Effect Handlers in Haskell, Evidently

Ningning Xie Daan Leijen

Haskell Symposium 2020

https://github.com/xnning/EvEff

Computational Effects

monad transformers

[Liang et al. 1995]

all you need is lifting

any type error can be resolved
by adding more lifting

Computational Effects

monad transformers
[Liang et al. 1995]

all you need is lifting
any type error can be resolved
by adding more lifting

algebraic effects
[Plotkin and Power 2003;
Plotkin and Pretnar 2013]

composable
modular

Computational Effects

monad transformers

[Liang et al. 1995]

all you need is lifting

any type error can be resolved
by adding more lifting

an alternative to

[Kammar et al. 2013;
Kiselyov et al. 2013;
Kiselyov and Ishii 2015;
Wu and Schrijvers 2015]

algebraic effects

[Plotkin and Power 2003;
Plotkin and Pretnar 2013]

composable
modular

Computational Effects

monad transformers
[Liang et al. 1995]

all you need is lifting
any type error can be resolved
by adding more lifting

an alternative to

[Kammar et al. 2013;
Kiselyov et al. 2013;
Kiselyov and Ishii 2015;
Wu and Schrijvers 2015]

[This paper]

algebraic effects
[Plotkin and Power 2003;
Plotkin and Pretnar 2013]

composable
modular

Contribution

ICFP 2020

Effect Handlers, Evidently

NINGNING XIE, Microsoft Research, USA
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany
DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom
PHILIPP SCHUSTER, University of Tübingen, Germany
DAAN LEIJEN, Microsoft Research, USA

polymorphic algebraic effects

evidence-passing translation

polymorphic evidence calculus

monadic multi-prompt translation

polymorphic lambda calculus

Contribution

ICFP 2020 **Effect Handlers, Evidently**

NINGNING XIE, Microsoft Research, USA
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany
DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom
PHILIPP SCHUSTER, University of Tübingen, Germany
DAAN LEIJEN, Microsoft Research, USA

Haskell 2020 a Haskell library `Control.Ev.Eff` of effect handlers via evidence-passing

polymorphic algebraic effects | evidence-passing translation | polymorphic evidence calculus | monadic multi-prompt translation | polymorphic lambda calculus

Overview

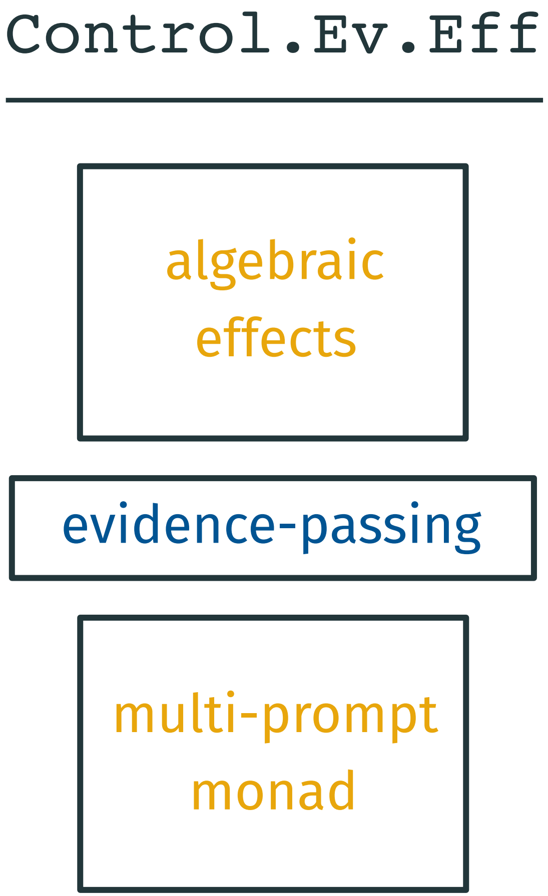
`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

Overview



- The library interface
  concise and simple.
- Implementations
  1. layered implementations
  2. optimizations: tail-resumptive operations are evaluated in-place
  3. the restriction: scoped resumptions
- Benchmarks

The library Interface

Library Interface

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect
operation

Library Interface

```
data Reader a e ans                                 effect
  = Reader{ ask :: Op () a e ans }                  operation

greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect

operation

```
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect monad

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }

greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect

operation

effect constraint

effect monad

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }

greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)

reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

effect

operation

effect constraint

effect monad

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect

operation

effect constraint

```
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect monad

```
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

implementation

Library Interface

```
data Reader a e ans                                  effect
  = Reader{ ask :: Op () a e ans }                   operation

                                      effect constraint
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()                              effect monad
           return ("hello " ++ s)

reader :: Eff (Reader String :* e) ans → Eff e ans
reader action                                                    implementation
  = handler (Reader{ ask = value "world" }) action
```

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect

operation

effect constraint

```
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect monad

handles

```
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

implementation

Library Interface

```haskell
data Reader a e ans                                  -- effect
  = Reader{ ask :: Op () a e ans }                   -- operation
```

effect constraint

```haskell
greet :: (Reader String :? e) ⇒ Eff e String          -- effect monad
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

handles

```haskell
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action                                         -- implementation
  = handler (Reader{ ask = value "world" }) action
```

Library Interface

```haskell
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect
operation

effect constraint

```haskell
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect monad

handles

```haskell
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

implementation

```haskell
helloWorld :: Eff e String
helloWorld = reader greet
```

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }
```

effect

operation

effect constraint

```
greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)
```

effect monad

handles

```
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

implementation

```
Reader String :? (Reader String :* e)
```

```
helloWorld :: Eff e String
helloWorld = reader greet
```

Library Interface

```
data Reader a e ans                                  effect
  = Reader{ ask :: Op () a e ans }                   operation

                                          effect constraint
greet :: (Reader String :? e) ⇒ Eff e String         effect monad
greet = do s <- perform ask ()
           return ("hello " ++ s)

                                       handles
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action                                        implementation
  = handler (Reader{ ask = value "world" }) action

Reader String :? (Reader String :* e) ✓
helloWorld :: Eff e String
helloWorld = reader greet
```

```
> runEff helloWorld
"hello world"
```

Library Interface

```
data Reader a e ans
  = Reader{ ask :: Op () a e ans }


greet :: (Reader String :? e) ⇒ Eff e String
greet = do s <- perform ask ()
           return ("hello " ++ s)


reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action


helloWorld :: Eff e String
helloWorld = reader greet
```

```
> runEff helloWorld
"hello world”
```

Operations

```
data Op a b e ans
```

Operations

```
data Op a b e ans

  value       :: a → Op () a e ans
     Reader{ ask = value "world" }
```

Operations

```
data Op a b e ans

  value      :: a → Op () a e ans
     Reader{ ask = value "world" }

  function   :: (a → Eff e b) → Op a b e ans
     Reader{ ask = function (\() -> "world”) }
```

Operations

```
data Op a b e ans

  value      :: a → Op () a e ans
     Reader{ ask = value "world" }

  function   :: (a → Eff e b) → Op a b e ans

     Reader{ ask = function (\() -> "world”) }

  operation  :: (a → (b → Eff e ans) → Eff e ans) → Op a b e ans

     Reader{ ask = operation (\() k -> k "world”) }
```

Implementation

Implementation

`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

Implementation

`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

```
data Eff e a
```

Implementation

`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

```
data Eff e a
```

```
data Ctl a
```

Implementation

`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

```
newtype Eff e a
  = Eff (Context e → Ctl a)
```

```
data Ctl a
```

Implementation

`Control.Ev.Eff`

algebraic effects

evidence-passing

multi-prompt monad

```
newtype Eff e a
  = Eff (Context e → Ctl a)
```

a vector of handlers is passed down as the context

```
data Ctl a
```

Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)
```

reader

incr

exception

ask()

ask()

Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```

reader

incr

exception

ask()

ask()

Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```

reader

incr

exception

ask()

ask()

Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```

reader

incr

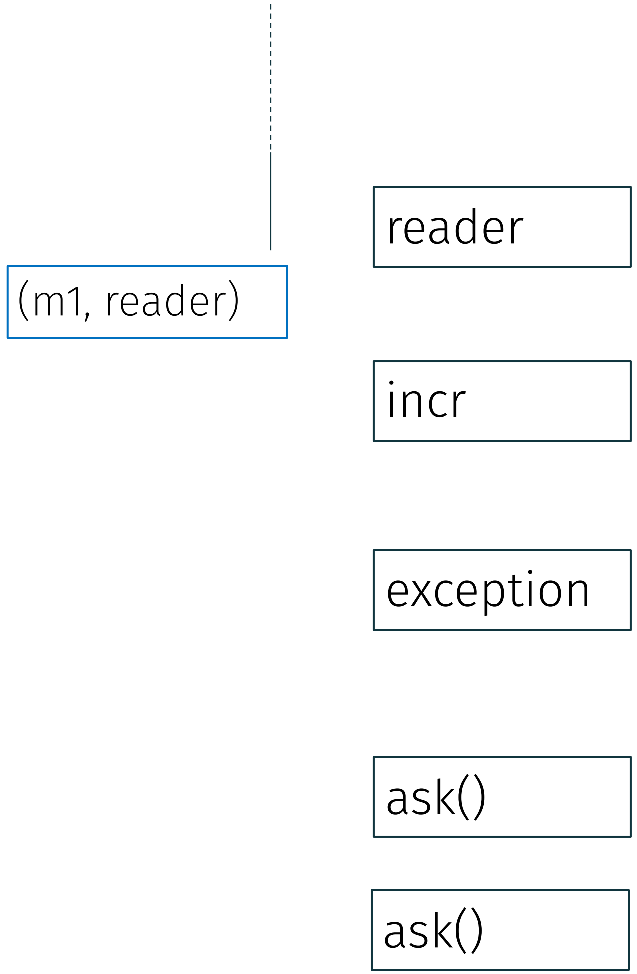
exception

ask()

ask()

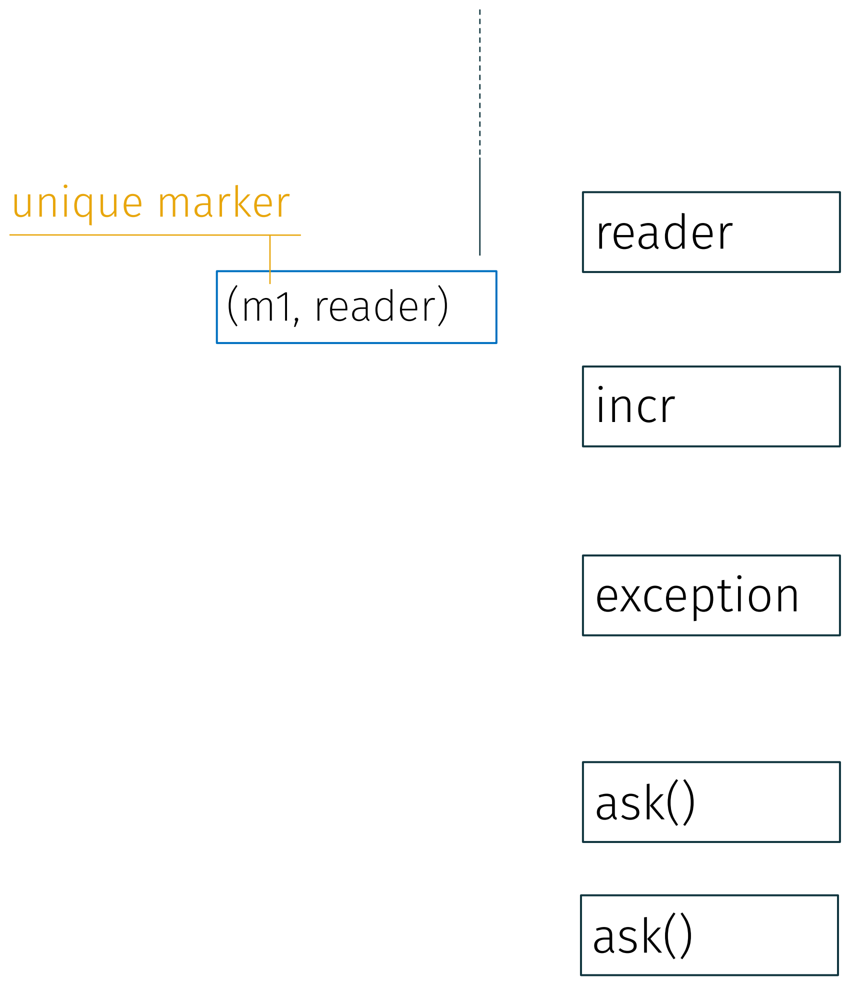
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



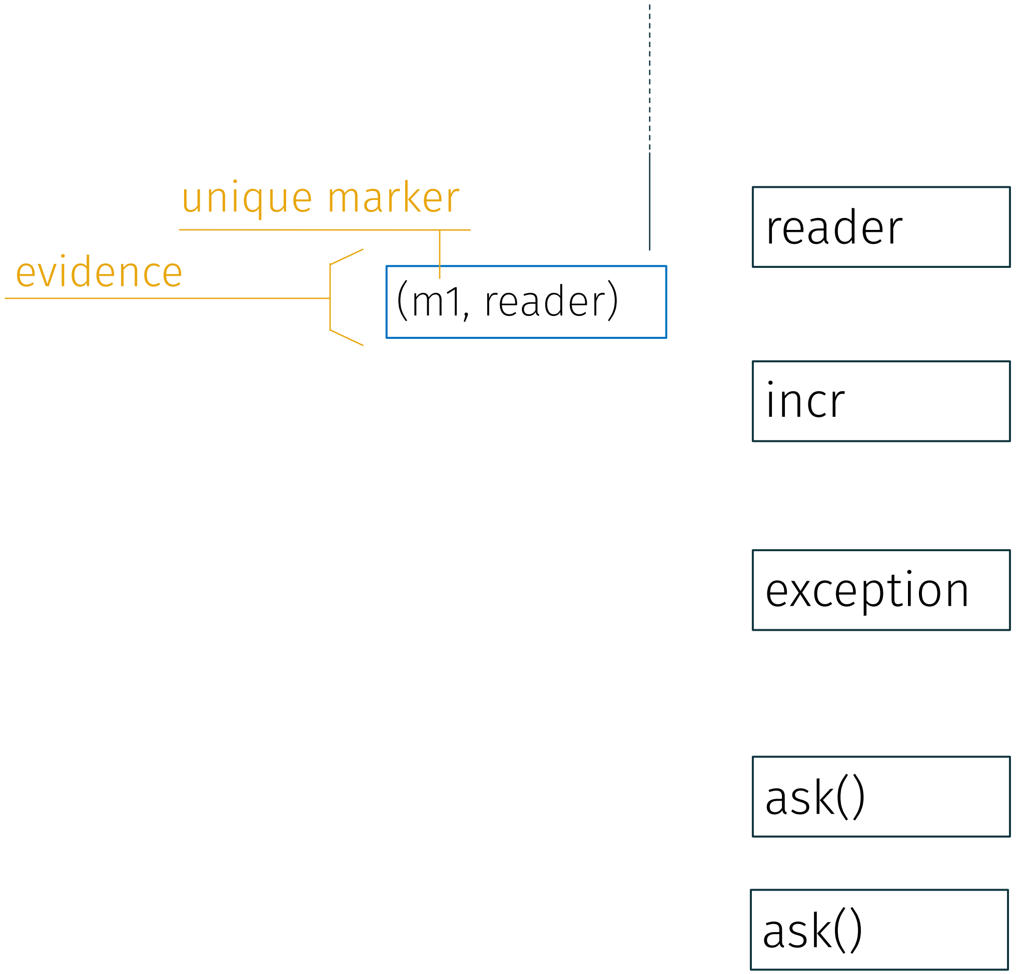
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
     operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
     operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



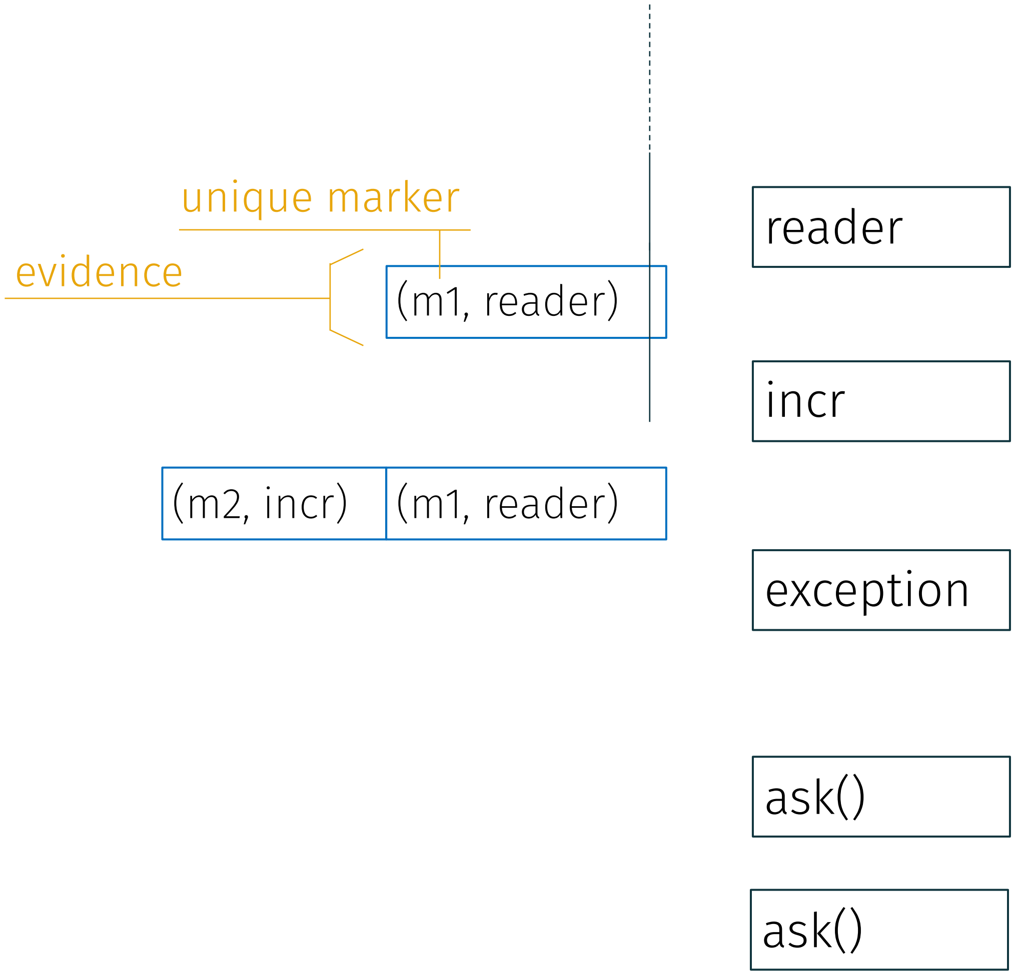
Evidence Passing

```
handler
   (Reader{ ask = value 1}) $
handler
   (Incr{ incr =
     operation (\x.\k. 1 + k ())}) $
handler
   (Exn{ fail =
     operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



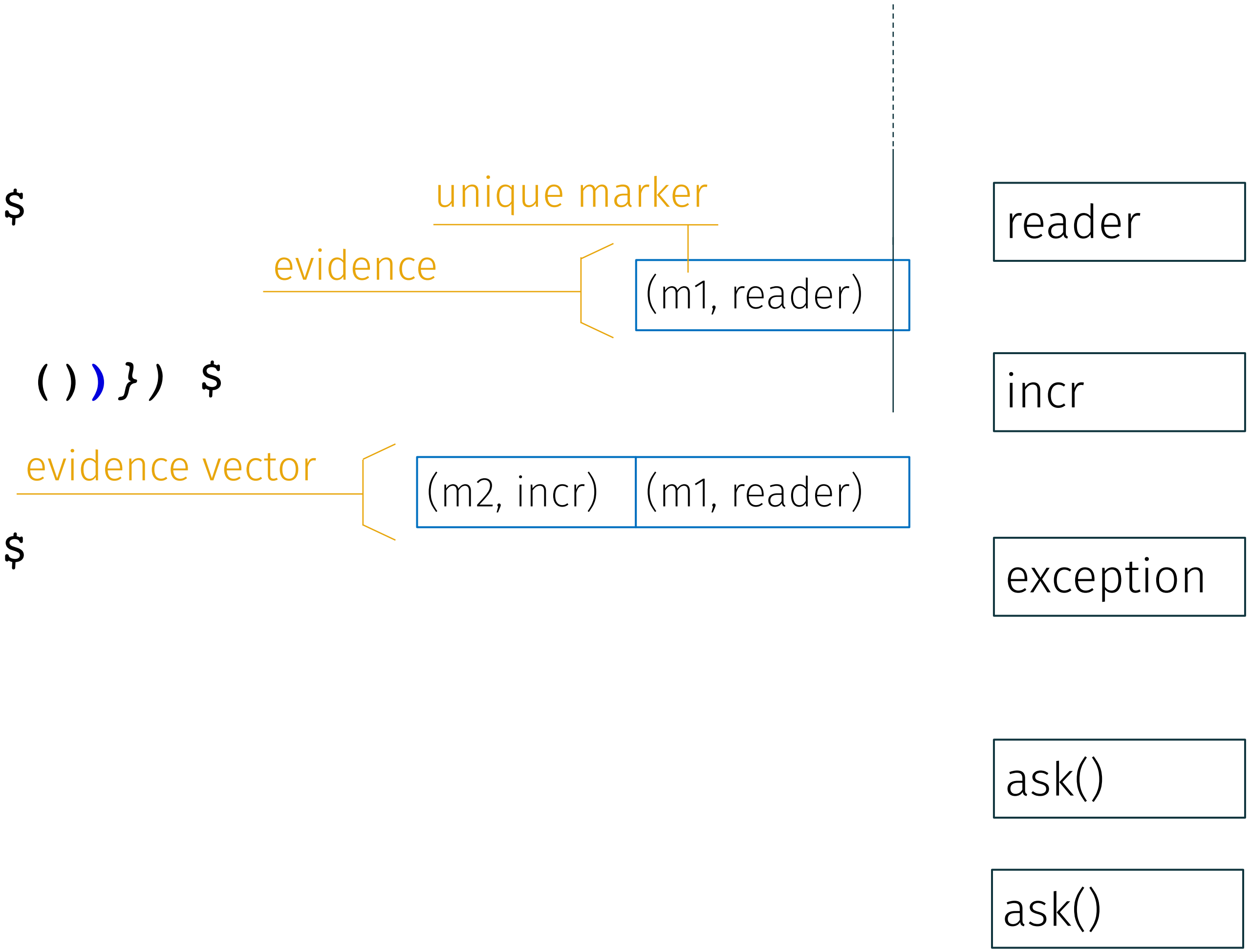
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
     operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
     operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```

unique marker

evidence

(m1, reader)

evidence vector

(m2, incr) (m1, reader)

reader

incr

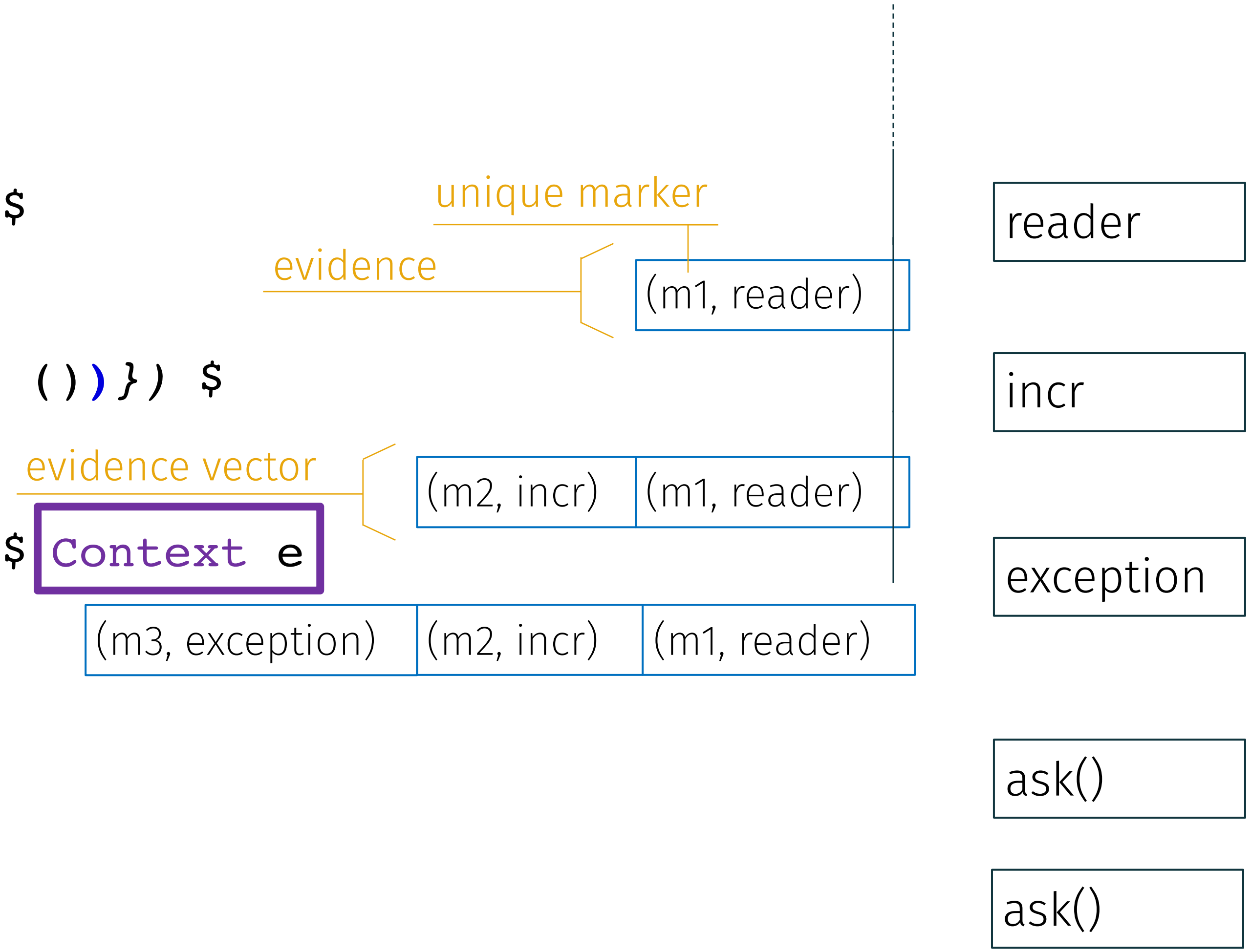
exception

ask()

ask()

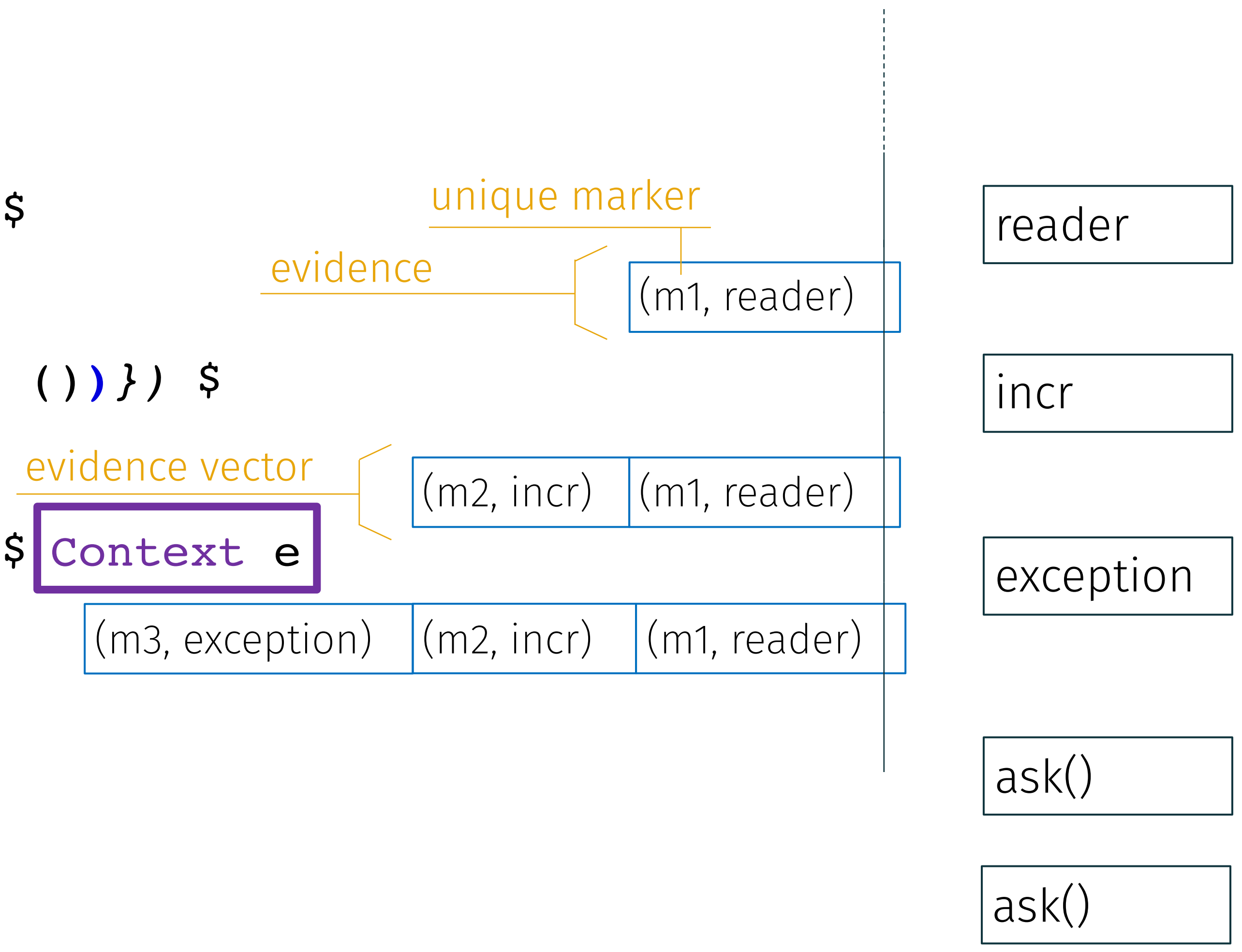
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



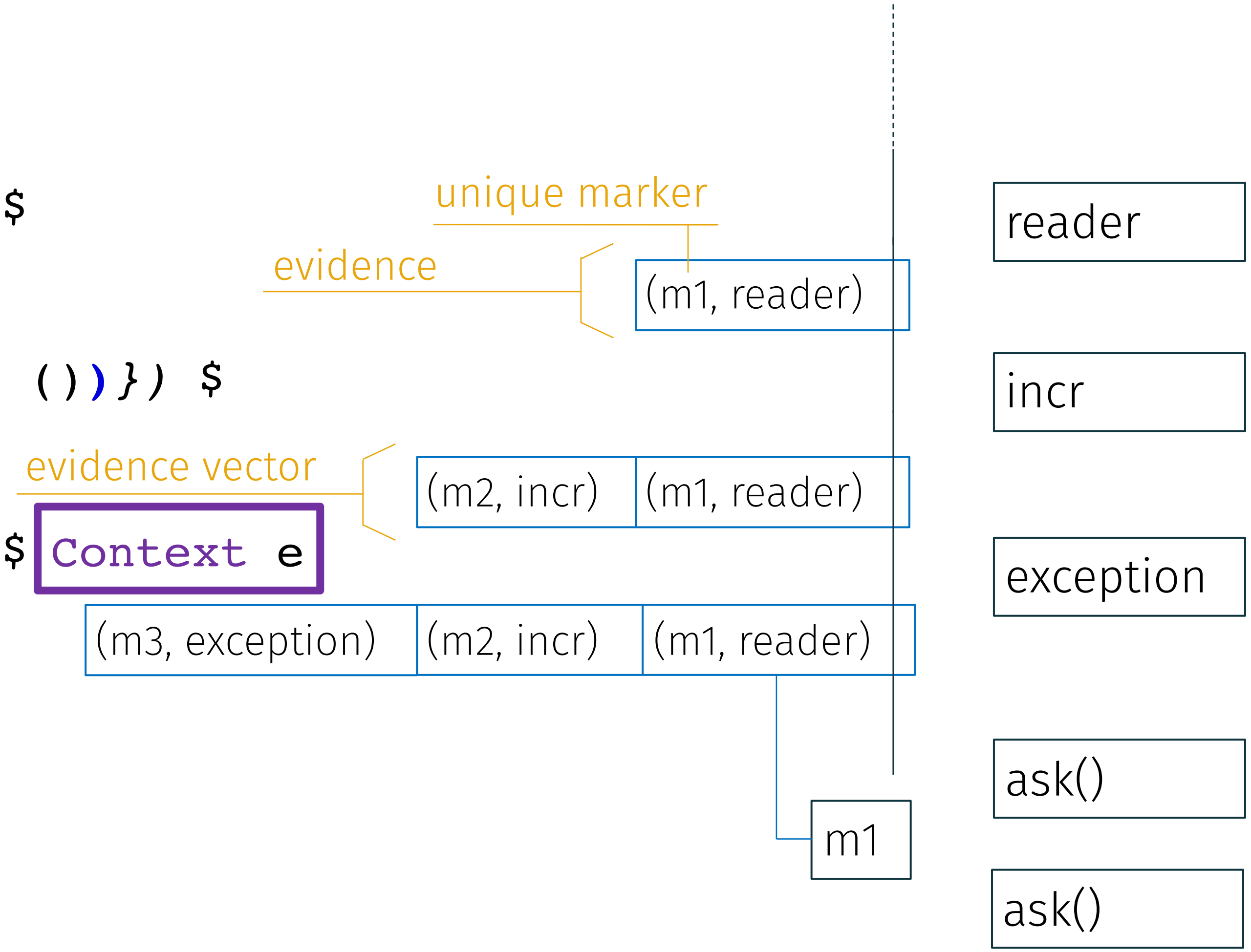
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
     operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
     operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



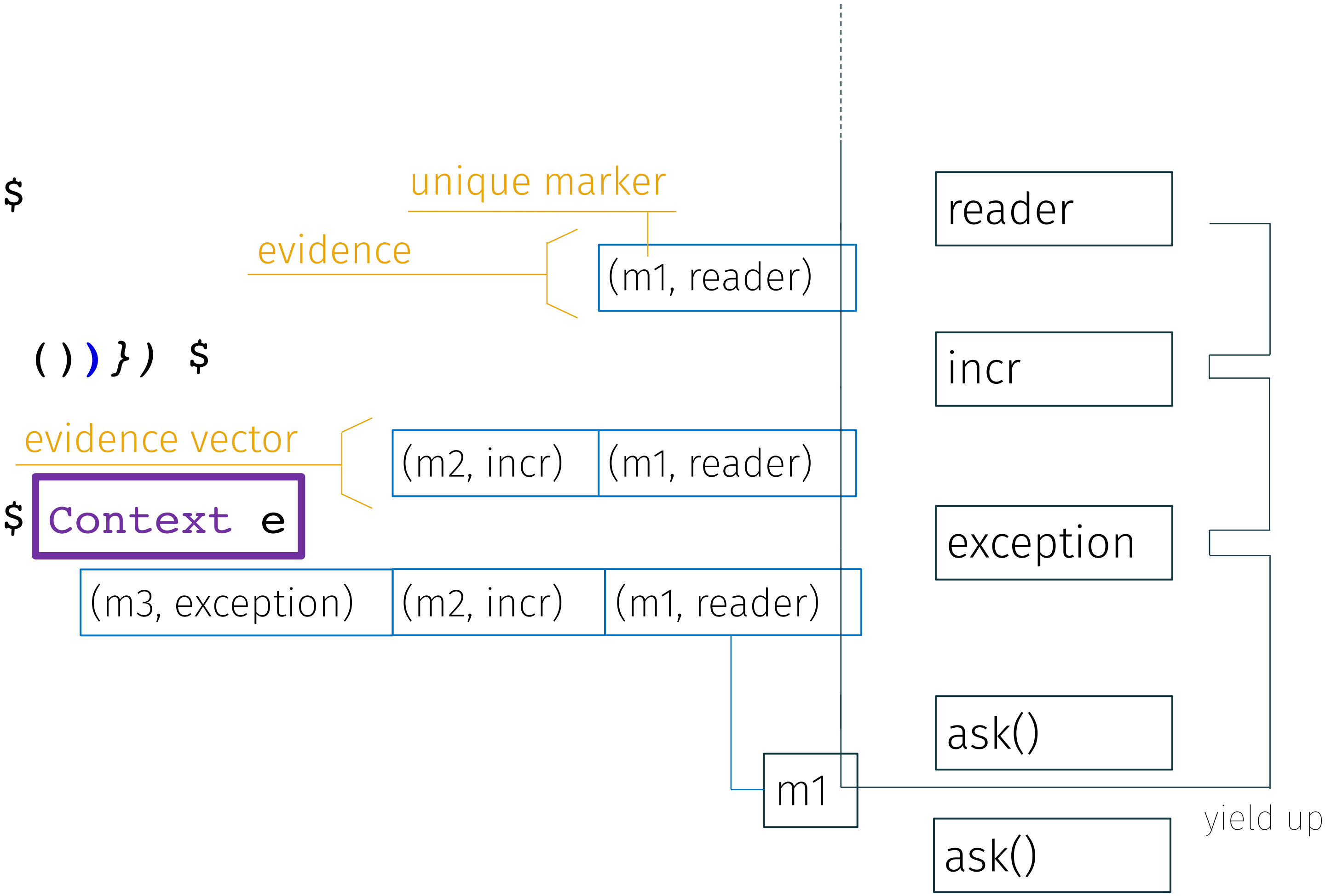
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



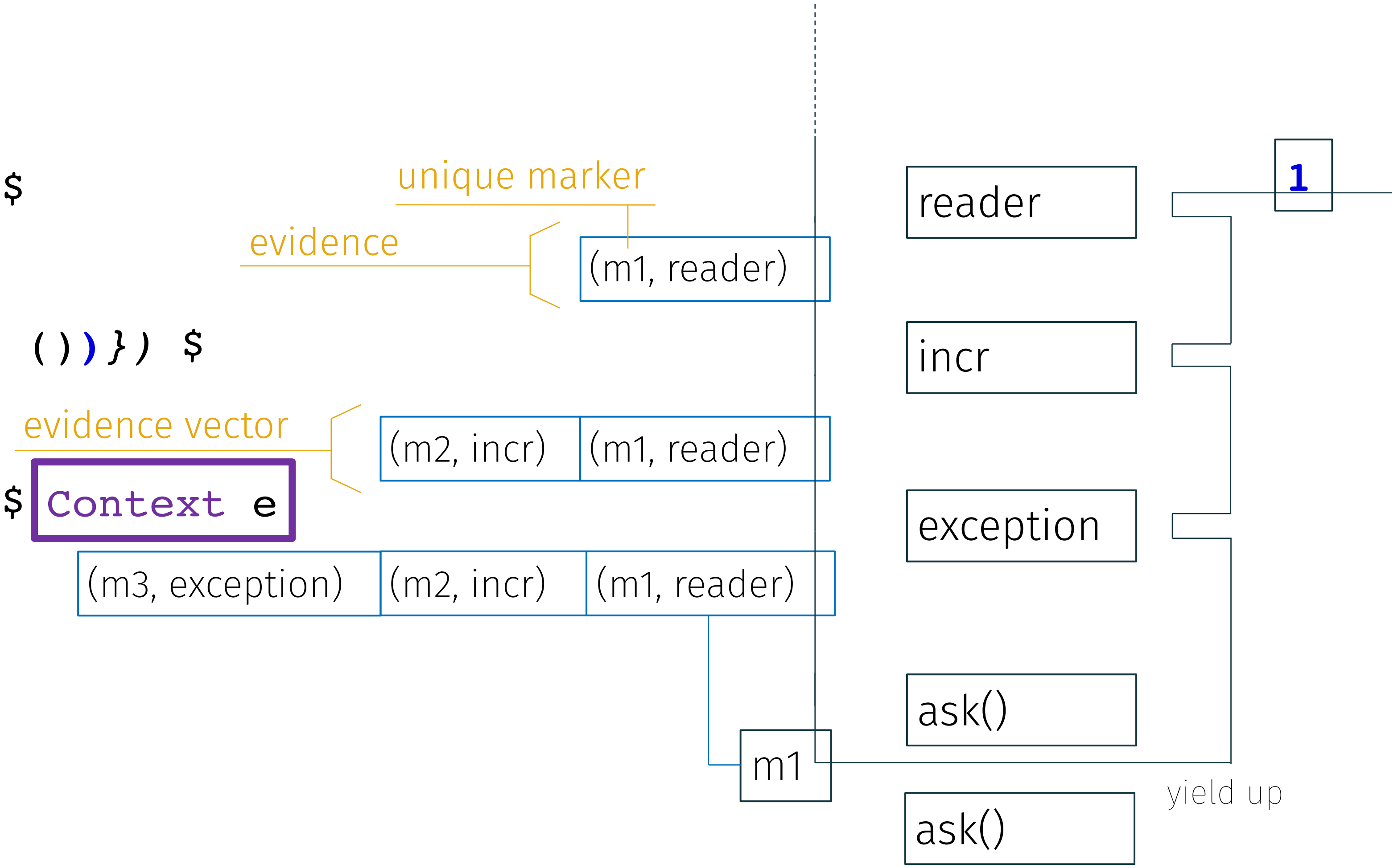
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



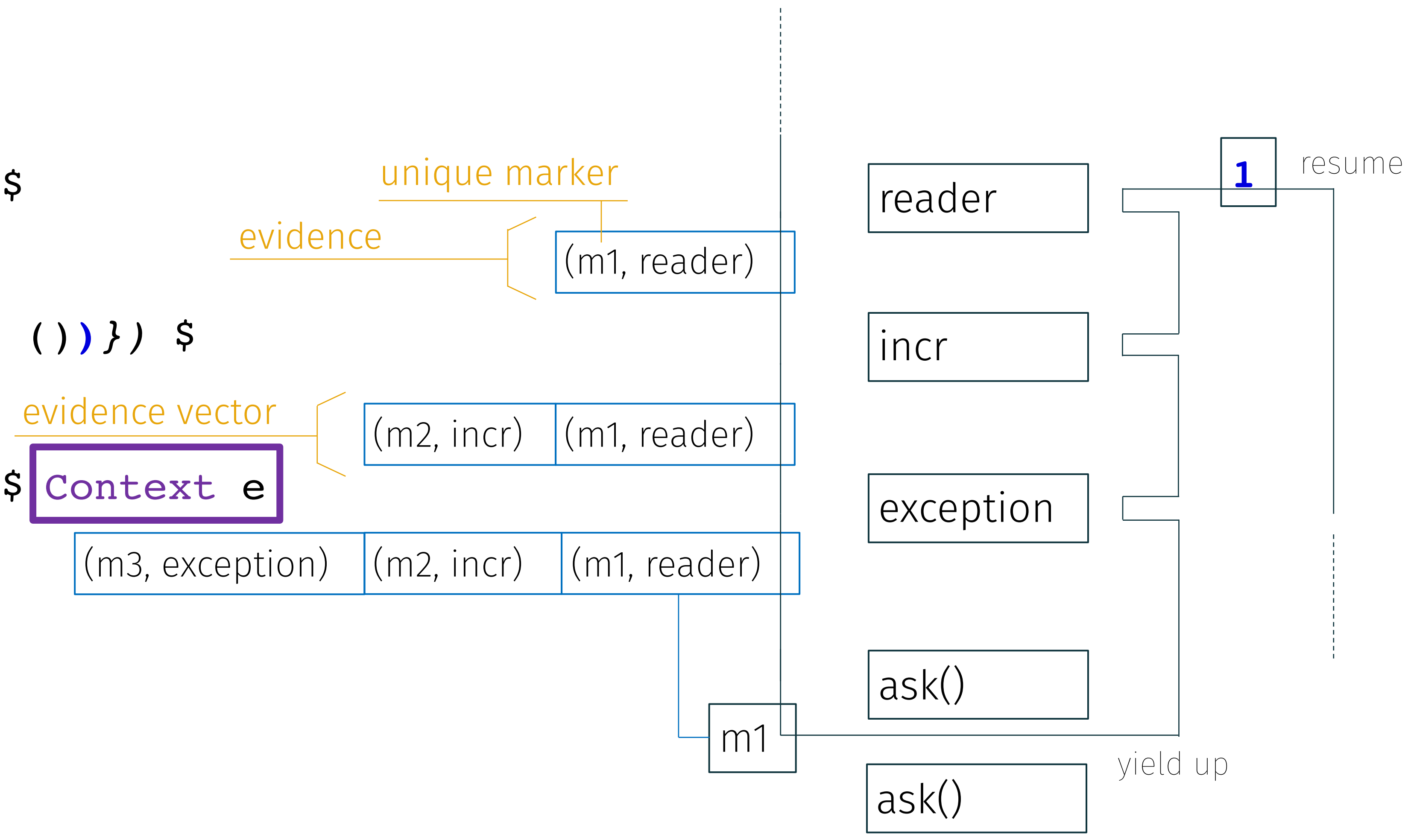
Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



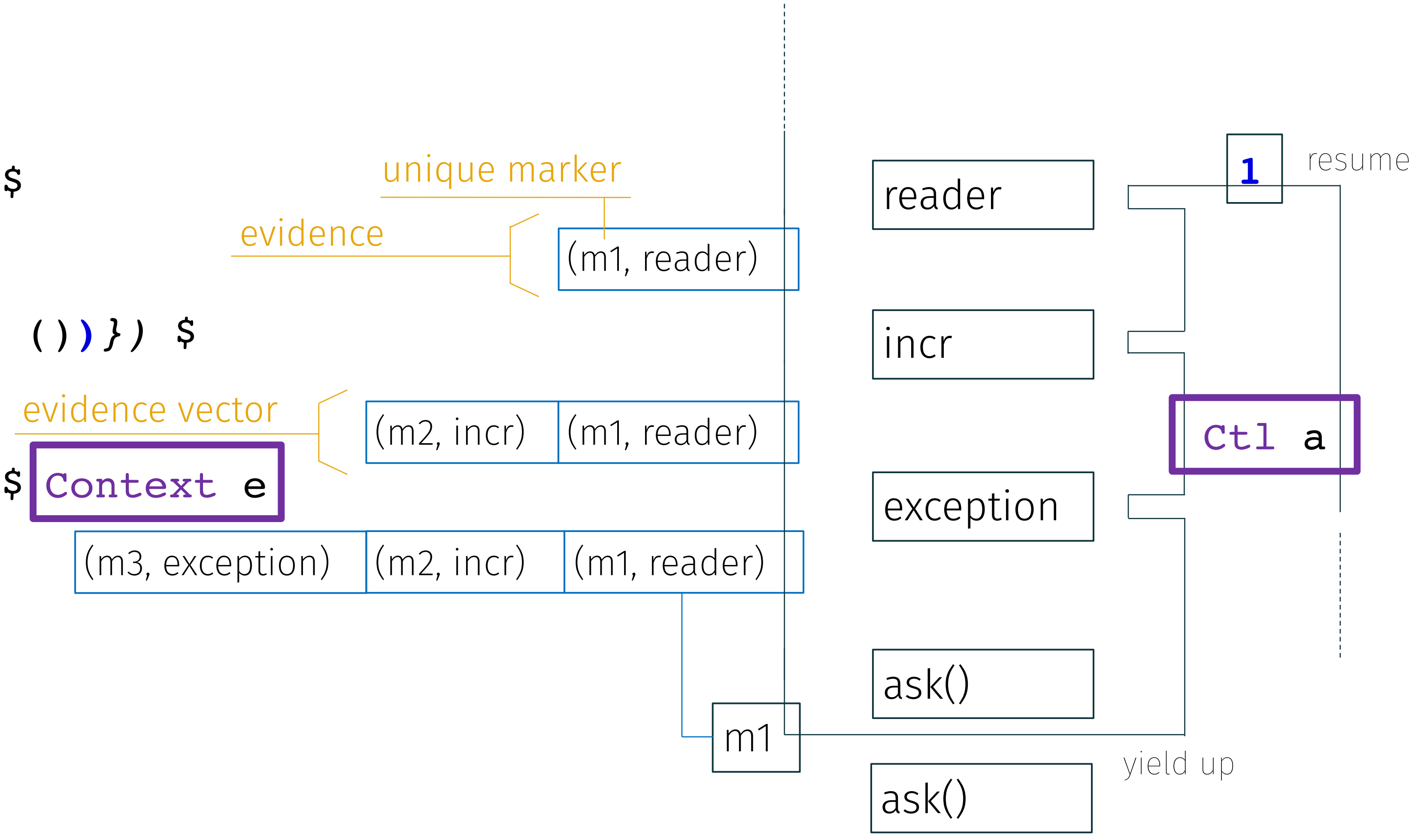
Evidence Passing

```
handler
   (Reader{ ask = value 1}) $
handler
   (Incr{ incr =
     operation (\x.\k. 1 + k ())}) $
handler
   (Exn{ fail =
     operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



Evidence Passing

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $ Context e
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2)  // 2
```



Multi-prompt Monad

```
data Ctl a
= Pure { result :: a }
| forall ans b.
  Yield {
     marker :: Marker ans,
     op     :: (b → Ctl ans) → Ctl ans,
     cont   :: b → Ctl a }
instance Monad Ctl
```

Multi-prompt Monad

```
data Ctl a
= Pure { result :: a }        ① a value result
| forall ans b.
  Yield {                     ② yielding to a prompt
     marker :: Marker ans,
     op     :: (b → Ctl ans) → Ctl ans,
     cont   :: b → Ctl a }
instance Monad Ctl
```

Multi-prompt Monad

```
                                      data Ctl a
                                      = Pure { result :: a }      ① a value result
                                      | forall ans b.
                                        Yield {                   ② yielding to a prompt
    the prompt to which it yields         marker :: Marker ans,
               the operation impl         op     :: (b → Ctl ans) → Ctl ans,
the partially built up continuation       cont   :: b → Ctl a }

                                      instance Monad Ctl
```

The Evidence Vector, or, The Context

(m3, exception)	(m2, incr)	(m1, reader)

The Evidence Vector, or, The Context

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
:: (Marker ans, Exception e1 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
             e    ans
data (h :: * → * → *) :* e
```

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

The Evidence Vector, or, The Context

```
                                     e    ans
                            data (h :: * → * → *) :* e

data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
                   :: (Marker ans, Exception e1 ans)
                   :: (Marker ans, Incr e2 ans)
                   :: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
  :: [ (Marker, Handler) ]
```

```
CCons m3 exception
```

The Evidence Vector, or, The Context

```
        e   ans
data (h :: * → * → *) :* e
```

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

❌

```
CCons m3 exception (CCons m2 incr
```

The Evidence Vector, or, The Context

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
          e    ans
          |    |
data (h :: * → * → *) :* e
```

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception)	(m2, incr)	(m1, reader)

```
:: [ (Marker, Handler) ]
```

❌

```
CCons m3 exception (CCons m2 incr (CCons m1 reader CNil)))
```

The Evidence Vector, or, The Context

```
               e    ans
data (h :: * → * → *) :* e
```

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans → Context e → Context (h :* e)
```

```
:: (Marker ans, Exception e1 ans)
:: (Marker ans, Incr e2 ans)
:: (Marker ans, (Reader Int) e3 ans)
```

(m3, exception) | (m2, incr) | (m1, reader)

```
:: [ (Marker, Handler) ]
```

```
CCons m3 exception (CCons m2 incr (CCons m1 reader CNil)))
```

```
:: Context (Exception :* Incr :* Reader Int :* ())
```

Effect Constraints (:?)

Effect Constraints (:?)

```
class h :? e where
  subContext :: Context e → SubContext h
```

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

Effect Constraints (:?)

```haskell
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```haskell
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

①
```haskell
instance h :? (h :* e) where
  subContext ctx = SubContext ctx
```

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

①
```
instance h :? (h :* e) where
  subContext ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance h :? (h :* e) where
     subContext ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance (h :? e) => h :? (h' :* e) where
     subContext (CCons _ _ ctx) = subContext ctx
```

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

①
```
instance h :? (h :* e) where
  subContext ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

②
```
instance (h :? e) => h :? (h′ :* e) where
  subContext (CCons _ _ ctx) = subContext ctx
```

```
> :load
error:
Overlapping instances for (h :? (h :* e))
```

Effect Constraints (:?)

```
data SubContext h
  = forall e. SubContext (Context (h :* e))
```

```
class h :? e where
  subContext :: Context e → SubContext h
```

(m3, h3)	(m2, h2)	(m1, h1)

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance h :? (h :* e) where
     subContext ctx = SubContext ctx
```

≠ ?

```
② instance (h :? e) => h :? (h′ :* e) where
     subContext (CCons _ _ ctx) = subContext ctx
```

```
> :load
error:
Overlapping instances for (h :? (h :* e))
```

Type-level Equality

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False    // datatype promotion
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance            InEq 'True h1 h2 e where
    subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance
                        InEq 'False h1 h2 e where
    subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance                InEq 'True h1 h2 e where
      subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance
                           InEq 'False h1 h2 e where
      subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Conte                      ntext h1
```

'True ~ HEqual h1 h2

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance                InEq 'True h1 h2 e where
       subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance
                            InEq 'False h1 h2 e where
       subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance (h1 ~ h2) ⇒ InEq 'True h1 h2 e where
      subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance
                    InEq 'False h1 h2 e where
      subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = ‘True
  HEqual h1 h2 = ‘False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance (h1 ~ h2) ⇒ InEq ‘True h1 h2 e where
     subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance
                         InEq ‘False h1 h2 e where
     subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance (h1 ~ h2) ⇒ InEq 'True h1 h2 e where
     subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance ('False ~ HEqual h1 h2
                         InEq 'False h1 h2 e where
     subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance (h1 ~ h2) ⇒ InEq 'True h1 h2 e where
      subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

```
② instance ('False ~ HEqual h1 h2
           , h1 :? e) ⇒  InEq 'False h1 h2 e where
      subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

Type-level Equality

1. type-level equality function

```
type family HEqual (h1 :: * → * → *) h2 where
  HEqual h1 h1 = 'True
  HEqual h1 h2 = 'False
```

2. effect constaints with type equality

```
class (heq ~ HEqual h1 h2) ⇒ InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) → SubContext h1
```

(m3, h3)	(m2, h2)	(m1, h1)

```
① instance (h1 ~ h2) ⇒ InEq 'True h1 h2 e where
     subContextEq ctx = SubContext ctx
```

(m3, h3)	(m2, h2)	(m1, h1)

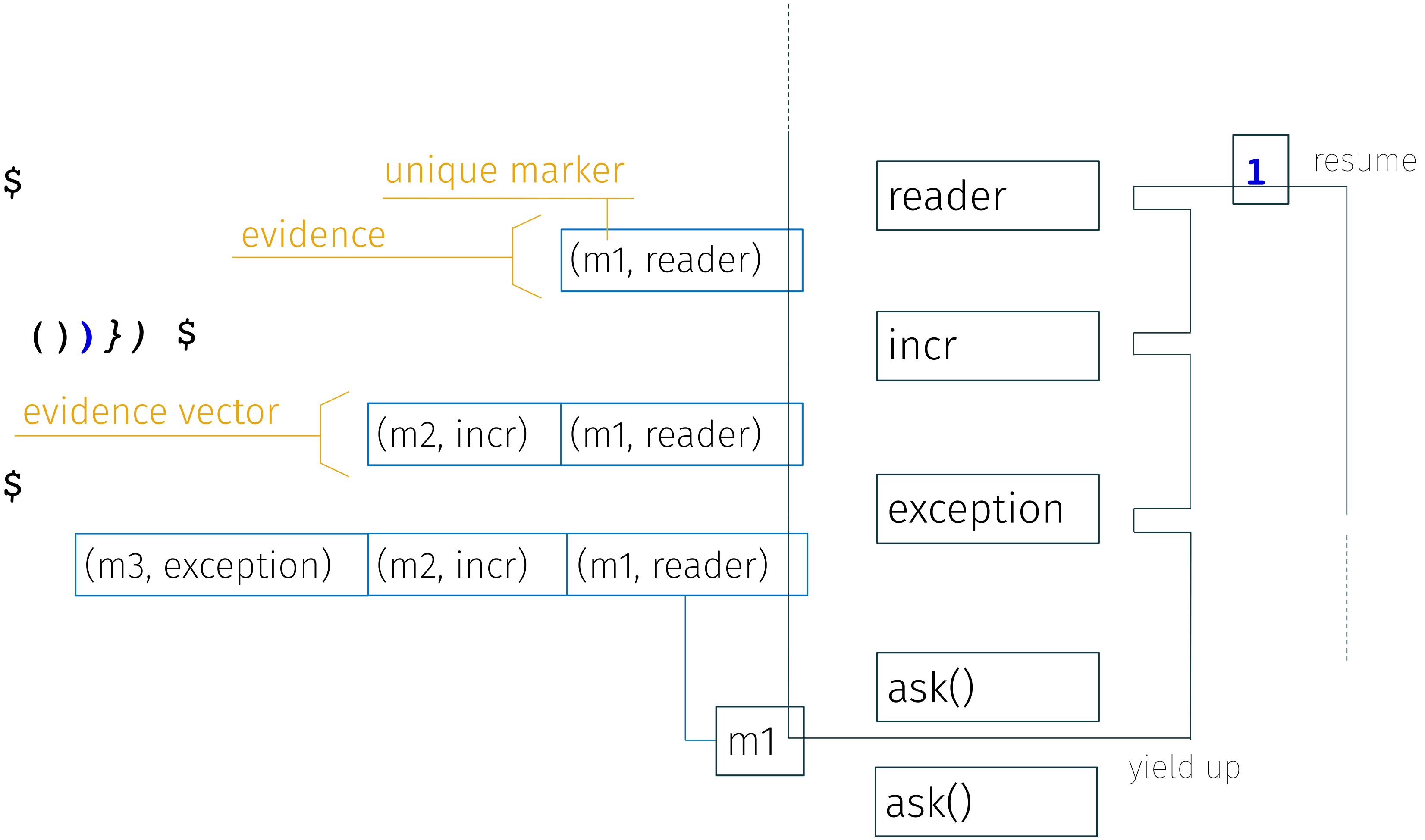
```
② instance ('False ~ HEqual h1 h2
           , h1 :? e) ⇒  InEq 'False h1 h2 e where
     subContextEq (CCons _ _ ctx) = subContext ctx
```

3. delegate

```
instance InEq (HEqual h1 h2) h1 h2 e) ⇒ h1 :? (h2 :* e) where
  subContext = subContextEq
```

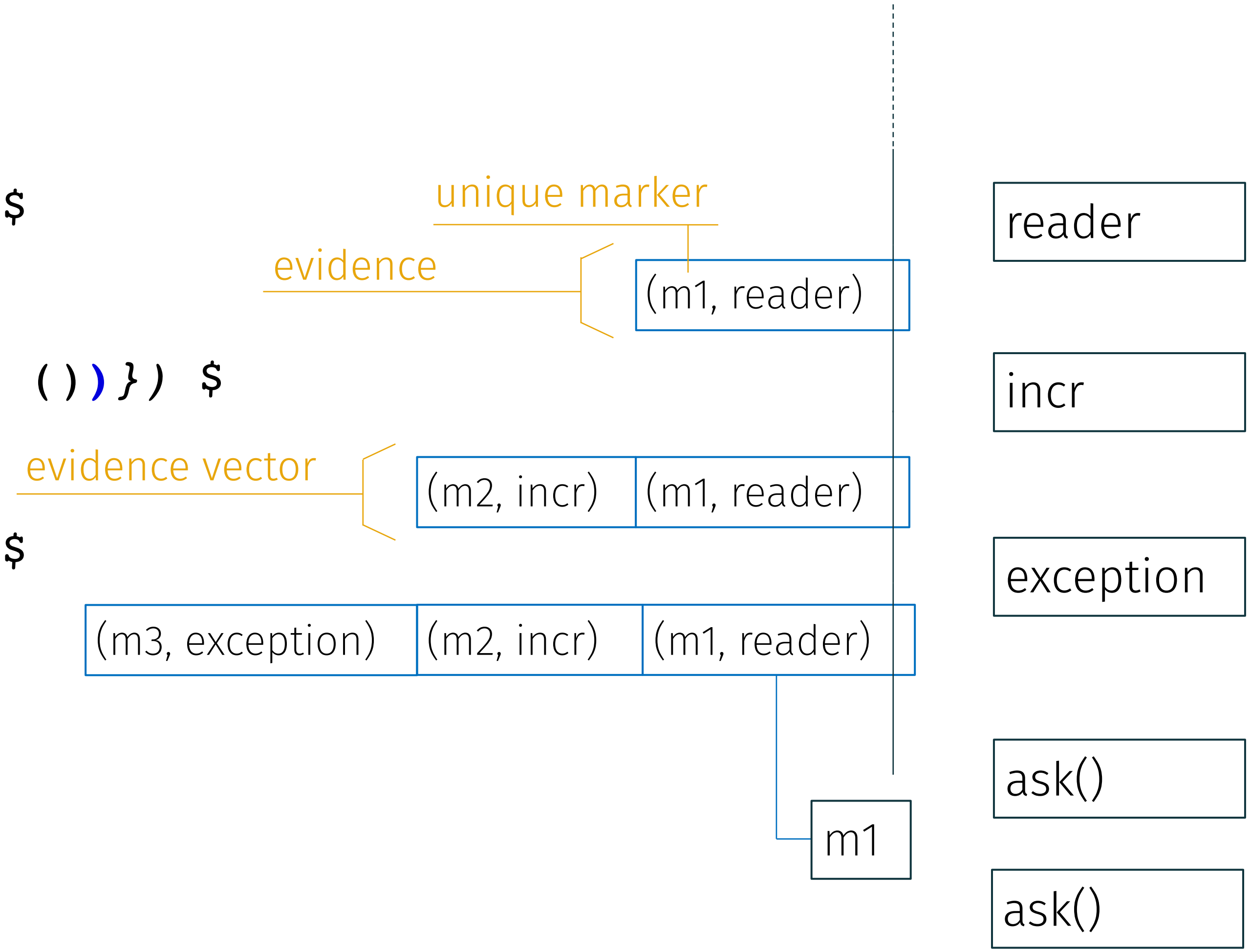
Optimization

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



Optimization

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```

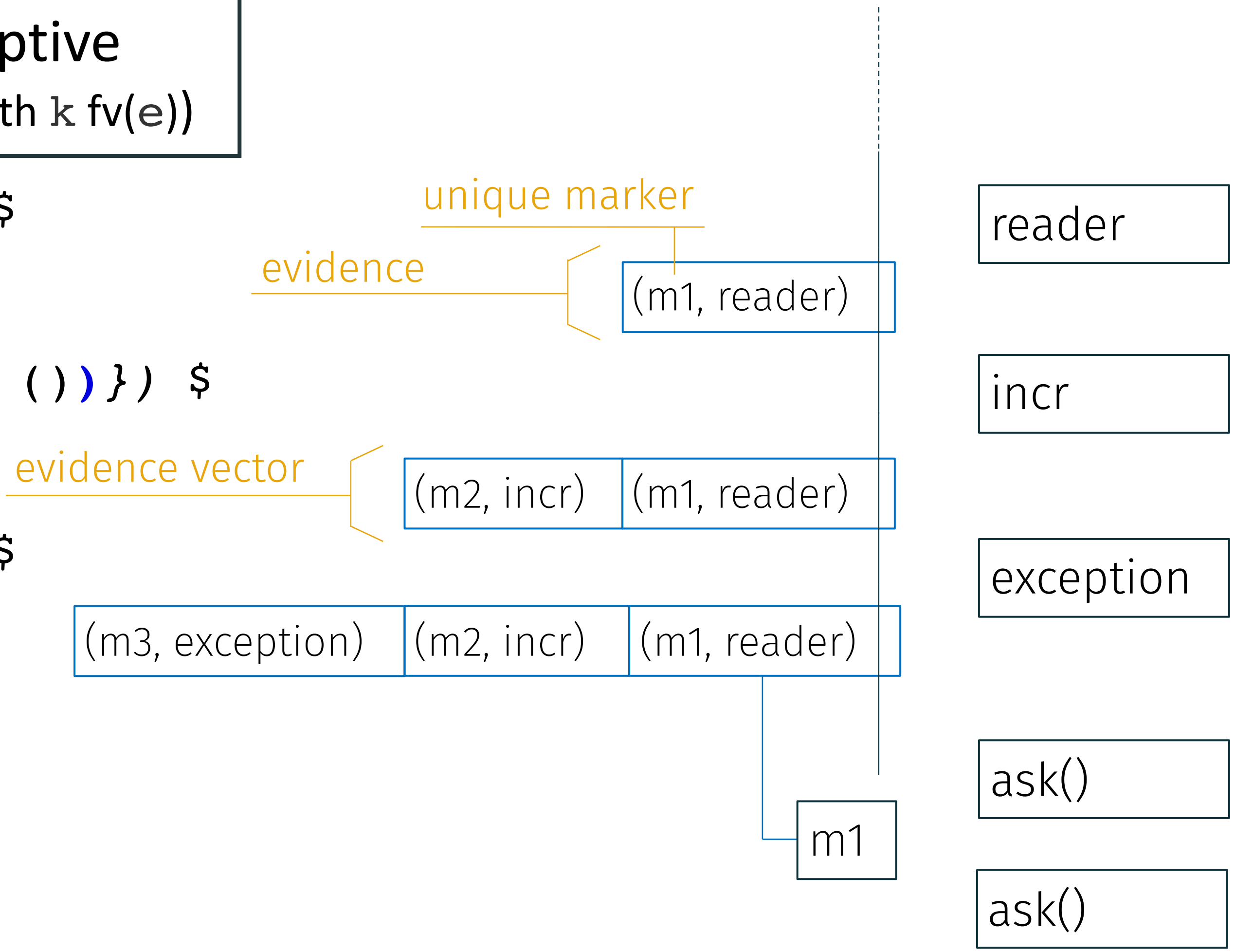


Optimization

tail-resumptive

(`\x.\k. k e` with `k` fv(e))

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```

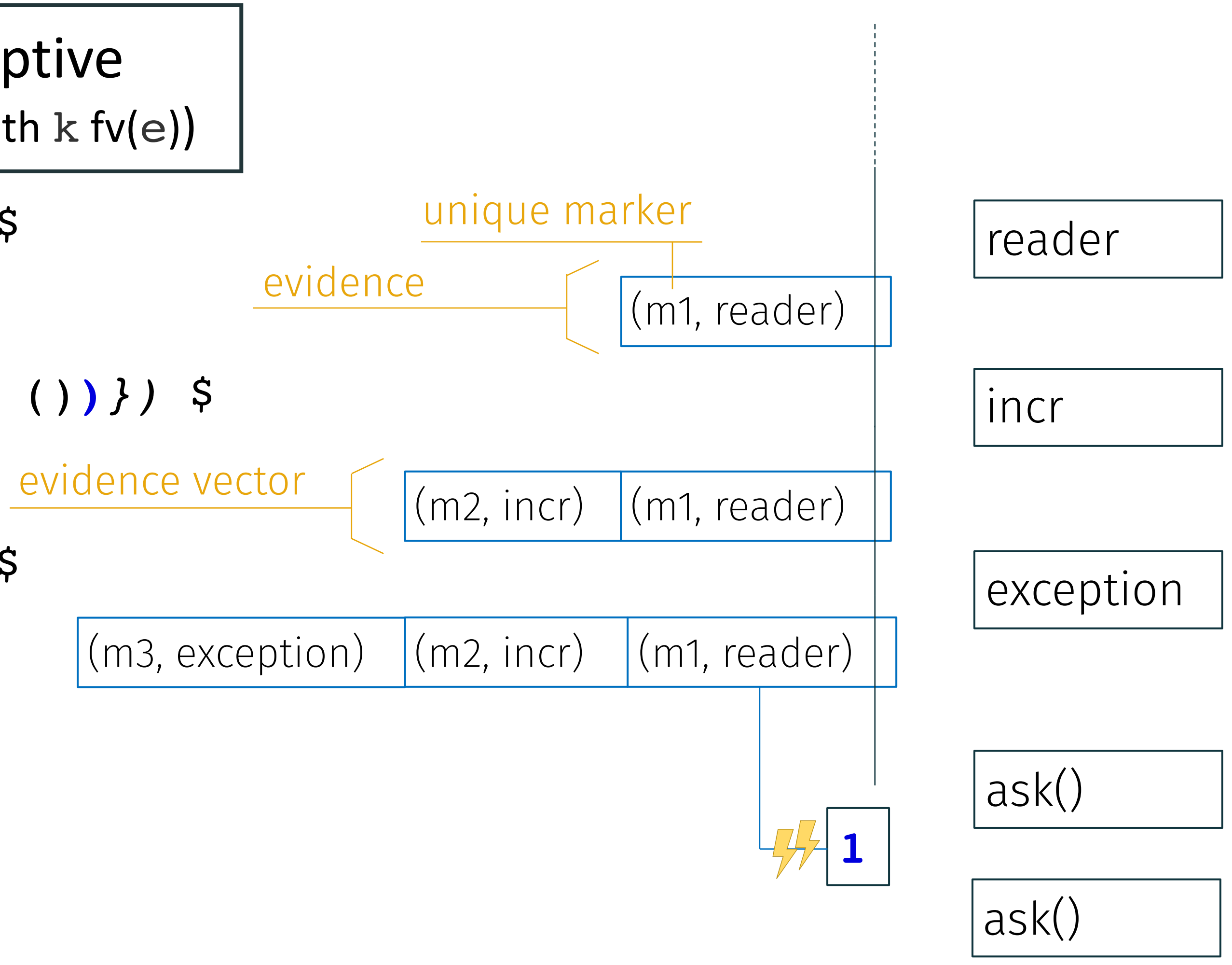


Optimization

tail-resumptive

(`\x.\k. k e` with `k` fv(e))

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```

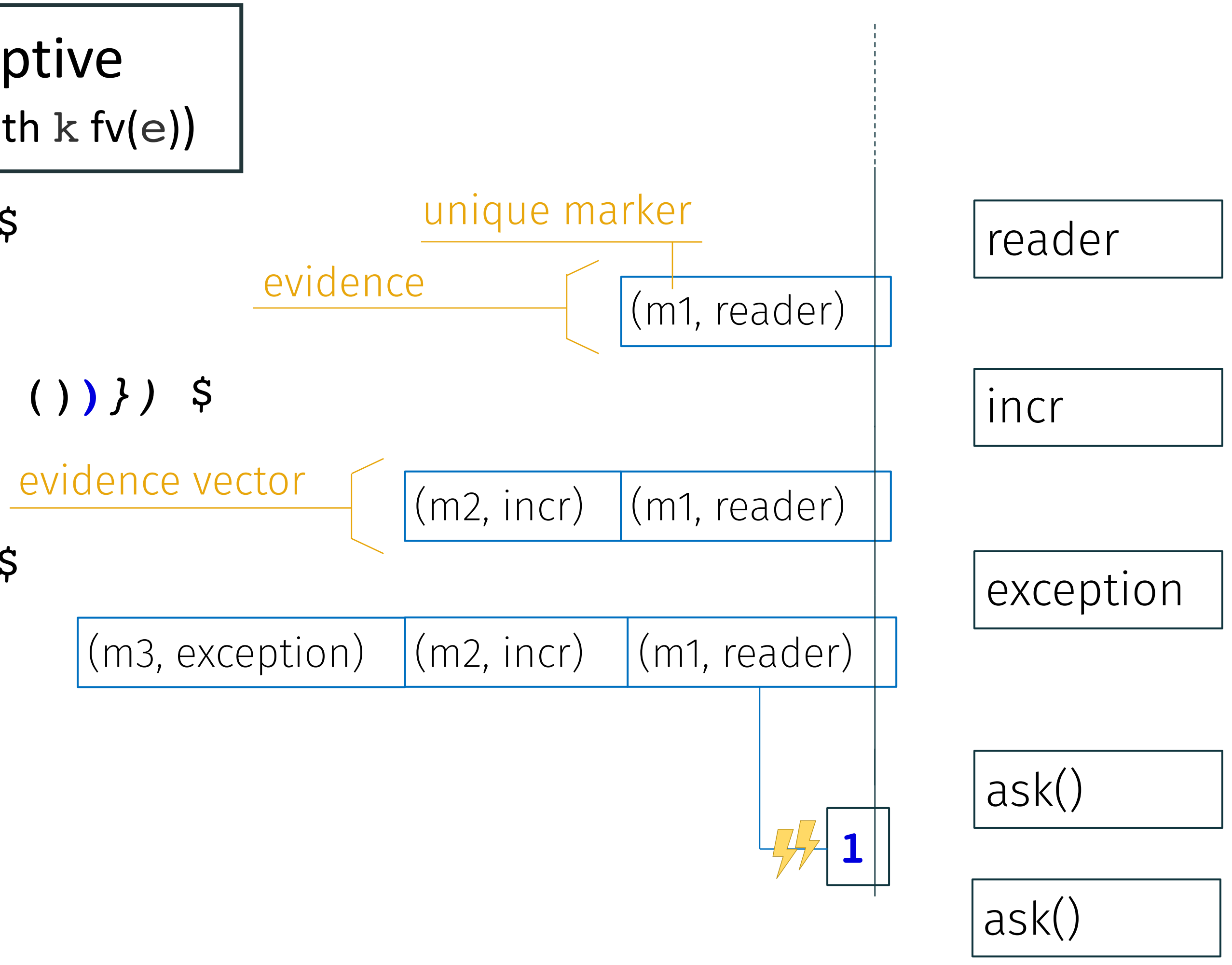


Optimization

tail-resumptive

(\x.\k. k e with k fv(e))

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```

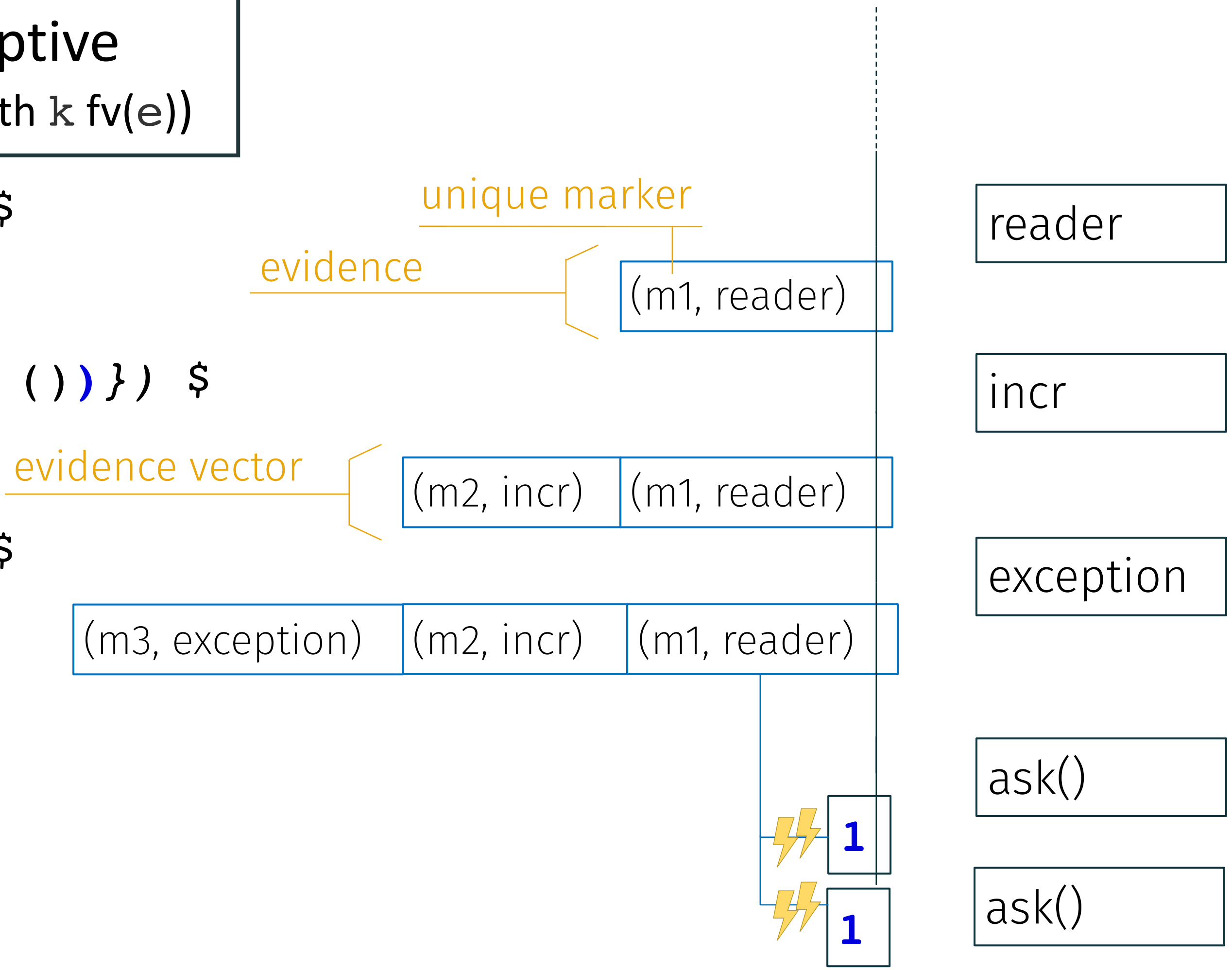


Optimization

tail-resumptive

(\x.\k. k e with k fv(e))

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```

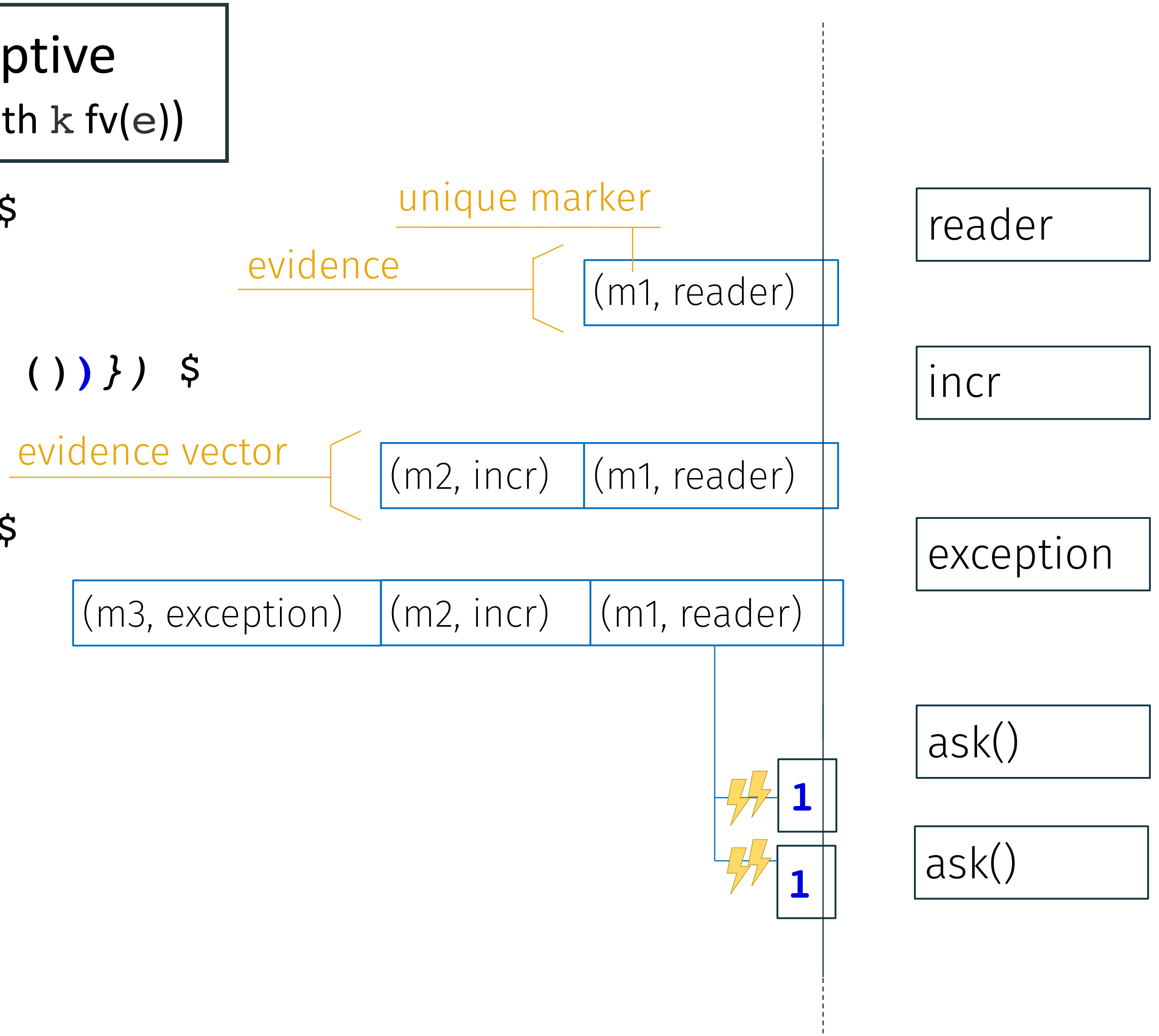


Optimization

tail-resumptive

(`\x.\k. k e` with `k` fv(e))

```
handler
  (Reader{ ask = value 1}) $
handler
  (Incr{ incr =
    operation (\x.\k. 1 + k ())}) $
handler
  (Exn{ fail =
    operation (\x.\k. 3)}) $
do x1 <- perform ask ()
   x2 <- perform ask ()
   return (x1 + x2) // 2
```



Optimization

Optimization

✓ 1. tail-resumptive operations (i.e., `value/function`) are evaluated in-place

✓ 2. non tail-resumptive operations (i.e., `operation`) *locally* decide which *marker* to yield to

Scoped Resumptions

- Restriction: resumptions can only be *resumed* in the same handler context as *captured*

- We believe that all important effect handlers in practice can be defined in terms of scoped resumptions

- Implemented as a dynamic check, called `guard`

Benchmarks

Benchmarks

EV	our `Control.Ev.Eff` library
EV NT	our `Control.Ev.Eff` library; handlers always **N**on **T**ail-resumptive
EE	the **E**xtensible **E**ffects library [Kiselyov and Ishii 2015]
FE	the **F**used **E**ffects library [Schrijvers et al. 2019; Wu and Schrijvers 2015b; Wu et al. 2014]
MTL	the **M**onad **T**ransformer **L**ibrary

Benchmarks
[Kiselyov and Ishii 2015]

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00×**

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	**Time**	**Speed**
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00×**

```
runCount5 :: (State Integer :? e) ⇒
             Integer → Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0]
  where f acc x | x ‘mod‘ 5 == 0
                = do i <- perform get ()
                     perform put (i+1)
                     return (max acc x)
        f acc x = return (max acc x)
```

Benchmarks

```haskell
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

```haskell
runCount5 :: (State Integer :? e) ⇒
             Integer → Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0]
  where f acc x | x ‘mod‘ 5 == 0
                = do i <- perform get ()
                     perform put (i+1)
                     return (max acc x)
        f acc x = return (max acc x)
```

(msec)	Time	Speed
Pure	247	0.34×
MTL	327	0.25×
RunST	256	0.32×
EE	129	0.64×
FE	136	0.61×
EV NT	99	0.84×
EV	83	**1.00**×

Benchmarks

```haskell
runCount :: (State Int :? e) ⇒  Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

```haskell
runCount5 :: (State Integer :? e) ⇒
             Integer → Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0]
  where f acc x | x ‘mod‘ 5 == 0
          = do i <- perform get ()
               perform put (i+1)
               return (max acc x)
        f acc x = return (max acc x)
```

(msec)	Time	Speed
Pure	247	0.34×
MTL	327	0.25×
RunST	256	0.32×
EE	129	0.64×
FE	136	0.61×
EV NT	99	0.84×
EV	83	**1.00**×

Benchmarks

```
runCount :: (State Int :? e) ⇒ Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

(msec)	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

```
runCount5 :: (State Integer :? e) ⇒
               Integer → Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0]
  where f acc x | x ‘mod‘ 5 == 0
          = do i <- perform get ()
               perform put (i+1)
               return (max acc x)
        f acc x = return (max acc x)
```

(msec)	Time	Speed
Pure	247	0.34×
MTL	327	0.25×
RunST	256	0.32×
EE	129	0.64×
FE	136	0.61×
EV NT	99	0.84×
EV	83	**1.00**×

Benchmarks

```haskell
runCount :: (State Int :? e) ⇒ Eff e Int
runCount = do i <- perform get ()
              if (i==0) then return i
              else do perform put (i - 1)
                      runCount
```

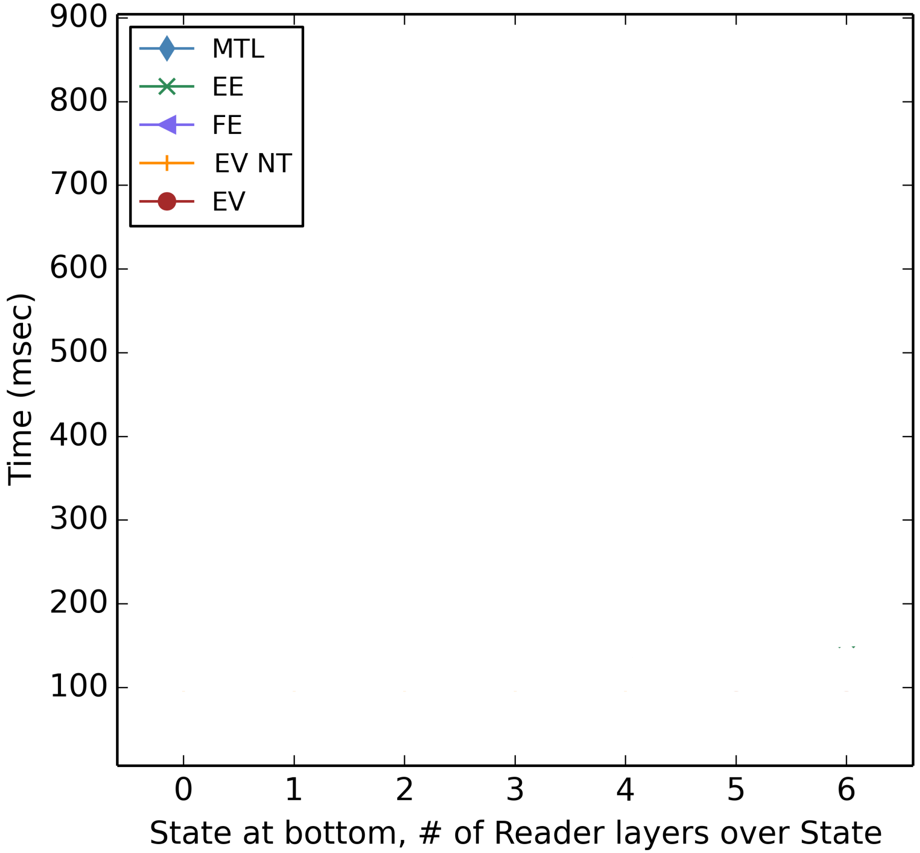
(msec)	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

```haskell
runCount5 :: (State Integer :? e) ⇒
               Integer → Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0]
  where f acc x | x `mod` 5 == 0
                = do i <- perform get ()
                     perform put (i+1)
                     return (max acc x)
        f acc x = return (max acc x)
```

(msec)	Time	Speed
Pure	247	0.34×
MTL	327	0.25×
RunST	256	0.32×
EE	129	0.64×
FE	136	0.61×
EV NT	99	0.84×
EV	83	**1.00**×

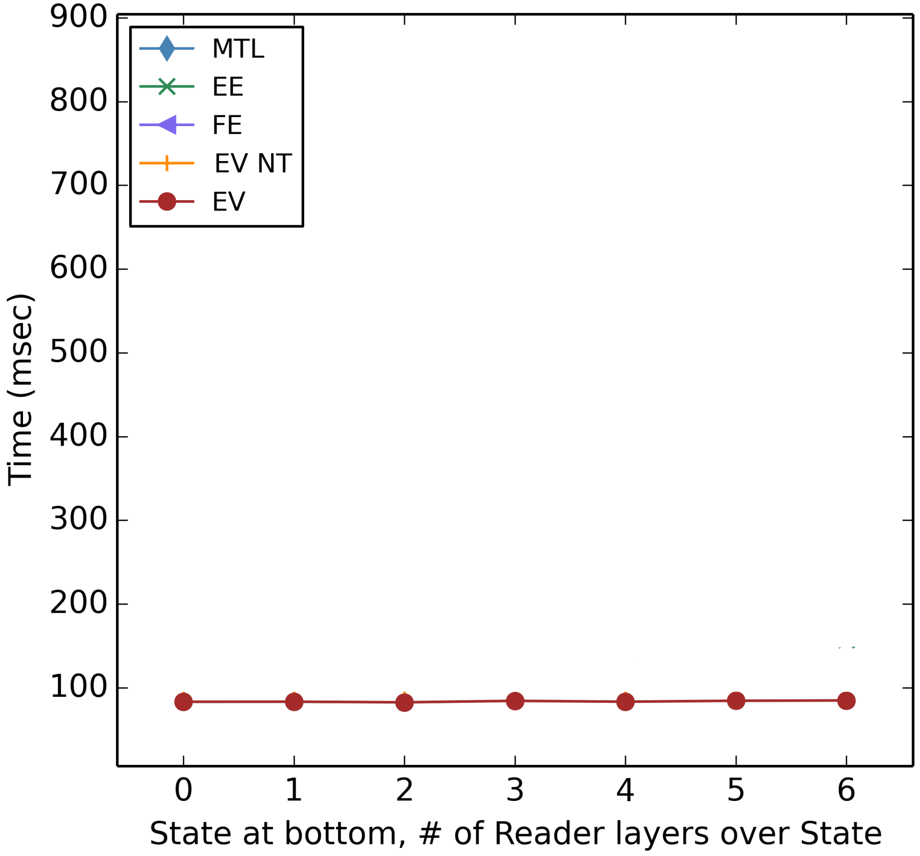
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



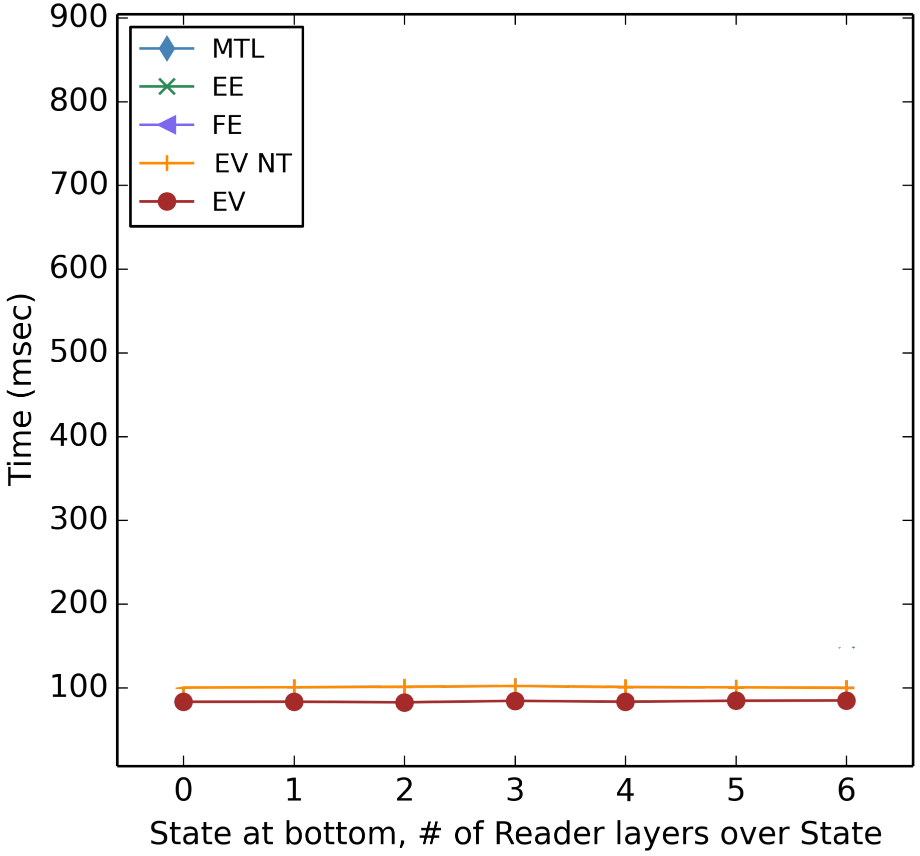
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



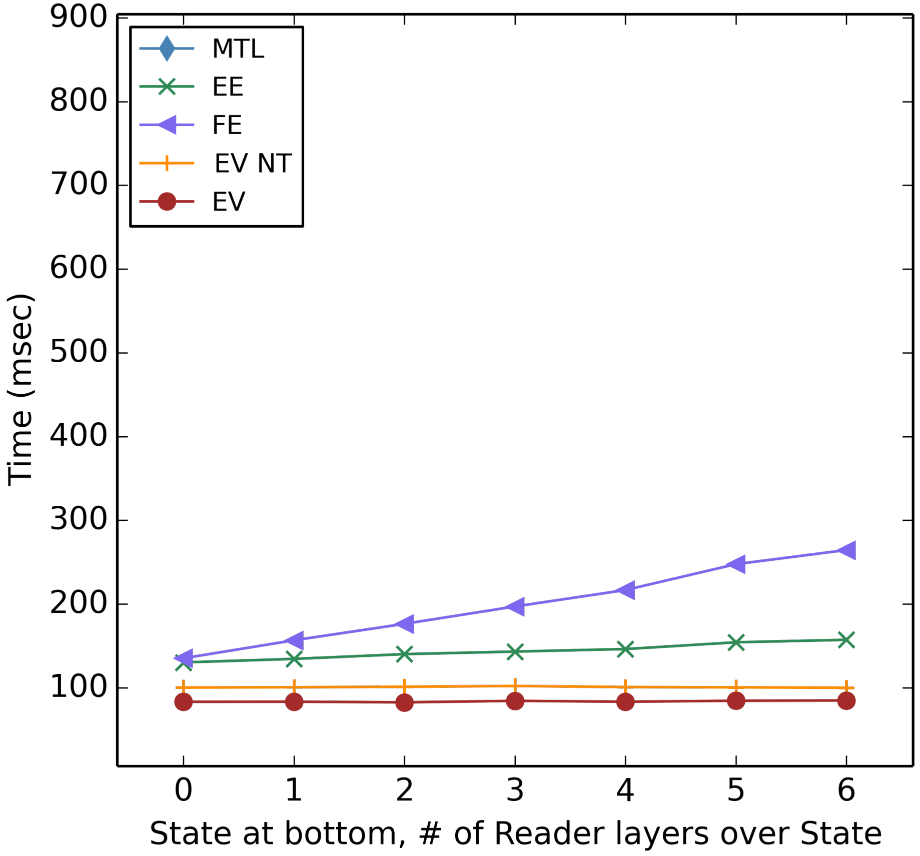
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



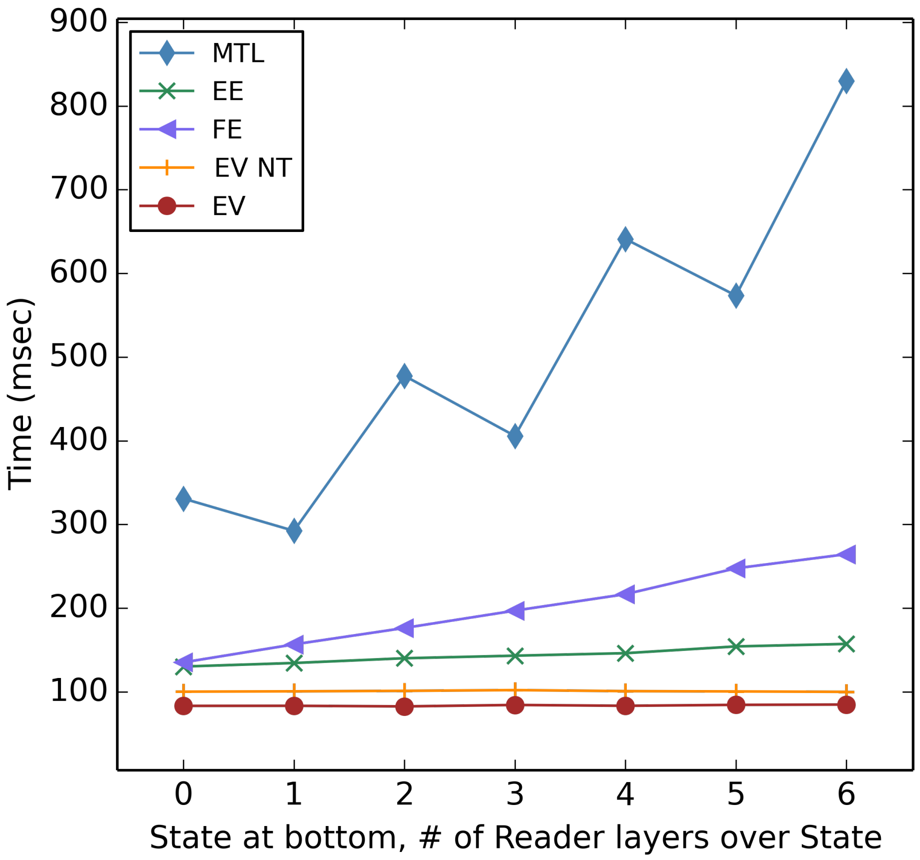
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



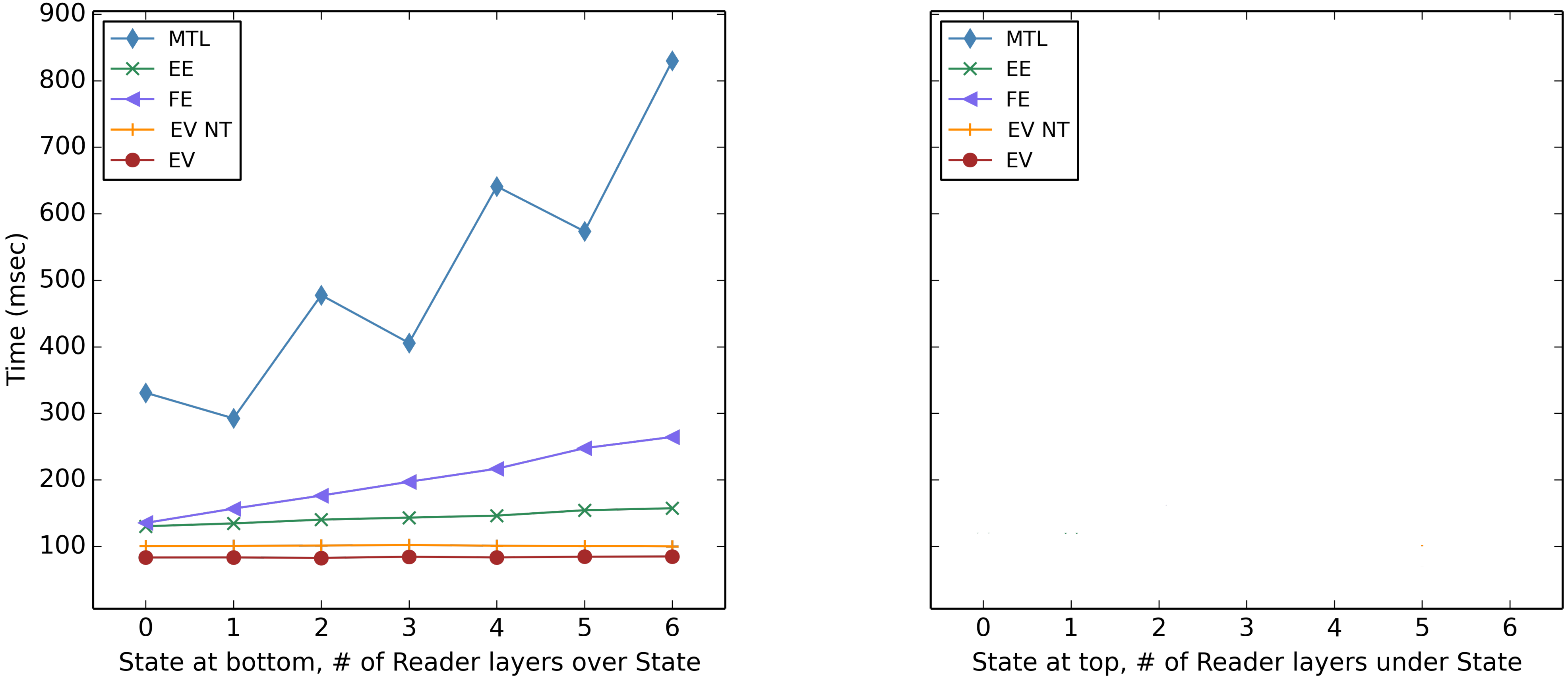
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



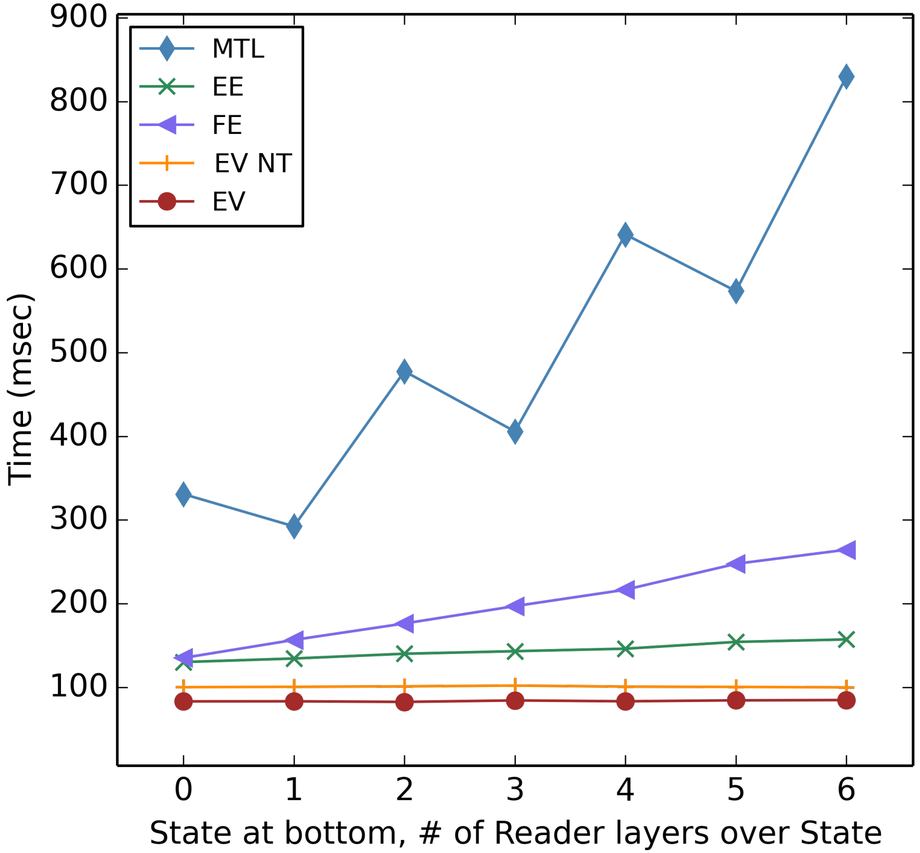
Benchmarks

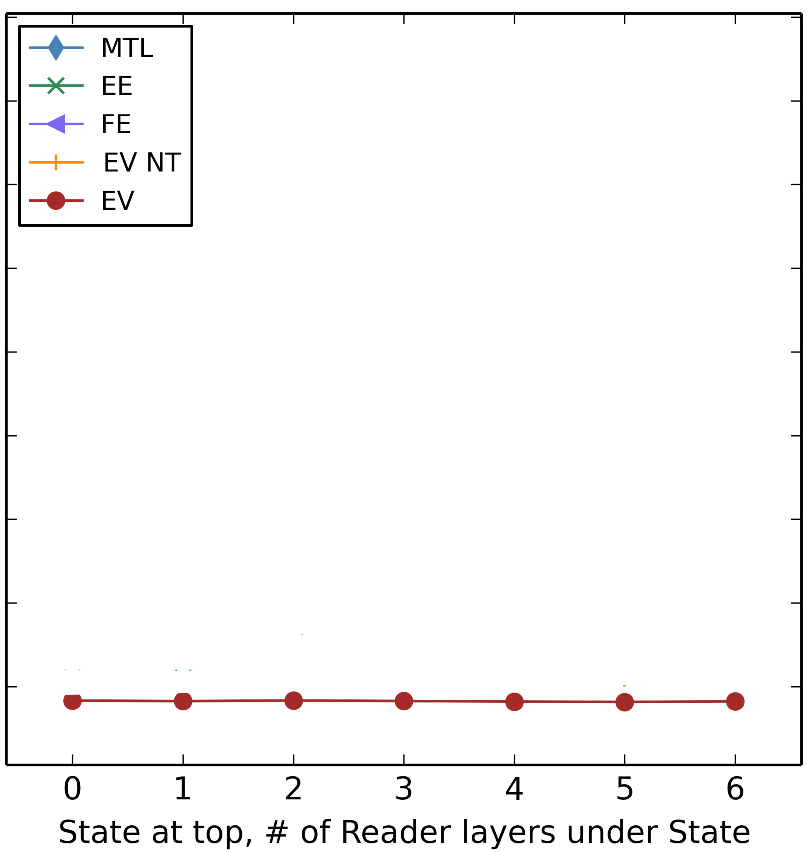
In the benchmark, we put many `Reader` layers under or over the target `State` layer.



Benchmarks

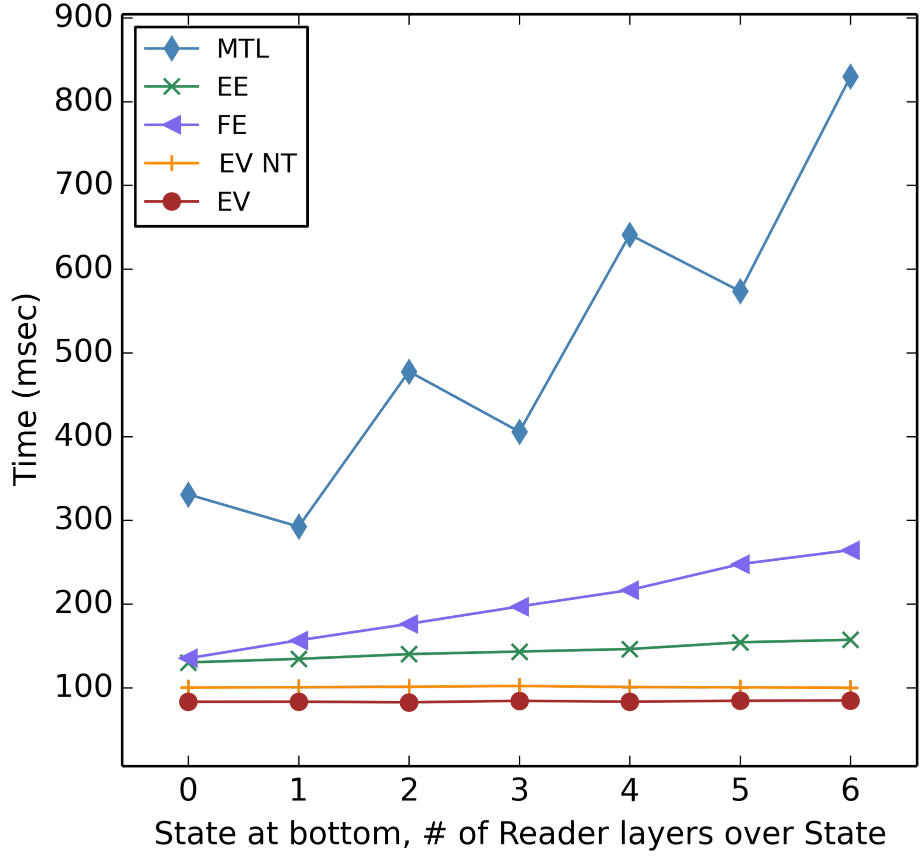
In the benchmark, we put many `Reader` layers under or over the target `State` layer.

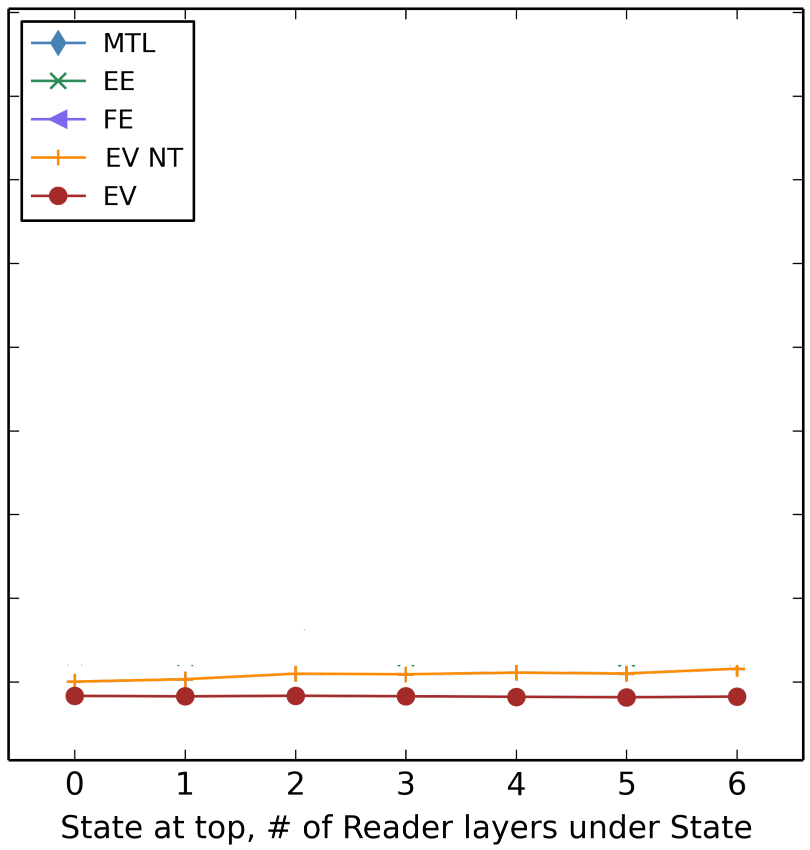




Benchmarks

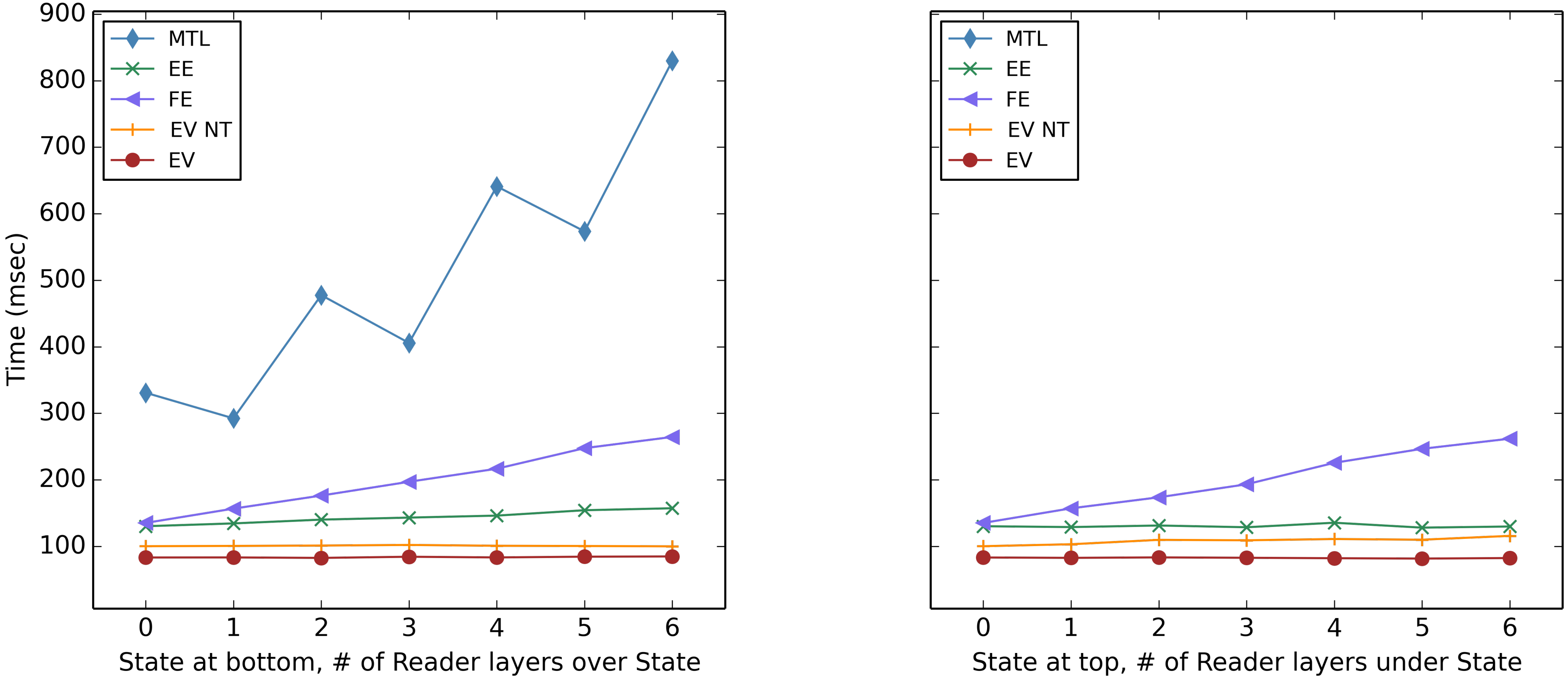
In the benchmark, we put many `Reader` layers under or over the target `State` layer.





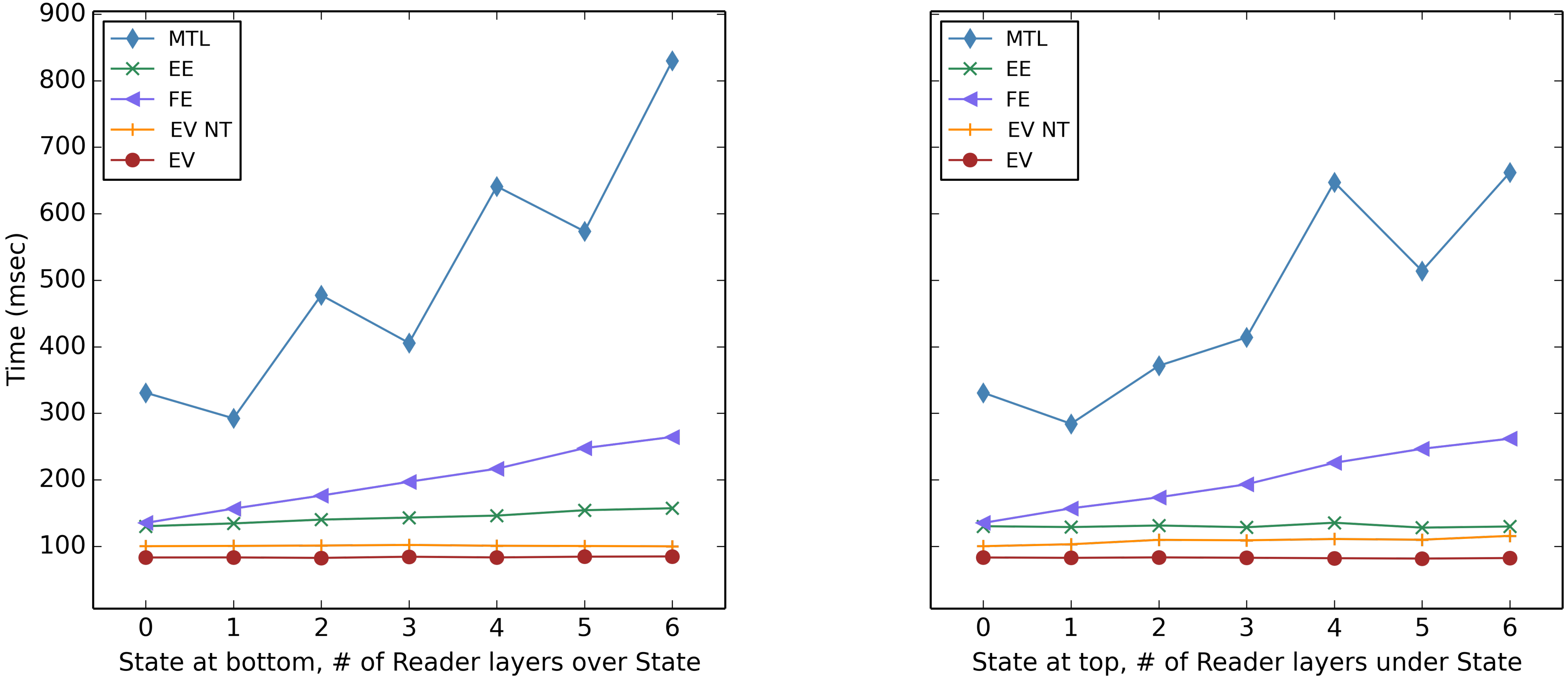
Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



Benchmarks

In the benchmark, we put many `Reader` layers under or over the target `State` layer.



More in the Paper

- Advanced handlers:
  handlers with return clauses, handlers with local state

- Our implementation ensures type safety

- More discussion

https://github.com/xnning/EvEff

More in the Paper

- Advanced handlers:
  handlers with return clauses, handlers with local state

- Our implementation ensures type safety

- More discussion

https://github.com/xnning/EvEff

Effect Handlers, Evidently

Ningning Xie Jonathan Brachthäuser
Daniel Hillerström Philipp Schuster Daan Leijen

ICFP 2020