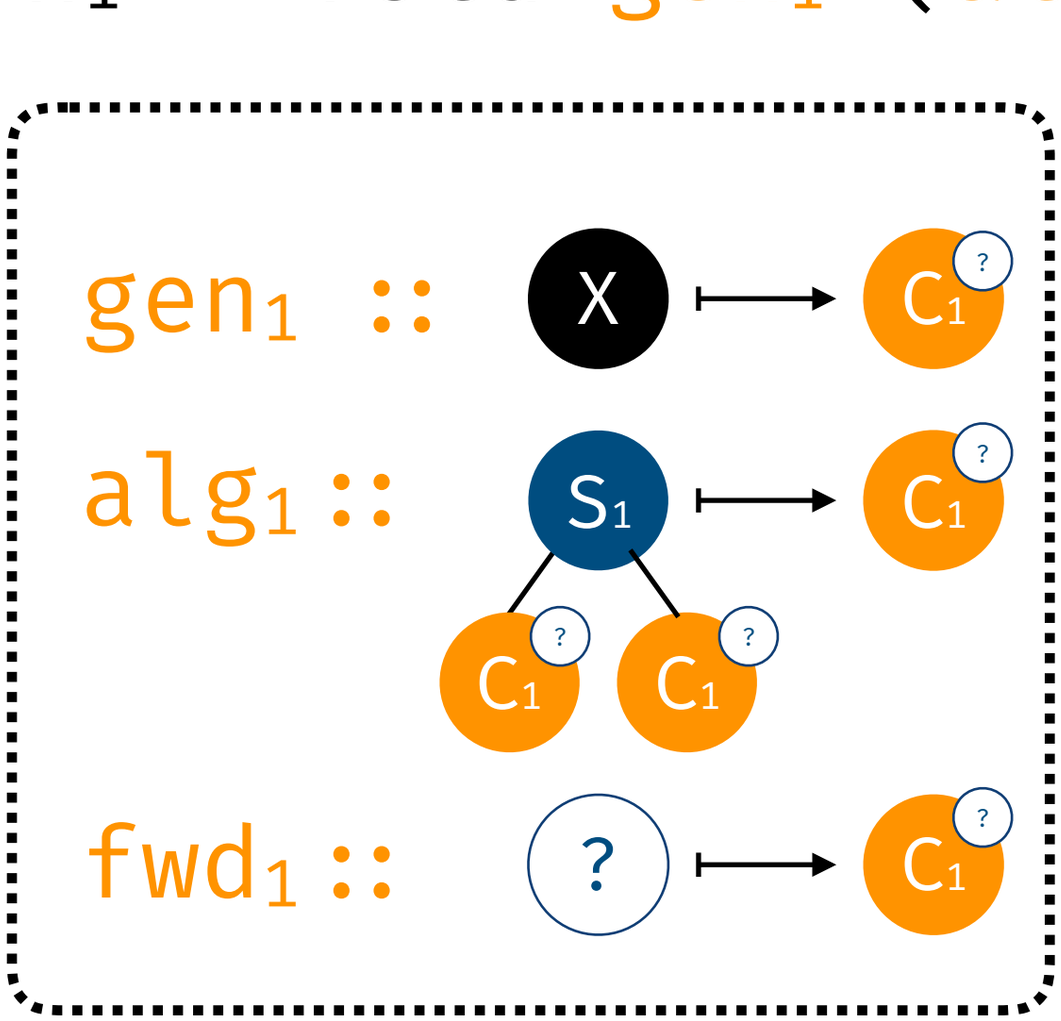


algebra
mediator

Composing Handlers

Sequential Handling

```
h₁ :: Free (S₁+S₂) X → C₁ (Free S₂)
h₁ = fold gen₁ (alg₁∇fwd₁)
```

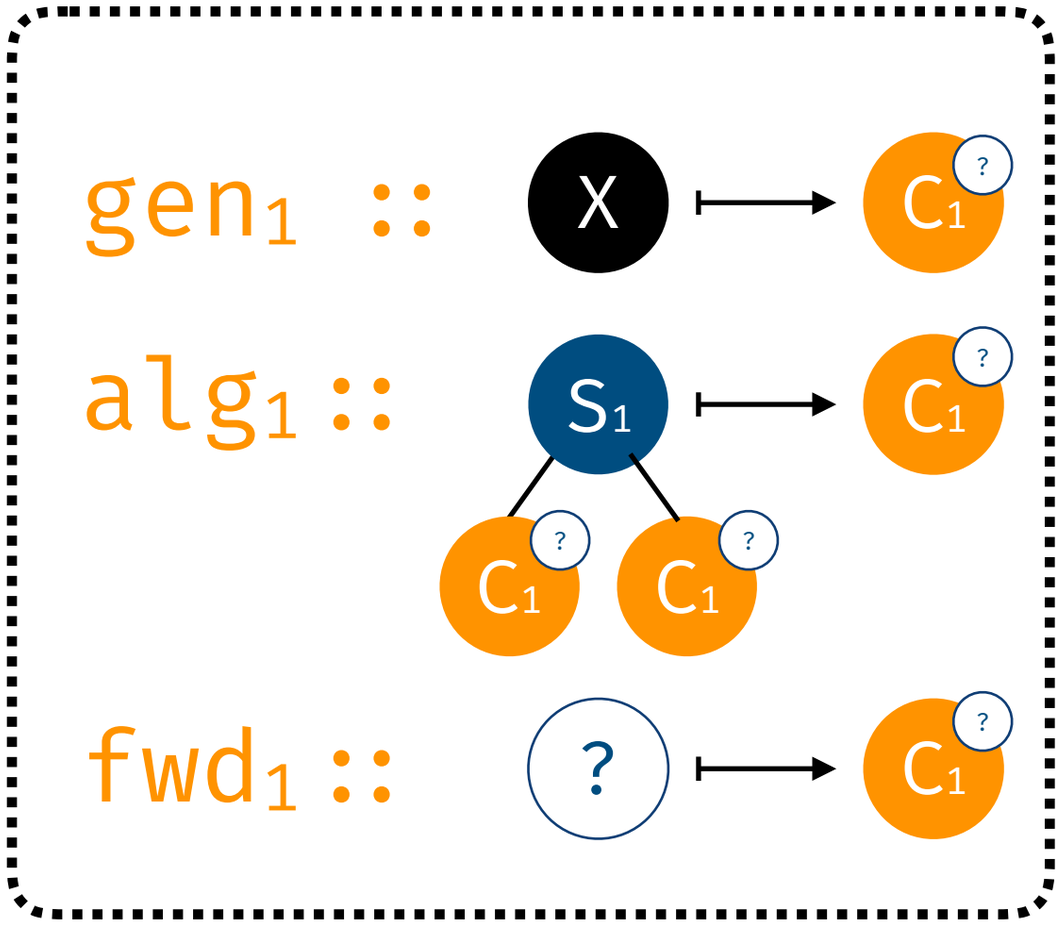


algebra mediator

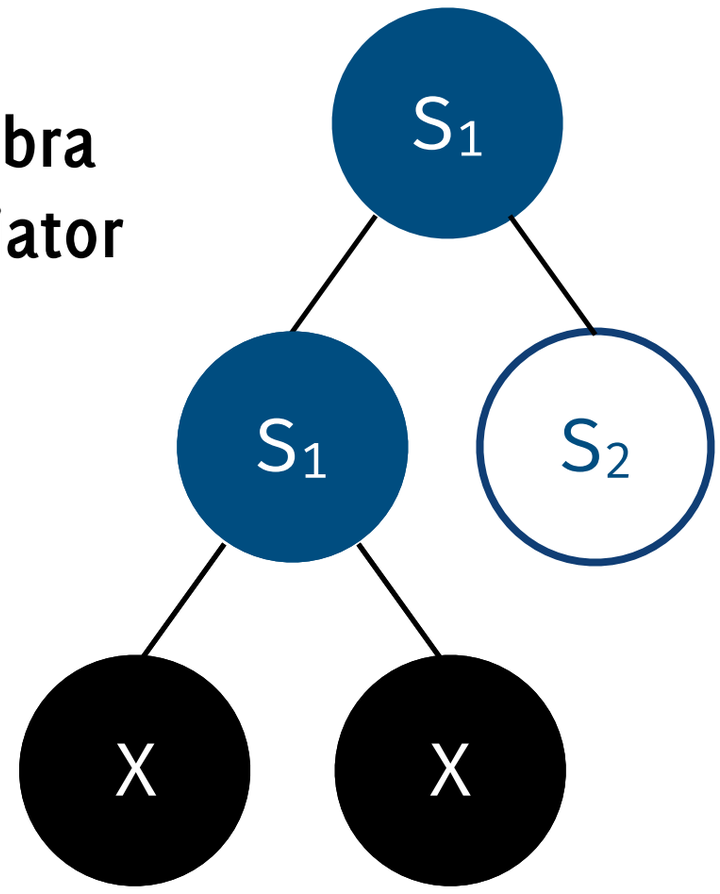
Composing Handlers

Sequential Handling

```
h₁ :: Free (S₁+S₂) X → C₁ (Free S₂)
h₁ = fold gen₁ (alg₁∇fwd₁)
```



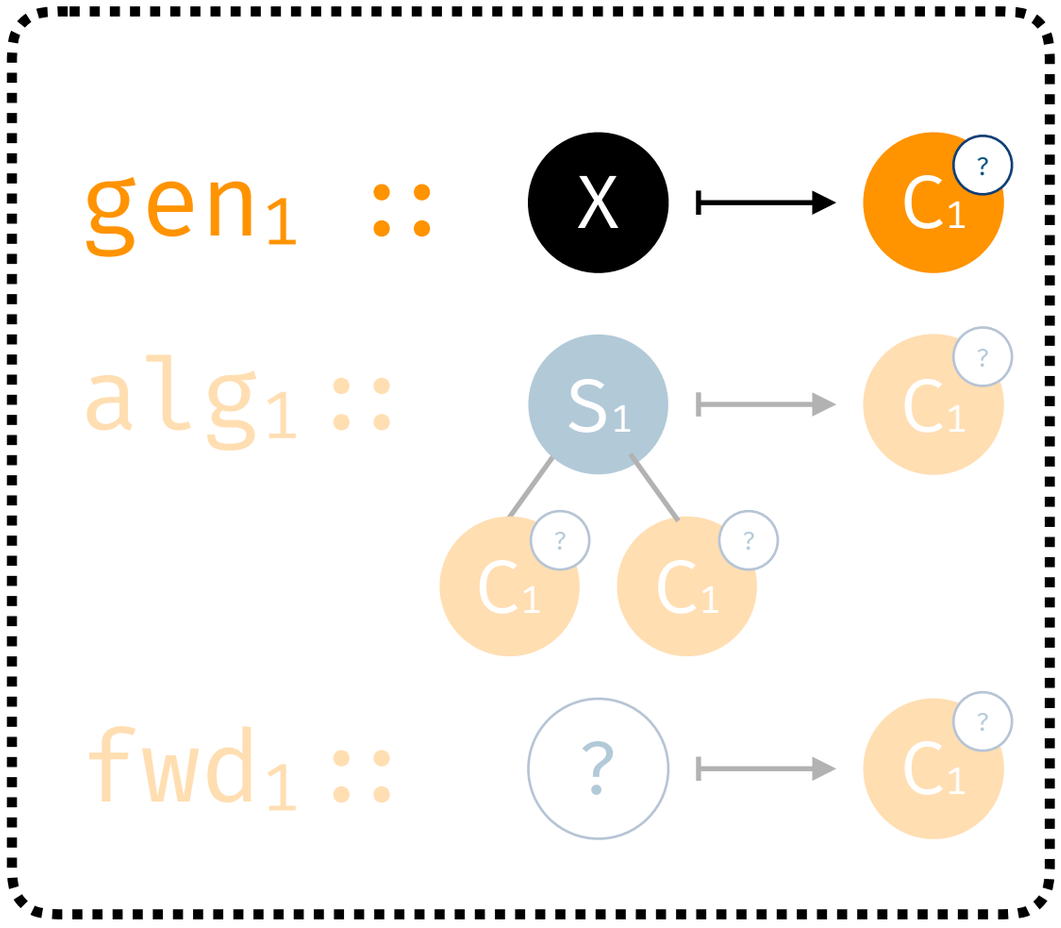
algebra mediator



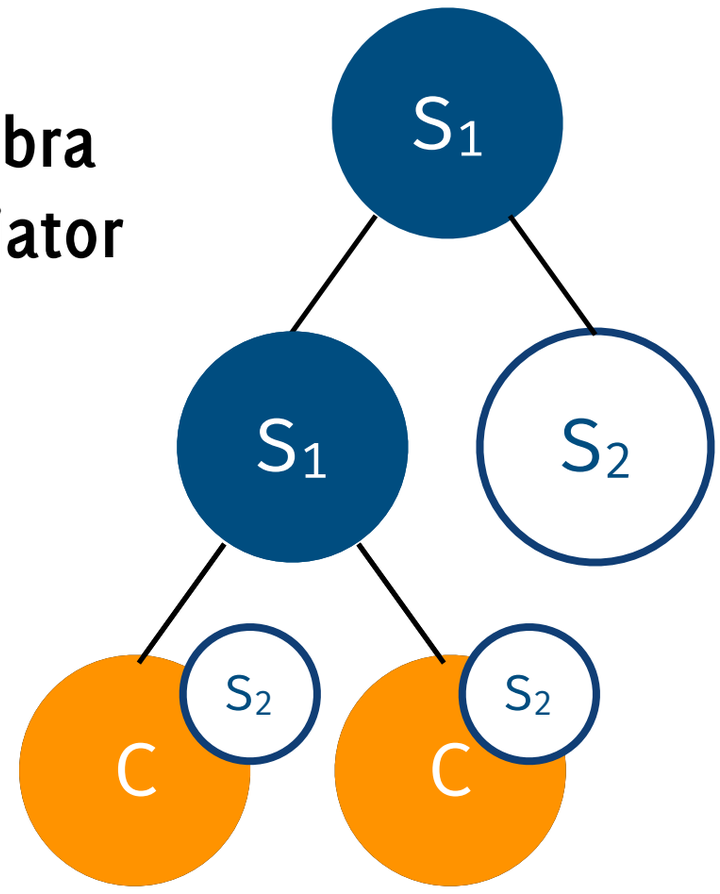
Composing Handlers

Sequential Handling

```
h₁ :: Free (S₁+S₂) X → C₁ (Free S₂)
h₁ = fold gen₁ (alg₁∇fwd₁)
```



algebra mediator

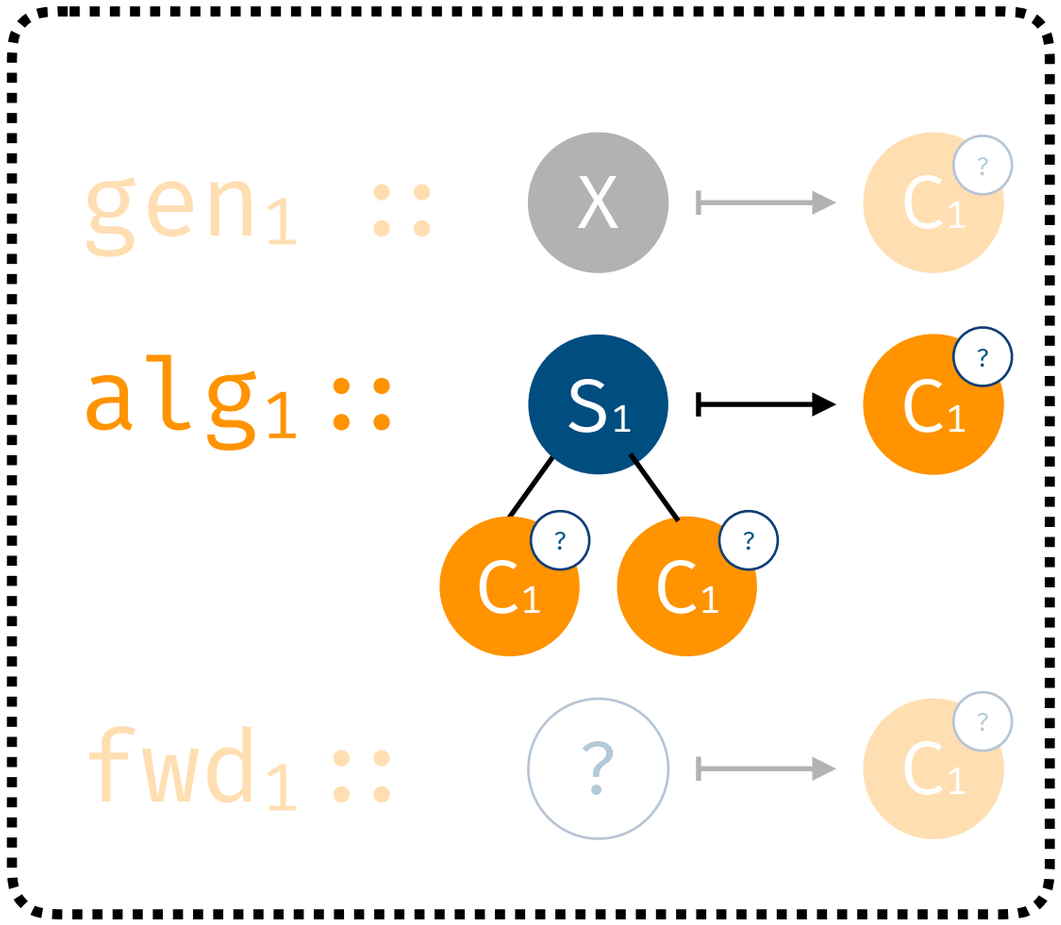


Composing Handlers

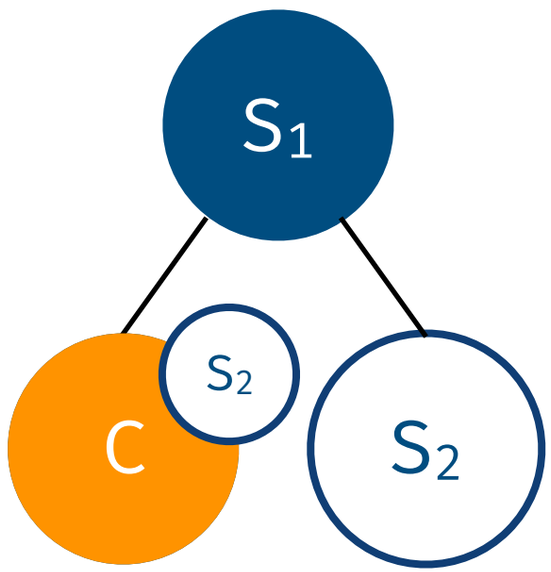
Sequential Handling

h_1 :: Free (S_1+S_2) X → C_1 (Free S_2)

h_1 = fold gen_1 ($alg_1 \nabla fwd_1$)



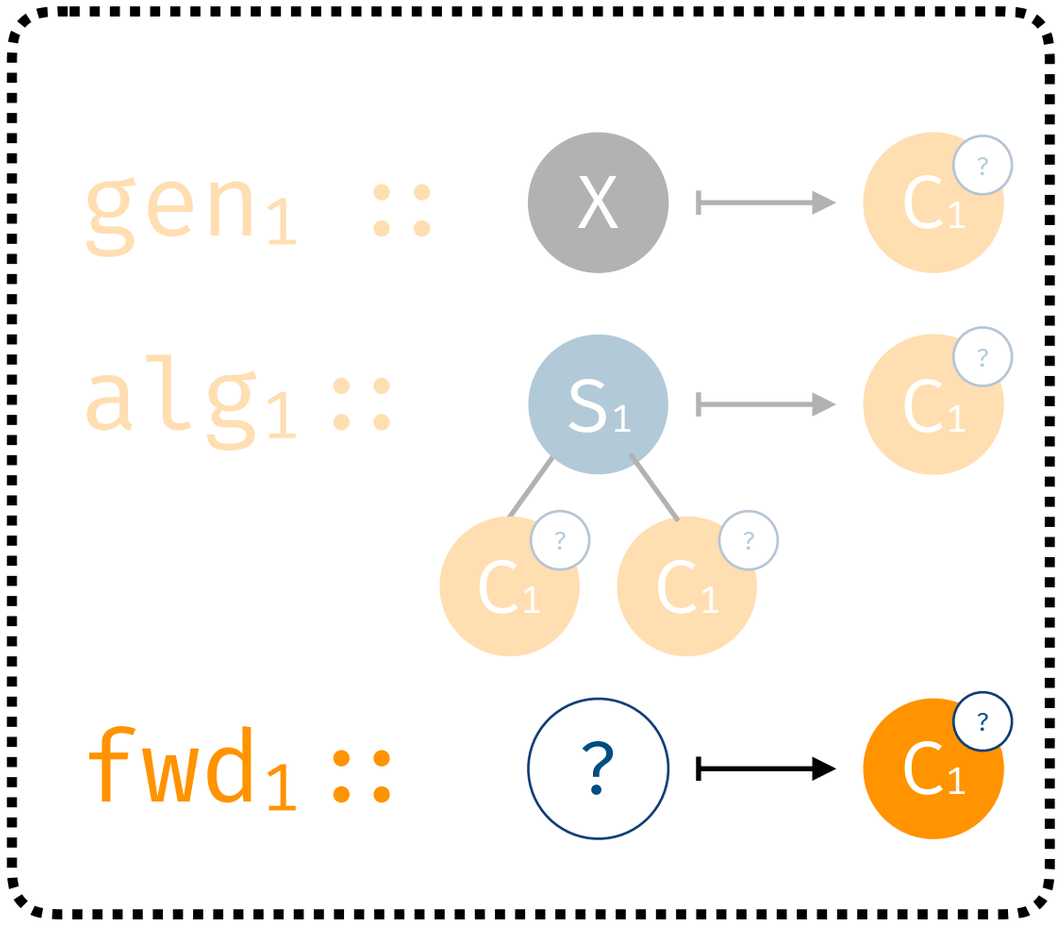
algebra mediator



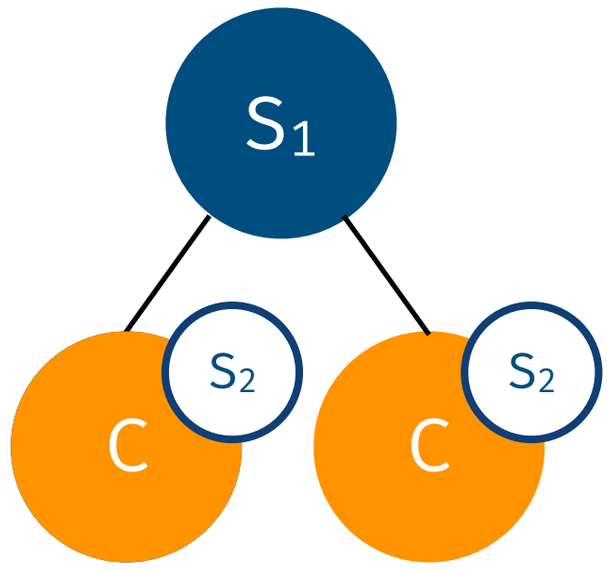
Composing Handlers

Sequential Handling

```
h₁ :: Free (S₁+S₂) X → C₁ (Free S₂)
h₁ = fold gen₁ (alg₁∇fwd₁)
```



algebra mediator

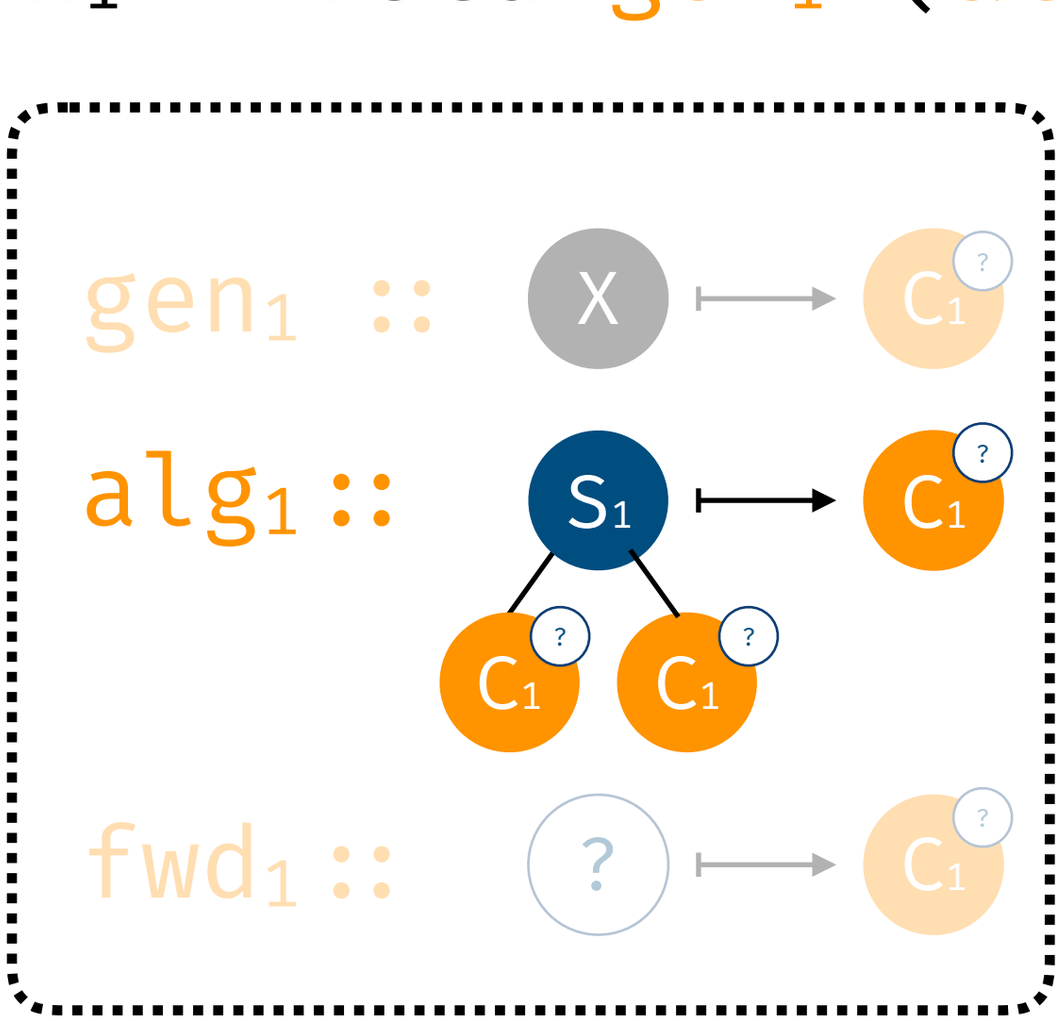


Composing Handlers

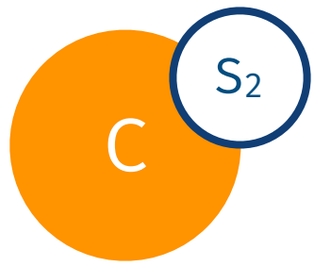
Sequential Handling

h_1 :: Free (S_1+S_2) X → C_1 (Free S_2)

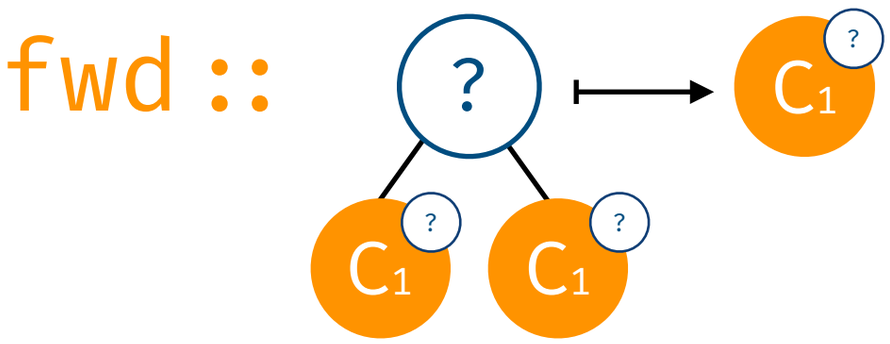
h_1 = fold gen_1 ($alg_1 \nabla fwd_1$)



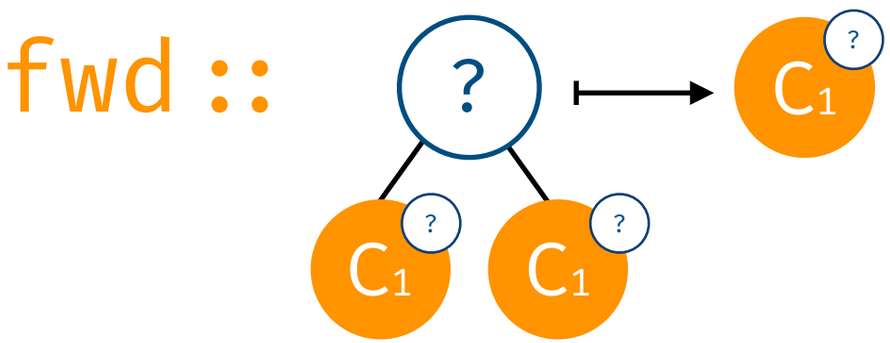
algebra mediator



Modular Carrier

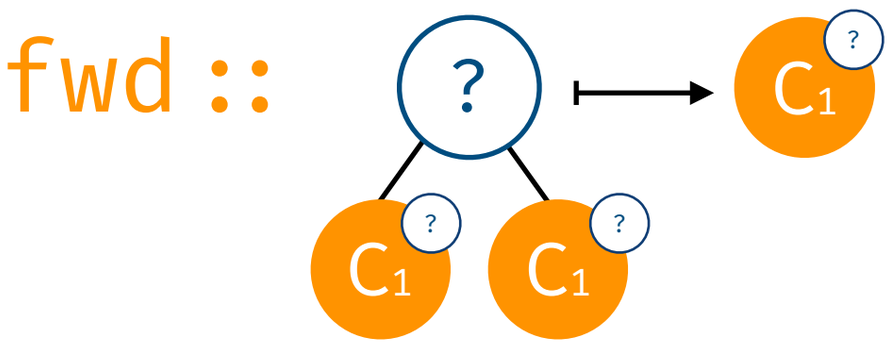


Modular Carrier




```
class ModularCarrier (c :: (* → *) → *) where
  fwd :: Monad m ⇒ m (c m) → c m
```

Modular Carrier




```
class ModularCarrier (c :: (* → *) → *) where
  fwd :: Monad m ⇒ m (c m) → c m
```

Example: Boolean State

```
newtype MSC s a m = MSC {runMSC :: s → m a}
```

5. Modular Carrier Instance

```
instance ModularCarrier (MSC s a) where
  fwd p = MSC (\s → do x ← p; runMSC x s)
```

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”
```

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”

>                 handleVOID $
                    finallyF $
    fold genF (algF∇fwdSig) $
            flip finallyS 0 $
    fold genS (algS∇fwdSig) $
                        prog
```

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”

 >                    handleVOID $
                        finallyF $
   fold genF (algF∇fwdSig) $
              flip finallyS 0 $
   fold genS (algS∇fwdSig) $
                          prog
```

handle STATE

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”
```

```
>                    handleVOID $
                       finallyF $
  fold genF (algF∇fwdSig) $
            flip finallyS 0 $
  fold genS (algS∇fwdSig) $
                          prog
```

handle FAIL

handle STATE

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”
```

```
>  handleVOID $
     finallyF $
   fold genF (algF∇fwdSig) $
          flip finallyS 0 $
   fold genS (algS∇fwdSig) $
                      prog
```

handle VOID

handle FAIL

handle STATE

Stacking Effects

```
prog :: Free (STATE Int + (FAIL + VOID)) Int
prog = inc >> fail “oops”
```

```
>  handleVOID $
     finallyF $
   fold genF (algF∇fwdSig) $
            flip finallyS 0 $
   fold genS (algS∇fwdSig) $
                     prog
```

handle VOID

handle FAIL

handle STATE

6. Finally Function

```
finallyS :: MSC s a m → s → m a
finallyS = runSMC
```

The Plan

Monad Transformers

Modular Algebraic Effects ⇒ Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Monad Transformers ⇒ Modular Algebraic Effects



The Plan

Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers



The Plan

Monad Transformers

Modular Algebraic Effects ⇒ Monad Transformers

Algebraic Operations

Algebraic Effects & Handlers

Modular Handlers

Monad Transformers ⇒ Modular Algebraic Effects



Modular Algebraic Effects

$\Rightarrow$

Monad Transformers

Modular Algebraic Effects



Monad Transformers

- Signature
- Carrier Type
- Generator
- ModularCarrier Instance
- Algebra
- Finally Function

- Representation Type
- Monad Instance
- Trans Instance
- Subclass Declaration
- Subclass Instance
- Lifting Subclass Instances
- Run Function

Modular Algebraic Effects



Monad Transformers

- Signature
  - Carrier Type
  - Generator
  - ModularCarrier Instance
  - Algebra
  - Finally Function
- Representation Type
- Monad Instance
- Trans Instance
- Subclass Declaration
- Subclass Instance
- Lifting Subclass Instances
- Run Function

Modular Algebraic Effects ⟹ Monad Transformers

Signature → Representation Type

Carrier Type

Generator

ModularCarrier Instance

Algebra

Finally Function → Run Function

Monad Instance

Trans Instance

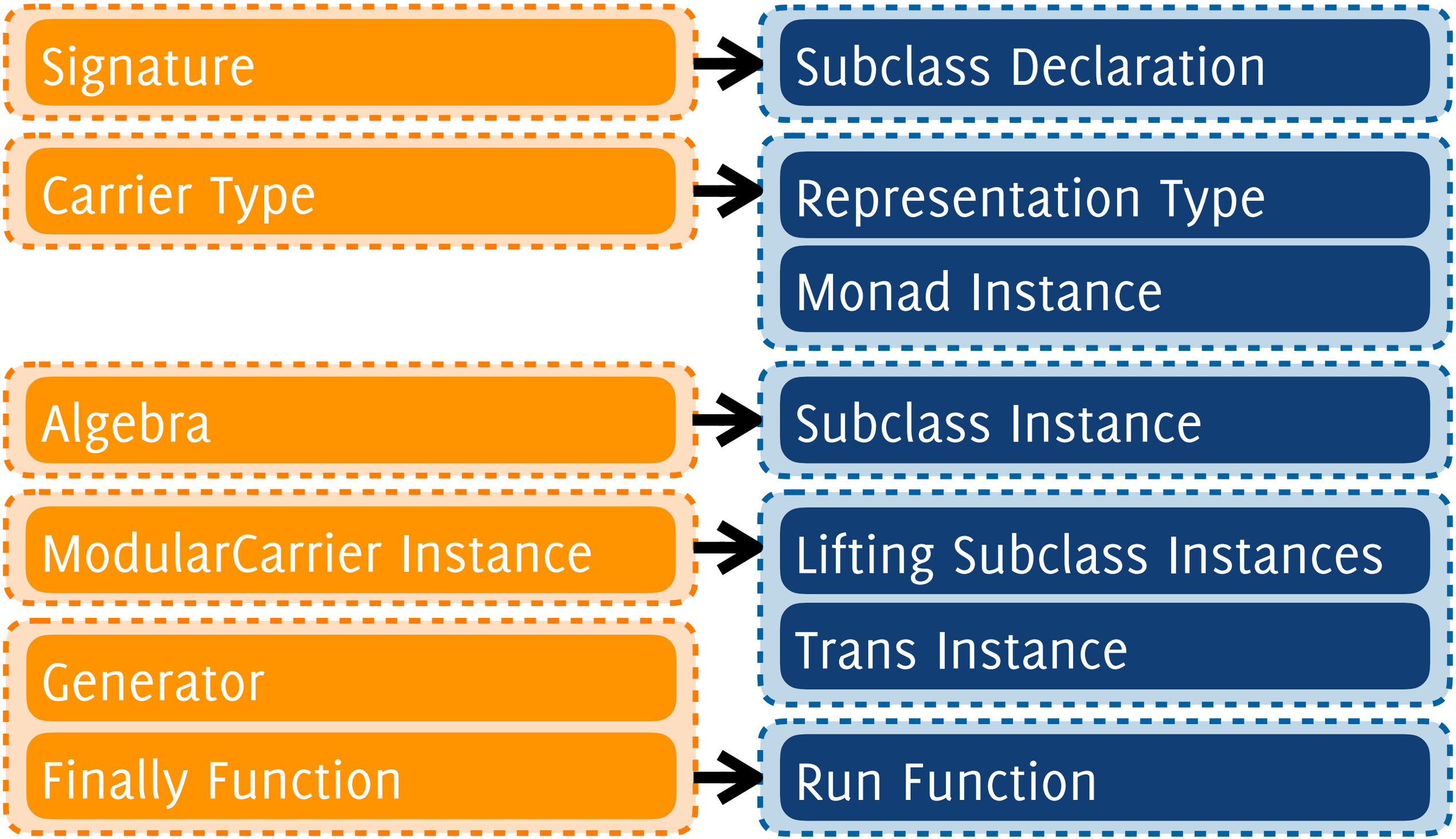
Subclass Declaration

Subclass Instance

Lifting Subclass Instances

Solution 1: Free Monad Transformer

Modular Algebraic Effects ⟹ Monad Transformers



Modular Algebraic Effects ⟹ Monad Transformers

Signature → Subclass Declaration

Carrier Type → Representation Type

Algebra → Monad Instance

Modular Combinator → Lifting Instances

Generator → Trans Instance

Finally Function → Run Function

Solution 2: Continuation Monad Transformer

Monad Transformers

$\Rightarrow$

Modular Algebraic Effects

Modular Algebraic Effects

- Signature
- Carrier Type
- Generator
- ModularCarrier Instance
- Algebra
- Finally Function



Monad Transformers

- Representation Type
- Monad Instance
- Trans Instance
- Subclass Declaration
- Subclass Instance
- Lifting Subclass Instances
- Run Function

Modular Algebraic Effects



Monad Transformers

Modular Algebraic Effects	Monad Transformers
Signature	Representation Type
Carrier Type	Monad Instance
Generator	Trans Instance
ModularCarrier Instance	Subclass Declaration
Algebra	Subclass Instance
Finally Function	Lifting Subclass Instances
	Run Function

Modular Algebraic Effects ⟸ Monad Transformers

Modular Algebraic Effects	Monad Transformers
Signature	Subclass Declaration
Carrier Type	Representation Type
Generator	Monad Instance: `return`
ModularCarrier Instance	Trans Instance
Algebra	Subclass Instance
Finally Function	Run Function

Modular Algebraic Effects ⟸ Monad Transformers

Signature ← Subclass Declaration ⚠

Carrier Type ← Representation Type

Generator ← Monad Instance: return

ModularCarrier ← Trans Instance

Algebra ← Subclass Instance

Finally Function ← Run Function

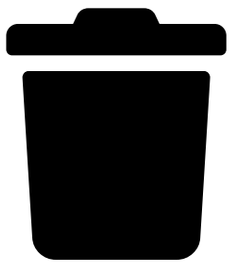


Modular Algebraic Effects ⟸ Monad Transformers

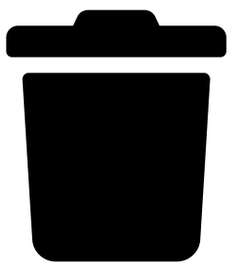
Monad Instance: ≫=

Lifting Subclass Instances

Modular Algebraic Effects ⟸ Monad Transformers

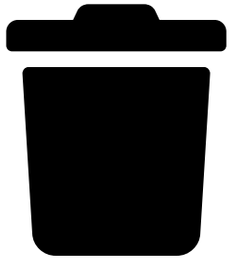


⟵ Monad Instance: ≫=



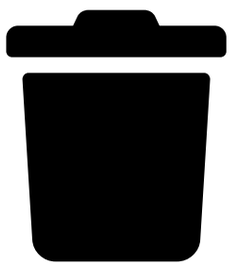
⟵ Lifting Subclass Instances

Modular Algebraic Effects ⟸ Monad Transformers



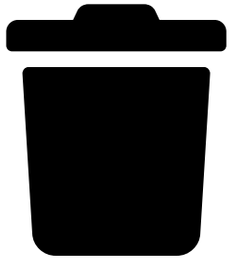
← Monad Instance: ≫=

Free Monad: ≫= 



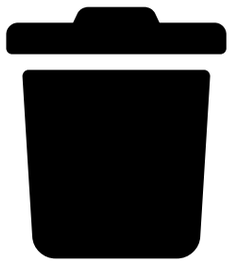
← Lifting Subclass Instances

Modular Algebraic Effects ⟸ Monad Transformers



← Monad Instance: ≫=

Free Monad: ≫= 



← Lifting Subclass Instances

Functor Co-Product 

Proceed to Checkout



Summary

Summary



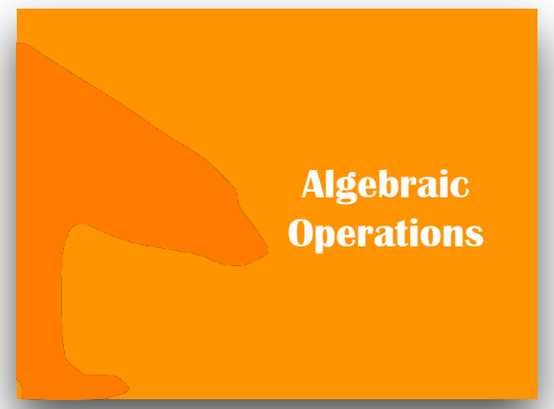
Summary

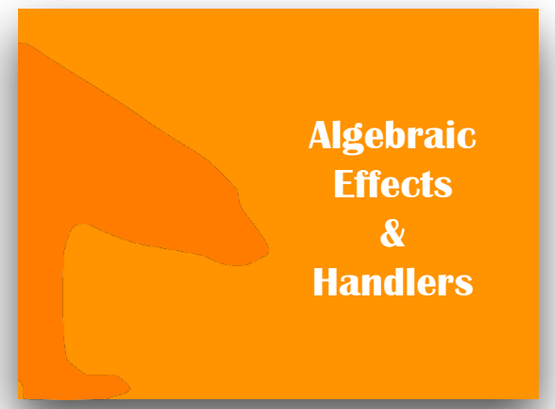




Summary


Monad
Transformers

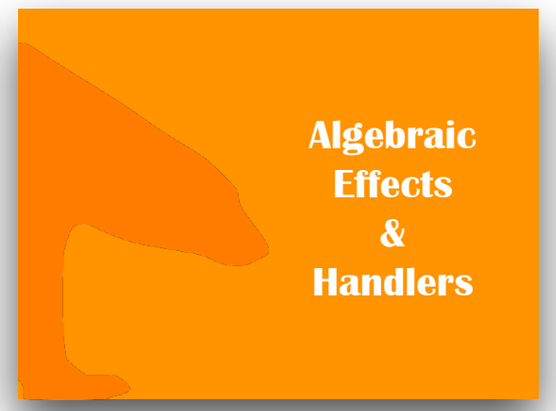

Algebraic
Operations

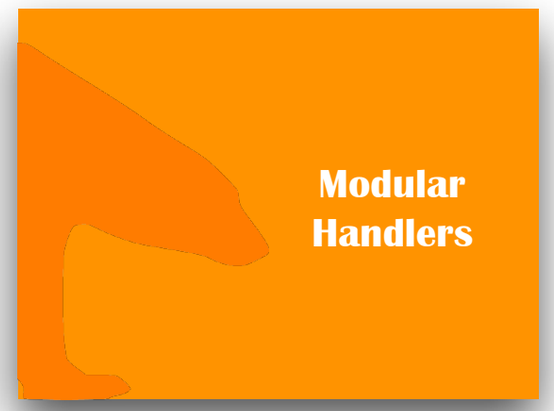

Algebraic
Effects
&
Handlers

Summary


Monad
Transformers


Algebraic
Operations


Algebraic
Effects
&
Handlers


Modular
Handlers

Summary

Monad Transformers

Modular Algebraic Effects ⇒ Monad Transformers



Algebraic Operations

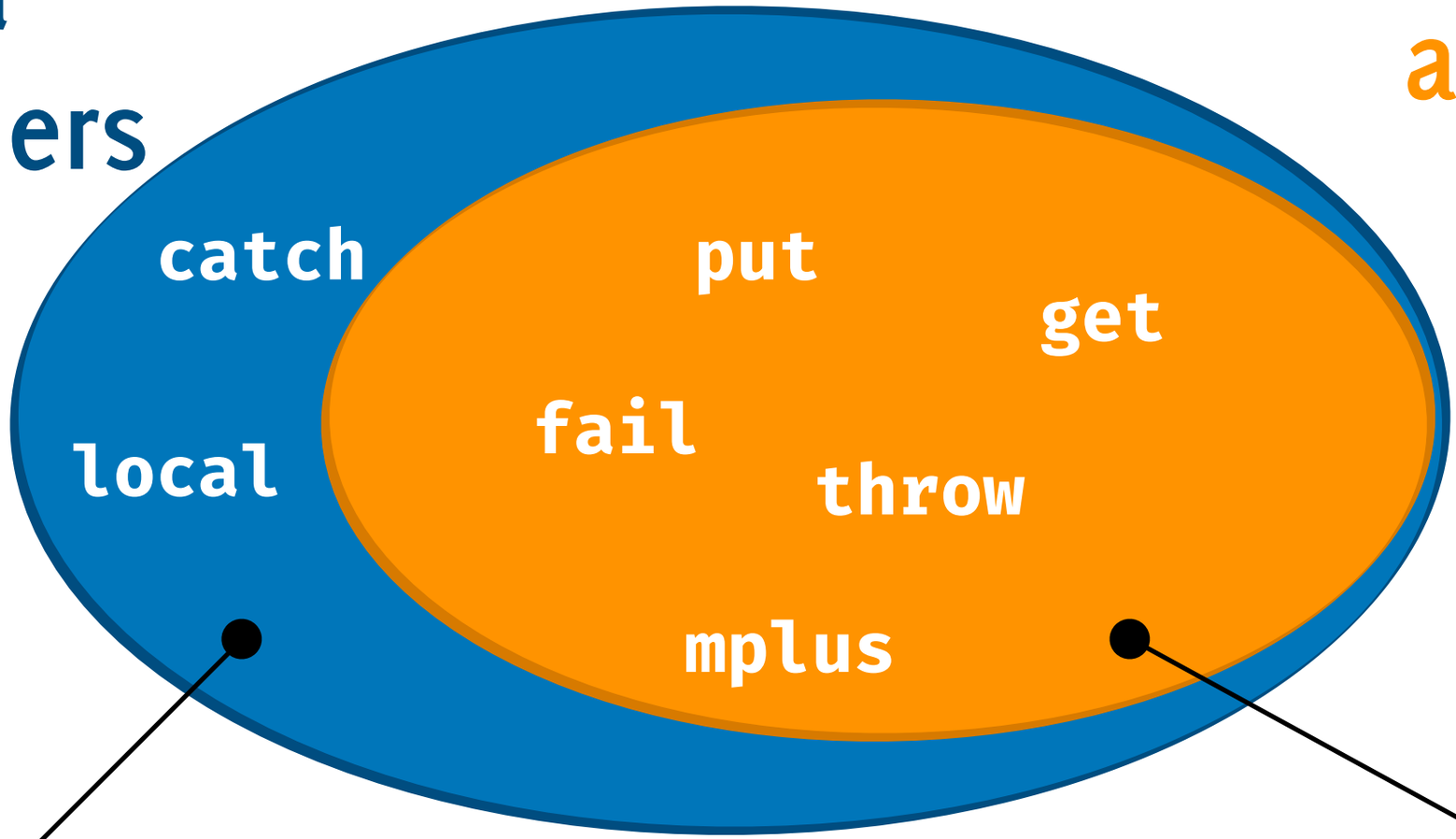
Algebraic Effects & Handlers

Modular Handlers

Expressivity Comparison

monad transformers

modular algebraic effects



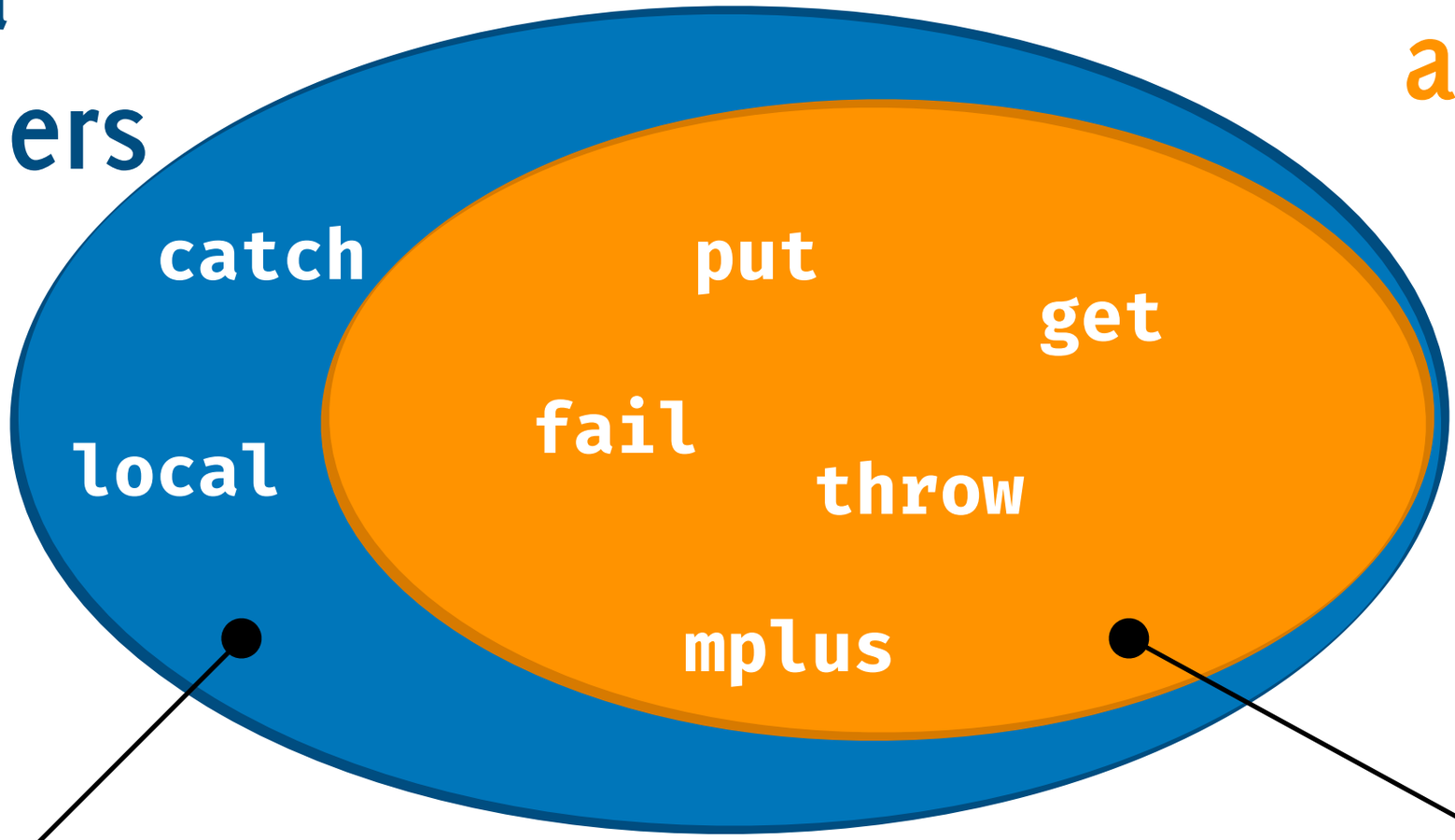
non-algebraic operations

algebraic operations

Expressivity Comparison

monad transformers

modular algebraic effects



non-algebraic operations

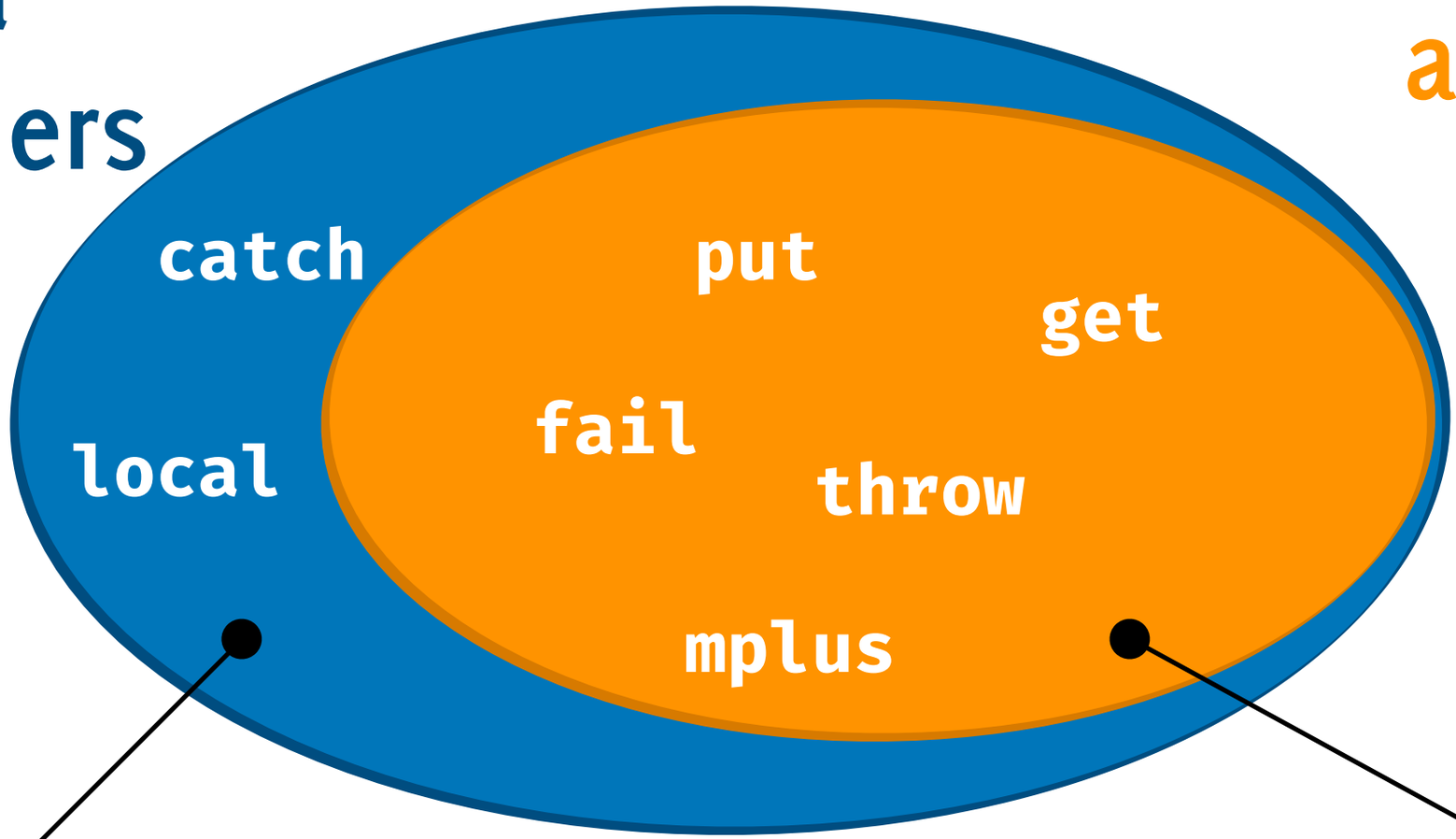
algebraic operations

free infrastructure

Expressivity Comparison

monad transformers

modular algebraic effects



non-algebraic operations

ad-hoc lifting instances

algebraic operations

free infrastructure

More in the Paper



Monad Transformers and Modular Algebraic Effects

What Binds Them Together

Tom Schrijvers
KU Leuven
Belgium

Maciej Piróg
University of Wrocław
Poland

Nicolas Wu
Imperial College London
United Kingdom

Mauro Jaskelioff
CIFASIS-CONICET
Universidad Nacional de Rosario
Argentina

Abstract

For over two decades, monad transformers have been the main *modular* approach for expressing purely functional side-effects in Haskell. Yet, in recent years algebraic effects have emerged as an alternative whose popularity is growing.

While the two approaches have been well-studied, there is still confusion about their relative merits and expressiveness, especially when it comes to their comparative modularity. This paper clarifies the connection between the two approaches—some of which is folklore—and spells out consequences that we believe should be better known.

We characterise a class of algebraic effects that is modular, and show how these correspond to a specific class of monad transformers. In particular, we show that our modular algebraic effects gives rise to monad transformers. Moreover, every monad transformer for algebraic operations gives rise to a modular effect handler.

CCS Concepts • **Software and its engineering → Functional languages**; *Control structures*; *Coroutines*; • **Theory of computation** → *Categorical semantics.*

Keywords Handlers, Effects, Monads, Transformers

ACM Reference Format:
Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3331545.3342595

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '19, August 22–23, 2019, Berlin, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342595

1 Introduction

For decades monads [29, 42] have dominated the scene of pure functional programming with effects, and the recent popularisation of algebraic effects & handlers [3, 7, 21, 23, 35] promises to change the landscape. However, with rapid change also comes confusion, and practitioners have been left uncertain about the advantages and limitations of the two competing approaches. This paper aims at clarifying the essential differences and similarities between them.

Working with combinations of multiple different effects demands a *modular* approach, where each effect is given semantics separately. This allows for the construction of complex custom effects from off-the-shelf building blocks.

A popular approach to achieving modularity for monads is with monad transformers [27]. Monad transformers extend an arbitrary monad with a new effect while at the same time ensuring that original effects are available. The desired combination of monads is achieved by stacking several monad transformers in a particular order.

In the algebraic effects approach modularity is conceptually achieved in two stages. First, the syntax of all operations involved in the effect are defined. Then a program is incrementally interpreted by several handlers, which in turn give the syntax of different effects a semantics.

Since each handler only knows about the part of the syntax of the effect it is handling, a modular approach to algebraic effects must provide a way of leaving unknown syntax uninterpreted and to be dealt with later by other handlers.

There are several properties that can be studied when comparing approaches, such as expressivity, ease of use, modularity, boilerplate automation, and efficiency. This paper focuses only on the essential *expressivity* of transformers and algebraic effects, leaving other important properties out of scope. To study expressivity, we formulate a minimal implementation of both approaches and abstract over all other aspects. In this way, we aim to explain the main ideas in an approachable manner and provide general insights that can be applied to everyone's favourite Haskell library. Much of what we present is folklore among experts, but the tale has yet to be collected in a single, consistent place for the wider Haskell community to enjoy, as we have done here.

After an introduction to monad transformers (Section 2) that fixes notation, the contributions of this paper are:

★ Detailed Explanation

★ More Examples

★ Laws & Proofs

★ Related Work

★ `callCC` Case Study

Vielen Dank!

Composing Handlers

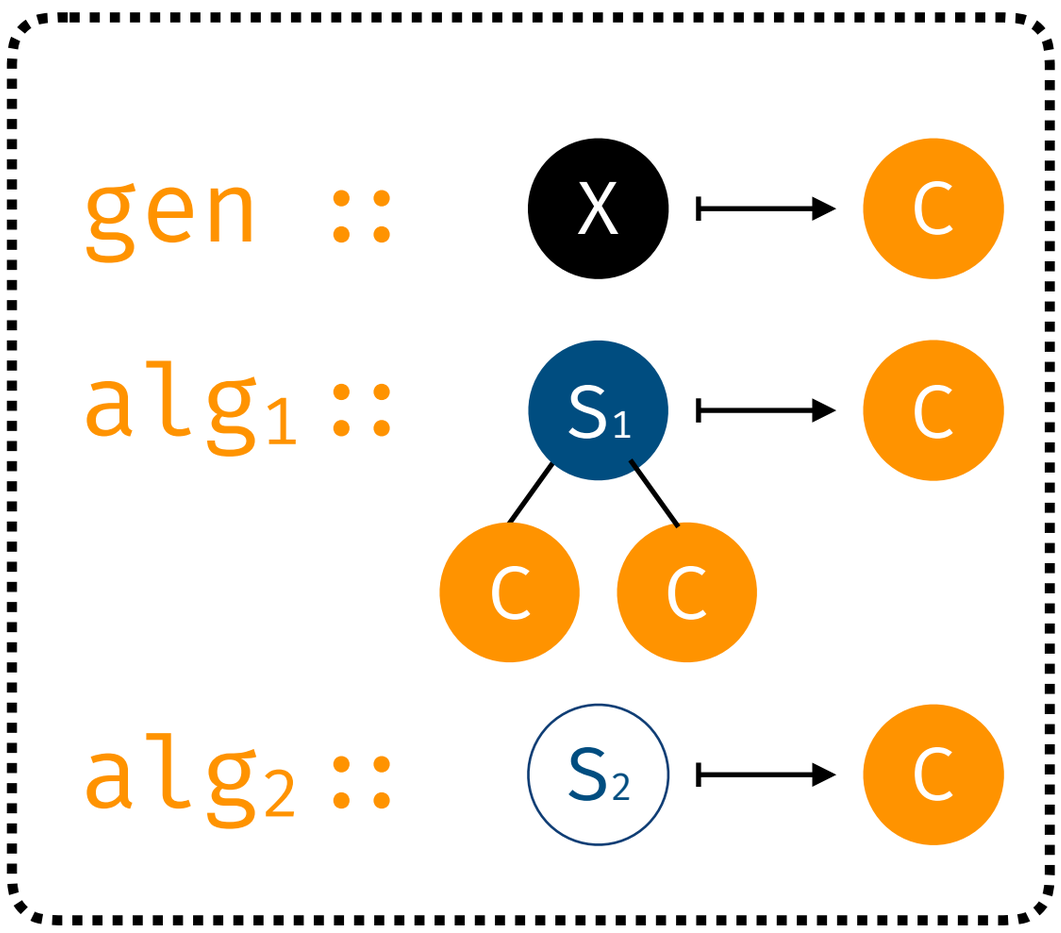
Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

Composing Handlers

Simultaneous Handling

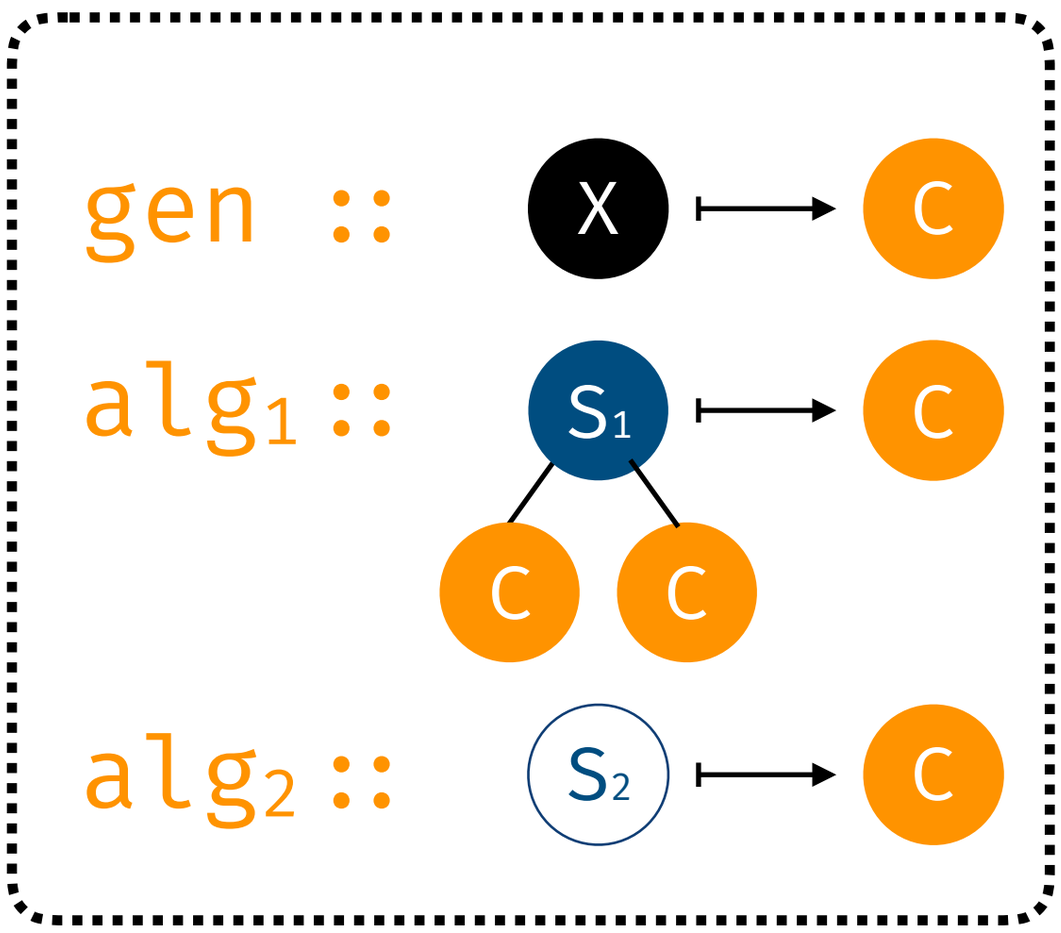
```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

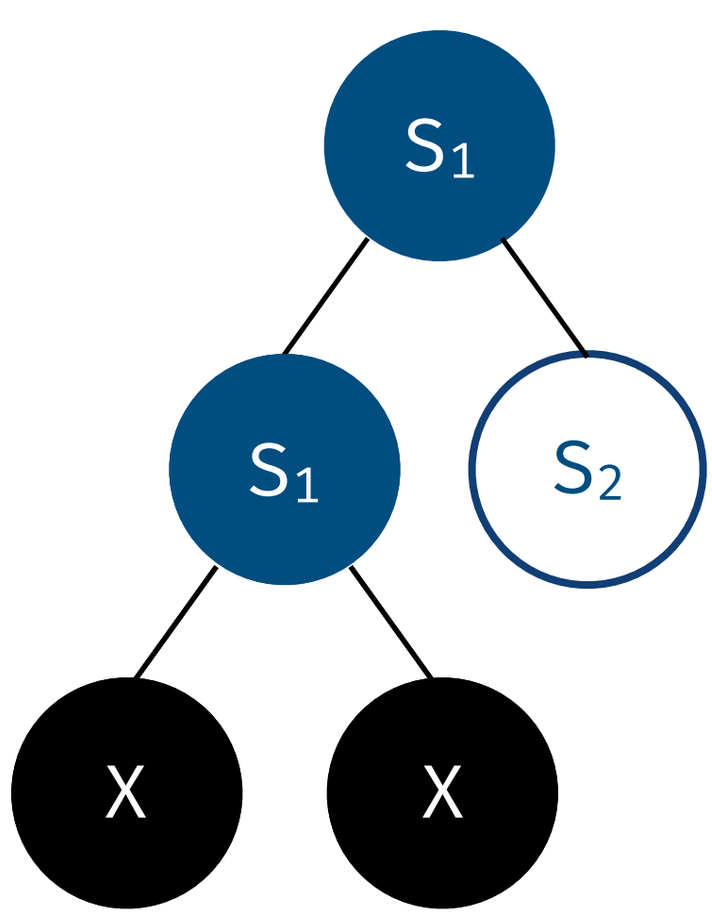


Composing Handlers

Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

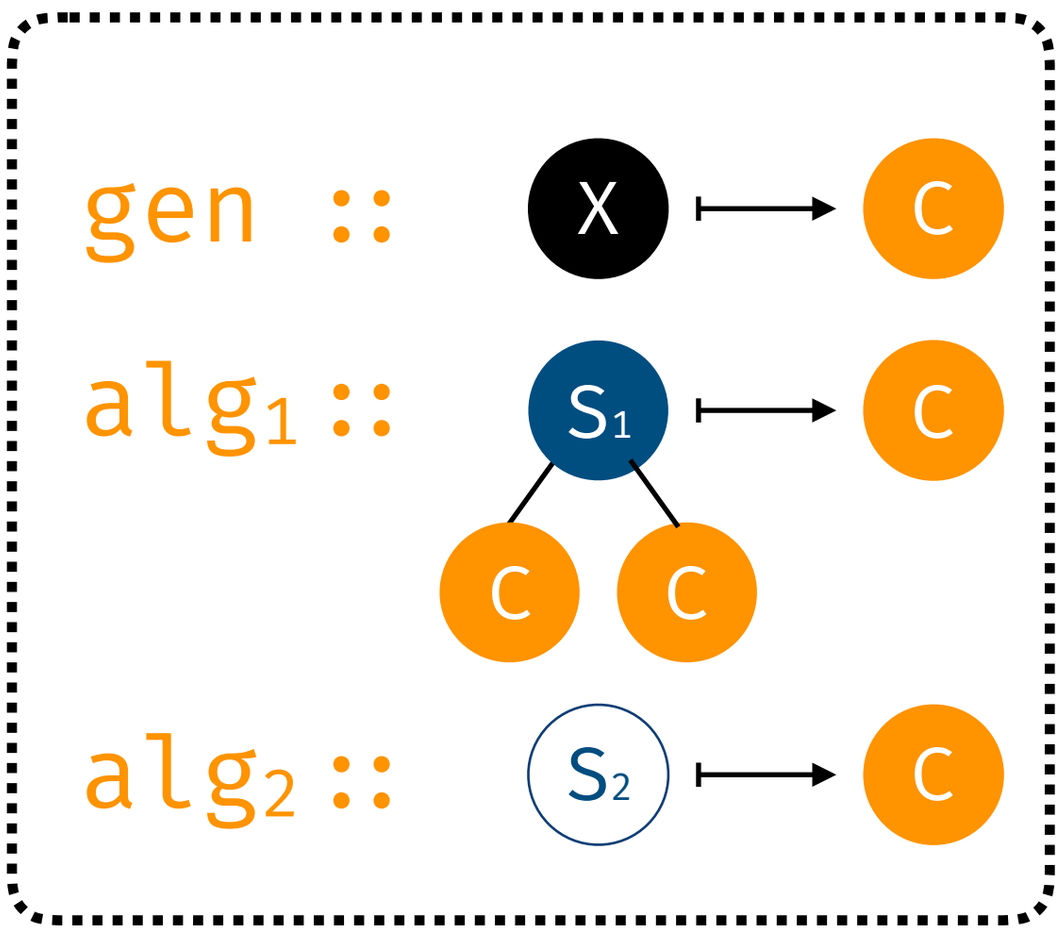


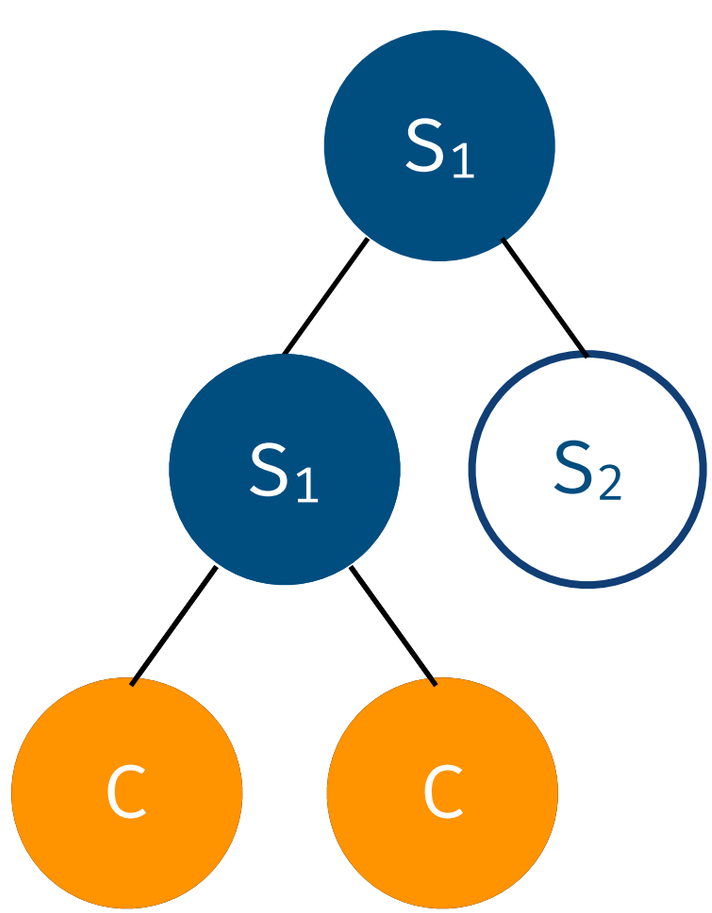


Composing Handlers

Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

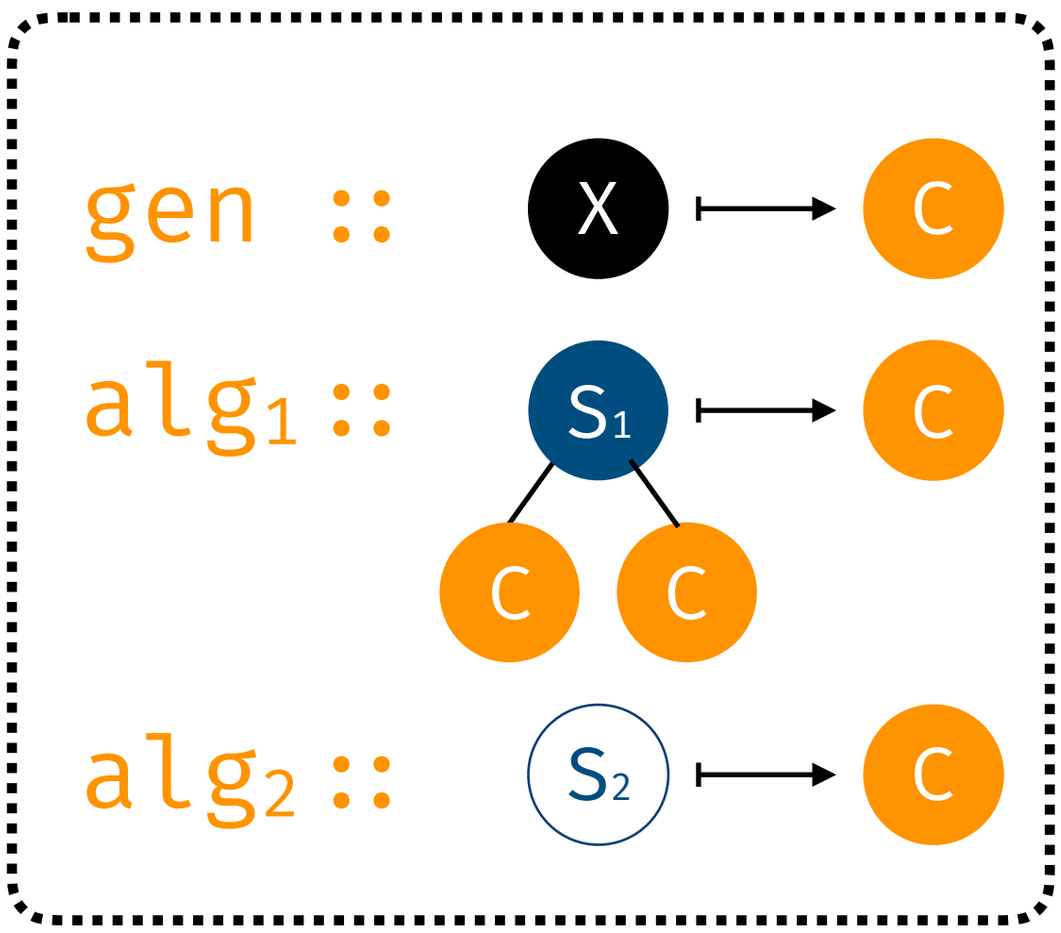


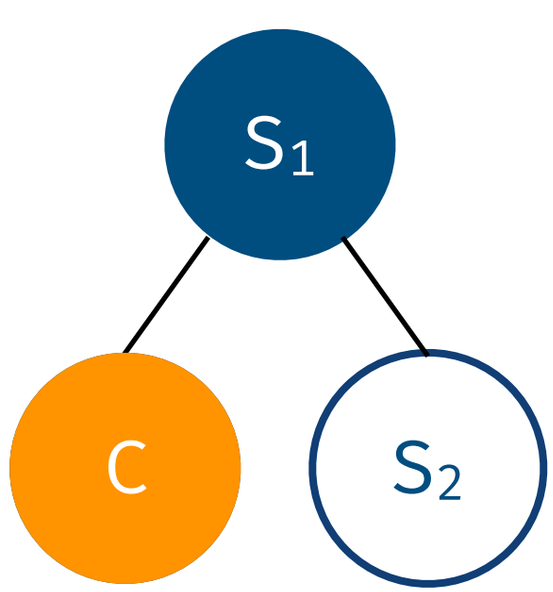


Composing Handlers

Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

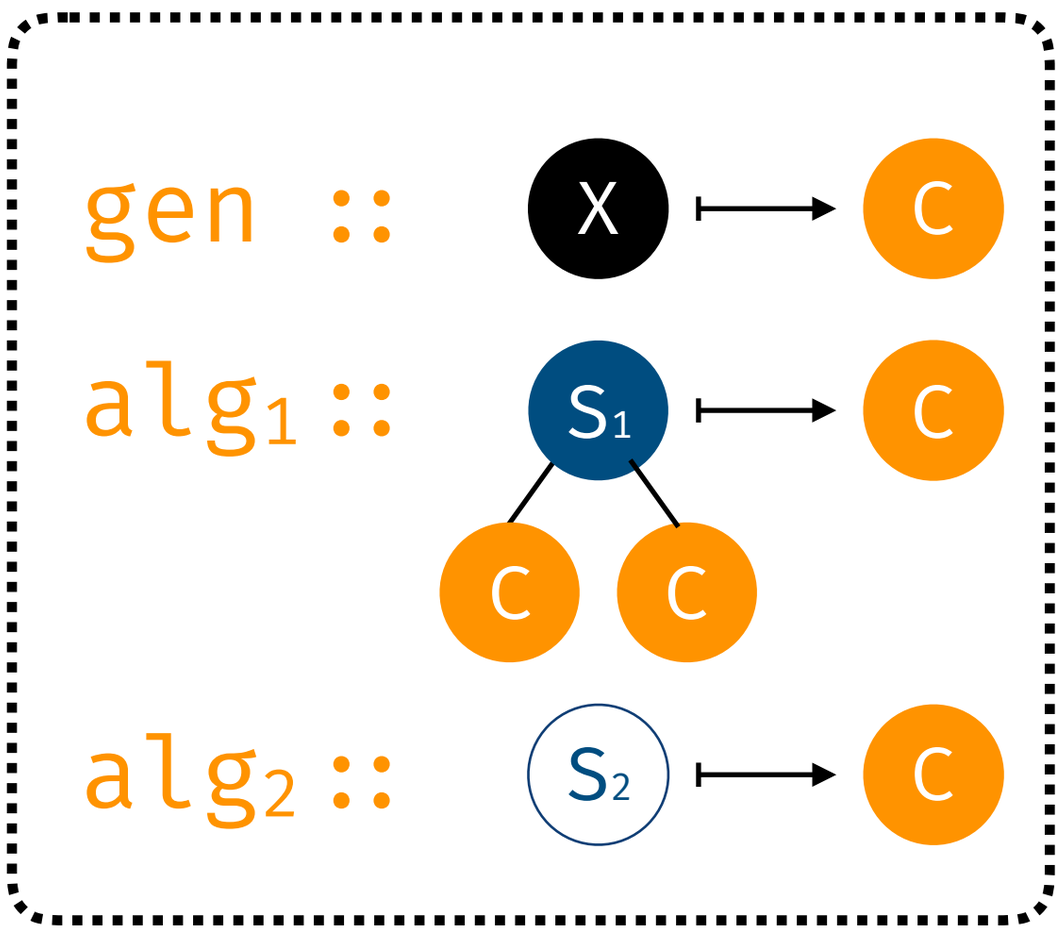


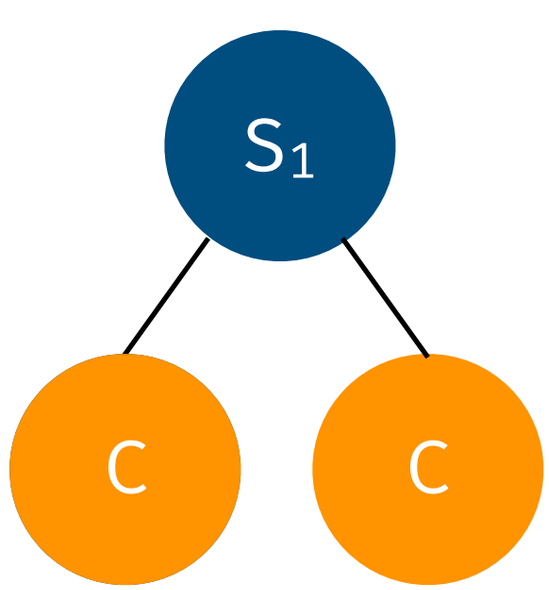


Composing Handlers

Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```

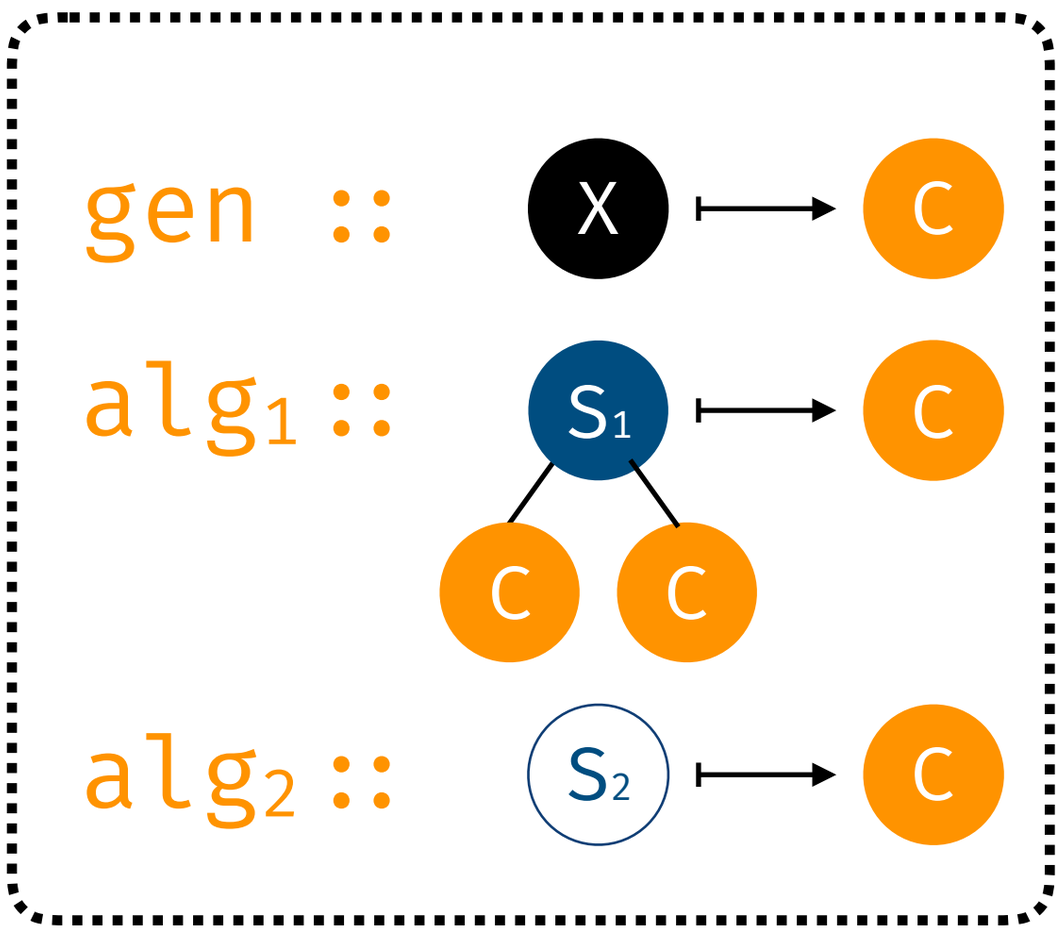




Composing Handlers

Simultaneous Handling

```
h :: Free (S₁+S₂) X → C
h = fold gen (alg₁∇alg₂)
```





Composing Handlers

Simultaneous Handling

$$(C, gen, alg_1)$$



$$(C, gen, alg_2)$$

$$=$$

$$(C, gen, alg_1 \nabla alg_2)$$

```
(∇)::(s1 c → c) → (s2 c → c) → ((s1 + s2) c → c)
(alg1 ∇ alg2) (Inl op) = alg1 op
(alg1 ∇ alg2) (Inr op) = alg2 op
```

Composing Handlers

Simultaneous Handling

$(\text{🌙}, \mathtt{gen}, \mathtt{alg}_1)$



$\mathbin{\circ\!\!\!\circ}$

$(\text{🌙}, \mathtt{gen}, \mathtt{alg}_2)$

$=$

$(\text{🌙}, \mathtt{gen}, \mathtt{alg}_1 \nabla \mathtt{alg}_2)$

"Once in a blue moon!"

```
(∇)::(s1 c → c) → (s2 c → c) → ((s1 + s2) c → c)
(alg1 ∇ alg2) (Inl op) = alg1 op
(alg1 ∇ alg2) (Inr op) = alg2 op
```