# Effect handlers in scope, evidently

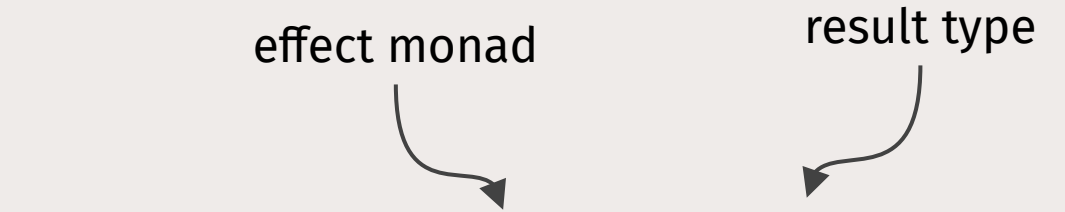
Scoped effects, evidence passing, and everything in between

Xiaoyan Ren @ HOPE ‘22
xy.r@outlook.com
https://github.com/re-xyr

“Sp”

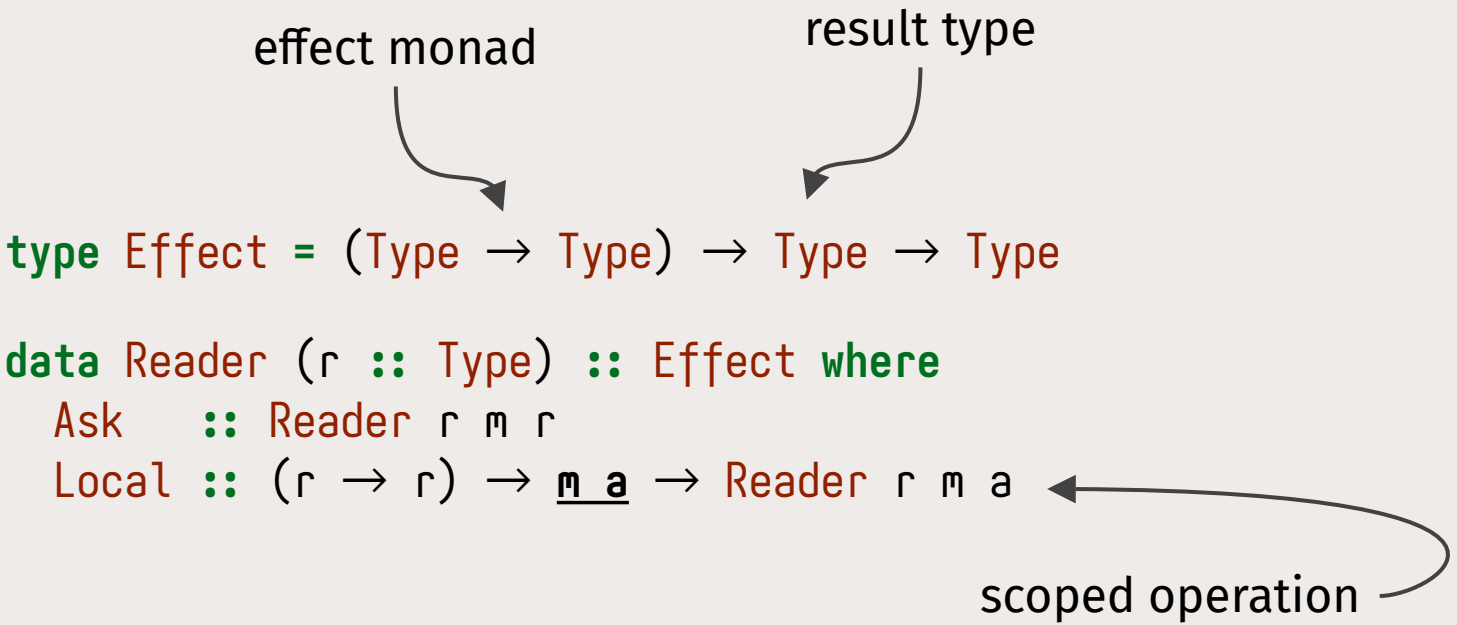
References will appear here

[0] Author name ‘22




```
type Effect = (Type → Type) → Type → Type

data Reader (r :: Type) :: Effect where
  Ask   :: Reader r m r
  Local :: (r → r) → m a → Reader r m a
```


effect monad
result type
type Effect = (Type → Type) → Type → Type
data Reader (r :: Type) :: Effect where
Ask :: Reader r m r
Local :: (r → r) → m a → Reader r m a
scoped operation

[3] Wu et al ‘14

```
try :: MonadError e m ⇒ m a → m (Either e a)

try = handle { Throw e → abort (Left e)
             ; pure x  → pure (Right x) }
```

```
withFile  :: MonadFile m ⇒ FilePath → (Handle → m a) → m a

readFile  :: MonadFile m ⇒ Handle → m String

writeFile :: MonadFile m ⇒ Handle → String → m ()
```

[4] King ‘21

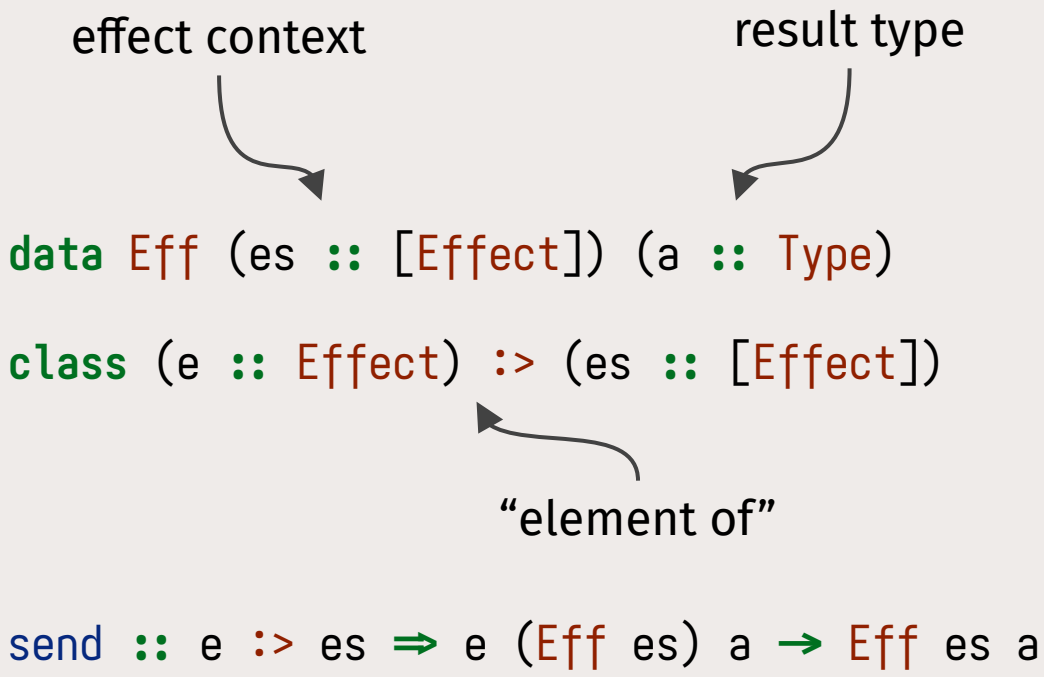
```
withFileCloud :: MonadFile m ⇒ FilePath → (Handle → m a) → m a
                                ≠ (because observable side effects)
withFileRAM :: MonadFile m ⇒ FilePath → (Handle → m a) → m a
                                ≠
withFileFS :: MonadFile m ⇒ FilePath → (Handle → m a) → m a

withFile   :: MonadFile m ⇒ FilePath → (Handle → m a) → m a

readFile  :: MonadFile m ⇒ Handle → m String

writeFile :: MonadFile m ⇒ Handle → String → m ()
```

[4] King ‘21


effect context
result type
data Eff (es :: [Effect]) (a :: Type)
class (e :: Effect) :> (es :: [Effect])
“element of”
send :: e :> es => e (Eff es) a -> Eff es a

*provide* implementation for an effect

```
interpret   :: Handler e es a → Eff (e : es) a → Eff es a

reinterpret :: Handler e (e' : es) a → Eff (e : es) a → Eff (e' : es) a
```

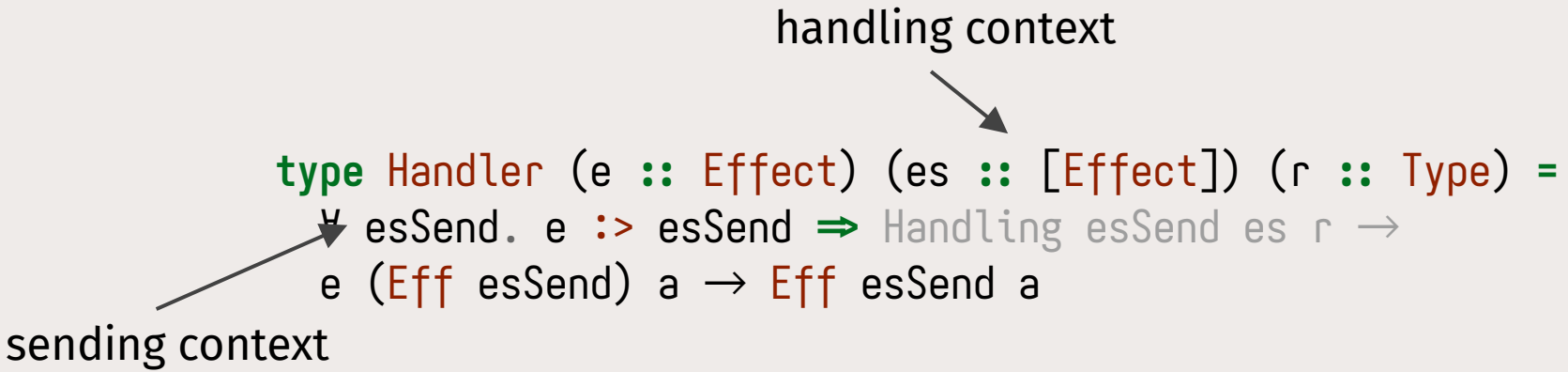
*change* implementation for an effect

```
interpose   :: e :> es ⇒ Handler e es a → Eff es a → Eff es a

reinterpose :: e :> es ⇒ Handler e (e' : es) a → Eff es a → Eff (e' : es) a
```

[11] King et al '16

```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```


handling context
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
∀ esSend. e :> esSend ⇒ Handling esSend es r →
e (Eff esSend) a → Eff esSend a
sending context

handling context

```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

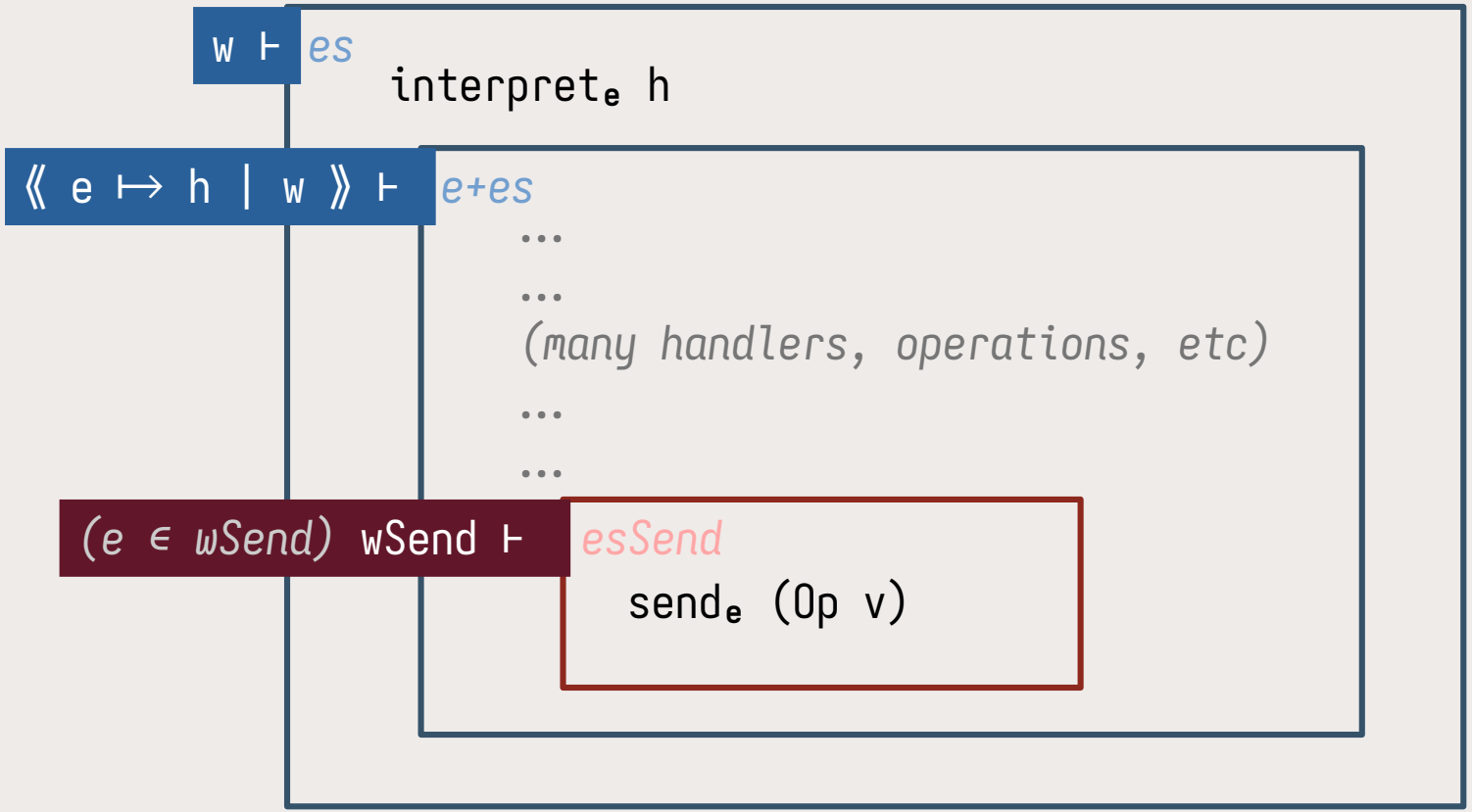
`interpose` can be used to
change the handler of the current effect
**locally** for a computation

handling context

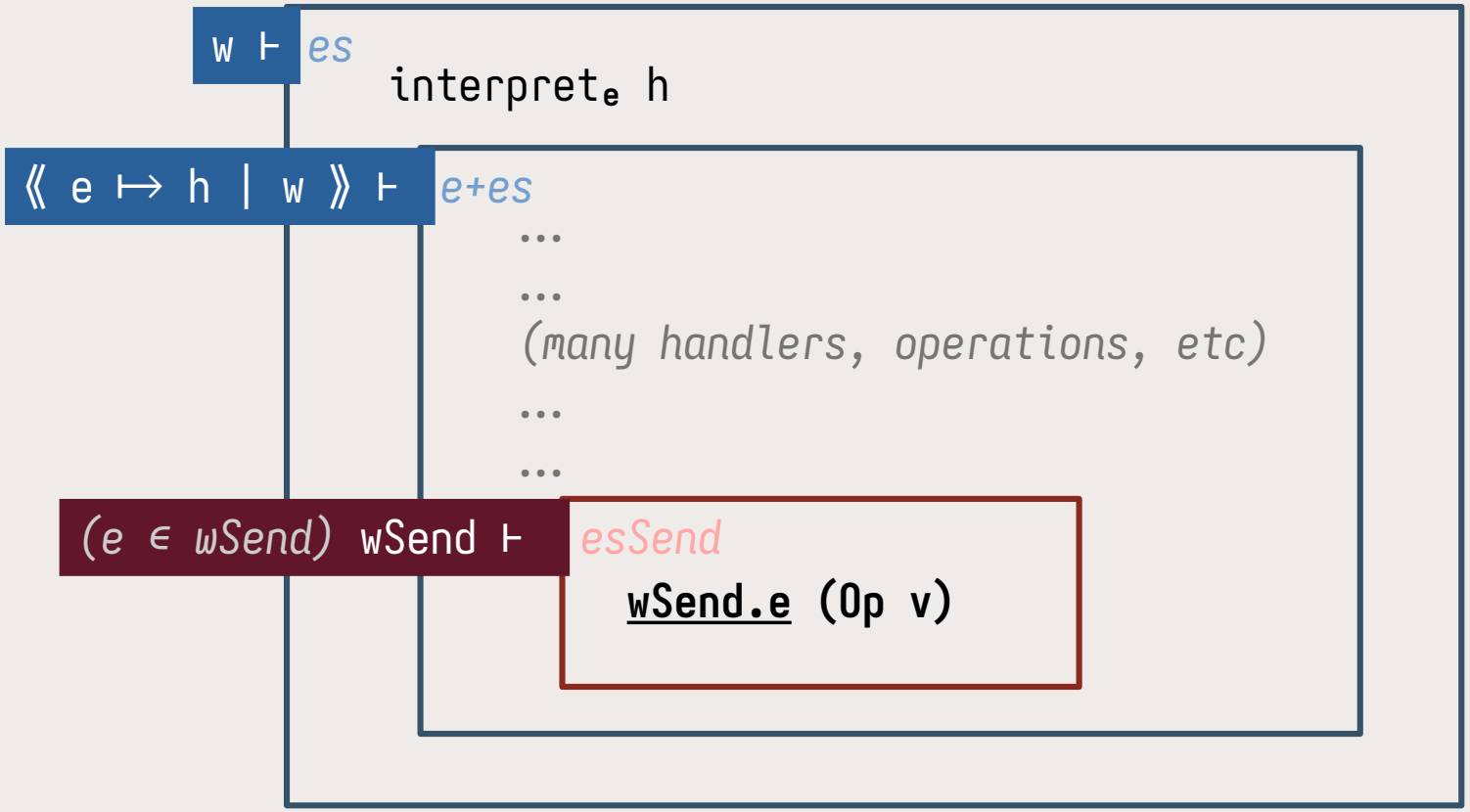
```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

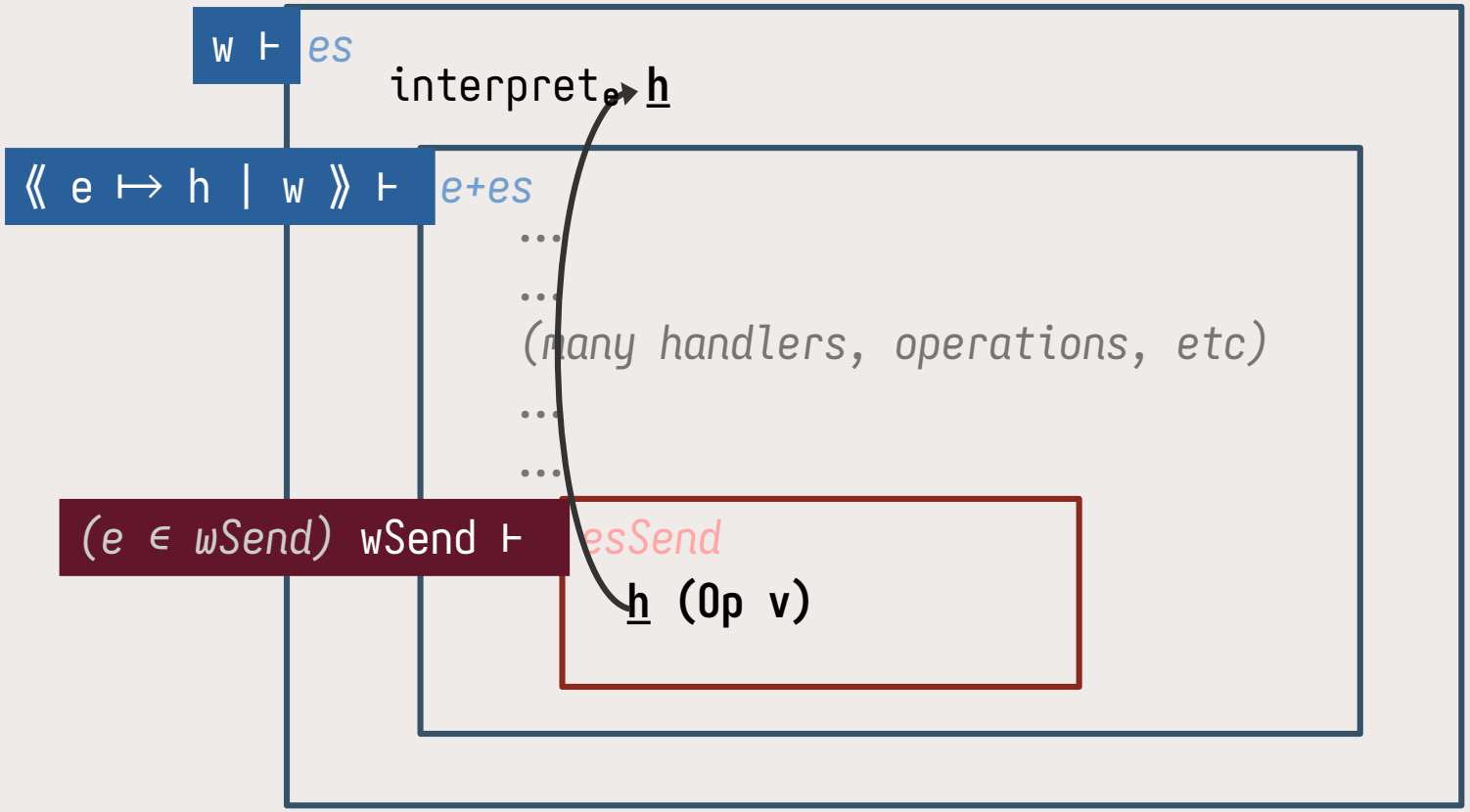
a `Handler` is a transformation
from operations to computations
in the **sending** context


w ⊢ es
interpret_e h
⟪ e ↦ h | w ⟫ ⊢ e+es
...
...
(many handlers, operations, etc)
...
...
(e ∈ wSend) wSend ⊢ esSend
send_e (Op v)

[2] Xie et al '20

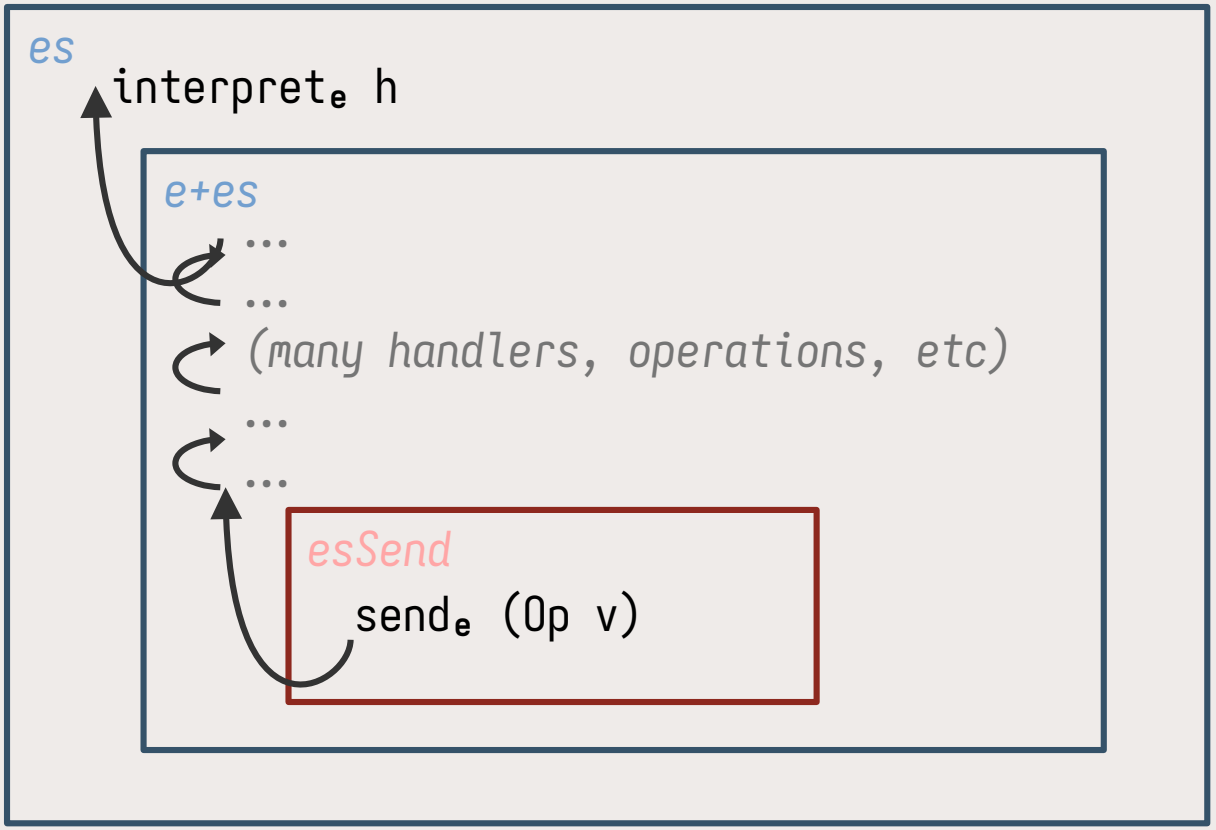

w ⊢ es
interpret_e h
⟪ e ↦ h | w ⟫ ⊢ e+es
...
...
(many handlers, operations, etc)
...
...
(e ∈ wSend) wSend ⊢ esSend
wSend.e (Op v)

[2] Xie et al '20




Running esSend computations in handlers is trivial in evidence-passing!

[2] Xie et al '20


es
interpret$_e$ h
e+es
...
...
(many handlers, operations, etc)
...
...
esSend
send$_e$ (Op v)

[1] Leijen ‘15

handling context

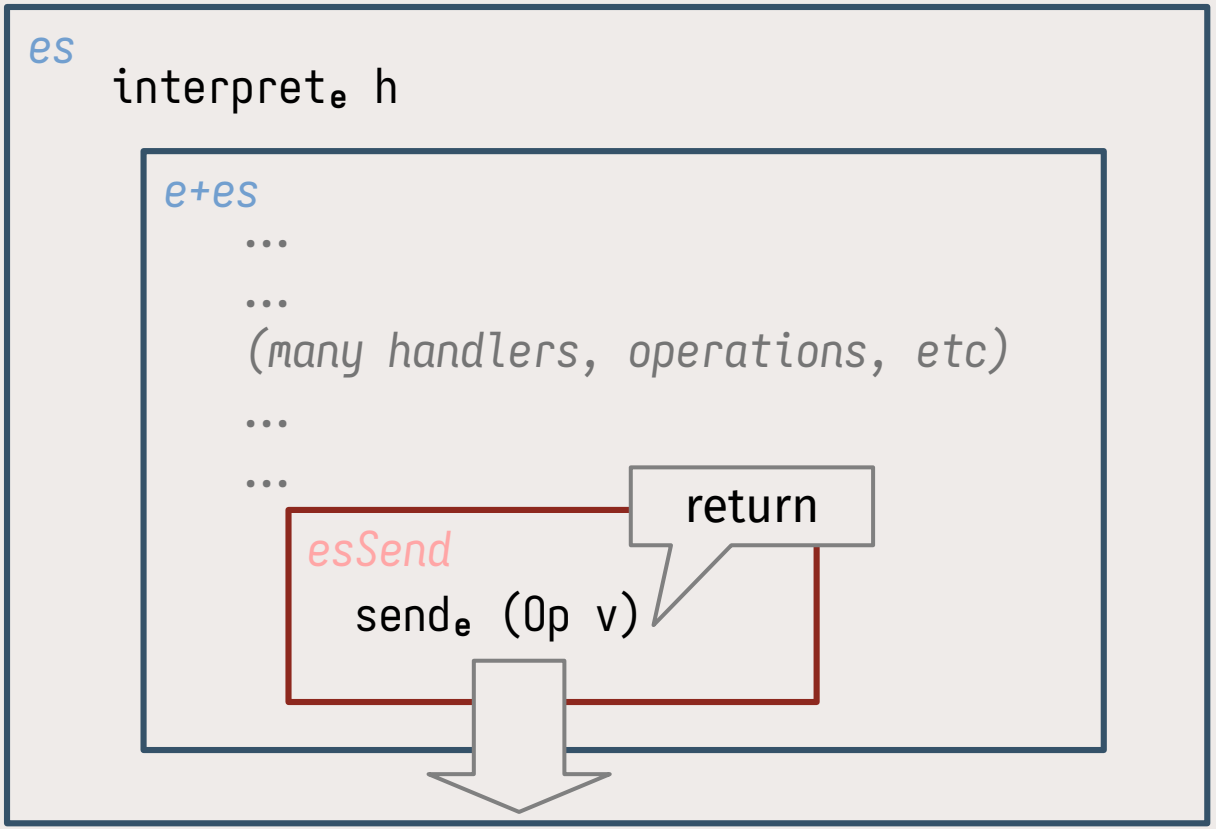
```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

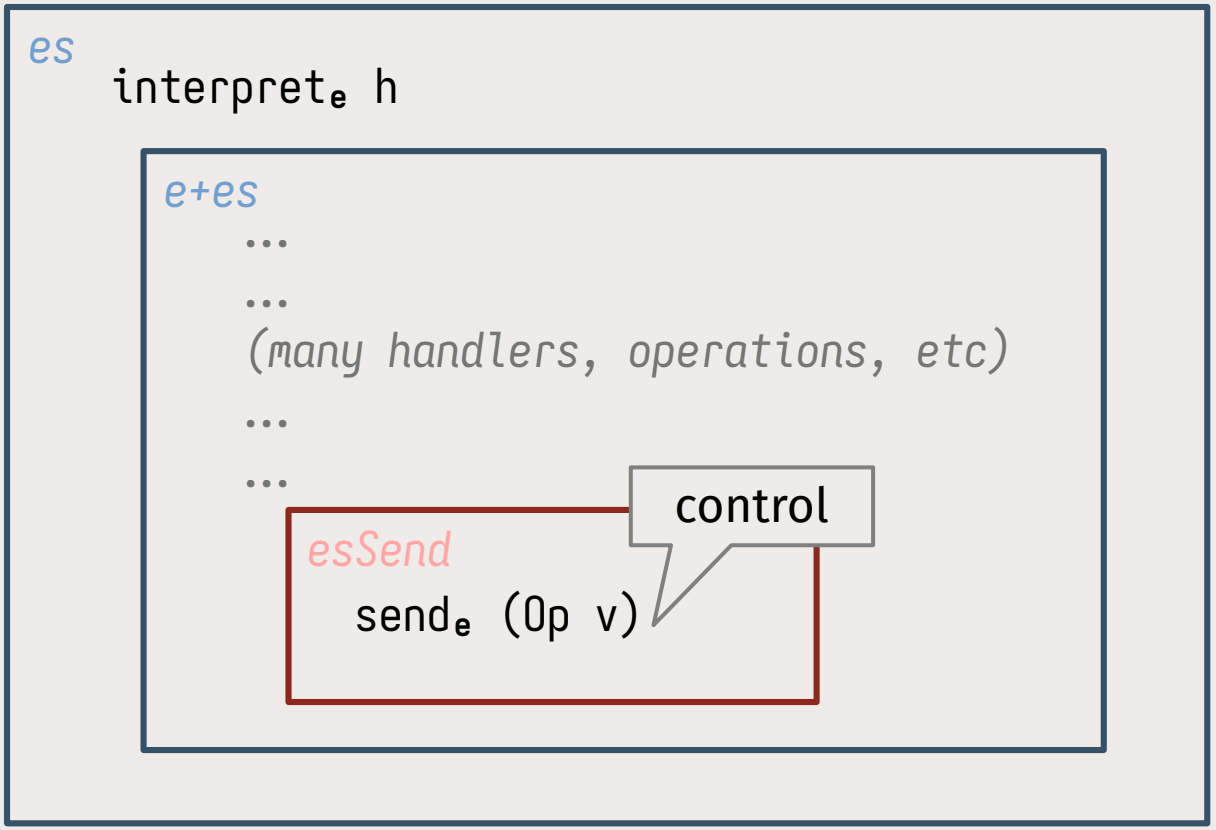
Q: How do we manipulate control flow?

```
control :: Handling esSend es r →
           ((a → Eff es r) → Eff es r) → Eff esSend a
```

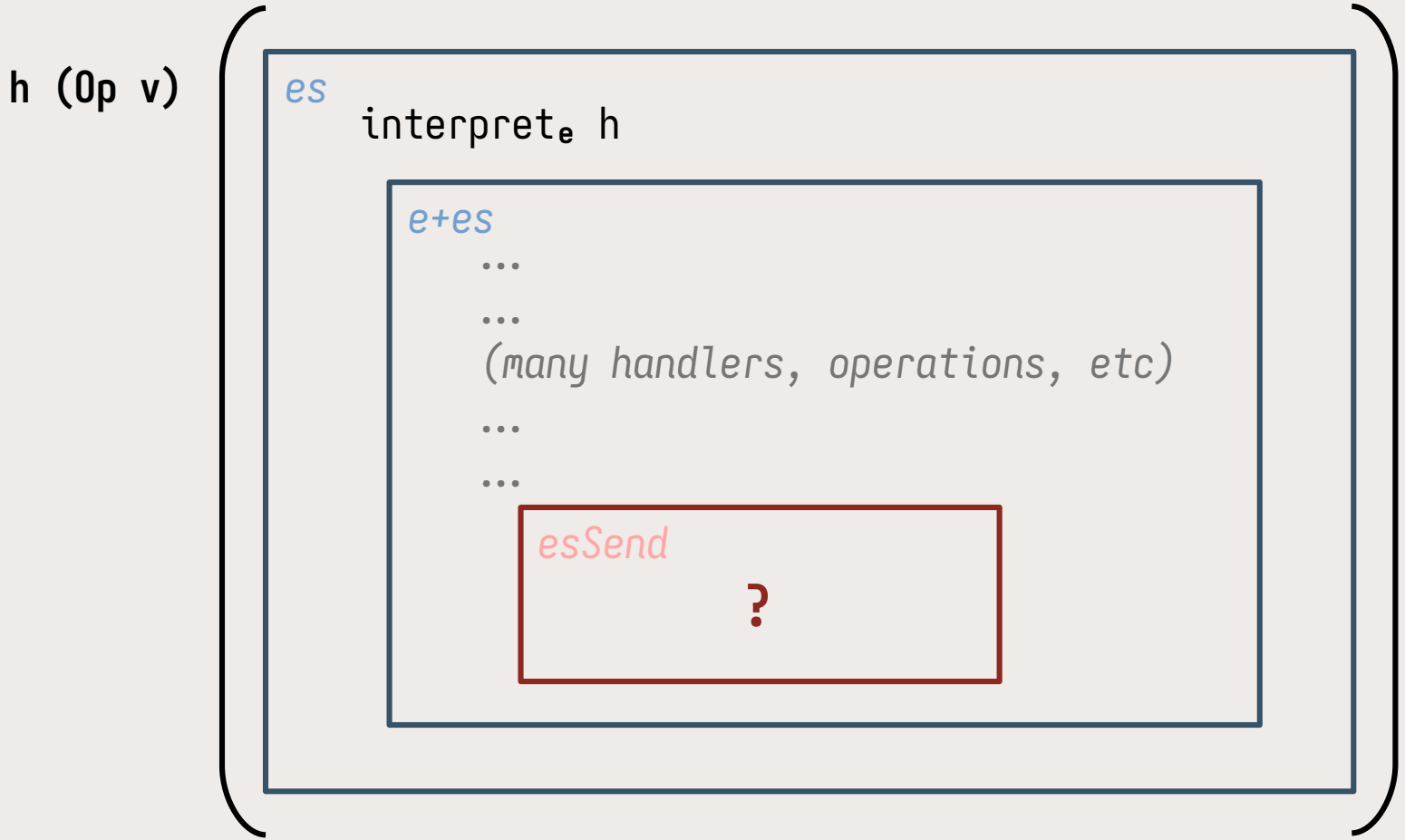
continuation


es
interpret$_e$ h
e+es
...
...
(many handlers, operations, etc)
...
...
esSend
send$_e$ (Op v)
return

[2] Xie et al ‘20


es
interpret$_e$ h
e+es
...
...
(many handlers, operations, etc)
...
...
control
esSend
send$_e$ (Op v)

[2] Xie et al '20


h (Op v)
es
interpret$_e$ h
e+es
...
...
(many handlers, operations, etc)
...
...
esSend
?

[2] Xie et al '20

handling context

```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

Q: How do we manipulate control flow?

```
control :: Handling esSend es r →
           ((a → Eff es r) → Eff es r) → Eff esSend a
```

**A: Handlers can yield, but only when they need to**

handling context

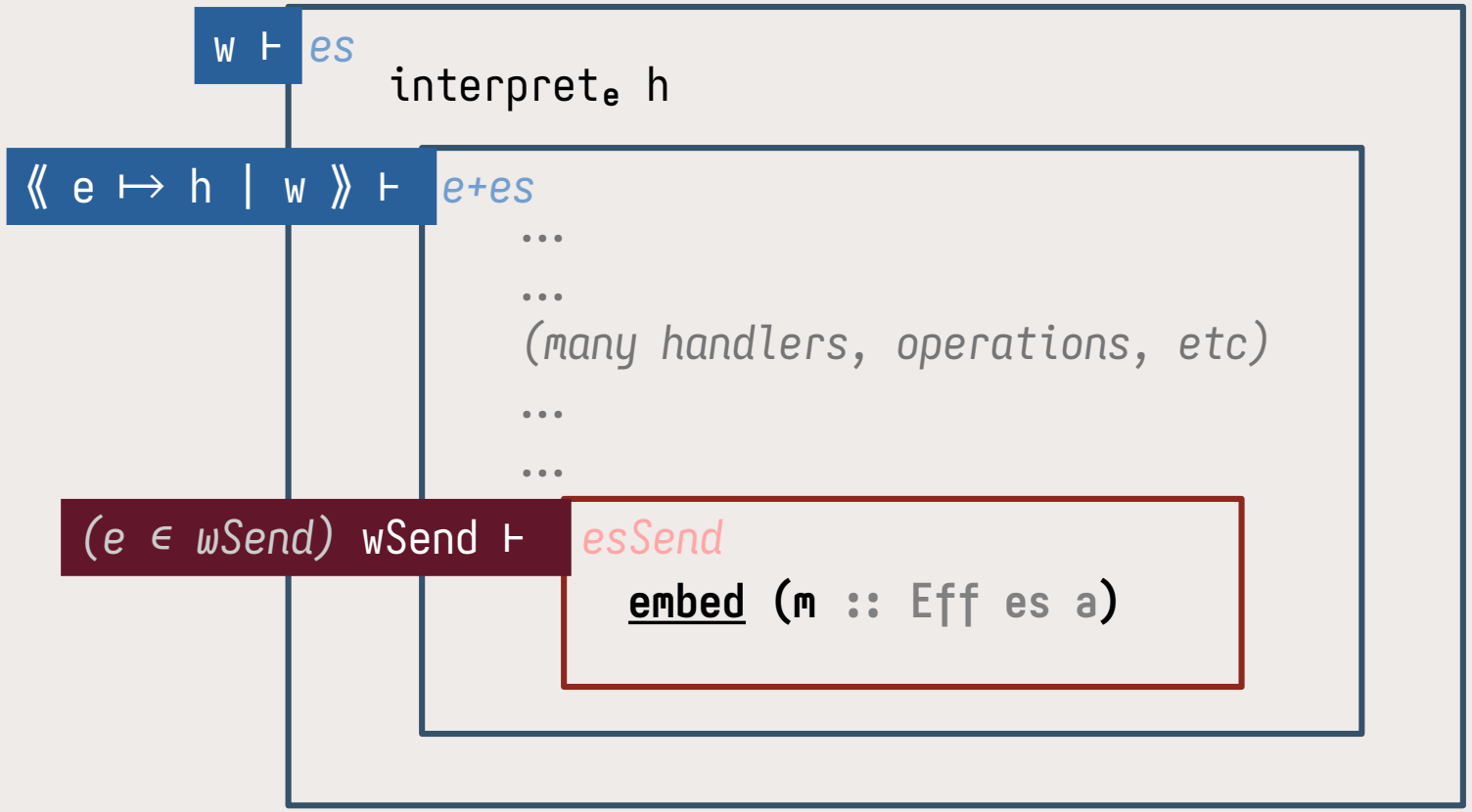
```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

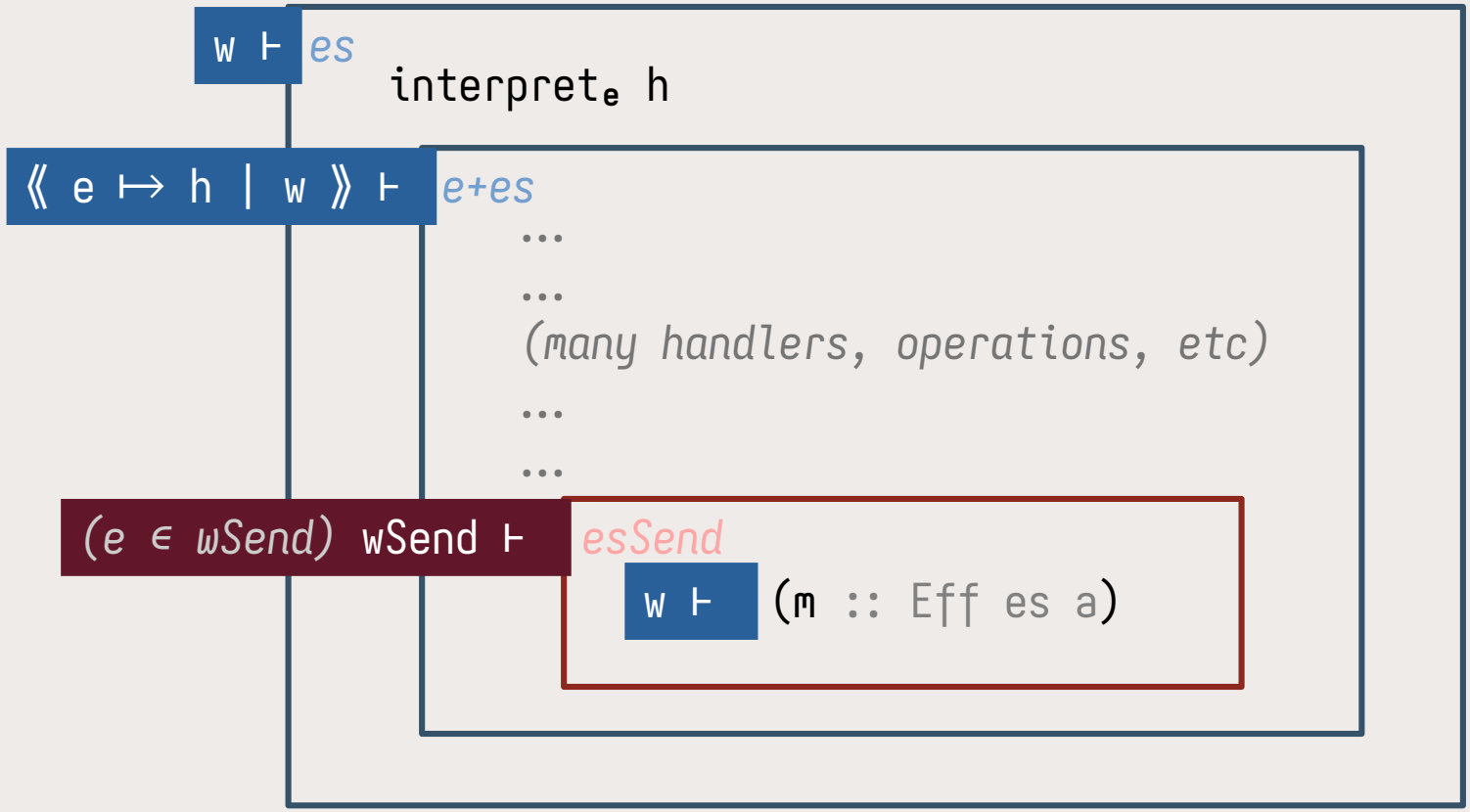
Q: How do we call operations in the *handling* context?

```
embed :: Handling esSend es r →
         Eff es a → Eff esSend a
```

# What does *embed* do actually?



*embed* = pass in the evidence vector from handle site



handling context

```
type Handler (e :: Effect) (es :: [Effect]) (r :: Type) =
  ∀ esSend. e :> esSend ⇒ Handling esSend es r →
  e (Eff esSend) a → Eff esSend a
```

sending context

Q: How do we call operations in the *handling* context?

```
embed :: Handling esSend es r →
         Eff es a → Eff esSend a
```

**A: With a primitive *embed* operation**

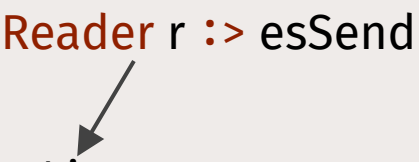
# Safety rules of embedding computations

- Inner contexts can safely embed computations in the outer context
- Outer context **cannot** embed computations in the inner context
  - `interprets` are skipped, so `controls` can stuck
- `es` = outer, `esSend` = inner, so we're fine embedding `es` in `esSend`

```haskell
data IOE :: Effect where
  LiftIO :: IO a -> IOE es a
```

```haskell
data Reader (r :: Type) :: Effect where
  Ask   :: Reader r m r
  Local :: (r → r) → m a → Reader r m a
```

```haskell
handleReader :: r -> Handler (Reader r) es a
handleReader r = \ctx -> \case
  Ask -> pure r
  Local f action -> interpose (handleReader $ f r) action
```




```haskell
runReader :: r -> Eff (Reader r : es) a -> Eff es a
runReader r action = interpret (handleReader r) action
```

# Compared to EvEff...

- Sp uses the same delimited continuation implementation, which has efficient tail-resumptive computations
- Sp adds support for embedding IO actions
- Sp uses an array instead of list for the evidence vector
- Sp adds support for scoped effects

[5] Xie et al ‘20

# Benchmarking

- **Sp:** our implementation based on `eveff`, with support of scoped effects
- **Ev:** the `eveff` library by Xie et al
- **Freer:** `freer-simple`, an effect library based on freer monads
- **Mtl:** `mtl`, the classic monad transformers library
- **Fused:** `fused-effects`, a library based on monad transformers supporting scoped effects
- **Sem:** `polysemy`, an library based on freer monads supporting scoped effects

[5] Xie et al ‘20
[11] King et al ‘16
[12] Rix et al ‘19
[13] Maguire et al ‘19

# Benchmarking

```haskell
-- Recursively decrement an Int state till 0
countdown :: Member (State Int) es => Eff es Int
countdown = do
  x <- get
  if x == 0
    then pure x
    else do
      put (x - 1)
      countdown
```

[6] Charles et al ‘16

# Benchmarking *what?*

```haskell
-- Recursively decrement an Int state till 0
countdown :: Member (State Int) es => Eff es Int
countdown = do
  x <- get
  if x == 0
    then pure x
    else do
      put (x - 1)
      countdown
```

[6] Charles et al ‘16

# Benchmarking effect invocation

```haskell
-- Recursively decrement an Int state till 0
countdown :: Member (State Int) es => Eff es Int
countdown = do
  x <- get
  if x == 0
    then pure x
    else do
      put (x - 1)
      countdown
```

[6] Charles et al ‘16

# Benchmarking normal control flow

```haskell
-- Recursively decrement an Int state till 0
countdown :: Member (State Int) es => Eff es Int
countdown = do
  x <- get
  if x == 0
    then pure x
    else do
      put (x - 1)
      countdown
```

[6] Charles et al ‘16

# Benchmark results

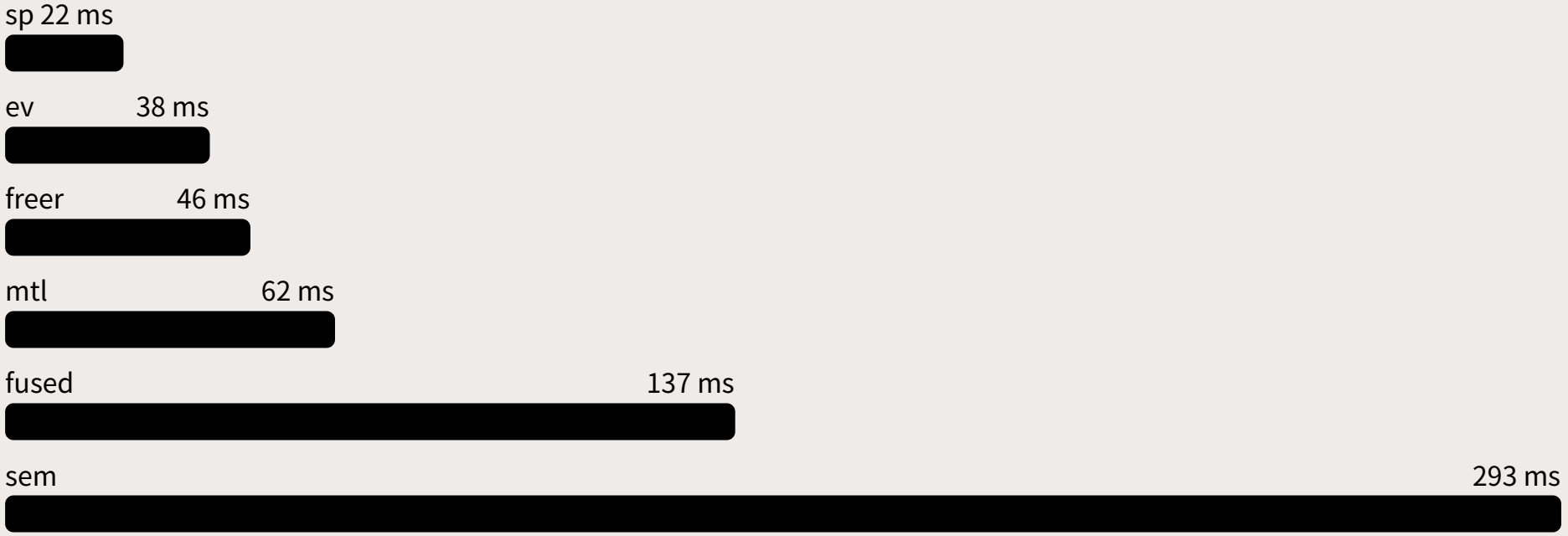
# *Misleading* benchmark results?


[7] King ‘20

# Change 1: {-# NOINLINE #-}


[7] King ‘20

# Change 1: {-# NOINLINE #-}




($n = 10^6$)

Transformer-based effect libraries heavily relies on **GHC optimization**

[7] King ‘20

# Change 2: Dummy effects

`[Reader () ×5, State Int, Reader () ×5]`


[8] Rybczak et al ‘21

# Change 2: Dummy effects

`[Reader () ×5, State Int, Reader () ×5]`


($n = 10^6$)

Transformer-based effect libraries relies on **a shallow effect context**

[8] Rybczak et al ‘21

# Benchmarking *yield*-intensive code

```haskell
-- Find all 3-tuples of Ints that are side lengths of a right triangle
pyth :: Member NonDet es => Int -> Eff es (Int, Int, Int)
pyth upbound = do
  x <- choice [1..upbound]
  y <- choice [1..upbound]
  z <- choice [1..upbound]
  if x*x + y*y == z*z
    then return (x,y,z)
    else empty
```

[5] Xie et al '20

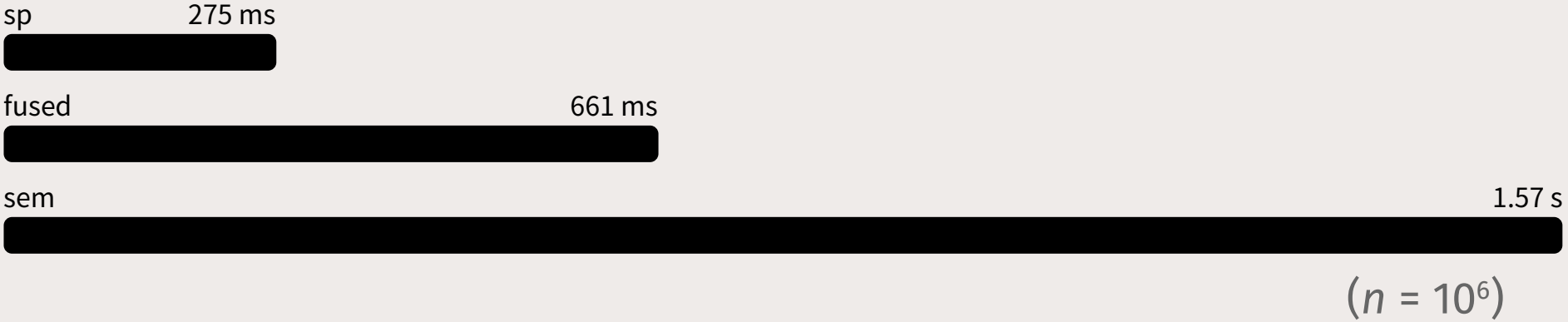
# Benchmarking *yield*-intensive code


($n$ = 128, algorithm O($n^3$))

# Benchmarking scoped effects

```haskell
-- Nest n local environment values
nesting :: Member (Reader Int) es => Int -> Eff es Int
nesting n = case n of
  0 -> ask
  n -> local (+1) (nesting (n - 1))
```

# Benchmarking scoped effects



# The performance of Sp

- Sp has good performance in *effect invocation*, *normal control flow*, yield-*intensive code*, and *scoped effects*
- The performance of Sp is **not** dependent on fragile compiler optimizations, nor a small effect context
- **Effect invocation** in Sp is even faster than Ev
  - …because Sp used an array instead of list to represent the evidence vector, making access O(1)
- **Normal control flow** in Sp is faster than other popular implementation
  - …because of the efficient evidence-passing model
- ***yield*-intensive code** in Sp is slightly slower than Ev
  - …because Sp's delimited control is modified to support IO actions
- **Scoped effects** in Sp is faster than other popular libraries supporting it
  - …again thanks to the efficiency of evidence-passing

# Limitations

- Higher-order IO functions
  - `catch :: Exception e ⇒ IO a → (e → IO a) → IO a`
- Embedding higher-order IO *loses* monadic state
- Delimited control monad *requires* monadic state
- Status quo: **Embedding higher-order IO** or **delimited control** - choose one

# Relevant & Future work

- `cleff`: Sp + Higher-order IO – delimited control
  - https://github.com/re-xyr/cleff
- Potential solution: IO-native delimited continuations in GHC, by Alexis King
  - https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7942
- A formal operational semantics of scoped effects?

[9] Ren ‘21
[10] King ‘22

# Why not…

- base your work on MpEff, where continuations need not be called in handler scope?
  - Well, I believe it's possible! I'm trying to make a prototype of that.
- make this work in *capability passing*, where handlers are individual arguments instead of stored in a vector?
  - No – I don't think we can define *embed* in that way
- Isn't it the case that you can't define the async effect either in `cleff`?
  - True, but at least you can use the `async` package on Hackage, which uses GHC green threads instead!

[14] Xie et al '21
[15] Brachthäuser et al '20

# Questions?

Sp: https://github.com/re-xyr/speff

(talk with me about effect systems! at the ICFP Discord or xy.r@outlook.com)

# References


1. Leijen, Daan (2016), *Algebraic effects for functional programming,* Microsoft Research.
2. Xie, Ningning et al (2020), *Effect handlers, evidently,* ICFP '20.
3. Wu, Nicholas et al (2014), *Effect handlers in scope,* Haskell '14.
4. King, Alexis (2021), *Unsolved problems of scoped effects, and what that means for eff,* Twitch stream recording.
5. Xie, Ningning et al (2020), *Effect handlers in Haskell, evidently,* Haskell '20.
6. Charles, Ollie et al (2019), *effect-zoo*: https://github.com/ocharles/effect-zoo.
7. King, Alexis (2020), *Effects for less,* ZuriHac '20.
8. Rybczak, Andrzej et al (2021), *effectful:* https://github.com/haskell-effectful/effectful.
9. Ren, Xiaoyan (2021), *cleff:* https://github.com/re-xyr/cleff.
10. King, Alexis (2022), *Native, first-class, delimited continuations:* https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7942.
11. King, Alexis et al (2016), *freer-simple:* https://github.com/lexi-lambda/freer-simple.
12. Rix, Rob et al (2019), *fused-effects:* https://github.com/fused-effects/fused-effects.
13. Maguire, Sandy et al (2019), *polysemy:* https://github.com/polysemy-research/polysemy.
14. Xie, Ningning et al (2021), *Generalized evidence passing for effect handlers,* ICFP '21.
15. Brachthäuser, Jonathan Immanuel et al (2020), *Effects as capabilities: effect handlers and lightweight effect polymorphism*, OOPSLA '20.