Effect Handlers, Evidently

Ningning Xie Jonathan Brachthäuser
Daniel Hillerström Philipp Schuster
Daan Leijen

ICFP 2020

In this paper

algebraic effects $\xrightarrow{\text{evidence passing}}$ **polymorphic lambda calculus**

Algebraic Effects 101

```
effect reader {
  ask : () -> int
}


handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

Algebraic Effects 101

effect

```
effect reader {
  ask : () -> int
}


handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

Algebraic Effects 101

operation effect

```
effect reader {
  ask : () -> int
}


handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

Algebraic Effects 101

operation effect

```
effect reader {
  ask : () -> int
}


handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

raise an effect

Algebraic Effects 101

operation

effect

```
effect reader {
  ask : () -> int
}
```

effect handler

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

raise an effect

Algebraic Effects 101

operation

effect

```
effect reader {
  ask : () -> int
}
```

effect handler

implementation

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask ()
)
```

raise an effect

Algebraic Effects 101

operation

effect

```
effect reader {
  ask : () -> int
}
```

effect handler

implementation

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask () // 2
)
```

raise an effect

Algebraic Effects 101

operation

effect

```
effect reader {
  ask : () -> int
}
```

effect handler

implementation

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask () // 2
)
```

raise an effect

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + 10 // 11
)
```

Algebraic Effects 101

operation

effect

```
effect reader {
  ask : () -> int
}
```

effect handler

implementation

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + perform ask () // 2
)
```

raise an effect

```
handler {
  ask -> \x.\k. k 1
} (\_.
  perform ask () + 10 // 11
)
```

```
handler {
  ask -> \x.\k. k 2
} (\_.
  perform ask () + perform ask () // 4
)
```

Algebraic Effects 101

composable and modular computational effects

algebraic effects

define a family of operations

effect handlers

give semantics to operations

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

incr

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

incr

exception

But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

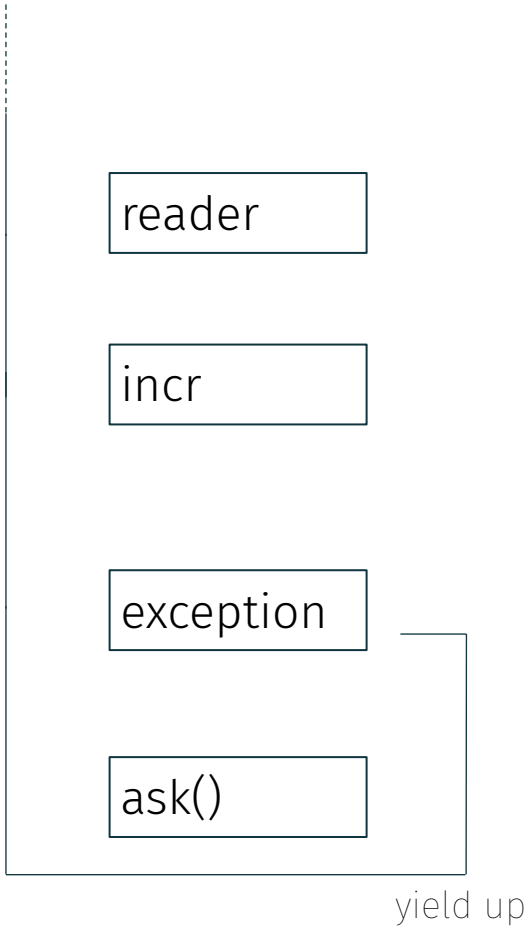
incr

exception

ask()

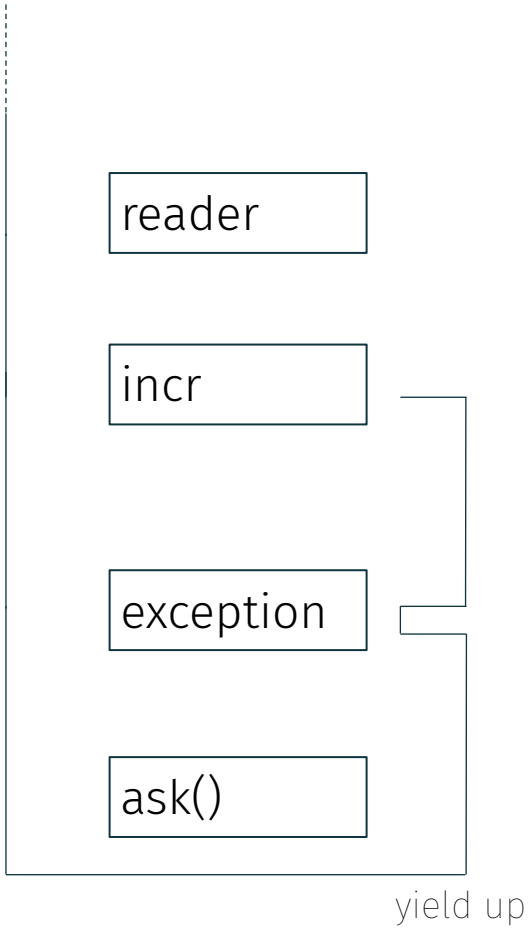
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



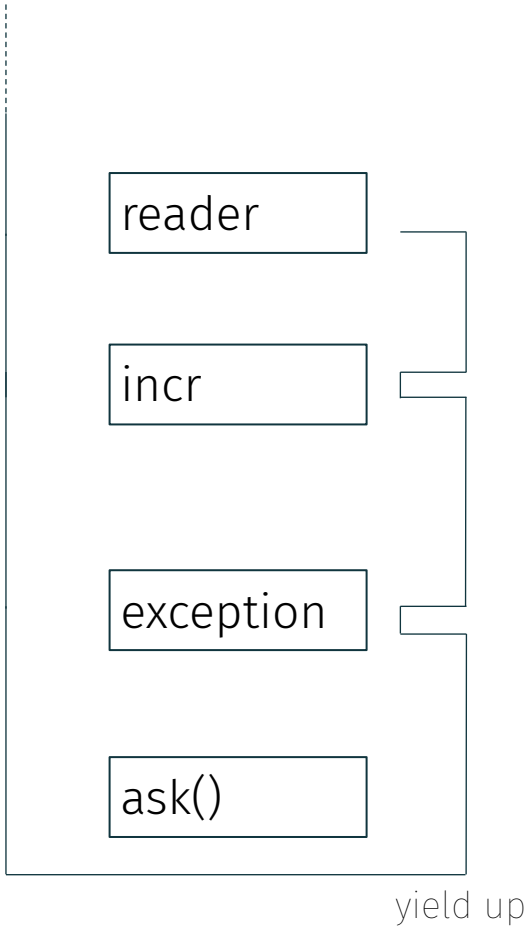
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



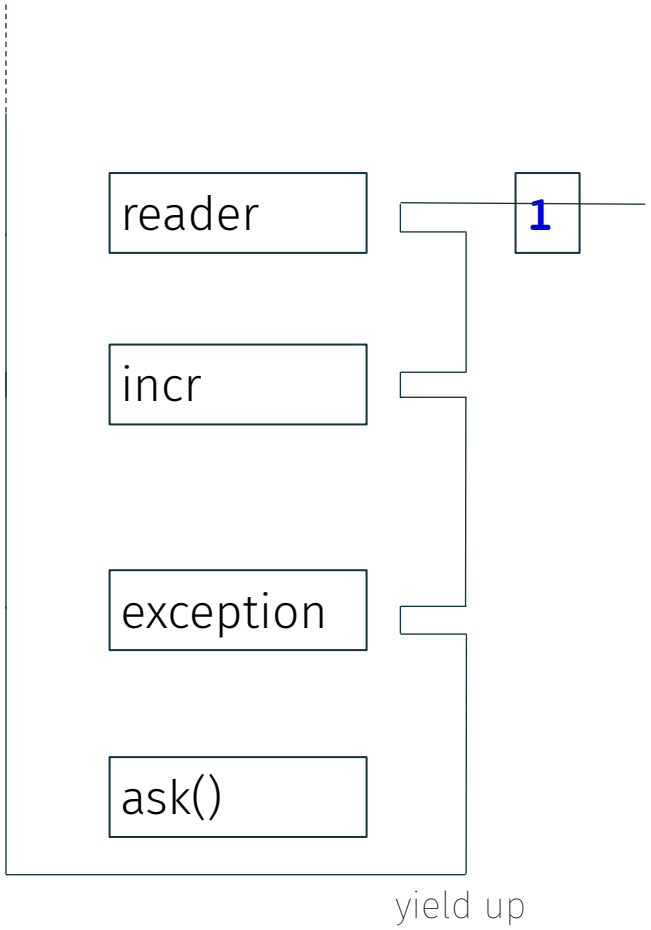
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



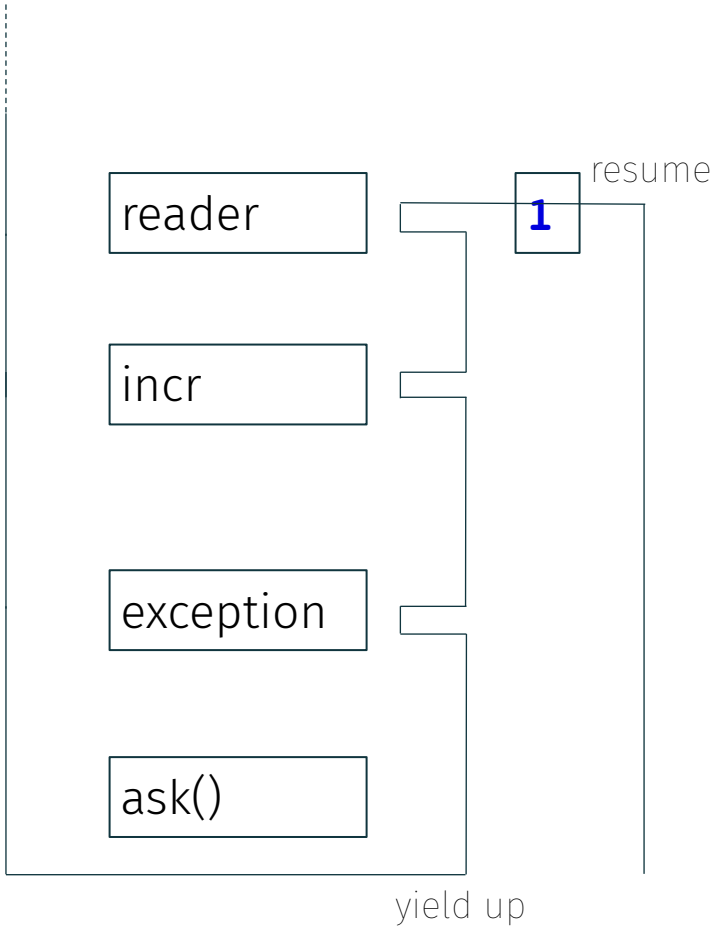
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



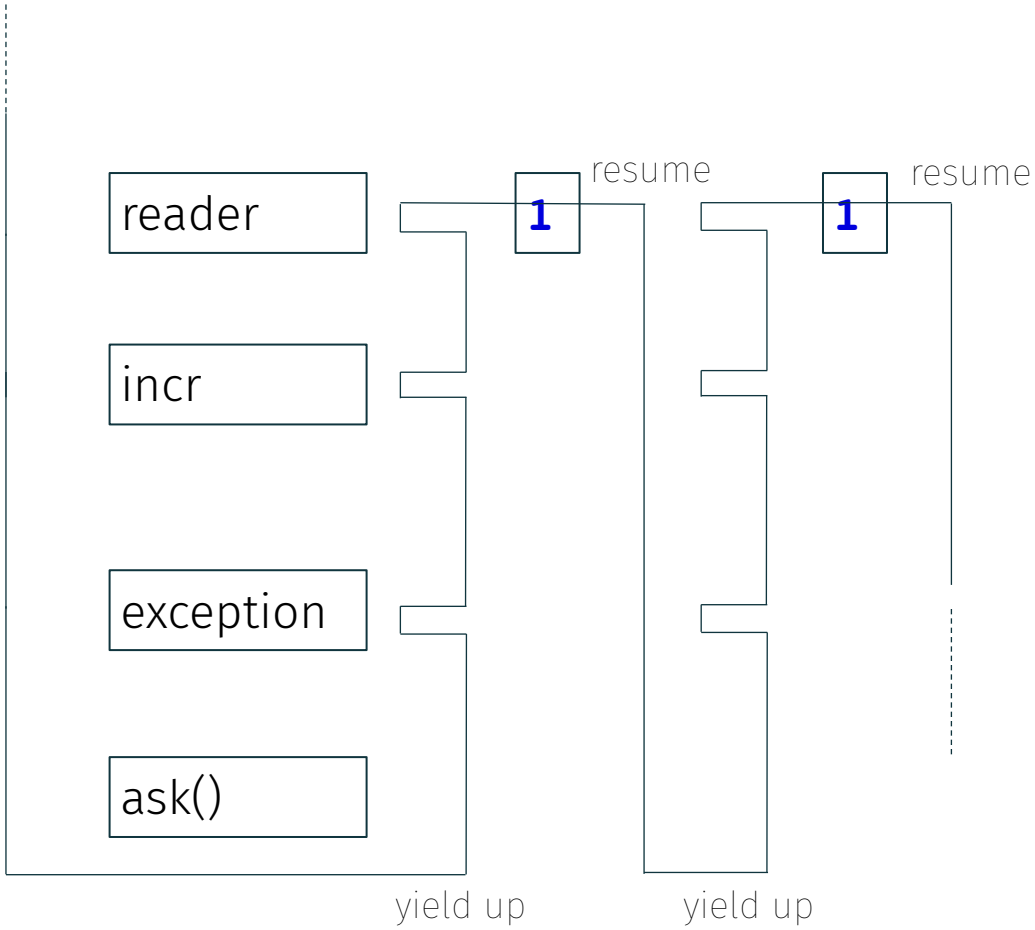
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



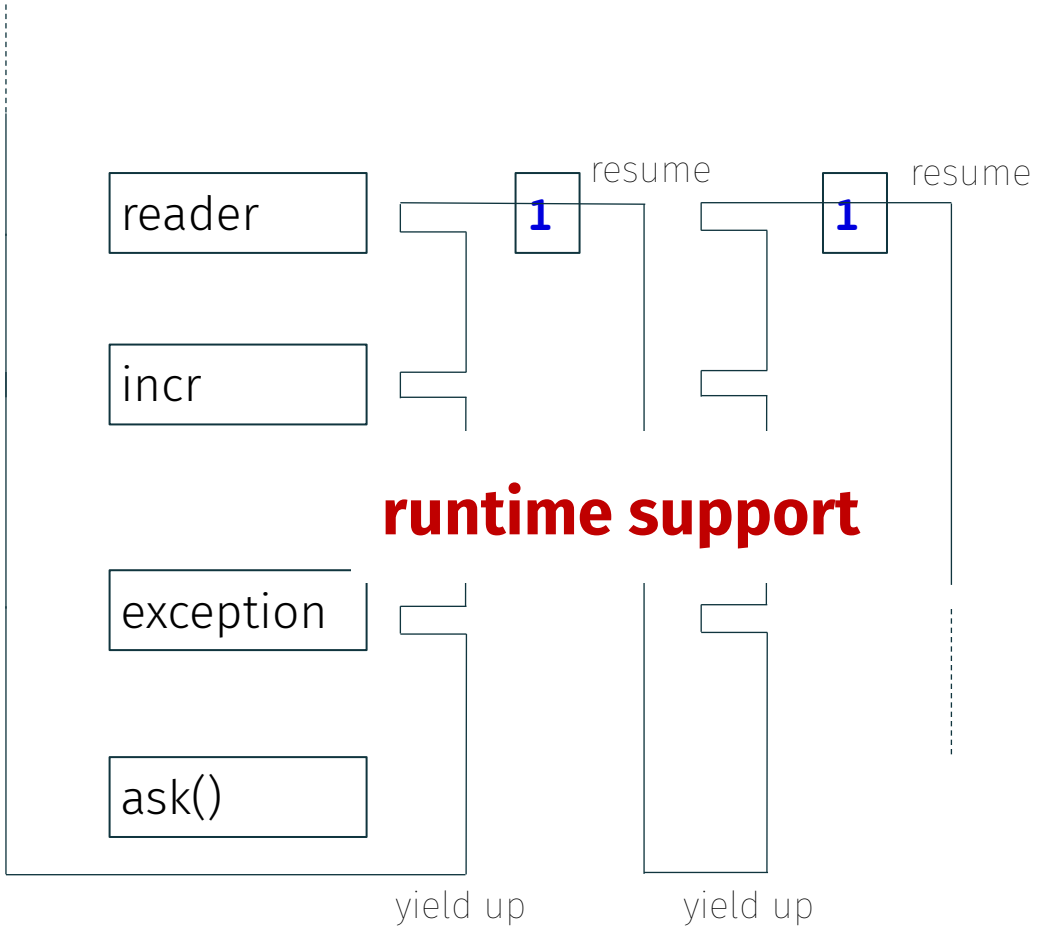
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



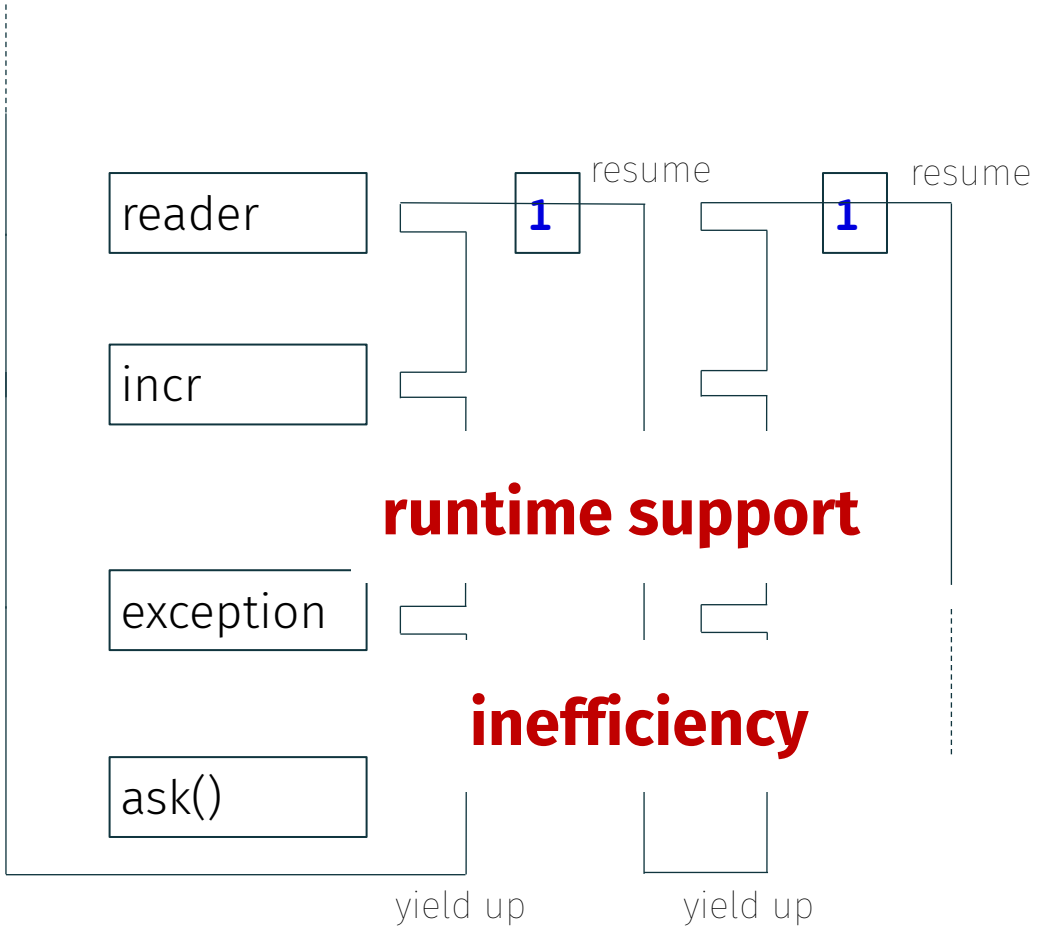
But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



But...

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Contribution

composable, modular, efficient, and easy-to-implement
computational effects

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

polymorphic
algebraic
effects
F^{ϵ}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

polymorphic
algebraic
effects
F^{ϵ}

polymorphic
lambda
calculus
F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

polymorphic algebraic effects F^{ϵ}	polymorphic evidence calculus F^{ev}	polymorphic lambda calculus F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

polymorphic
algebraic
effects
F^{ϵ}

2. evidence-
passing
translation
$\rightsquigarrow$

polymorphic
evidence
calculus
F^{ev}

polymorphic
lambda
calculus
F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

1. scoped resumptions

polymorphic algebraic effects F^{ϵ}	2. evidence-passing translation $\rightsquigarrow$	polymorphic evidence calculus F^{ev}	polymorphic lambda calculus F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

1. scoped resumptions

polymorphic algebraic effects F^{ϵ}	2. evidence-passing translation $\rightsquigarrow$	polymorphic evidence calculus F^{ev}	3. monadic multi-prompt translation $\rightsquigarrow$	polymorphic lambda calculus F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

1. scoped resumptions

polymorphic algebraic effects F^{ϵ}

2. evidence-passing translation ⇝

polymorphic evidence calculus F^{ev}

3. monadic multi-prompt translation ⇝

polymorphic lambda calculus F^{v}

Contribution

composable, modular, efficient, and easy-to-implement
computational effects

1. scoped resumptions

polymorphic algebraic effects F^{ϵ}

2. evidence-passing translation ⇝

polymorphic evidence calculus F^{ev}

3. monadic multi-prompt translation ⇝

polymorphic lambda calculus F^{v}

Scoped Resumptions

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected

1

1

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected	actual
1	1
1	

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected	actual
1	1
1	2

```
f (handler h1 (\_. handler hevil (\_. e)))
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected	actual
1	1
1	2

```
f (handler h1 (\_. handler hevil (\_. e)))
```

```
hevil = { evil -> \x k. k }
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected	actual
1	1
1	2

```
f (handler h1 (\_. handler hevil (\_. e)))
```

```
h2 = { ask -> \x k. k 2 }
f = \k. handler h2 (\_. k ())
```

```
hevil = { evil -> \x k. k }
```

Scoped Resumptions

```
h1 = { ask -> \x k. k 1 }
```

```
e = perform ask ();
    perform evil ();
    perform ask ()
```

expected	actual
1	1
1	2

```
f (handler h1 (\_. handler hevil (\_. e)))
```

```
h2 = { ask -> \x k. k 2 }
f = \k. handler h2 (\_. k ())
```

```
hevil = { evil -> \x k. k }
```

Scoped Resumptions

- Resumptions can only be *resumed* in the same handler context as *captured*; thus the example is rejected.
- We believe that all important effect handlers in practice can be defined in terms of scoped resumptions
- Implemented as a dynamic check, called `guard`

Evidence Passing

Evidence Passing

a vector of handlers is passed down as an implicit parameter to all operation invocations

Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

incr

exception

ask()

ask()

Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

reader

incr

exception

ask()

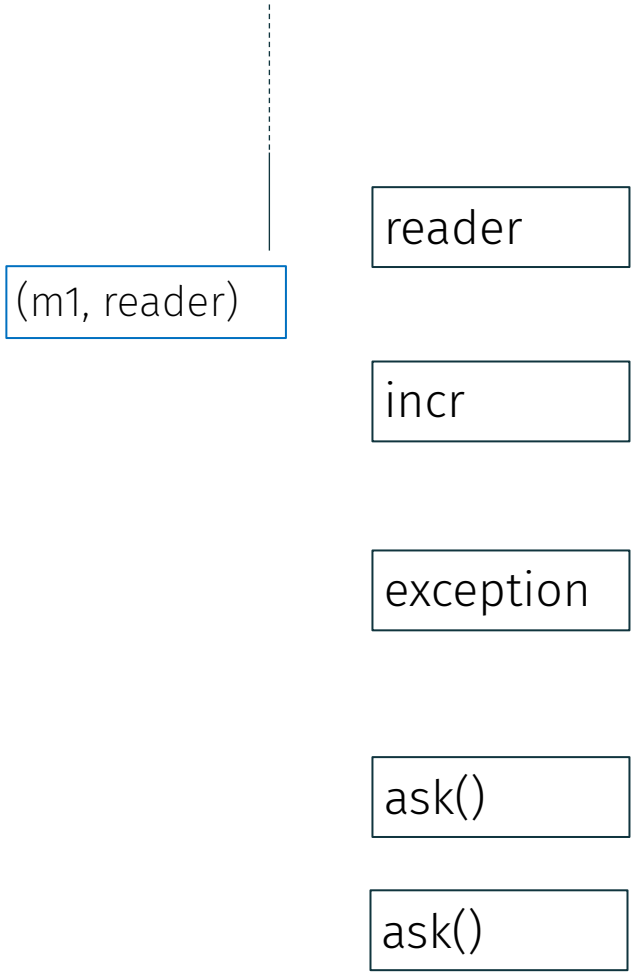
ask()

Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```

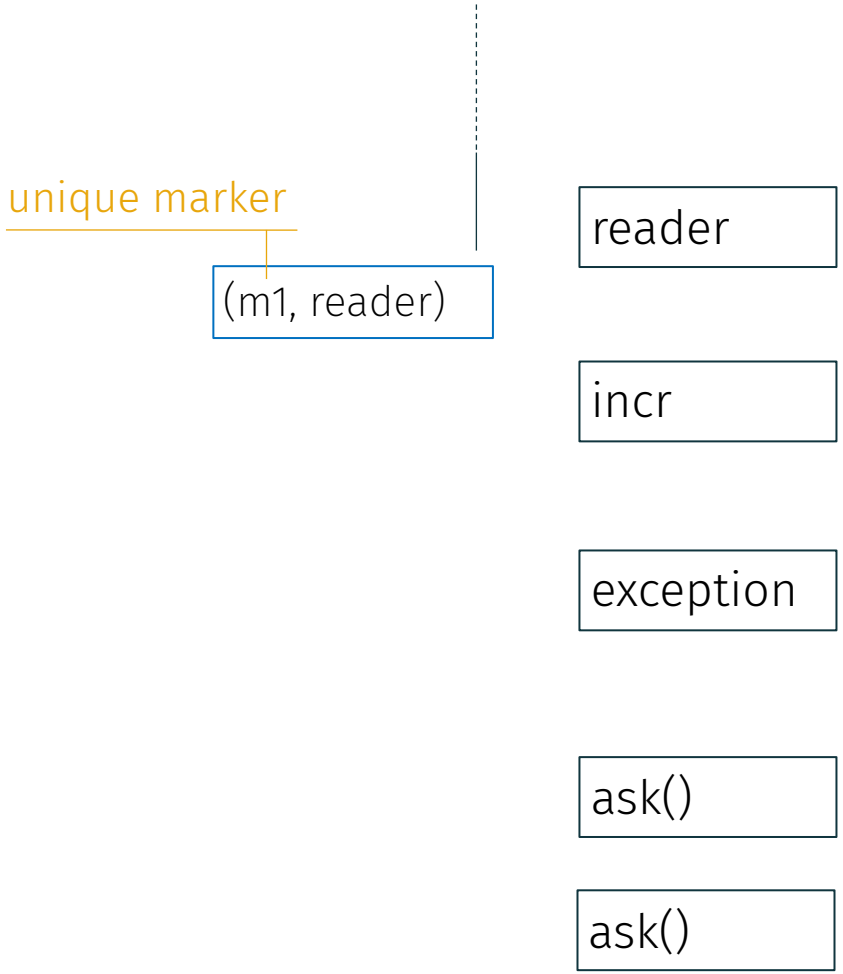
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



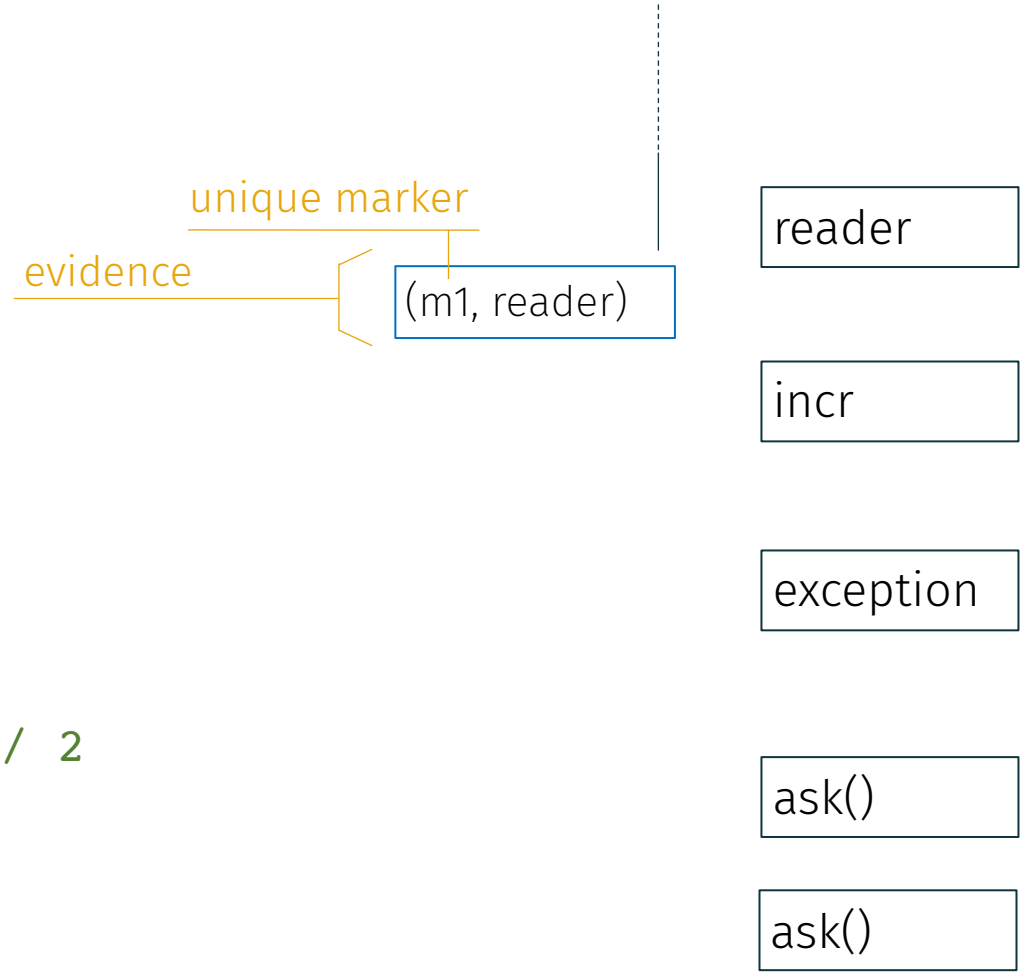
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



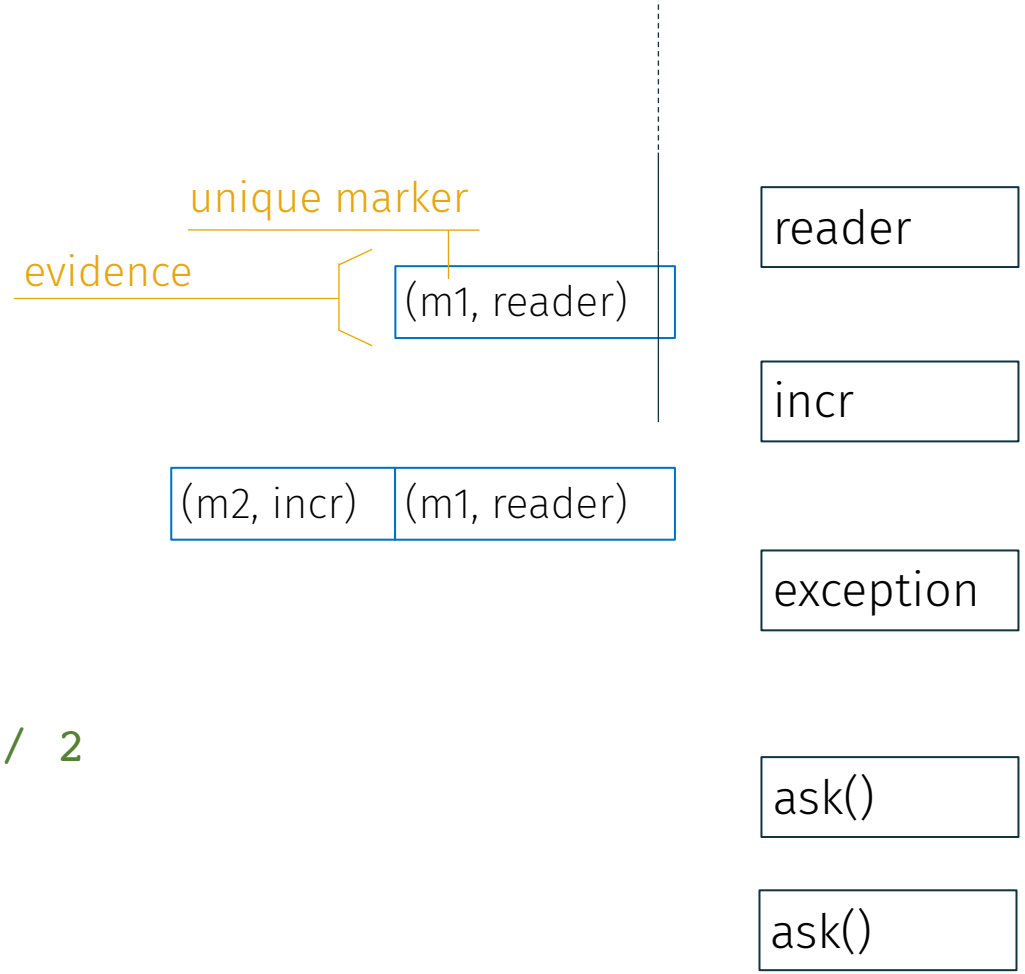
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



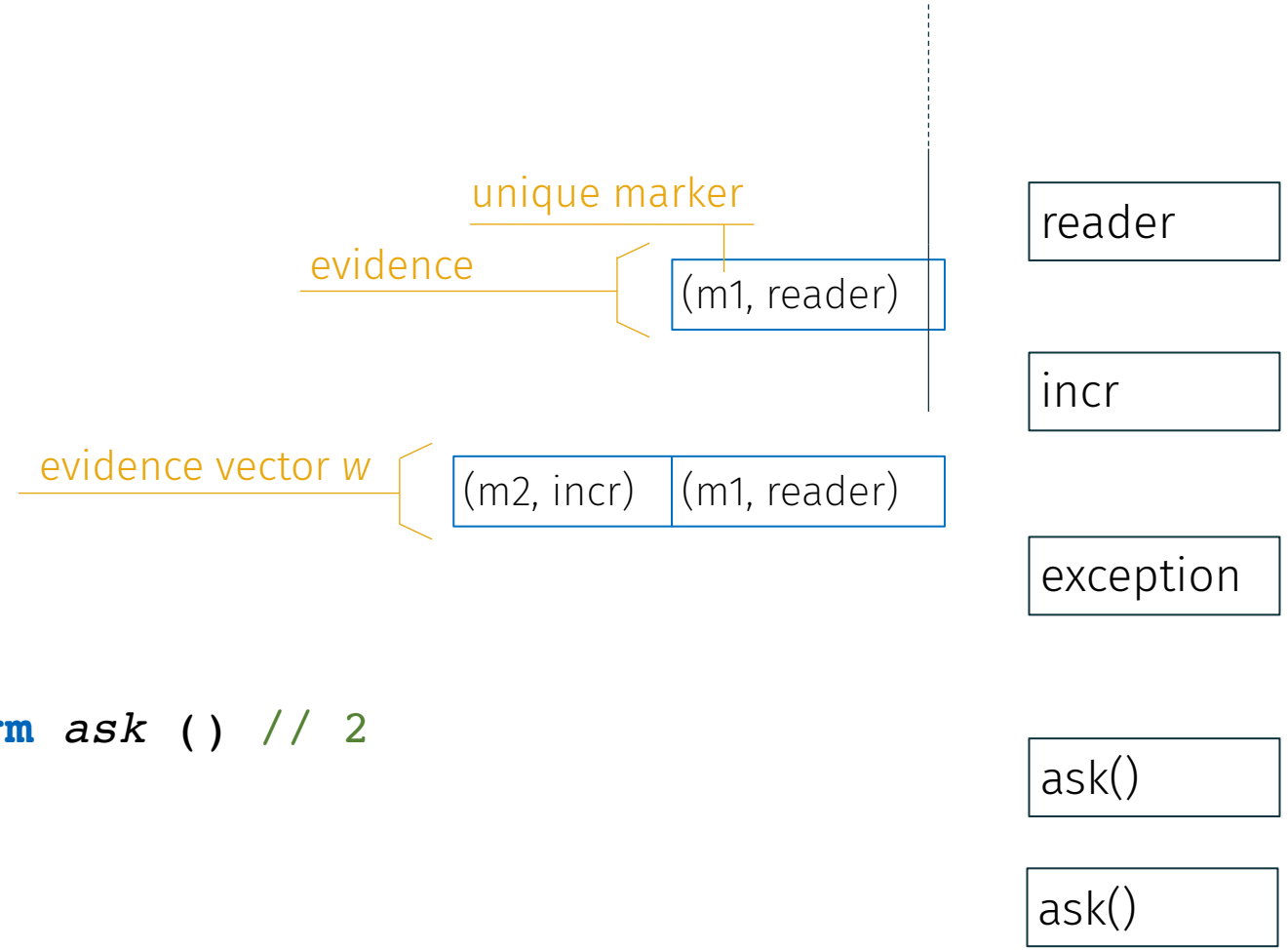
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



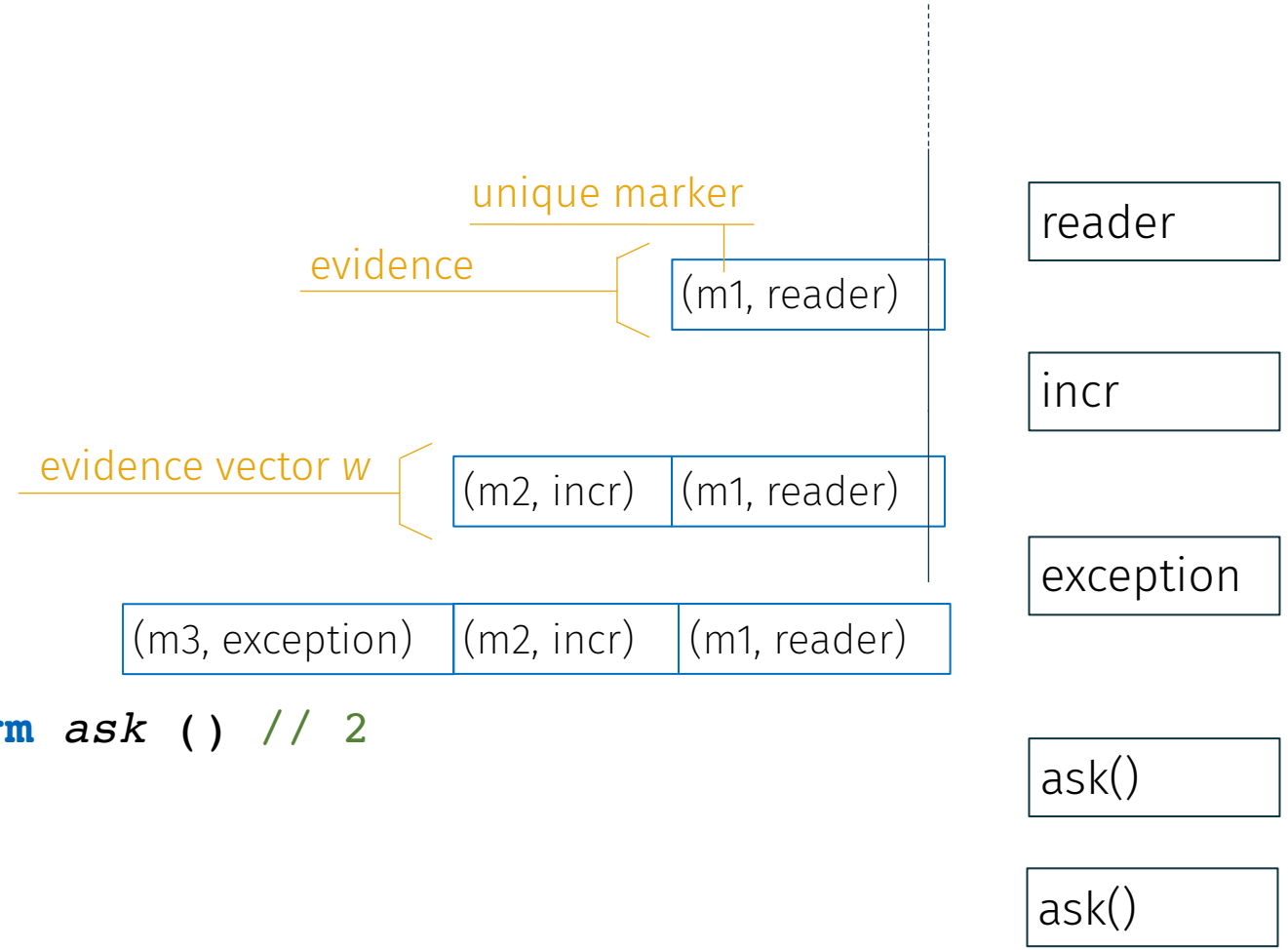
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



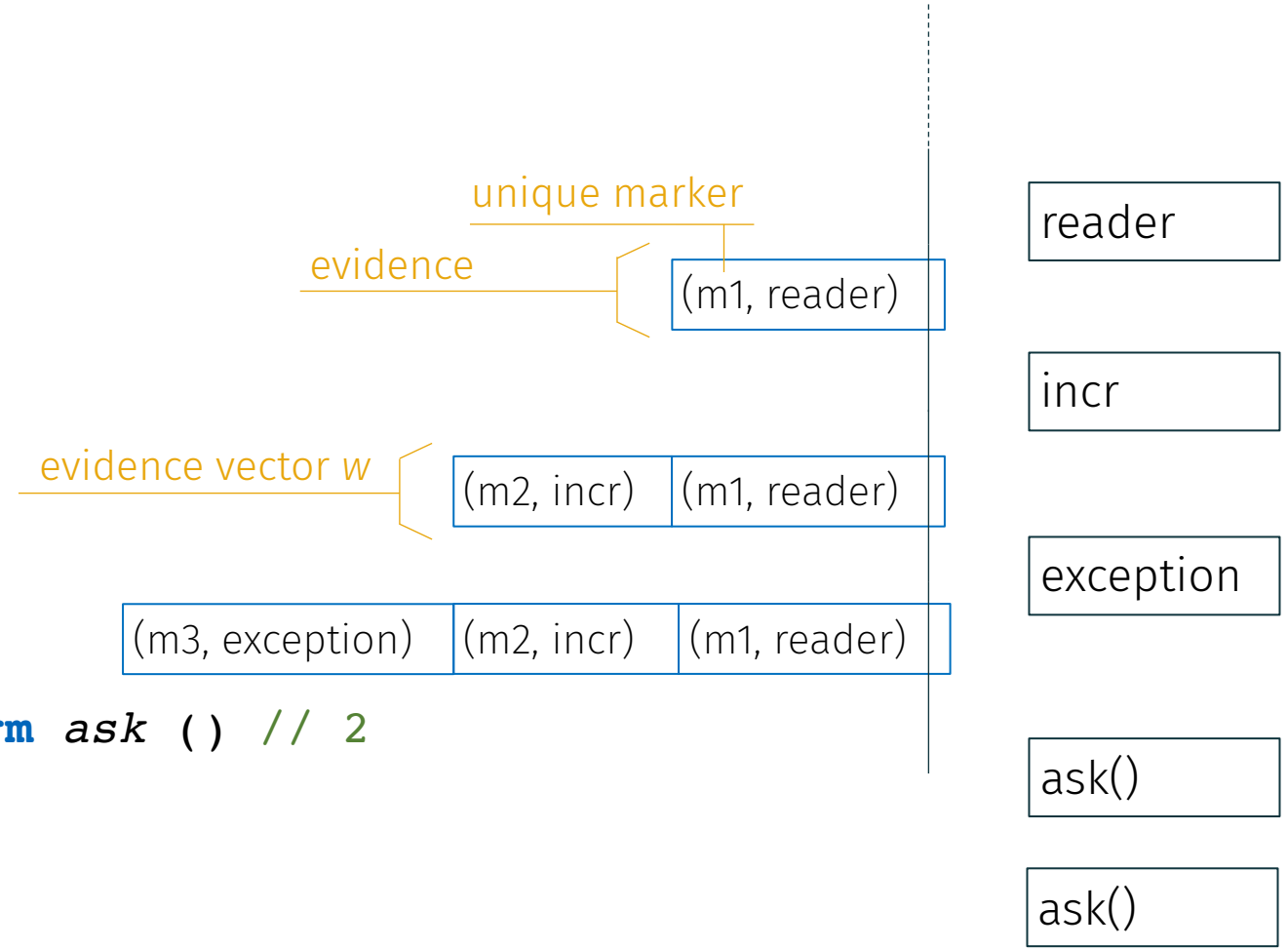
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



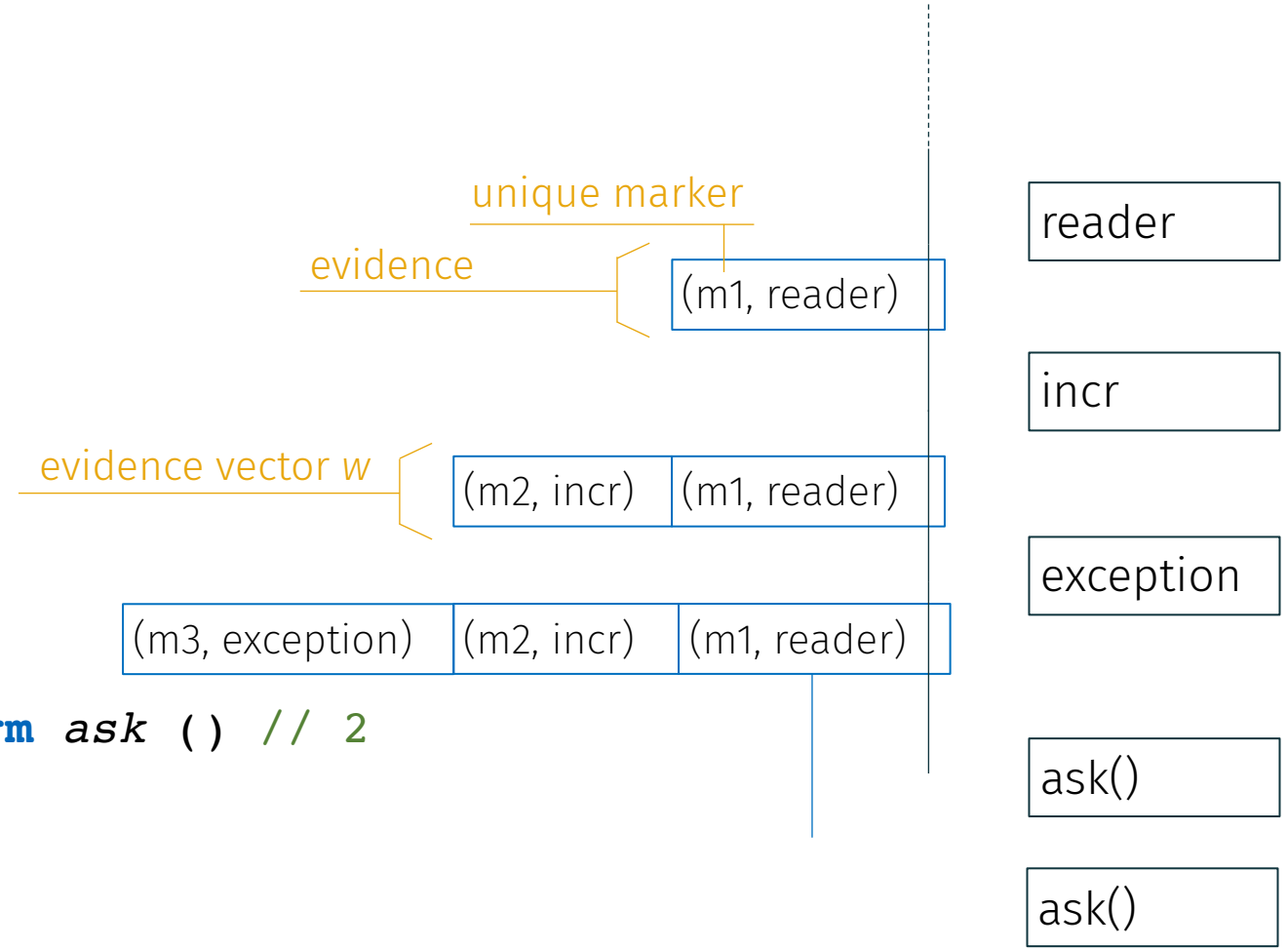
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



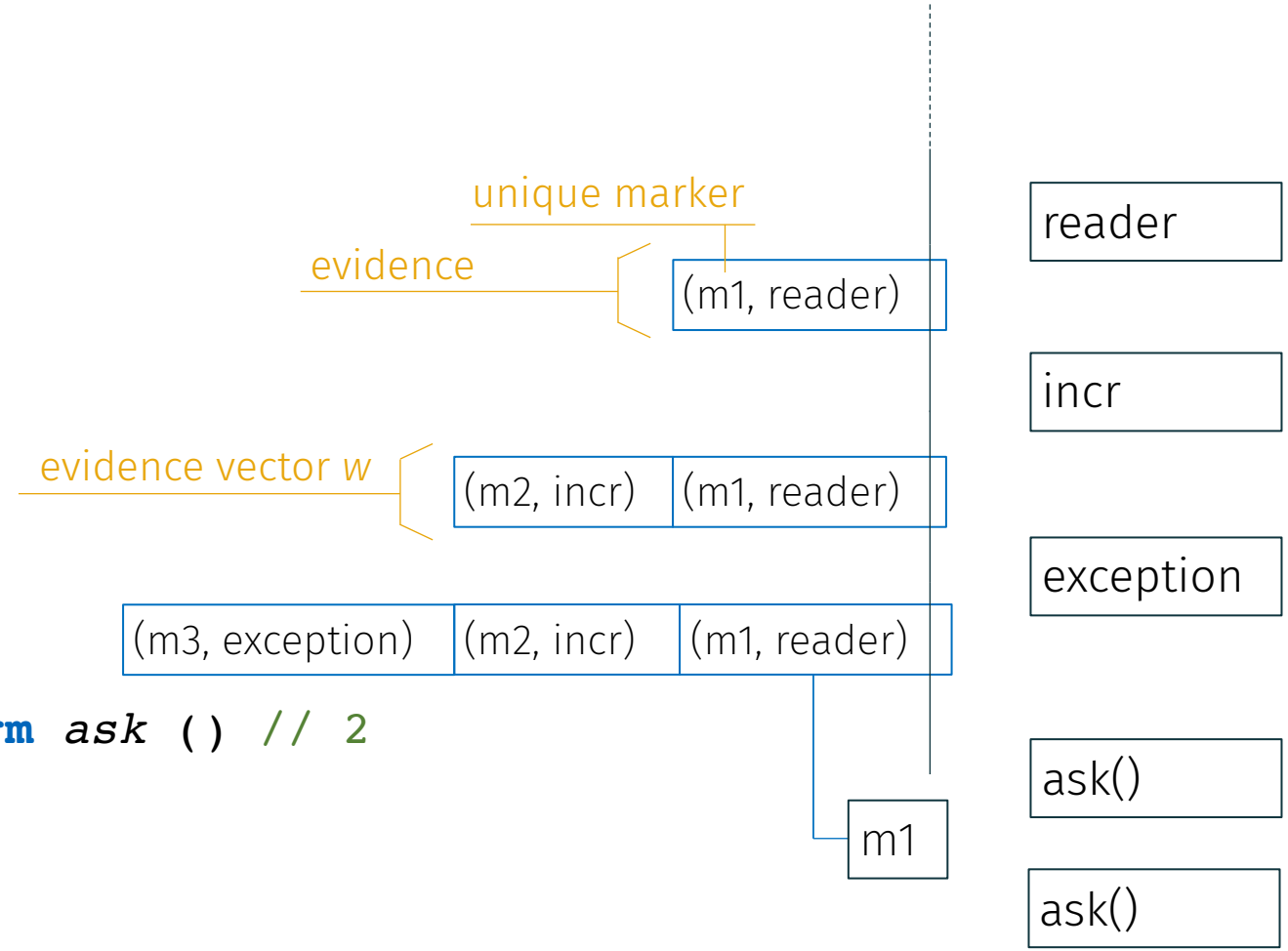
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



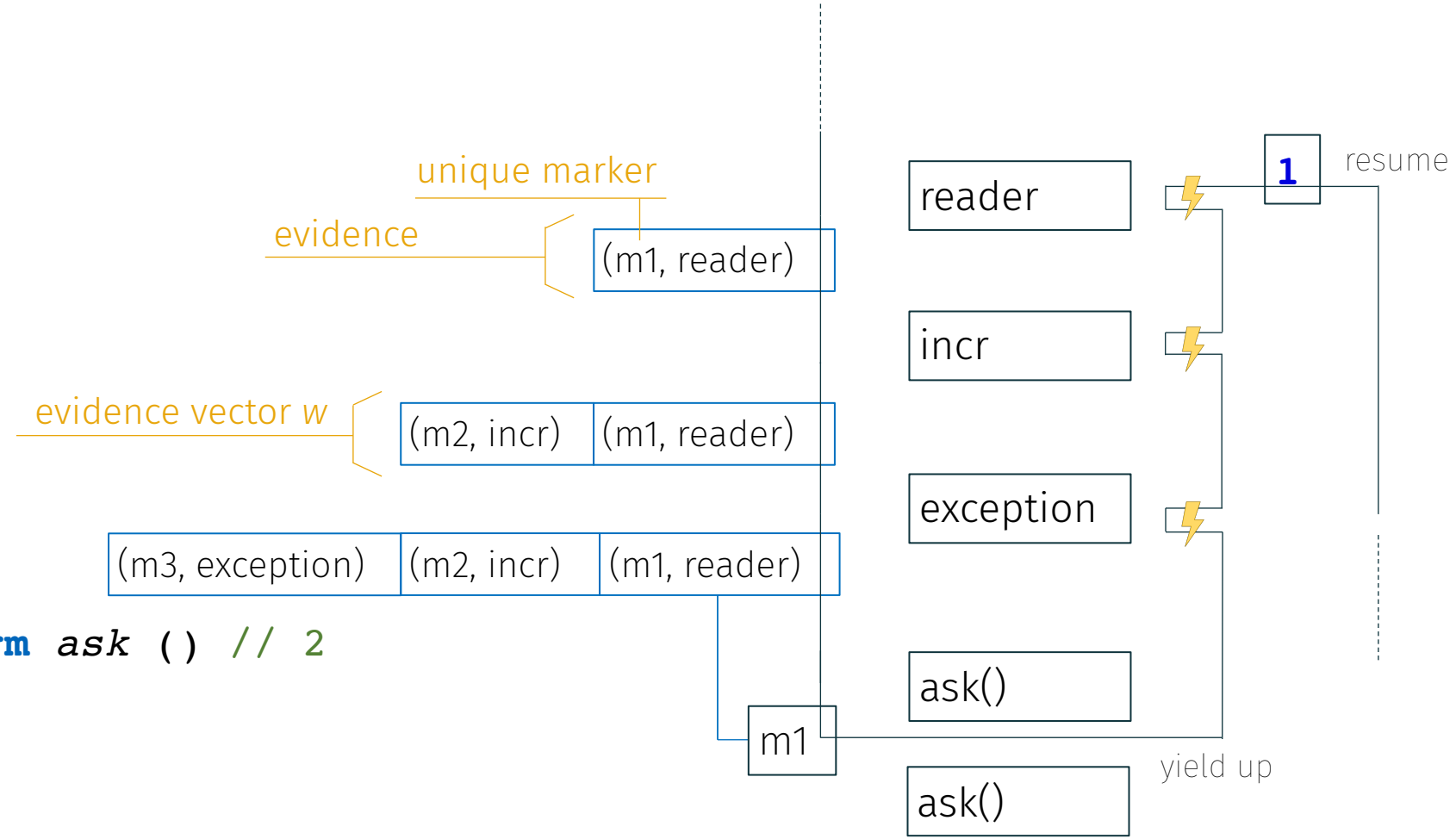
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



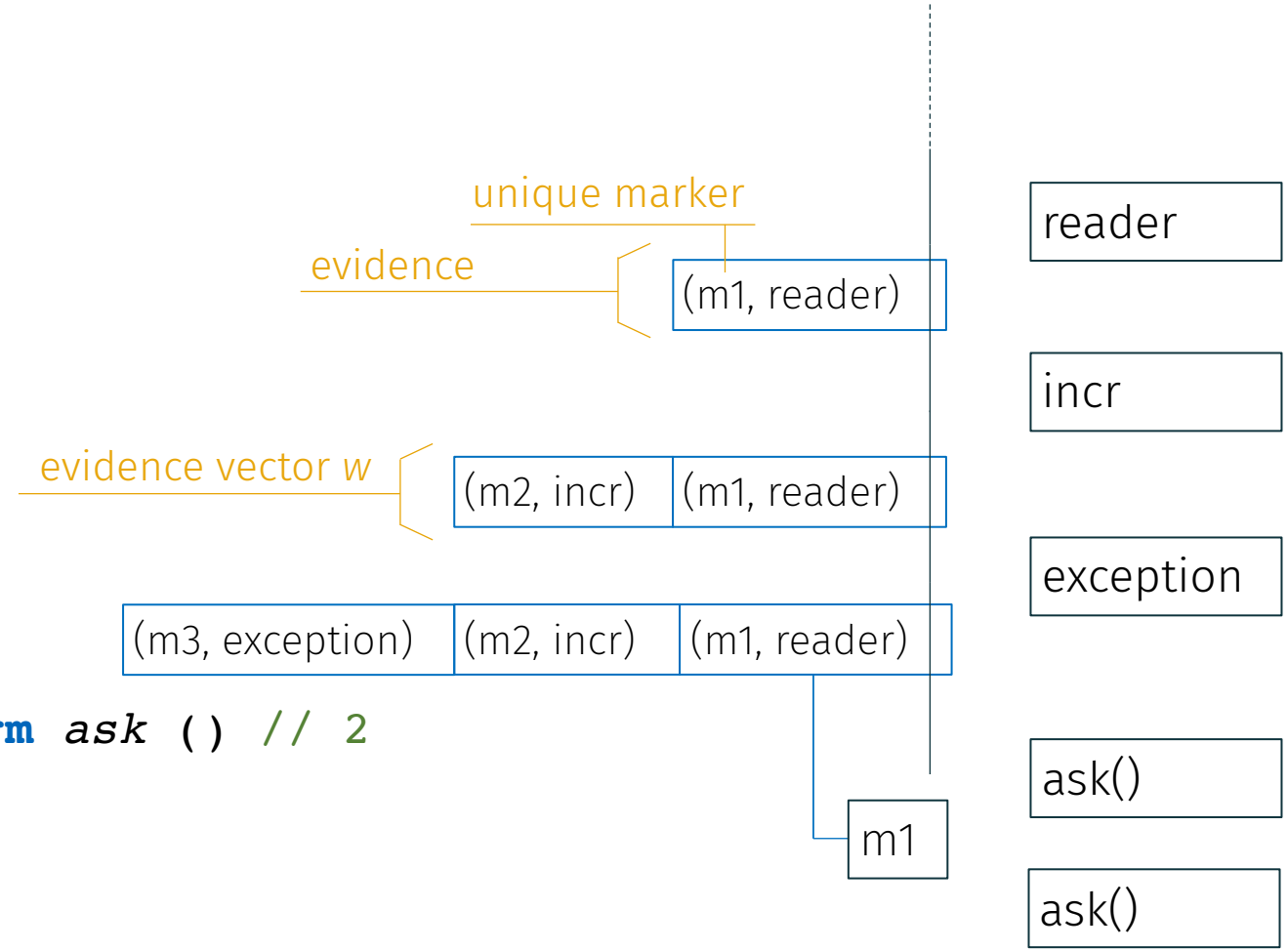
Evidence Passing

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

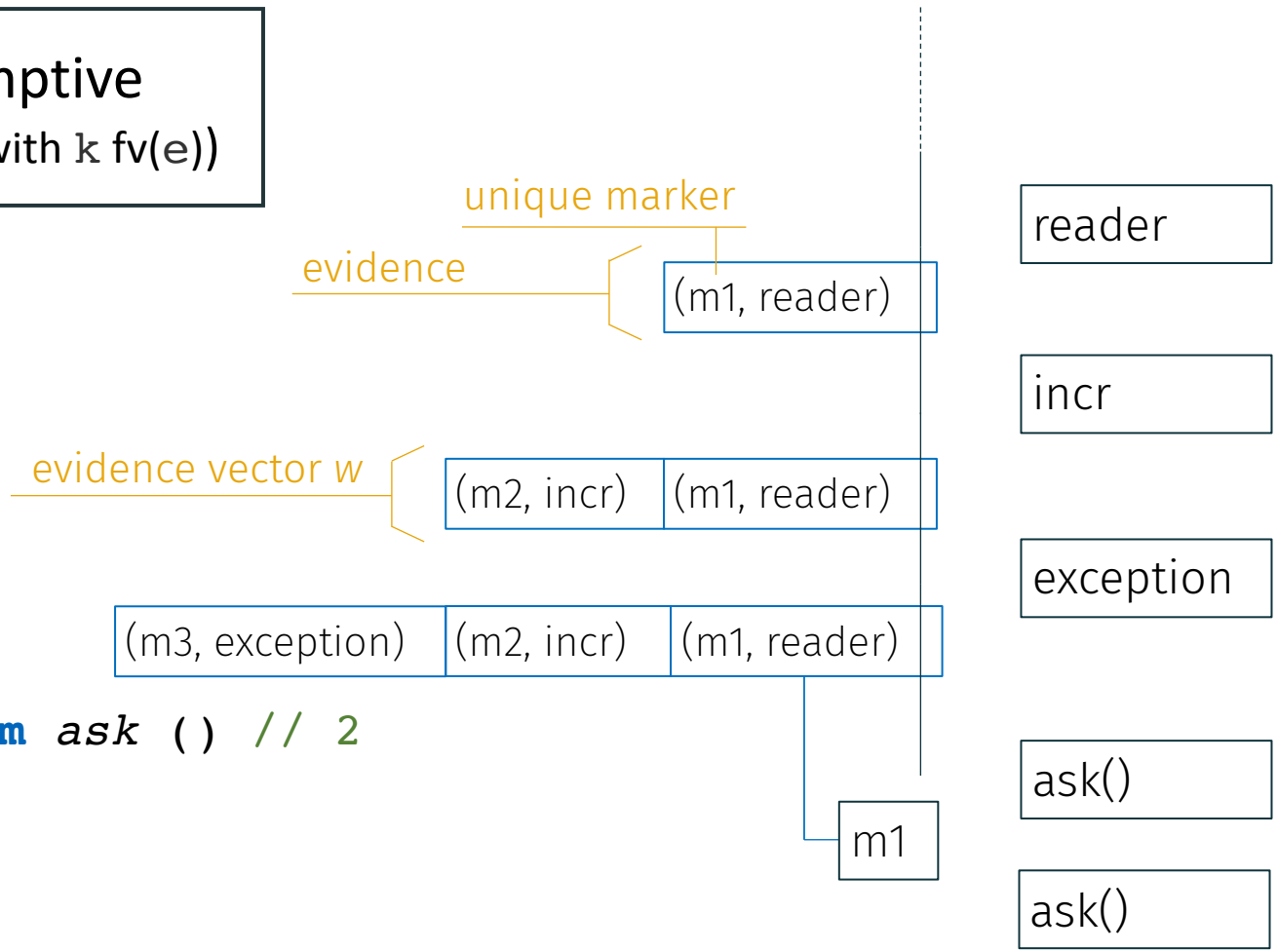
```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

tail-resumptive
(`\x.\k. k e` with `k` fv(e))

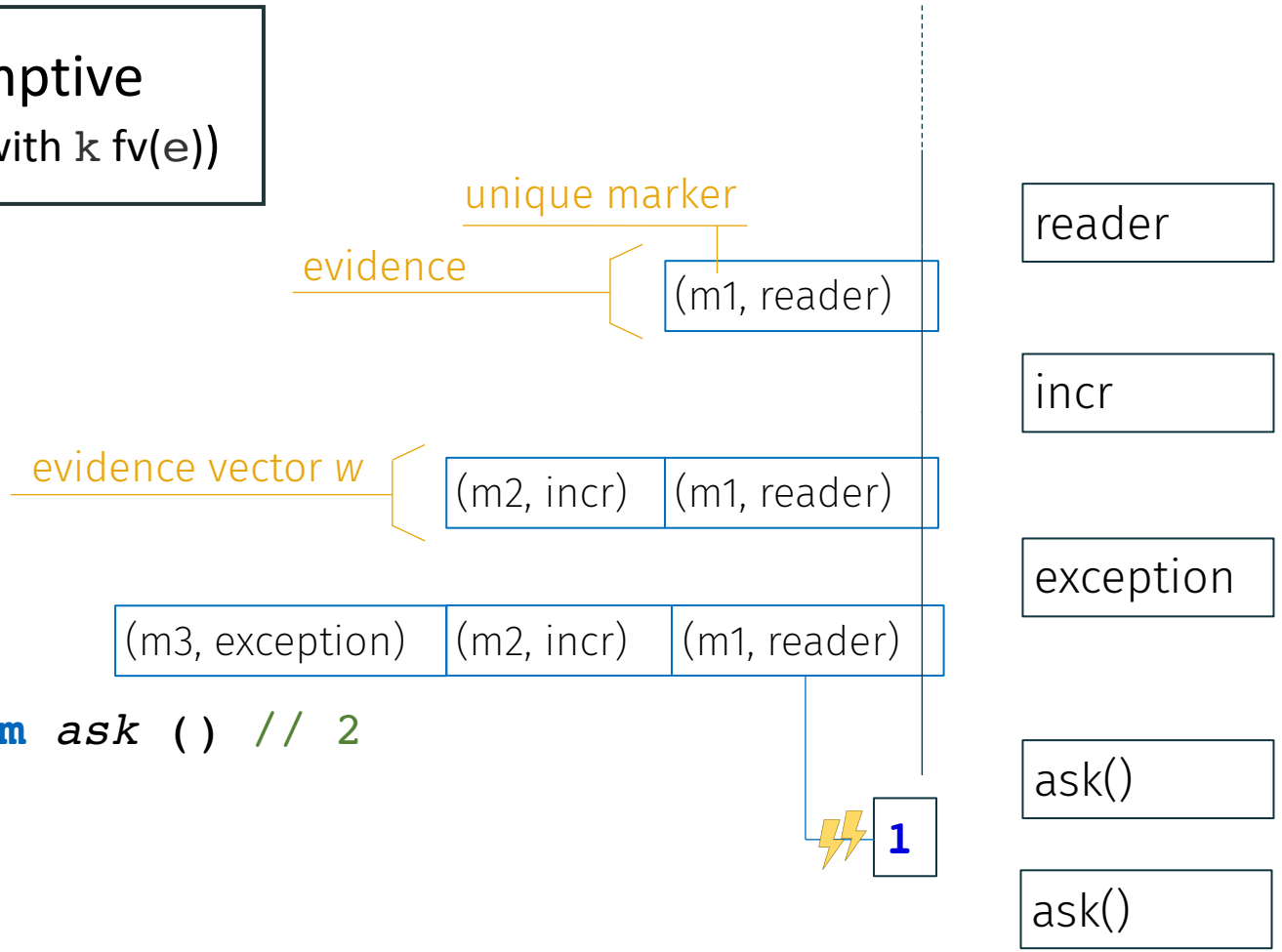
```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

tail-resumptive
(`\x.\k. k e` with `k` fv(e))

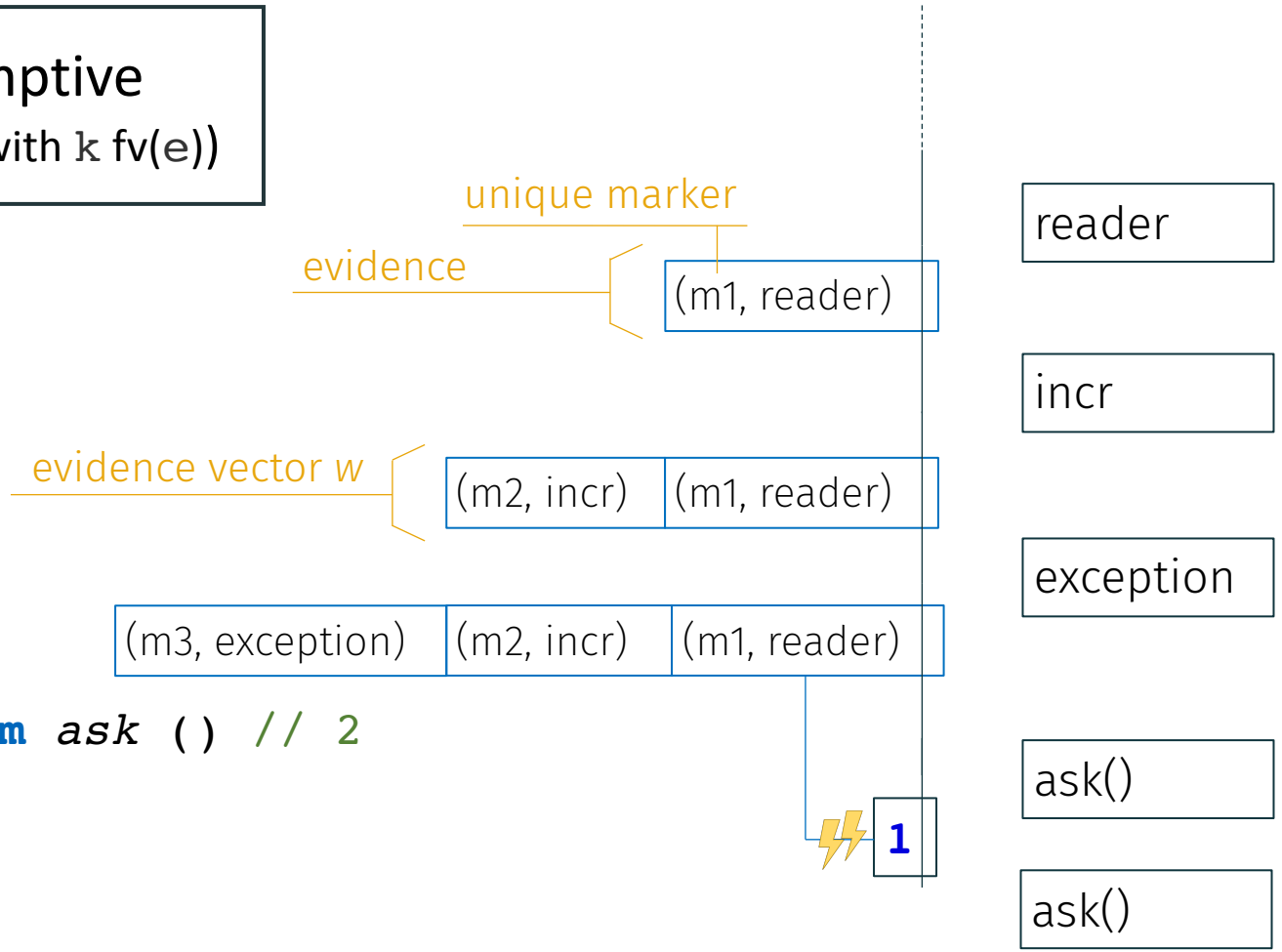
```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

tail-resumptive
(`\x.\k. k e` with `k` fv(e))

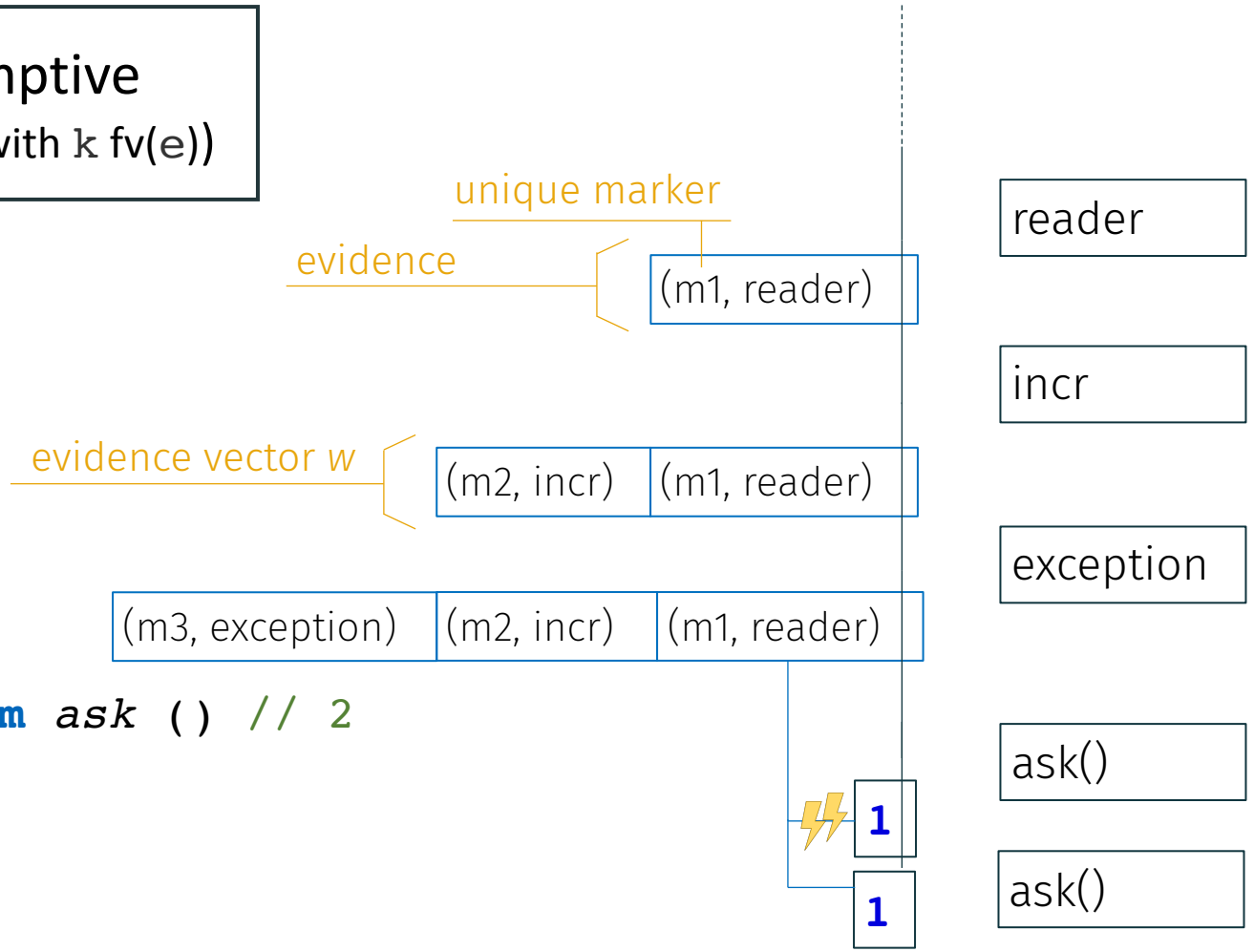
```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

tail-resumptive
(`\x.\k. k e` with `k` fv(e))

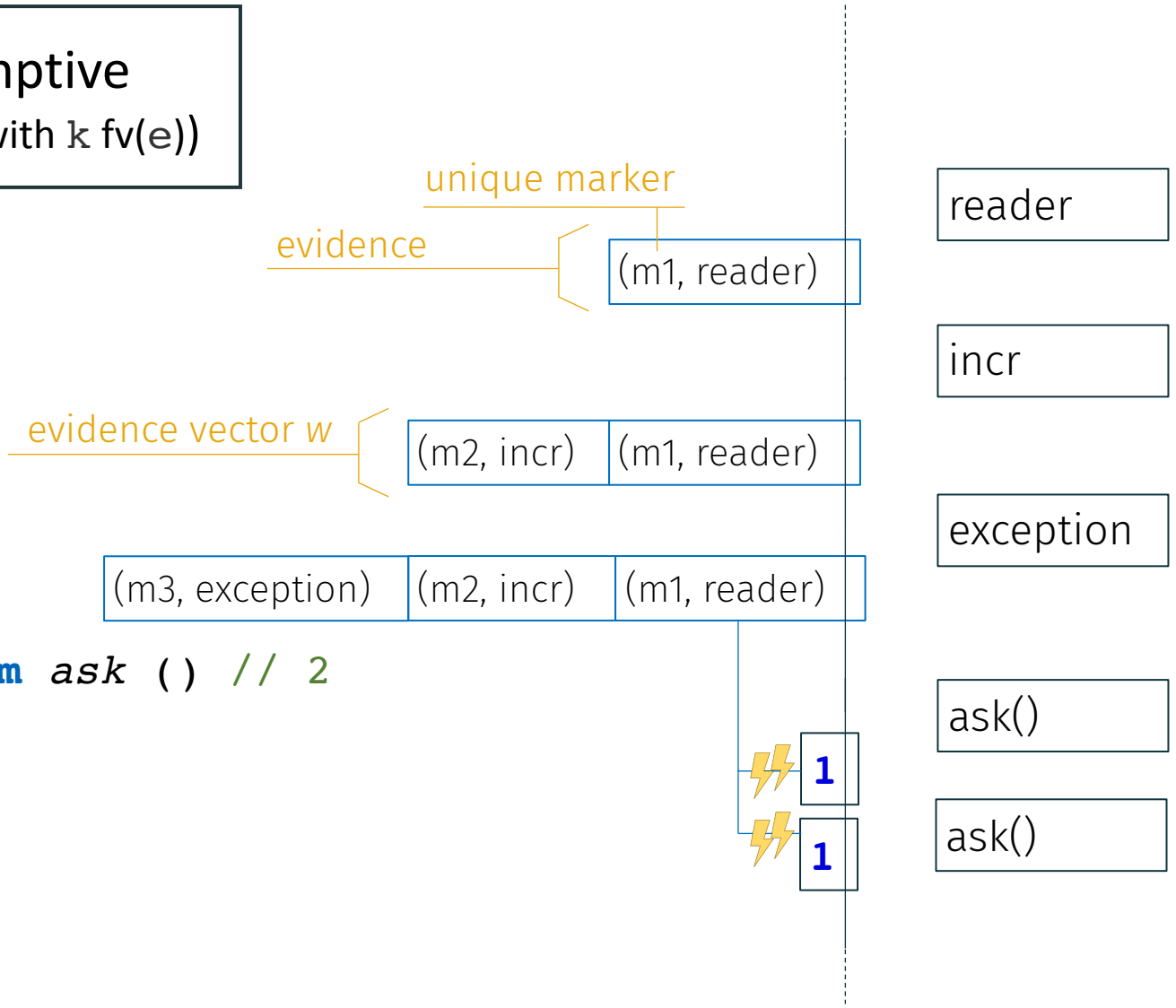
```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Evidence Passing

tail-resumptive
(`\x.\k. k e` with `k` fv(e))

```
handler {
  ask -> \x.\k. k 1
} (\_.
handler {
  incr -> \x.\k. 1 + k ()
} (\_.
handler {
  fail -> \x.\k. 3
} (\_.
  perform ask () + perform ask () // 2
)))
```



Efficiency

Theorem 5. (*Evidence Correspondence*)
If $\varnothing; \langle\!\langle\rangle\!\rangle \Vdash \mathsf{E}[\mathit{perform}\ op\ \overline{\sigma}\ w\ v] : \sigma \mid \langle\rangle$ then E has the form $\mathsf{E}_1 \cdot \mathsf{handle}^{w'}_{m}\ h \cdot \mathsf{E}_2$ with $op \in \Sigma(l)$, $op \notin \mathrm{bop}(\mathsf{E}_2)$, $op \longrightarrow f \in h$, and the evidence corresponds exactly to dynamic execution context such that $w.l = (m, h)$.

Efficiency

- Tail-resumptive operations (e.g., `ask`) can evaluate *in-place*
- Non tail-resumptive operations (e.g., `incr`) yield up, with *locally* deciding which *marker* to yield to

Monadic Multi-prompt Translation

$$\mathsf{handle}_m\ h\ \mathsf{E}[\mathsf{perform}\ op\ (m, h)\ v]$$

$$\rightsquigarrow$$

$$\mathsf{prompt}_m\ \mathsf{E}[\mathsf{yield}_m\ (\lambda k.\ (h.op)\ v\ k)]$$

Monadic Multi-prompt Translation

$$\mathsf{handle}_m\ h\ \mathsf{E}[\mathsf{perform}\ op\ (m, h)\ v]$$

$$\rightsquigarrow$$

$$\mathsf{prompt}_m\ \mathsf{E}[\mathsf{yield}_m\ (\lambda k.\ (h.op)\ v\ k)]$$

[Gunter, Rémy, and Riecke 1995]

Monadic Multi-prompt Translation

$$\mathsf{handle}_m\ h\ \mathsf{E}[\mathsf{perform}\ op\ (m, h)\ v]$$

$$\rightsquigarrow$$

$$\mathsf{prompt}_m\ \mathsf{E}[\mathsf{yield}_m\ (\lambda k.\ (h.op)\ v\ k)]$$

[Gunter, Rémy, and Riecke 1995]

Monadic Multi-prompt Translation

$$\mathsf{handle}_m\ \cancel{h}\ \mathsf{E}[\mathsf{perform}\ op\ (m, h)\ v]$$

$$\rightsquigarrow$$

$$\mathsf{prompt}_m\ \mathsf{E}[\mathsf{yield}_m\ (\lambda k.\ (h.op)\ v\ k)]$$

[Gunter, Rémy, and Riecke 1995]

Monadic Multi-prompt Translation

$$\mathsf{handle}_m\ \cancel{h}\ \mathsf{E}[\mathsf{perform}\ op\ (m, h)\ v]$$

$$\rightsquigarrow$$

$$\mathsf{prompt}_m\ \mathsf{E}[\mathsf{yield}_m\ (\lambda k.\ (h.op)\ v\ k)]$$

[Gunter, Rémy, and Riecke 1995]

implementations
based on
dynamic search for the handler

[Dolan et al. 2015; Leijen 2014;
Lindley et al. 2017]

implementations
using
multi-prompt delimited control

[Biernacki et al. 2019;
Brachthäuser and Schuster 2017;
Zhang and Myers 2019]

Implementation based on polymorphic lambda calculus

multi-prompt monad

$$
\begin{aligned}
&\text{data mon } \mu\ \alpha\ = \\
&\quad \mid \text{pure} : \alpha \to \text{mon } \mu\ \alpha \\
&\quad \mid \text{yield} : \forall \beta\ r\ \mu'.\ \text{marker } \mu'\ r \to (\text{evv } \mu' \to (\text{evv } \mu' \to \beta \to \text{mon } \mu'\ r) \to \text{mon } \mu'\ r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to (\text{mon } \mu\ \beta \to \text{mon } \mu\ \alpha) \to \text{mon } \mu\ \alpha
\end{aligned}
$$

Implementation based on polymorphic lambda calculus

multi-prompt monad

$$
\begin{array}{l}
\text{data mon } \mu\ \alpha\ = \\
\quad \mid \text{pure} : \alpha \to \text{mon } \mu\ \alpha \\
\quad \mid \text{yield} : \forall \beta\ r\ \mu'.\ \text{marker } \mu'\ r \to (\text{evv } \mu' \to (\text{evv } \mu' \to \beta \to \text{mon } \mu'\ r) \to \text{mon } \mu'\ r) \\
\qquad\qquad\qquad\qquad\qquad\qquad \to (\text{mon } \mu\ \beta \to \text{mon } \mu\ \alpha) \to \text{mon } \mu\ \alpha
\end{array}
$$

✓ No special runtime support needed

Implementation based on polymorphic lambda calculus

multi-prompt monad

$$
\begin{array}{l}
\text{data mon } \mu\ \alpha\ = \\
\quad \mid \text{pure} : \alpha \to \text{mon } \mu\ \alpha \\
\quad \mid \text{yield} : \forall \beta\ r\ \mu'.\ \text{marker } \mu'\ r \to (\text{evv } \mu' \to (\text{evv } \mu' \to \beta \to \text{mon } \mu'\ r) \to \text{mon } \mu'\ r) \\
\qquad\qquad\qquad\qquad\qquad\qquad \to (\text{mon } \mu\ \beta \to \text{mon } \mu\ \alpha) \to \text{mon } \mu\ \alpha
\end{array}
$$

- ✓ No special runtime support needed
- ✓ Advanced compilation strategies can be used (e.g., reference counting [Ullrich and Moura 2019])

Summary

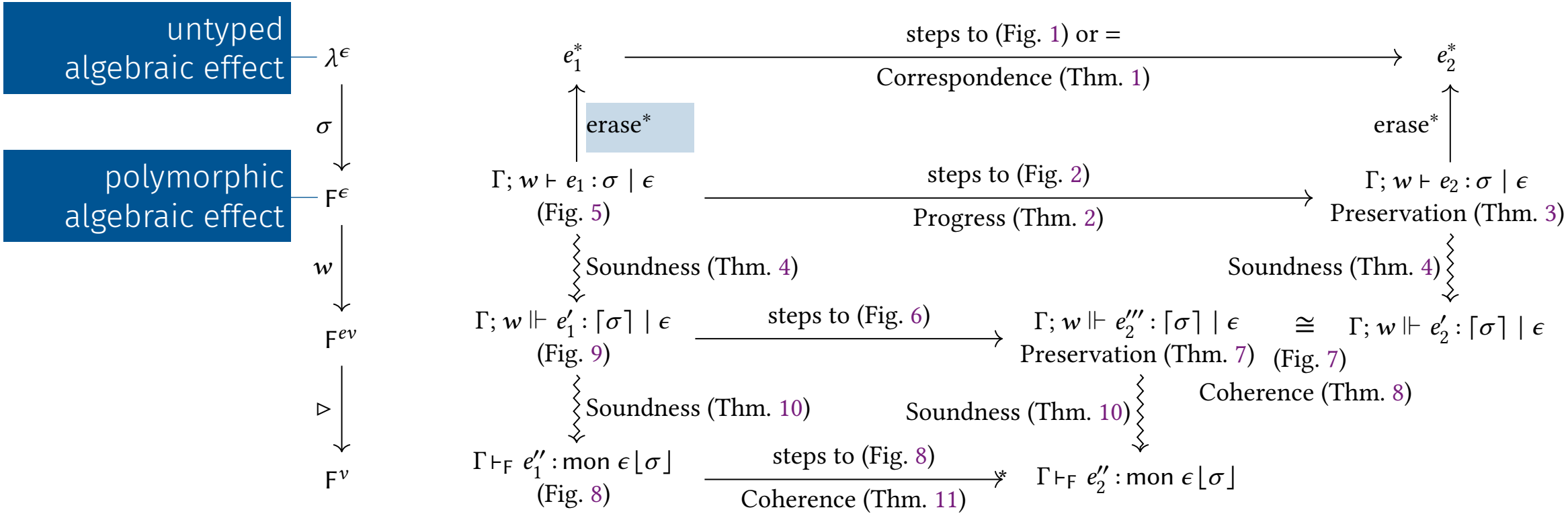
$$
\begin{tikzcd}
\lambda^{\epsilon} \arrow[d, "\sigma"'] & e_1^* \arrow[rrr, "\text{steps to (Fig. 1) or } =", "\text{Correspondence (Thm. 1)}"'] & & & e_2^* \\
\mathsf{F}^{\epsilon} \arrow[d, "w"'] & \Gamma; w \vdash e_1 : \sigma \mid \epsilon \ \text{(Fig. 5)} \arrow[u, "\text{erase}^*"'] \arrow[rrr, "\text{steps to (Fig. 2)}", "\text{Progress (Thm. 2)}"'] \arrow[d, squiggly, "\text{Soundness (Thm. 4)}"] & & & \Gamma; w \vdash e_2 : \sigma \mid \epsilon \ \text{Preservation (Thm. 3)} \arrow[u, "\text{erase}^*"] \arrow[d, squiggly, "\text{Soundness (Thm. 4)}"'] \\
\mathsf{F}^{ev} \arrow[d, "\triangleright"'] & \Gamma; w \Vdash e_1' : \lceil \sigma \rceil \mid \epsilon \ \text{(Fig. 9)} \arrow[rrr, "\text{steps to (Fig. 6)}"] \arrow[d, squiggly, "\text{Soundness (Thm. 10)}"] & & & \Gamma; w \Vdash e_2''' : \lceil \sigma \rceil \mid \epsilon \ \text{Preservation (Thm. 7)} \arrow[r, phantom, "\cong \ \text{(Fig. 7)}"] \arrow[d, squiggly, "\text{Soundness (Thm. 10)}"'] & \Gamma; w \Vdash e_2' : \lceil \sigma \rceil \mid \epsilon \arrow[l, squiggly, "\text{Coherence (Thm. 8)}"] \\
\mathsf{F}^{v} & \Gamma \vdash_{\mathsf{F}} e_1'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor \ \text{(Fig. 8)} \arrow[rrr, "\text{steps to (Fig. 8)}", "\text{Coherence (Thm. 11)}"'] & & & \Gamma \vdash_{\mathsf{F}} e_2'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor
\end{tikzcd}
$$

Summary

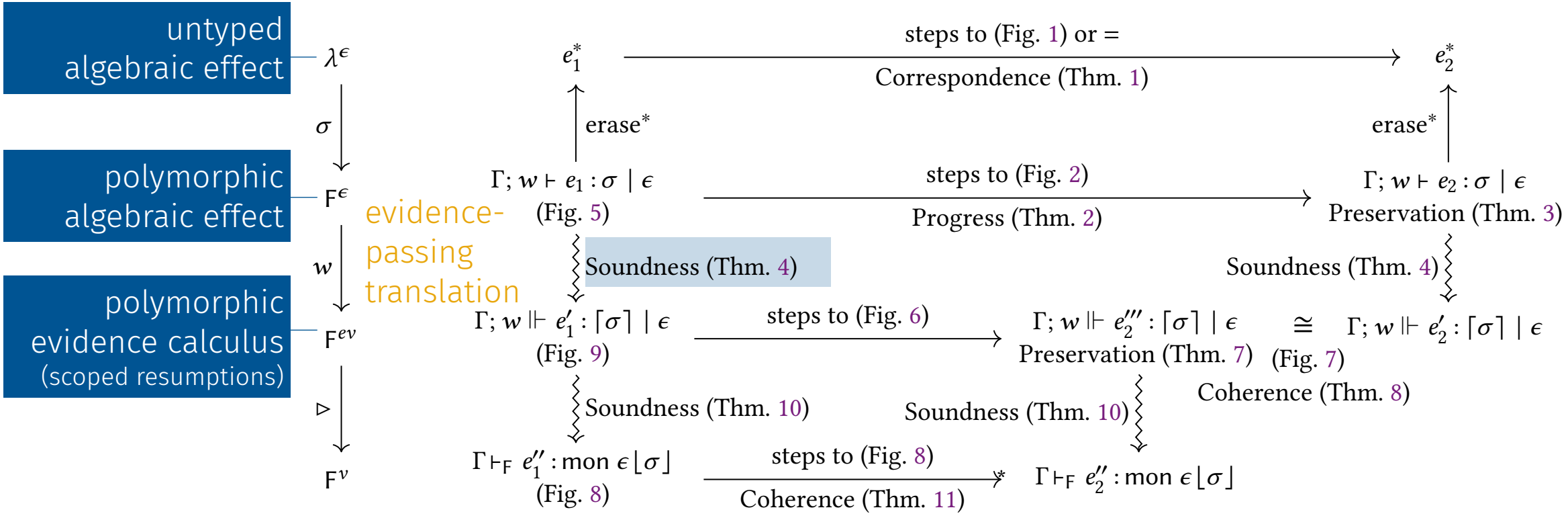
polymorphic algebraic effect

$$
\begin{tikzcd}
\lambda^{\epsilon} \arrow[d, "\sigma"'] & e_1^* \arrow[rrr, "\text{steps to (Fig. 1) or } =", "\text{Correspondence (Thm. 1)}"'] & & & e_2^* \\
\mathsf{F}^{\epsilon} \arrow[d, "w"'] & \Gamma; w \vdash e_1 : \sigma \mid \epsilon \ \text{(Fig. 5)} \arrow[u, "\text{erase}^*"'] \arrow[rrr, "\text{steps to (Fig. 2)}", "\text{Progress (Thm. 2)}"'] \arrow[d, squiggly, "\text{Soundness (Thm. 4)}"] & & & \Gamma; w \vdash e_2 : \sigma \mid \epsilon \ \text{Preservation (Thm. 3)} \arrow[u, "\text{erase}^*"] \arrow[d, squiggly, "\text{Soundness (Thm. 4)}"'] \\
\mathsf{F}^{ev} \arrow[d, "\triangleright"'] & \Gamma; w \Vdash e_1' : \lceil \sigma \rceil \mid \epsilon \ \text{(Fig. 9)} \arrow[rr, "\text{steps to (Fig. 6)}"] \arrow[d, squiggly, "\text{Soundness (Thm. 10)}"] & & \Gamma; w \Vdash e_2''' : \lceil \sigma \rceil \mid \epsilon \ \text{Preservation (Thm. 7)} \arrow[r, phantom, "\cong \ \text{(Fig. 7)}"] \arrow[d, squiggly, "\text{Soundness (Thm. 10)}"'] & \Gamma; w \Vdash e_2' : \lceil \sigma \rceil \mid \epsilon \\
\mathsf{F}^{v} & \Gamma \vdash_{\mathsf{F}} e_1'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor \ \text{(Fig. 8)} \arrow[rr, "\text{steps to (Fig. 8)}", "\text{Coherence (Thm. 11)}"'] & & \Gamma \vdash_{\mathsf{F}} e_2'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor & & \text{Coherence (Thm. 8)}
\end{tikzcd}
$$

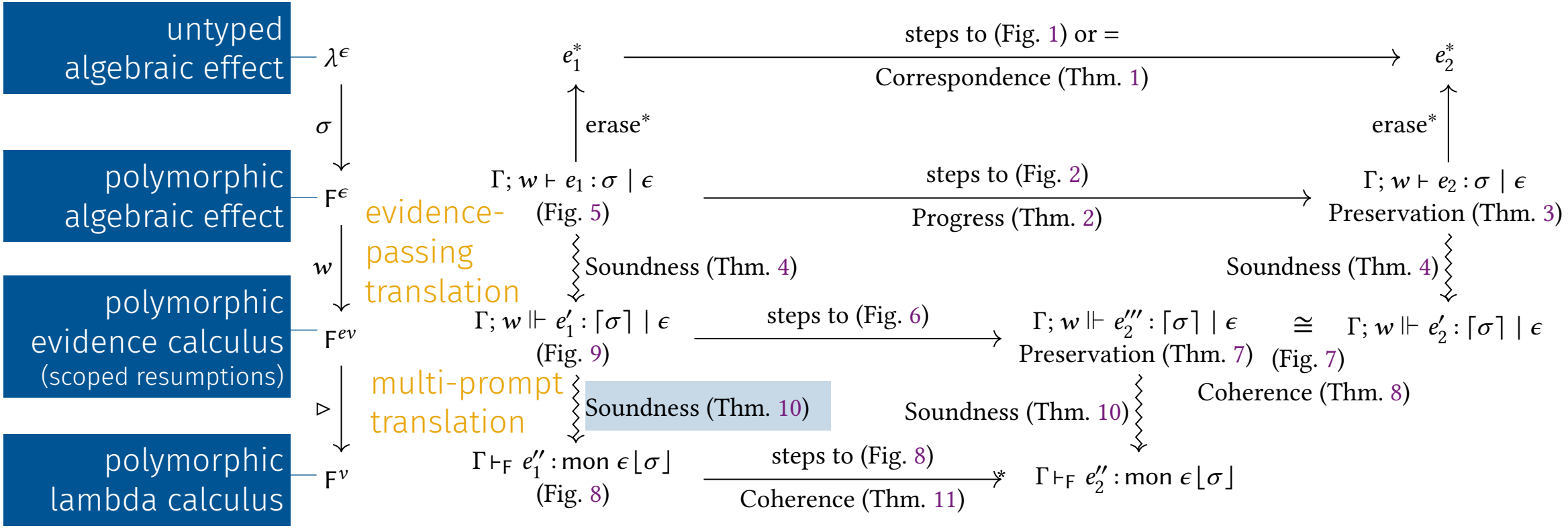
Summary


untyped algebraic effect
polymorphic algebraic effect
λ^{ϵ}
σ
F^{ϵ}
w
F^{ev}
$\triangleright$
F^{v}
e_1^*
steps to (Fig. 1) or =
Correspondence (Thm. 1)
e_2^*
erase*
erase*
$\Gamma; w \vdash e_1 : \sigma \mid \epsilon$
(Fig. 5)
steps to (Fig. 2)
Progress (Thm. 2)
$\Gamma; w \vdash e_2 : \sigma \mid \epsilon$
Preservation (Thm. 3)
Soundness (Thm. 4)
Soundness (Thm. 4)
$\Gamma; w \Vdash e_1' : \lceil \sigma \rceil \mid \epsilon$
(Fig. 9)
steps to (Fig. 6)
$\Gamma; w \Vdash e_2''' : \lceil \sigma \rceil \mid \epsilon$
Preservation (Thm. 7)
$\cong$
(Fig. 7)
$\Gamma; w \Vdash e_2' : \lceil \sigma \rceil \mid \epsilon$
Coherence (Thm. 8)
Soundness (Thm. 10)
Soundness (Thm. 10)
$\Gamma \vdash_{\mathsf{F}} e_1'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor$
(Fig. 8)
steps to (Fig. 8)
Coherence (Thm. 11)
$\Gamma \vdash_{\mathsf{F}} e_2'' : \mathsf{mon}\ \epsilon \lfloor \sigma \rfloor$

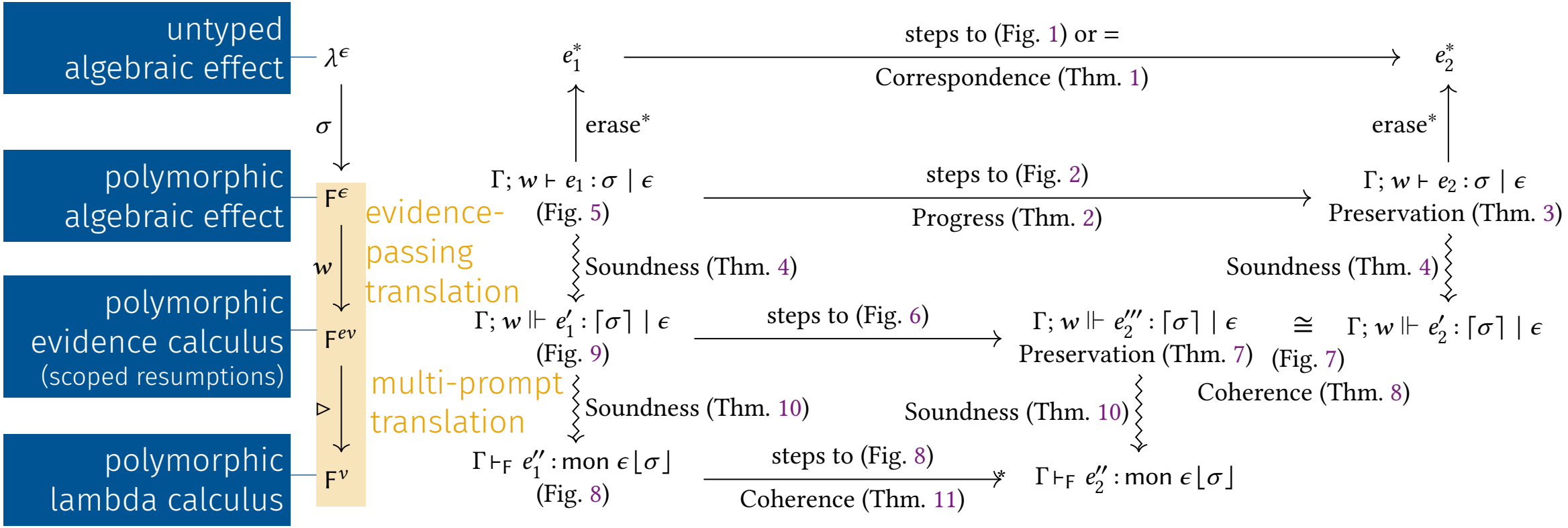
Summary


untyped algebraic effect
polymorphic algebraic effect
polymorphic evidence calculus (scoped resumptions)
λ^ϵ
σ
F^ϵ
w
F^{ev}
$\triangleright$
F^v
evidence-passing translation
e_1^*
steps to (Fig. 1) or =
Correspondence (Thm. 1)
e_2^*
erase*
erase*
$\Gamma; w \vdash e_1 : \sigma \mid \epsilon$
(Fig. 5)
steps to (Fig. 2)
Progress (Thm. 2)
$\Gamma; w \vdash e_2 : \sigma \mid \epsilon$
Preservation (Thm. 3)
Soundness (Thm. 4)
Soundness (Thm. 4)
$\Gamma; w \Vdash e_1' : \lceil\sigma\rceil \mid \epsilon$
(Fig. 9)
steps to (Fig. 6)
$\Gamma; w \Vdash e_2''' : \lceil\sigma\rceil \mid \epsilon$
Preservation (Thm. 7)
$\cong$
(Fig. 7)
$\Gamma; w \Vdash e_2' : \lceil\sigma\rceil \mid \epsilon$
Coherence (Thm. 8)
Soundness (Thm. 10)
Soundness (Thm. 10)
$\Gamma \vdash_{\mathsf{F}} e_1'' : \mathrm{mon}\ \epsilon \lfloor\sigma\rfloor$
(Fig. 8)
steps to (Fig. 8)
Coherence (Thm. 11)
$\Gamma \vdash_{\mathsf{F}} e_2'' : \mathrm{mon}\ \epsilon \lfloor\sigma\rfloor$

Summary


untyped algebraic effect
λ^{ϵ}
σ
polymorphic algebraic effect
F^{ϵ}
w
polymorphic evidence calculus (scoped resumptions)
F^{ev}
$\triangleright$
polymorphic lambda calculus
F^{v}
evidence-passing translation
multi-prompt translation
e_1^*
steps to (Fig. 1) or =
Correspondence (Thm. 1)
e_2^*
erase*
erase*
$\Gamma; w \vdash e_1 : \sigma \mid \epsilon$
(Fig. 5)
steps to (Fig. 2)
Progress (Thm. 2)
$\Gamma; w \vdash e_2 : \sigma \mid \epsilon$
Preservation (Thm. 3)
Soundness (Thm. 4)
Soundness (Thm. 4)
$\Gamma; w \Vdash e_1' : \lceil \sigma \rceil \mid \epsilon$
(Fig. 9)
steps to (Fig. 6)
$\Gamma; w \Vdash e_2''' : \lceil \sigma \rceil \mid \epsilon$
Preservation (Thm. 7)
$\cong$
(Fig. 7)
$\Gamma; w \Vdash e_2' : \lceil \sigma \rceil \mid \epsilon$
Coherence (Thm. 8)
Soundness (Thm. 10)
Soundness (Thm. 10)
$\Gamma \vdash_{\mathsf{F}} e_1'' : \mathrm{mon}\ \epsilon \lfloor \sigma \rfloor$
(Fig. 8)
steps to (Fig. 8)
Coherence (Thm. 11)
$\Gamma \vdash_{\mathsf{F}} e_2'' : \mathrm{mon}\ \epsilon \lfloor \sigma \rfloor$

Summary


untyped algebraic effect
λ^{ϵ}
σ
polymorphic algebraic effect
F^{ϵ}
w
polymorphic evidence calculus (scoped resumptions)
F^{ev}
$\triangleright$
polymorphic lambda calculus
F^{v}
evidence-passing translation
multi-prompt translation
e_1^*
steps to (Fig. 1) or =
Correspondence (Thm. 1)
e_2^*
erase*
erase*
$\Gamma; w \vdash e_1 : \sigma \mid \epsilon$
(Fig. 5)
steps to (Fig. 2)
Progress (Thm. 2)
$\Gamma; w \vdash e_2 : \sigma \mid \epsilon$
Preservation (Thm. 3)
Soundness (Thm. 4)
Soundness (Thm. 4)
$\Gamma; w \Vdash e_1' : \lceil \sigma \rceil \mid \epsilon$
(Fig. 9)
steps to (Fig. 6)
$\Gamma; w \Vdash e_2''' : \lceil \sigma \rceil \mid \epsilon$
Preservation (Thm. 7)
$\cong$
(Fig. 7)
$\Gamma; w \Vdash e_2' : \lceil \sigma \rceil \mid \epsilon$
Coherence (Thm. 8)
Soundness (Thm. 10)
Soundness (Thm. 10)
$\Gamma \vdash_{\mathsf{F}} e_1'' : \mathrm{mon}\ \epsilon \lfloor \sigma \rfloor$
(Fig. 8)
steps to (Fig. 8)
Coherence (Thm. 11)
$\Gamma \vdash_{\mathsf{F}} e_2'' : \mathrm{mon}\ \epsilon \lfloor \sigma \rfloor$

More in the Paper

- Formalizations of calculi and translations
- Implementation in the Koka programming language, and initial benchmarks showing evidence translation enables efficient implementations
- More discussion

https://github.com/koka-lang/koka

More in the Paper

- Formalizations of calculi and translations
- Implementation in the Koka programming language, and initial benchmarks showing evidence translation enables efficient implementations
- More discussion

https://github.com/koka-lang/koka

More in the Paper

- Formalizations of calculi and translations
- Implementation in the Koka programming language, and initial benchmarks showing evidence translation enables efficient implementations
- More discussion

https://github.com/koka-lang/koka

Effect Handlers in Haskell, Evidently

Ningning Xie Daan Leijen

Haskell Symposium 2020

https://github.com/xnning/EvEff