Generalized Evidence Passing for Effect Handlers

Efficient Compilation of Effect Handlers to C

Ningning Xie

香 港 大 學

THE UNIVERSITY OF HONG KONG

Daan Leijen

Microsoft Research

ICFP 2021

Research contributions

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

Research contributions

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

Algebraic effects

Efficient C (with no special runtime support)

Research contributions

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

Algebraic effects

Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Efficient C (with no special runtime support)

Research contributions

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

- Algebraic effects
- Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
- Evidence-passing semantics

Efficient C (with no special runtime support)

Research contributions

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

Algebraic effects

Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Evidence-passing semantics

Bubbling Yields [Pretnar 2015]

Efficient C (with no special runtime support)

Research contributions

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

- Algebraic effects
- Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
- Evidence-passing semantics
- Bubbling Yields [Pretnar 2015]
- Monadic translation

Efficient C (with no special runtime support)

Research contributions

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

Algebraic effects

Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Evidence-passing semantics
- optimization of tail-resumptive operations
- insertion- versus canonical ordered evidence vector

Bubbling Yields [Pretnar 2015]
- short-cut resumption [Kiselyov and Ishii 2015]

Monadic translation
- bind-inlining and join-point sharing

Efficient C (with no special runtime support)

Research contributions

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

PLDI 2021

Perceus: Garbage Free Reference Counting with Reuse

Alex Reinking*
Microsoft Research
Redmond, WA, USA
alex_reinking@berkeley.edu

Ningning Xie*
University of Hong Kong
Hong Kong, China
nnxie@cs.hku.hk

Leonardo de Moura
Microsoft Research
Redmond, WA, USA
leonardo@microsoft.com

Daan Leijen
Microsoft Research
Redmond, WA, USA
daan@microsoft.com

Algebraic effects

Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Evidence-passing semantics
- optimization of tail-resumptive operations
- insertion- versus canonical ordered evidence vector

Bubbling Yields [Pretnar 2015]
- short-cut resumption [Kiselyov and Ishii 2015]

Monadic translation
- bind-inlining and join-point sharing

Efficient C (with no special runtime support)

Research contributions

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

PLDI 2021

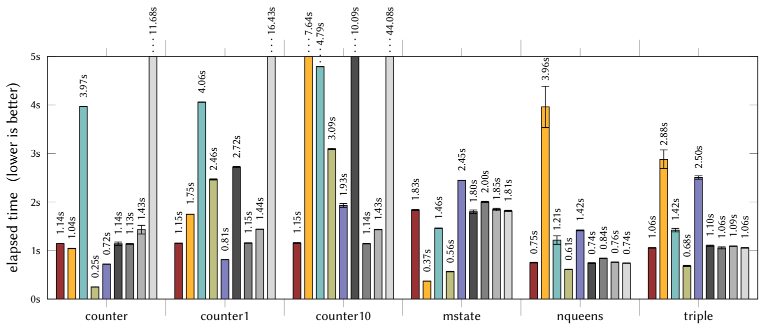
Perceus: Garbage Free Reference Counting with Reuse

Alex Reinking*
Microsoft Research
Redmond, WA, USA
alex_reinking@berkeley.edu

Ningning Xie*
University of Hong Kong
Hong Kong, China
nnxie@cs.hku.hk

Leonardo de Moura
Microsoft Research
Redmond, WA, USA
leonardo@microsoft.com

Daan Leijen
Microsoft Research
Redmond, WA, USA
daan@microsoft.com



- Algebraic effects
- Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
- Evidence-passing semantics
  - optimization of tail-resumptive operations
  - insertion- versus canonical ordered evidence vector
- Bubbling Yields [Pretnar 2015]
  - short-cut resumption [Kiselyov and Ishii 2015]
- Monadic translation
  - bind-inlining and join-point sharing

Efficient C (with no special runtime support)

Algebraic effects 101

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

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

Algebraic effects 101

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

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

Algebraic effects 101

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

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

Algebraic effects 101

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

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

Algebraic effects 101

effect

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

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

Algebraic effects 101

effect

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

operation

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

Algebraic effects 101

effect

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

operation

effect handler

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

Algebraic effects 101

effect

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

operation

effect handler

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

implementation

Algebraic effects 101

effect

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

operation

effect handler

```
handler {
  ask -> \x.\k. k 1
}
```

implementation

computation

```
(\_.
  perform ask () + perform ask ()
)
```

Algebraic effects 101

effect

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

operation

effect handler

```
handler {
  ask -> \x.\k. k 1
}
```

implementation

computation

```
(\_.
  perform ask () + perform ask ()
)
```

perform an effect

Algebraic effects 101

effect

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

operation

effect handler

```
handler {
  ask -> \x.\k. k 1
}
```

implementation

computation

```
(\_.
  perform ask () + perform ask ()
) // 2
```

perform an effect

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

② $(\mathit{handler})$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ (return) $\text{handle}\ h\ v \longrightarrow v$

④ $(\mathit{perform})$ $\text{handle}\ h\ \mathsf{E}[\text{perform}\ \mathit{op}\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])$
$\quad$ iff $\mathit{op} \notin \text{bop}(\mathsf{E}) \wedge (\mathit{op} \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① (app) $\quad (\lambda x.\ e)\ v \quad \longrightarrow \quad e[x{:=}v]$

② $(handler)$ $\quad \text{handler}\ h\ f \quad \longrightarrow \quad \text{handle}\ h\ (f\ ())$

③ $(return)$ $\quad \text{handle}\ h\ v \quad \longrightarrow \quad v$

④ $(perform)$ $\quad \text{handle}\ h\ \mathsf{E}[\text{perform}\ op\ v] \quad \longrightarrow \quad f\ v\ (\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])$

$\text{iff}\ op \notin \text{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$

Operational semantics of untyped algebraic effects

①	(app)	$(\lambda x.\ e)\ v$	$\longrightarrow$	$e[x{:=}v]$
②	$(handler)$	$\text{handler}\ h\ f$	$\longrightarrow$	$\text{handle}\ h\ (f\ ())$
③	$(return)$	$\text{handle}\ h\ v$	$\longrightarrow$	v
④	$(perform)$	$\text{handle}\ h\ \mathsf{E}[\text{perform}\ op\ v]$	$\longrightarrow$	$f\ v\ (\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])$ iff $op \notin \text{bop}(\mathsf{E}) \land (op \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(\mathit{handler})$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ (return) $\text{handle}\ h\ v \longrightarrow v$

④ $(\mathit{perform})$ $\text{handle}\ h\ \mathsf{E}[\text{perform}\ \mathit{op}\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])$
$\text{iff}\ \mathit{op} \notin \text{bop}(\mathsf{E}) \wedge (\mathit{op} \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① (*app*) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② (*handler*) $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ (*return*) $\text{handle}\ h\ v \longrightarrow v$

④ (*perform*) $\text{handle}\ h\ \mathrm{E}[\text{perform}\ op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])$
iff $op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(handler)$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ $(return)$ $\text{handle}\ h\ v \longrightarrow v$

④ $(perform)$ $\text{handle}\ h\ \mathsf{E}[\text{perform}\ op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])$
iff $op \notin \text{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(\mathit{handler})$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ (return) $\text{handle}\ h\ v \longrightarrow v$

evaluation context

④ $(\mathit{perform})$ $\text{handle}\ h\ \boxed{\mathrm{E}}[\text{perform}\ \mathit{op}\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])$

iff $\mathit{op} \notin \text{bop}(\mathrm{E}) \land (\mathit{op} \mapsto f) \in h$

Operational semantics of untyped algebraic effects

① $(\textit{app})$ $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(\textit{handler})$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ $(\textit{return})$ $\text{handle}\ h\ v \longrightarrow v$

evaluation context

④ $(\textit{perform})$ $\text{handle}\ h\ \boxed{\mathrm{E}}[\text{perform}\ op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])$

iff $op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$

h is the innermost handler

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(handler)$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ $(return)$ $\text{handle}\ h\ v \longrightarrow v$

evaluation context

④ $(perform)$ $\text{handle}\ h\ \boxed{\mathrm{E}}[\text{perform}\ op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])$
$\text{iff}\ op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$

h is the innermost handler

get the operation implementation f

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(handler)$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ $(return)$ $\text{handle}\ h\ v \longrightarrow v$

evaluation context

operation argument

④ $(perform)$ $\text{handle}\ h\ \boxed{\mathrm{E}}[\text{perform}\ op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])$

iff $op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$

h is the innermost handler

get the operation implementation f

Operational semantics of untyped algebraic effects

① (app) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

a unit-taking function as a computation

② $(handler)$ $\text{handler}\ h\ f \longrightarrow \text{handle}\ h\ (f\ ())$

③ $(return)$ $\text{handle}\ h\ v \longrightarrow v$

evaluation context

operation argument

resumption

④ $(perform)$ $\text{handle}\ h\ \boxed{\mathrm{E}}[\text{perform}\ op\ v] \longrightarrow f\ v\ \boxed{(\lambda x.\ \text{handle}\ h\ \mathrm{E}[x])}$

iff $op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$

h is the innermost handler

get the operation implementation f

The problem: compiling effect handlers efficiently

$$(\textit{perform}) \quad \text{handle } h \; \mathsf{E}[\text{perform } op \; v] \longrightarrow f \; v \; (\lambda x.\, \text{handle } h \; \mathsf{E}[x])$$
$$\text{iff } op \notin \text{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$$

The problem: compiling effect handlers efficiently

$$(\textit{perform})\quad \mathrm{handle}\ h\ \mathsf{E}[\mathrm{perform}\ op\ v] \longrightarrow f\ \ v\ (\lambda x.\ \mathrm{handle}\ h\ \mathsf{E}[x])$$
$$\text{iff } op \notin \mathrm{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$$

Two potentially expensive runtime operations:

The problem: compiling effect handlers efficiently

evaluation context

$$(\textit{perform}) \quad \text{handle } h\ \boxed{\mathsf{E}}[\text{perform } op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle } h\ \mathsf{E}[x])$$
$$\text{iff } op \notin \text{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$$

Two potentially expensive runtime operations:

The problem: compiling effect handlers efficiently

evaluation context

$$(\mathit{perform}) \quad \text{handle } h\, \boxed{\mathsf{E}}[\text{perform } op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle } h\ \mathsf{E}[x])$$

$$\text{iff } op \notin \text{bop}(\mathsf{E}) \land (op \mapsto f) \in h$$

h is the innermost handler

Two potentially expensive runtime operations:

The problem: compiling effect handlers efficiently

evaluation context

$$(\mathit{perform}) \quad \text{handle } h\ \boxed{\mathsf{E}}[\text{perform } op\ v] \longrightarrow f\ v\ (\lambda x.\ \text{handle } h\ \mathsf{E}[x])$$

$$\text{iff } op \notin \text{bop}(\mathsf{E}) \land (op \mapsto f) \in h$$

h is the innermost handler

Two potentially expensive runtime operations:

1. **Searching**
   a *linear* search through the current evaluation context to find the innermost handler for op

The problem: compiling effect handlers efficiently

evaluation context

resumption

$$(\textit{perform}) \quad \text{handle}\ h\ \boxed{\mathsf{E}}[\text{perform}\ op\ v] \longrightarrow f\ v\ \boxed{(\lambda x.\ \text{handle}\ h\ \mathsf{E}[x])}$$

$$\text{iff}\ op \notin \text{bop}(\mathsf{E}) \wedge (op \mapsto f) \in h$$

h is the innermost handler

Two potentially expensive runtime operations:

1. **Searching**
   a *linear* search through the current evaluation context to find the innermost handler for op

The problem: compiling effect handlers efficiently

evaluation context

resumption

$$(\textit{perform}) \quad \text{handle } h\ \boxed{\mathrm{E}}[\text{perform } op\ v] \longrightarrow f\ v\ \boxed{(\lambda x.\ \text{handle } h\ \mathrm{E}[x])}$$

$$\text{iff } op \notin \text{bop}(\mathrm{E}) \land (op \mapsto f) \in h$$

h is the innermost handler

Two potentially expensive runtime operations:

1. **Searching**
   a *linear* search through the current evaluation context to find the innermost handler for op
2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

The problem: compiling effect handlers efficiently

evaluation context

resumption

$$(\mathit{perform}) \quad \text{handle } h\ \boxed{\mathrm{E}}[\text{perform } op\ v] \longrightarrow f\ v\ \boxed{(\lambda x.\ \text{handle } h\ \mathrm{E}[x])}$$

$$\text{iff } op \notin \text{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$$

h is the innermost handler

Two potentially expensive runtime operations:

1. **Searching**
   a *linear* search through the current evaluation context to find the innermost handler for op
2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

This work:

Multi-prompt delimited control

Evidence-passing semantics

Monadic translation

efficient C

Reader, formally

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

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Reader, formally

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

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Reader, formally

```
handler
  h1
(\_.
  perform ask () + perform ask ()
)
```

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Reader, formally

```
handler h1 (\_. perform ask () + perform ask ())
```

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Reader, formally

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ $(\textit{handler})$ $\quad$ handler $h\,f$ $\qquad\qquad\longrightarrow\qquad$ handle $h\,(f\;())$

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects

Multi-prompt delimited control

Evidence-passing semantics

Bubbling

Monadic translation

efficient C

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) $\quad$ handler $h\ f \quad \longrightarrow \quad$ handle $h\ (f\ ())$

$\longmapsto$ (*app*) $\quad (\lambda x.\ e)\ v \quad \longrightarrow \quad e[x{:=}v]$

Multi-prompt delimited control

Evidence-passing semantics

Bubbling

Monadic translation

Algebraic effects → efficient C

Reader, formally

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ (*handler*) handler $h\,f$ $\longrightarrow$ handle $h\,(f\ ())$

$\mapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

```
f = \x.\k. k 1
h1 = ask -> f
```

Multi-prompt delimited control

Evidence-passing semantics

Bubbling

Monadic translation

Algebraic effects

efficient C

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) handler $h\ f$ $\longrightarrow$ handle $h\ (f\ ())$

$\longmapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) handle h E[perform $op\ v$] $\longrightarrow$ $f\ v\ (\lambda x.$ handle h E$[x])$
iff $op \notin$ bop(E) $\wedge$ $(op \mapsto f) \in h$

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) handler $h\ f$ $\longrightarrow$ handle $h\ (f\ ())$

$\longmapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) handle h E[perform op v] $\longrightarrow$ $f\ v\ (\lambda x.$ handle h E$[x])$
iff $op \notin \text{bop}(\text{E}) \wedge (op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ()))
```

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) handler $h\,f$ $\longrightarrow$ handle $h\,(f\ ())$

$\longmapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) handle h E[perform op v] $\longrightarrow$ $f\ v\ (\lambda x.$ handle h E$[x])$
iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ())
```

$\longmapsto$ (*app*)*

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ (*handler*) handler $h\ f \longrightarrow$ handle $h\ (f\ ())$

$\mapsto$ (*app*) $(\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\mapsto$ (*perform*) handle h E[perform $op\ v$] $\longrightarrow f\ v\ (\lambda x.$ handle h E$[x])$
iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ()))
```

$\mapsto$ $(app)^*$

```
(\x. handle h1 (x + perform ask ())) 1
```

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ (*handler*) handler $h\, f$ $\longrightarrow$ handle $h\, (f\, ())$

$\mapsto$ (*app*) $(\lambda x.\, e)\; v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\mapsto$ (*perform*) handle h E[perform op v] $\longrightarrow$ $f\; v\; (\lambda x.$ handle h E$[x])$
iff $op \notin$ bop(E) $\wedge$ $(op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ()))
```

$\mapsto$ (*app*)*

```
(\x. handle h1 (x + perform ask ())) 1
```

$\mapsto$ (*app*)

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Reader, formally

```
f  = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) handler $h\,f$ $\longrightarrow$ handle $h\,(f\,())$

$\longmapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) handle h E[perform op v] $\longrightarrow$ $f\ v\ (\lambda x.$ handle h E[x])
iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ()))
```

$\longmapsto$ (*app*)*

```
(\x. handle h1 (x + perform ask ())) 1
```

$\longmapsto$ (*app*)

```
handle h1 (1 + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Reader, formally

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) handler $h\,f$ $\longrightarrow$ handle $h\,(f\,())$

$\longmapsto$ (*app*) $(\lambda x.\ e)\ v$ $\longrightarrow$ $e[x{:=}v]$

```
handle h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) handle h E[perform $op\ v$] $\longrightarrow$ $f\ v\ (\lambda x.$ handle h E[x])
iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
f () (\x. handle h1 (x + perform ask ()))
```

$\longmapsto$ (*app*)*

```
(\x. handle h1 (x + perform ask ())) 1
```

$\longmapsto$ (*app*)

```
handle h1 (1 + perform ask ())
```

$\longmapsto^*$ **2**

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Multi-prompt semantics

separating searching from capturing

Algebraic effects · Multi-prompt delimited control · Evidence-passing semantics · Bubbling · Monadic translation · efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
    handler h1 (\_. perform ask () + perform ask ())
↦ (handler) (app)
    handle h1 (perform ask () + perform ask ())
↦ (perform)
    f () (\x. handle h1 (x + perform ask ()))
↦ (app)*
    (\x. handle h1 (x + perform ask ())) 1
↦ (app)
    handle h1 (1 + perform ask ())
↦* 2
```

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
    handler h1 (\_. perform ask () + perform ask ())
⟼  (handler) (app)
    handle h1 (perform ask () + perform ask ())
⟼  (perform)
    f () (\x. handle h1 (x + perform ask ()))
⟼* 2
```

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
   handler h1 (\_. perform ask () + perform ask ())
↦ (handler) (app)
   handle h1 (perform ask () + perform ask ())
↦ (perform)
   f () (\x. handle h1 (x + perform ask ()))
↦* 2
```

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*)

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^*$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\mapsto$ (*perform*)

```
f () (\x. prompt m1 h1 (x + perform ask ()))
```

$\mapsto^*$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ *(perform)*

```
f () (\x. prompt m1 h1 (x + perform ask ()))
```

$\longmapsto^*$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \text{bop}(E) \wedge (op \mapsto f) \in h$

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

Multi-prompt semantics

separating searching from capturing

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

```
f = \x.\k. k 1
h1 = ask -> f
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler)* *(app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ *(perform)* iff $op \notin \mathrm{bop}(E) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

operation implementation partially applied to operation argument

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^*$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^*$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \text{bop}(\text{E}) \land (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

$\longmapsto$ (*prompt*)

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

$\longmapsto$ (*prompt*)

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ (*perform*) iff $op \notin \text{bop}(\text{E}) \wedge (op \mapsto f) \in h$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

$\longmapsto$ (*prompt*)

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^*$ **2**

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\mapsto$ $(\mathit{perform})\quad \text{iff } op \notin \text{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$ searching

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

$\mapsto$ *(prompt)*

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\mapsto^*$ **2**

Multi-prompt semantics

separating searching from capturing

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ $(\mathit{handler})\ (\mathit{app})$

```
prompt m1 h1 (perform ask () + perform ask ())
```

m1: a unique marker identifying handlers

$\longmapsto$ $(\mathit{perform})$ iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$ searching

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

yielding to a handler identified by m1

$\longmapsto$ (prompt) capturing

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

$\longmapsto^{*}$ 2

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ $(handler)$ (app)

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ $(perform)$ iff $op \notin \mathrm{bop}(\mathrm{E}) \wedge (op \mapsto f) \in h$ searching

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ $(handler)$ (app)

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ $(perform)$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$$\overbrace{\texttt{handler h1 (\textbackslash\_. perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\rangle\!\rangle}$$

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*)

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$\overbrace{\texttt{handler h1 (\textbackslash\_. perform ask () + perform ask ())}}^{\langle\!\langle\rangle\!\rangle}$

$\longmapsto$ *(handler) (app)*

$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle}$ `(perform ask () + perform ask ())`

$\longmapsto$ *(perform)*

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$$\underbrace{\texttt{handler h1 (\textbackslash\_. perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}_{\langle\!\langle\rangle\!\rangle}$$

$\longmapsto$ (*handler*) (*app*)

$$\underbrace{\texttt{prompt m1 h1}}_{\langle\!\langle\rangle\!\rangle} \; \underbrace{\texttt{(perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}_{\langle\!\langle \mathit{read}: (m1, h1) \rangle\!\rangle}$$

$\longmapsto$ (*perform*)

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$\langle\!\langle\rangle\!\rangle$

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler) (app)*

$\langle\!\langle\rangle\!\rangle$ $\quad\quad$ $\langle\!\langle\, read : (m1, h1)\, \rangle\!\rangle$

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ *(perform)* $\langle\!\langle\, read : (m1, h1)\, \rangle\!\rangle$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$$\overbrace{\texttt{handler h1 (\textbackslash\_. perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\rangle\!\rangle}$$

$\longmapsto$ $(handler)$ (app)

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle}\ \overbrace{\texttt{(perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\, read: (m1, h1)\, \rangle\!\rangle}$$

$\longmapsto$ $(perform)$ $\langle\!\langle\, read: (m1, h1)\, \rangle\!\rangle.read = (m1, h1)$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$$\overbrace{\texttt{handler h1 (\textbackslash\_. perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\rangle\!\rangle}$$

$\longmapsto$ $(\mathit{handler})$ (app)

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle}\ \overbrace{\texttt{(perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle}$$

$\longmapsto$ $(\mathit{perform})$ $\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle.read = (m1, h1)$

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle}\ \overbrace{\texttt{(yield m1 (\textbackslash k. f () k) + perform } \textit{ask} \texttt{ ())}}^{\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle}$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Evidence-passing semantics

make performs local: push down the current handlers as an evidence vector

```
f = \x.\k. k 1
h1 = ask -> f
```

$\langle\!\langle\rangle\!\rangle$

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto (\mathit{handler})\ (\mathit{app})$

$\langle\!\langle\rangle\!\rangle \qquad \langle\!\langle\ \mathit{read}\colon (m1, h1)\ \rangle\!\rangle$

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto (\mathit{perform})$ $\langle\!\langle\ \mathit{read}\colon (m1, h1)\ \rangle\!\rangle.\mathit{read} = (m1, h1)$ constant time

$\langle\!\langle\rangle\!\rangle \qquad \langle\!\langle\ \mathit{read}\colon (m1, h1)\ \rangle\!\rangle$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

```
f = \x.\k. k 1
h1 = ask -> f
```

$$\overbrace{\texttt{handler h1 (\textbackslash\_. perform } \mathit{ask} \texttt{ () + perform } \mathit{ask} \texttt{ ())}}$$

$\longmapsto$ $(\mathit{handler})$ (app)

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle} \; \overbrace{\texttt{(perform } \mathit{ask} \texttt{ () + perform } \mathit{ask} \texttt{ ())}}^{\langle\!\langle\, read : (m1, h1) \,\rangle\!\rangle}$$

$\longmapsto$ $(\mathit{perform})$ $\langle\!\langle\, read : (m1, h1) \,\rangle\!\rangle.read = (m1, h1)$

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\!\langle\rangle\!\rangle} \; \overbrace{\texttt{(yield m1 (\textbackslash k. f () k) + perform } \mathit{ask} \texttt{ ())}}^{\langle\!\langle\, read : (m1, h1) \,\rangle\!\rangle}$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

```
f = \x.\k. k 1
h1 = ask -> f
```

tail-resumptive operations

$op \mapsto \lambda x.\, \lambda k.\; k\; e$
where $k \notin \mathrm{fv}(e)$

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler) (app)*

$\langle\!\langle\rangle\!\rangle$ $\langle\!\langle\, read\colon (m1, h1)\,\rangle\!\rangle$

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ *(perform)* $\langle\!\langle\, read\colon (m1, h1)\,\rangle\!\rangle.read = (m1, h1)$

$\langle\!\langle\rangle\!\rangle$ $\langle\!\langle\, read\colon (m1, h1)\,\rangle\!\rangle$

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

```
f = \x.\k. k 1
h1 = ask -> f
```

tail-resumptive operations

$op \mapsto \lambda x.\, \lambda k.\, k\ e$
where $k \notin \mathrm{fv}(e)$

$$\overbrace{\texttt{handler h1 (\textbackslash\_. perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}$$

$\longmapsto$ $(handler)$ (app)

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\langle\rangle\rangle}\ \overbrace{\texttt{(perform } \textit{ask} \texttt{ () + perform } \textit{ask} \texttt{ ())}}^{\langle\langle\, read: (m1, h1)\,\rangle\rangle}$$

$\longmapsto$ $(perform)$ $\langle\langle\, read: (m1, h1)\,\rangle\rangle.read = (m1, h1)$

$$\overbrace{\texttt{prompt m1 h1}}^{\langle\langle\rangle\rangle}\ \overbrace{\texttt{(1 + perform } \textit{ask} \texttt{ ())}}^{\langle\langle\, read: (m1, h1)\,\rangle\rangle}$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

```
f = \x.\k. k 1
h1 = ask -> f
```

tail-resumptive operations

$op \mapsto \lambda x.\, \lambda k.\, k\ e$
where $k \notin \mathrm{fv}(e)$

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

$$\underbrace{\texttt{prompt m1 h1}}_{\langle\!\langle\rangle\!\rangle}\ \underbrace{\texttt{(perform ask () + perform ask ())}}_{\langle\!\langle\, read : (m1, h1)\, \rangle\!\rangle}$$

$\longmapsto$ (*perform*) $\langle\!\langle\, read : (m1, h1)\, \rangle\!\rangle.read = (m1, h1)$

$$\underbrace{\texttt{prompt m1 h1}}_{\langle\!\langle\rangle\!\rangle}\ \underbrace{\texttt{(under read 1 + perform ask ())}}_{\langle\!\langle\, read : (m1, h1)\, \rangle\!\rangle}$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

```
f = \x.\k. k 1
h1 = ask -> f
```

tail-resumptive operations

$op \mapsto \lambda x.\, \lambda k.\, k\ e$
where $k \notin \mathrm{fv}(e)$

`handler h1 (\_. perform ask () + perform ask ())`

$\longmapsto$ (*handler*) (*app*)

$\underbrace{\texttt{prompt m1 h1}}_{\langle\!\langle\rangle\!\rangle}$ $\underbrace{\texttt{(perform ask () + perform ask ())}}_{\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle}$

$\longmapsto$ (*perform*) $\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle.read = (m1, h1)$

$\underbrace{\texttt{prompt m1 h1}}_{\langle\!\langle\rangle\!\rangle}$ ($\underbrace{\texttt{under read 1}}_{\langle\!\langle\rangle\!\rangle}$ + $\underbrace{\texttt{perform ask ())}}_{\langle\!\langle\, read : (m1, h1)\,\rangle\!\rangle}$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\mapsto$ *(perform)* searching

```
prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
```

$\mapsto$ *(prompt)* capturing

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
    handler h1 (\_. perform ask () + perform ask ())
↦  (handler) (app)
    prompt m1 h1 (perform ask () + perform ask ())
↦  (perform)  searching
    prompt m1 h1 (yield m1 (\k. f () k) + perform ask ())
↦  (prompt)  capturing
    f () (\x. prompt m1 h1 (x + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\mapsto$ *(handler)* *(app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\mapsto$ *(perform)* searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k)
```

$\mapsto$ *(prompt)* capturing

```
f () (\x. prompt m1 h1 (x + perform ask ())
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
↦ (handler) (app)
prompt m1 h1 (perform ask () + perform ask ())
↦ (perform)   searching
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k)
↦ (prompt)    capturing
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

↦ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

↦ *(perform)* searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k)
```

↦ *(prompt)* capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

⟼ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

⟼ (*perform*) searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

⟼ (*prompt*) capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

⟼ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

⟼ *(perform)* searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

⟼ *(prompt)* capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f  = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ *(perform)* searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

$\longmapsto$ (app_1) bubbling

$\longmapsto$ *(prompt)* capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

$\longmapsto$ (*perform*) searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

$\longmapsto$ (app_1) bubbling

```
prompt m1 h1 •
```

$\longmapsto$ (*prompt*) capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

```
handler h1 (\_. perform ask () + perform ask ())
```

⟼ (*handler*) (*app*)

```
prompt m1 h1 (perform ask () + perform ask ())
```

⟼ (*perform*) searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

⟼ (app_1) bubbling

```
prompt m1 h1 • yield m1 (\k. f () k) (\x. (+ perform ask ()) • x)
```

⟼ (*prompt*) capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Bubbling yields

make yields local: bubbling it up until it meets its corresponding prompt frame

```
handler h1 (\_. perform ask () + perform ask ())
```

$\longmapsto$ *(handler) (app)*

```
prompt m1 h1 (perform ask () + perform ask ())
```

```
f = \x.\k. 1 + k 1
h1 = ask -> f
```

$\longmapsto$ *(perform)* searching

```
prompt m1 h1 • (+ perform ask ()) • yield m1 (\k. f () k) (\x. x)
```

partially built-up resumption

$\longmapsto$ (app_1) bubbling

```
prompt m1 h1 • yield m1 (\k. f () k) (\x. (+ perform ask ()) • x)
```

$\longmapsto$ *(prompt)* capturing

```
f () (\x. prompt m1 h1 • (+ perform ask ()) • x)
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
```

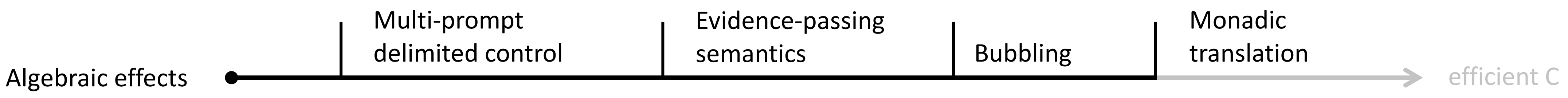
Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
```

$\rightsquigarrow$



Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

$$\text{type Mon } \mu\ \alpha \;=\; \text{Evv } \mu \longrightarrow \text{Ctl } \mu\ \alpha$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

$$\text{type Mon } \mu\ \alpha\ =\ \boxed{\text{Evv } \mu \to}\ \text{Ctl } \mu\ \alpha$$

Multi-prompt delimited control

Evidence-passing semantics

Bubbling

Monadic translation

Algebraic effects

efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

$$\text{type Mon } \mu\ \alpha \;=\; \boxed{\text{Evv } \mu \rightarrow} \boxed{\text{Ctl } \mu\ \alpha}$$

control monad

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

$$\text{type Mon}\ \mu\ \alpha\ =\ \boxed{\text{Evv}\ \mu \rightarrow}\boxed{\text{Ctl}\ \mu\ \alpha}$$

control monad

$$\begin{aligned} e \rhd g \quad = \ & \lambda w.\ \text{case}\ e\ w\ \text{of Pure}\ x \rightarrow g\ x\ w \\ & \qquad\qquad\qquad \text{Yield}\ m\ f\ k \rightarrow \text{Yield}\ m\ f\ (\lambda x.\ k\ x \rhd g) \end{aligned}$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation — efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷(\x. perform ask () ▷(\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

$$\text{type Mon}\ \mu\ \alpha\ =\ \boxed{\text{Evv}\ \mu \rightarrow}\boxed{\text{Ctl}\ \mu\ \alpha}$$

control monad

$$e \rhd g \quad = \ \lambda w.\ \text{case}\ e\ w\ \text{of Pure}\ x \rightarrow g\ x\ w$$

pass the result and the current evidence

$$\text{Yield}\ m\ f\ k \rightarrow \text{Yield}\ m\ f\ (\lambda x.\ k\ x \rhd g)$$

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

$$\text{type Mon}\ \mu\ \alpha\ =\ \boxed{\text{Evv}\ \mu \longrightarrow}\boxed{\text{Ctl}\ \mu\ \alpha}$$

control monad

$$\begin{aligned} e \triangleright g \quad &= \ \lambda w.\ \text{case}\ e\ w\ \text{of Pure}\ x && \longrightarrow g\ x\ w \\ & && \text{Yield}\ m\ f\ k \longrightarrow \text{Yield}\ m\ f\ (\lambda x. \boxed{k\ x \triangleright g)} \end{aligned}$$

pass the result and the current evidence

bubbling

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

```c
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx->w[0], unit, ctx );
  if (ctx->is_yielding) { yield_extend(&join2,ctx); return 0; }
  int y = perform_ask( ctx->w[0], unit, ctx );
  if (ctx->is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
  return (x+y); }
```

Algebraic effects → Multi-prompt delimited control → Evidence-passing semantics → Bubbling → Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

evidence passing

```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
  return (x+y); }
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

evidence passing

constant-time look-up

```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join₂,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join₁(x,ctx),ctx); return 0; }
  return (x+y); }
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

evidence passing

constant-time look-up

control monad

```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join₂,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join₁(x,ctx),ctx); return 0; }
  return (x+y); }
```

Algebraic effects → Multi-prompt delimited control → Evidence-passing semantics → Bubbling → Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

evidence passing

constant-time look-up

control monad

bubbling

```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join₂,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join₁(x,ctx),ctx); return 0; }
  return (x+y); }
```

Algebraic effects — Multi-prompt delimited control — Evidence-passing semantics — Bubbling — Monadic translation → efficient C

Compiling to C

```
handler h1 (\_. perform ask () + perform ask ())
⇝
handler h1 (\_. perform ask () ▷ (\x. perform ask () ▷ (\y. Pure (x + y))))
```

evidence passing

constant-time look-up

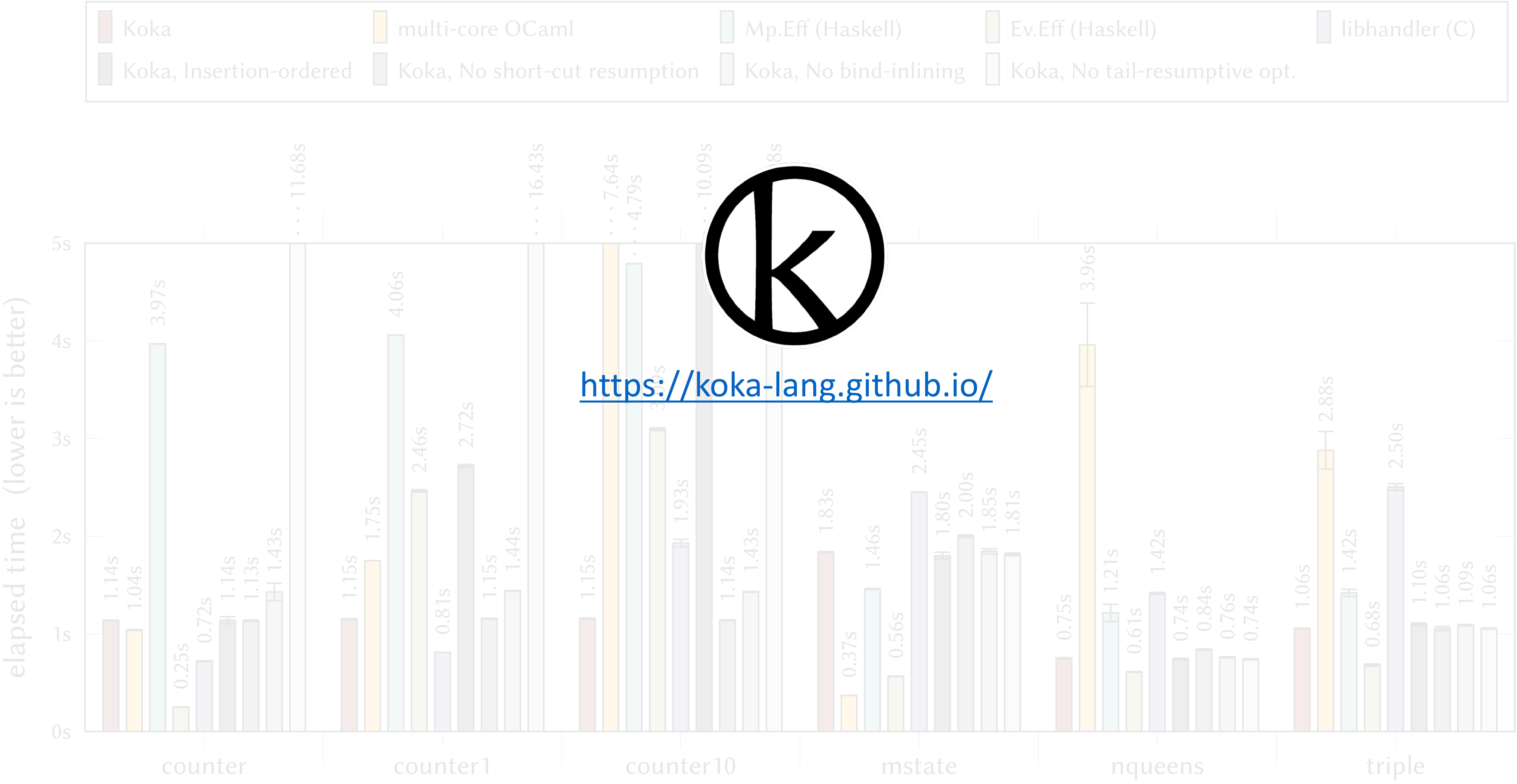
control monad

bubbling

```
int expr( unit_t u, context_t* ctx ) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
  return (x+y); }
```

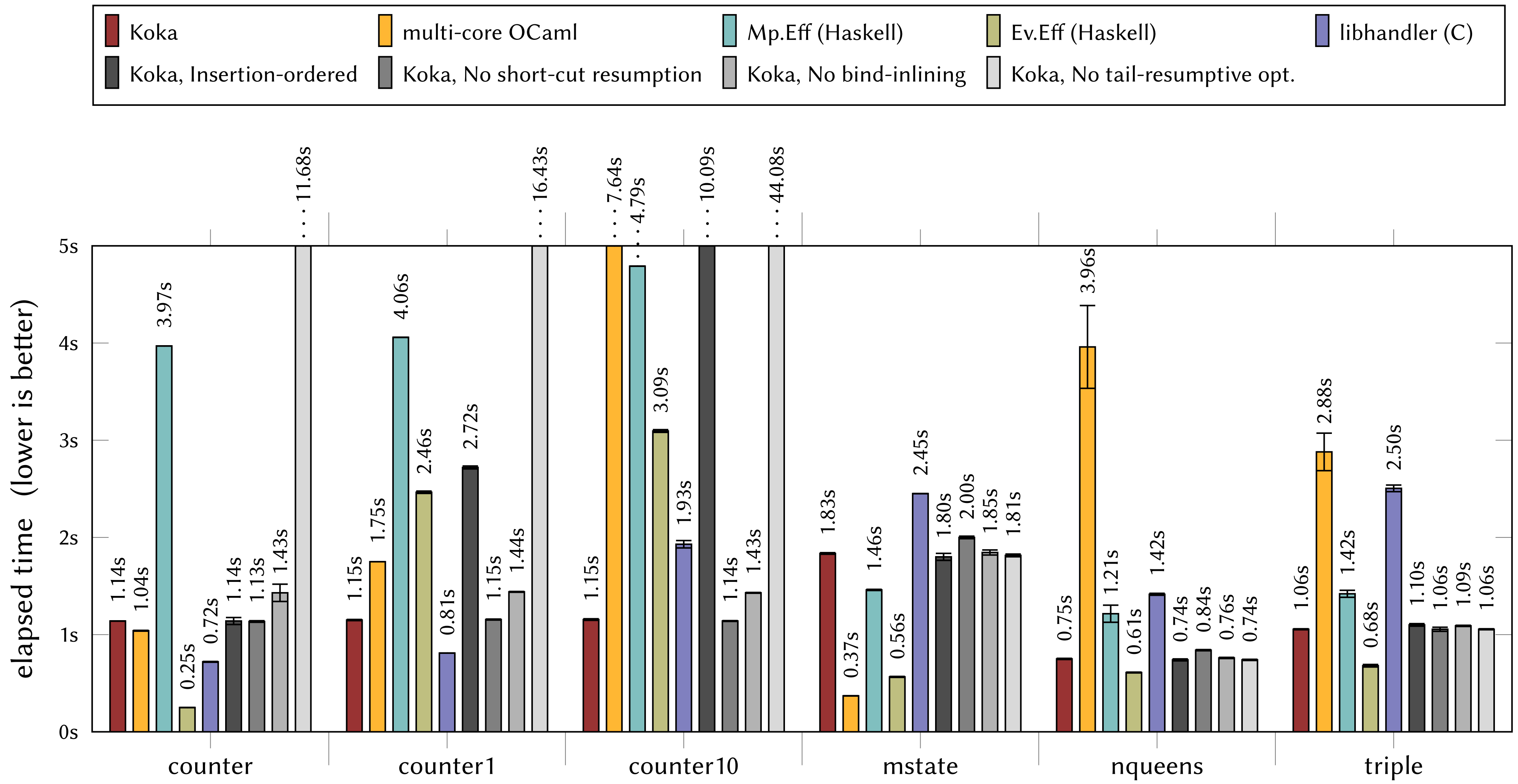
Algebraic effects → Multi-prompt delimited control → Evidence-passing semantics → Bubbling → Monadic translation → efficient C

Benchmarks



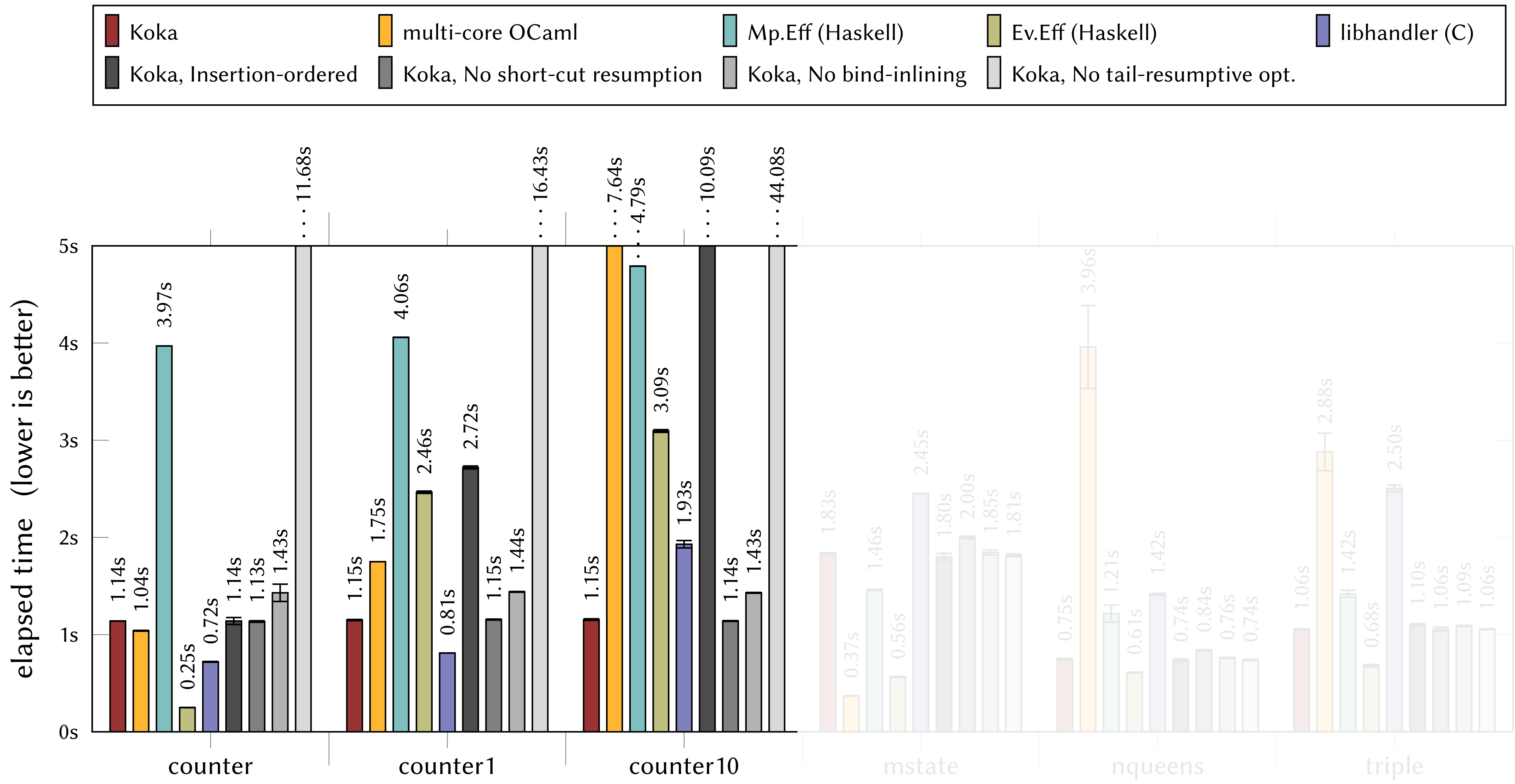
https://koka-lang.github.io/

Benchmarks



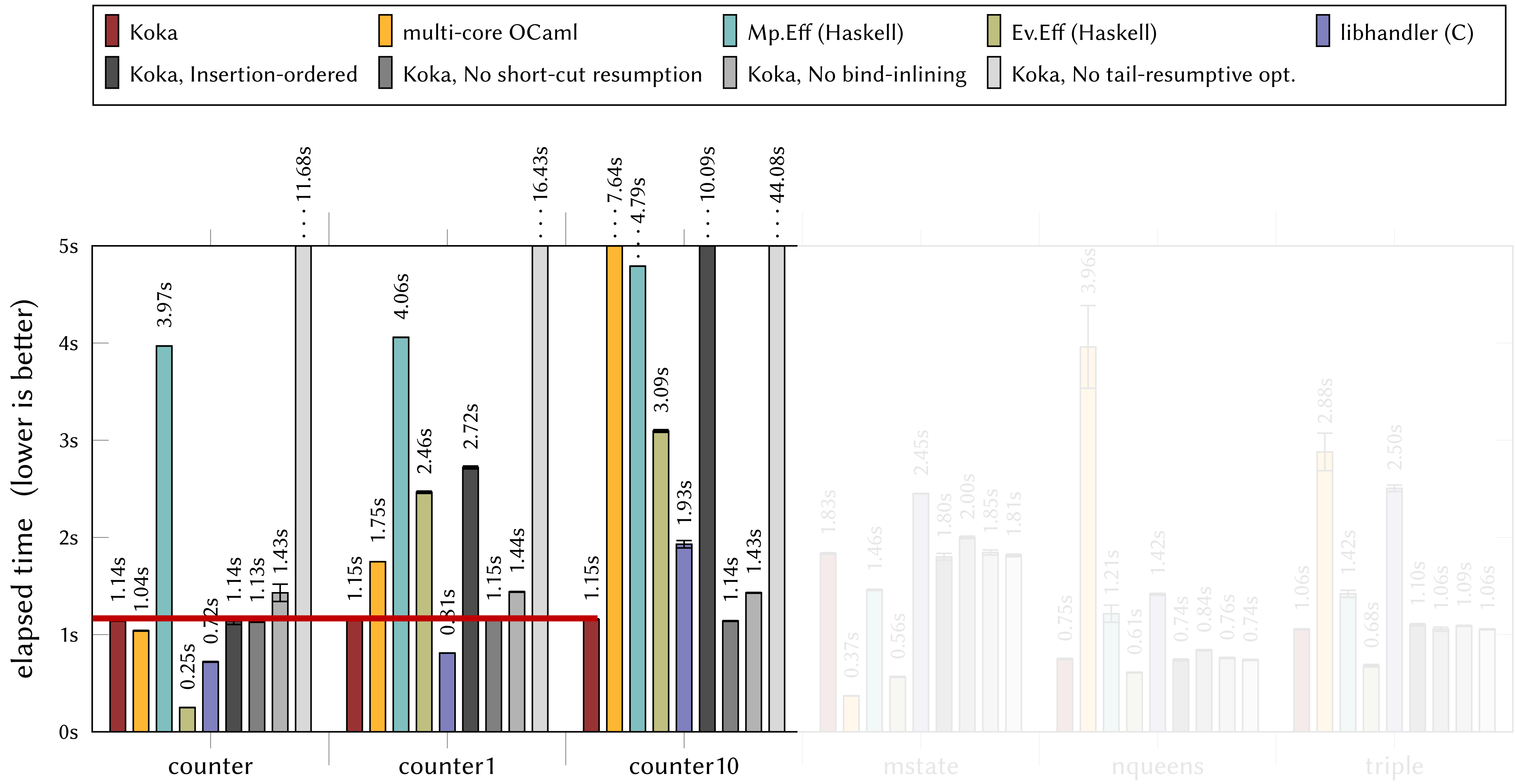
Koka
multi-core OCaml
Mp.Eff (Haskell)
Ev.Eff (Haskell)
libhandler (C)
Koka, Insertion-ordered
Koka, No short-cut resumption
Koka, No bind-inlining
Koka, No tail-resumptive opt.
elapsed time (lower is better)
0s
1s
2s
3s
4s
5s
counter
1.14s 1.04s 3.97s 0.25s 0.72s 1.14s 1.13s 1.43s 11.68s
counter1
1.15s 1.75s 4.06s 2.46s 0.81s 2.72s 1.15s 1.44s 16.43s
counter10
1.15s 7.64s 4.79s 3.09s 1.93s 10.09s 1.14s 1.43s 44.08s
mstate
1.83s 0.37s 1.46s 0.56s 2.45s 1.80s 2.00s 1.85s 1.81s
nqueens
0.75s 3.96s 1.21s 0.61s 1.42s 0.74s 0.84s 0.76s 0.74s
triple
1.06s 2.88s 1.42s 0.68s 2.50s 1.10s 1.06s 1.09s 1.06s

Benchmarks



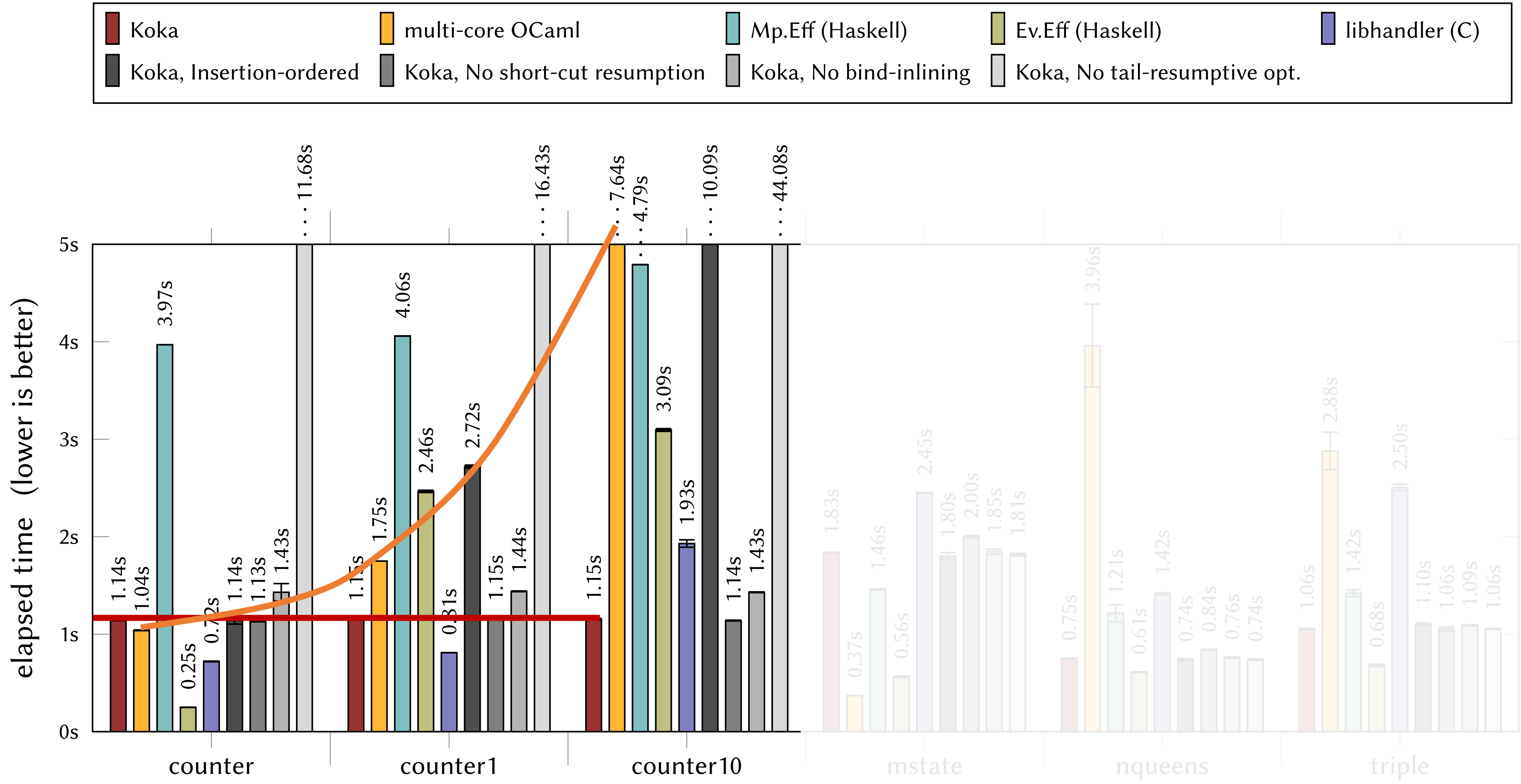
Koka
multi-core OCaml
Mp.Eff (Haskell)
Ev.Eff (Haskell)
libhandler (C)
Koka, Insertion-ordered
Koka, No short-cut resumption
Koka, No bind-inlining
Koka, No tail-resumptive opt.
elapsed time (lower is better)
0s
1s
2s
3s
4s
5s
1.14s
1.04s
3.97s
0.25s
0.72s
1.14s
1.13s
1.43s
11.68s
1.15s
1.75s
4.06s
2.46s
0.81s
2.72s
1.15s
1.44s
16.43s
1.15s
7.64s
4.79s
3.09s
1.93s
10.09s
1.14s
1.43s
44.08s
counter
counter1
counter10
mstate
nqueens
triple

Benchmarks



Koka
multi-core OCaml
Mp.Eff (Haskell)
Ev.Eff (Haskell)
libhandler (C)
Koka, Insertion-ordered
Koka, No short-cut resumption
Koka, No bind-inlining
Koka, No tail-resumptive opt.
elapsed time (lower is better)
0s
1s
2s
3s
4s
5s
counter
1.14s
1.04s
3.97s
0.25s
0.72s
1.14s
1.13s
1.43s
11.68s
counter1
1.15s
1.75s
4.06s
2.46s
0.81s
2.72s
1.15s
1.44s
16.43s
counter10
1.15s
7.64s
4.79s
3.09s
1.93s
10.09s
1.14s
1.43s
44.08s
mstate
1.83s
0.37s
1.46s
0.56s
2.45s
1.80s
2.00s
1.85s
1.81s
nqueens
0.75s
3.96s
1.21s
0.61s
1.42s
0.74s
0.84s
0.76s
0.74s
triple
1.06s
2.88s
1.42s
0.68s
2.50s
1.10s
1.06s
1.09s
1.06s

Benchmarks


Koka
multi-core OCaml
Mp.Eff (Haskell)
Ev.Eff (Haskell)
libhandler (C)
Koka, Insertion-ordered
Koka, No short-cut resumption
Koka, No bind-inlining
Koka, No tail-resumptive opt.
elapsed time (lower is better)
0s
1s
2s
3s
4s
5s
1.14s
1.04s
3.97s
0.25s
1.14s
1.13s
1.43s
11.68s
1.75s
4.06s
2.46s
2.72s
1.15s
1.44s
16.43s
1.15s
7.64s
4.79s
3.09s
1.93s
10.09s
1.14s
1.43s
44.08s
counter
counter1
counter10
mstate
nqueens
triple

Excited to know more?

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

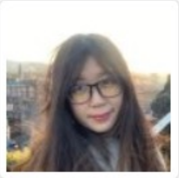
Programming with Effect Handlers and FBIP in Koka

Who *Daan Leijen, Ningning Xie*

Track ICFP 2021 Tutorials

When **(EST) Thu 26 Aug 2021 12:30 - 14:00 at Tutorials** - Programming with Effect Handlers and FBIP in Koka 1

Daan Leijen
Microsoft Research
United States



Ningning Xie
University of Hong Kong
China

Excited to know more?

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

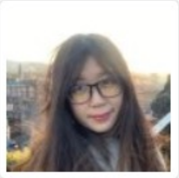
Programming with Effect Handlers and FBIP in Koka

Who *Daan Leijen, Ningning Xie*

Track ICFP 2021 Tutorials

When (EST) Thu 26 Aug 2021 12:30 - 14:00 at **Tutorials** - Programming with Effect Handlers and FBIP in Koka 1

Daan Leijen
Microsoft Research
United States



Ningning Xie
University of Hong Kong
China

Hackage :: [Package]

Search · Browse · What's new · Upload · User accounts

mpeff: Efficient effect handlers based on evidence-passing semantics

[control, effect, library, mit] [Propose Tags]

See the Control.Mp.Eff module or README.md for further information

[Skip to Readme]

Versions [RSS] [faq]
0.1.0.0

Change log

https://hackage.haskell.org/package/mpeff

Generalized Evidence Passing for Effect Handlers

Efficient Compilation of Effect Handlers to C

Ningning Xie

香　港　大　學
THE UNIVERSITY OF HONG KONG

Daan Leijen

Microsoft Research

ICFP 2021