EFFICIENT COMPILATION OF ALGEBRAIC EFFECT HANDLERS

Georgios Karachalias

Matija Pretnar

Filip Koprivec

Tom Schrijvers

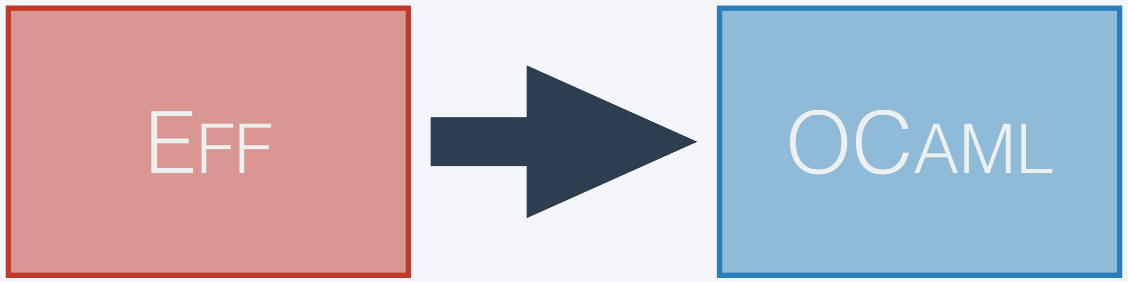
This material is based upon work supported by the Air Force Office of Scientific Research under award numbers FA9550-14-1-0096, FA9550-17-1-0326 and FA9550-21-1-0024.

```
effect Put: int -> unit
effect Get: unit -> int

let rec loop n =
  if n = 0 then () else
    perform (Put (perform (Get ()) + 1));
    loop (n - 1)

let state_handler = handler
  | effect (Put s') k -> (fun _ -> k () s')
  | effect (Get ()) k -> (fun s -> k s s)
  | _ -> (fun s -> s)

let main n =
  (with state_handler handle loop n) 0
```

*"Eff brings home the bacon,
but it is too slow
because it is interpreted."*

– Matija Pretnar, 21st July 2021

MAKING **EFF** **GREAT** AGAIN



Eff

OCaml

```
effect Put: int -> unit
effect Get: unit -> int



type ‘a computation =
   | Return of ‘a
   | Put of int * (unit -> ‘a computation)
   | Get of unit * (int -> ‘a computation)

let rec (>>=) comp k =
  match comp with
   | Return x -> k x
   | Put (s, k’) ->
       Put (s, fun _ -> k’ () >>= k)
   | Get (_, k’) ->
       Get ((), fun y -> k’ y >>= k)
```

```
let rec loop n =
  if n = 0 then () else
    perform (Put (perform (Get ()) + 1));
    loop (n - 1)
```

```
let rec loop n =
  equal n >>= fun f ->
  f 0     >>= fun b ->
  if b then return () else
    get ()  >>= fun s  ->
    plus s  >>= fun g  ->
    g 1     >>= fun s' ->
    put s'  >>= fun _  ->
    minus n >>= fun h  ->
    h 1     >>= fun n' ->
    loop n'
```

```
let equal =
  fun x ->
    return (fun y ->
      return (x = y))
```

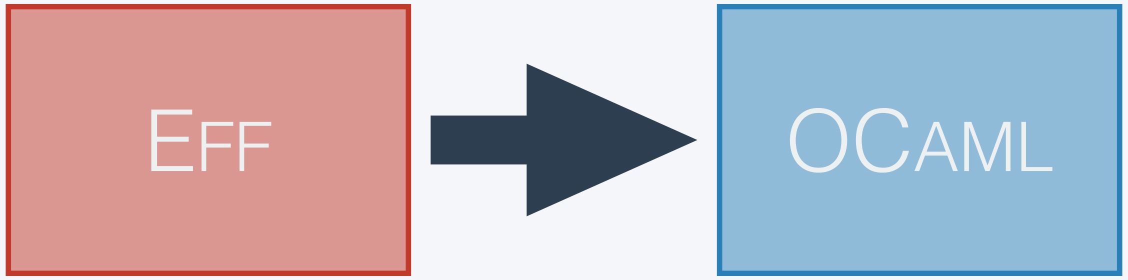
```
effect Put: int -> unit
effect Get: unit -> int


type (‘a, ‘b) handler_clauses = {
  return : ‘a -> ‘b;
  put : int -> (unit -> ‘b) -> ‘b;
  get : unit -> (int -> ‘b) -> ‘b
}

let rec handle hcls =
  function
  | Return x -> hcls.return x
  | Put (x, k) ->
     cl.put x (fun y -> handle hcls (k y))
  | Get (x, k) ->
     cl.get x (fun y -> handle hcls (k y))
```

```
let state_handler = handler
  | effect (Put s') k -> (fun _ -> k () s')
  | effect (Get ()) k -> (fun s -> k s s)
  | _ -> (fun s -> s)
```

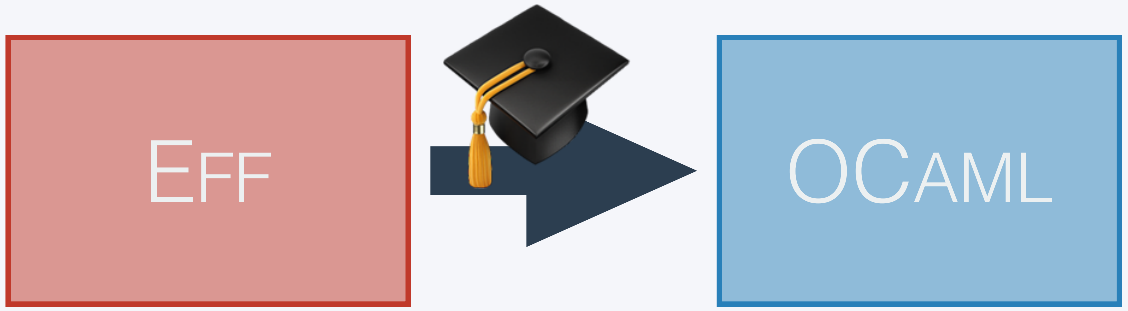
```
let state_handler = handler {
  put = (fun s' k -> return
    (fun _ -> k () >>= fun f -> f s'));
  get = (fun () k -> return
    (fun s -> k s >>= fun f -> f s));
  return = (fun _ -> return
    (fun s -> return s));
}
```

```
let main n =
  (with state_handler handle loop n) 0
```

```
let main n =
  state_handler (loop n) >>= (fun f -> f 0)
```



EFF
OCAML

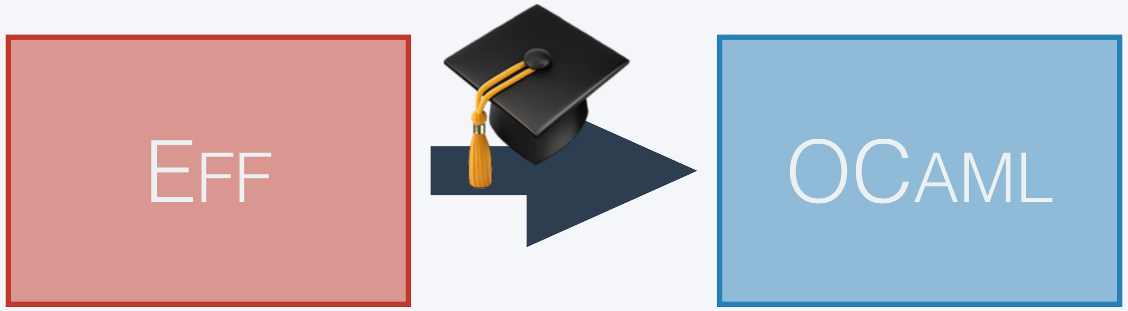


Eff

OCaml

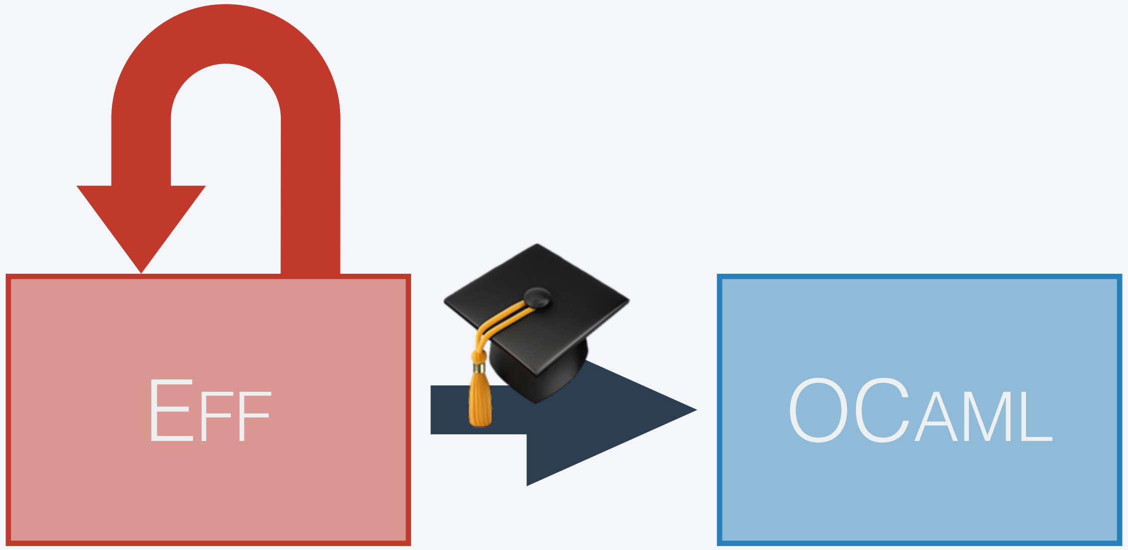
```
let rec loop n =
  equal n >>= fun f ->
  f 0     >>= fun b ->
  if b then return () else
    get ()  >>= fun s  ->
    plus s  >>= fun g  ->
    g 1     >>= fun s' ->
    put s'  >>= fun _  ->
    minus n >>= fun h  ->
    h 1     >>= fun n' ->
    loop n'
```

```
let rec loop n =
  let f = (=) n in
  let b = f 0 in
  if b then return () else
    get () >>= fun s  ->
    let g = (+) s in
    let s' = g 1 in
    put s' >>= fun _  ->
    let h = (-) n in
    let n' = h 1 in
    loop n'
```



Eff

OCaml


EFF
OCAML

```
effect Put: int -> unit
effect Get: unit -> int

let rec loop n =
  if n = 0 then () else
    perform (Put (perform (Get ()) + 1));
    loop (n - 1)

let state_handler = handler
  | effect (Put s') k -> (fun _ -> k () s')
  | effect (Get ()) k -> (fun s -> k s s)
  | _ -> (fun s -> s)

let main n =
  (with state_handler handle loop n) 0
```

```
let main n =
  let rec state_handler_loop m s =
    if m = 0 then s
             else state_handler_loop (m - 1) (s + 1)
  in
  state_handler_loop n 0
```

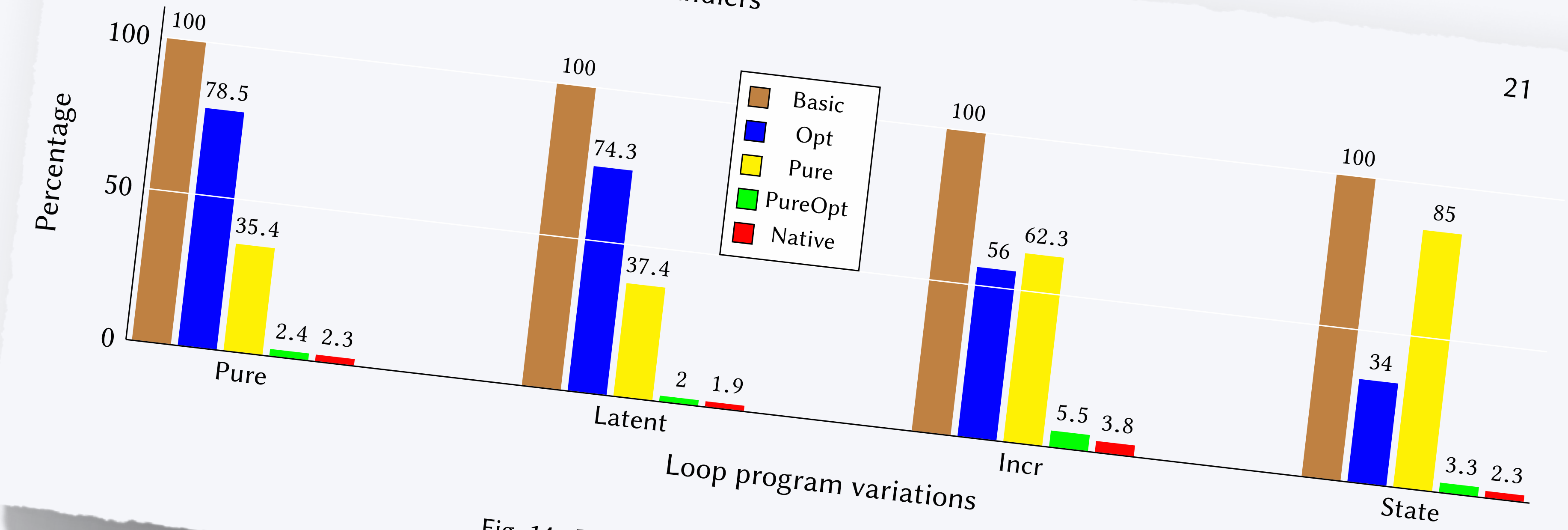
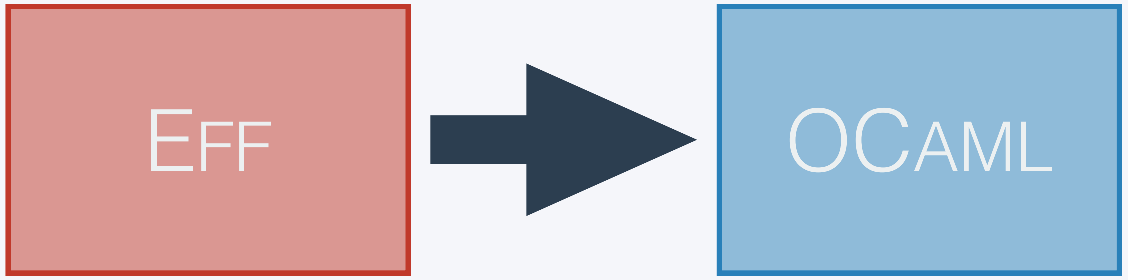
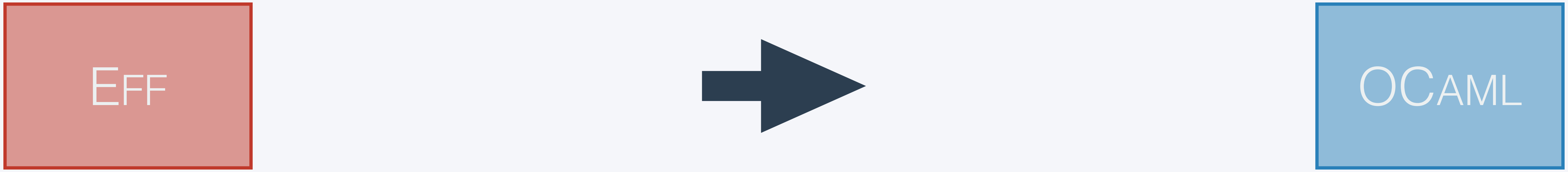


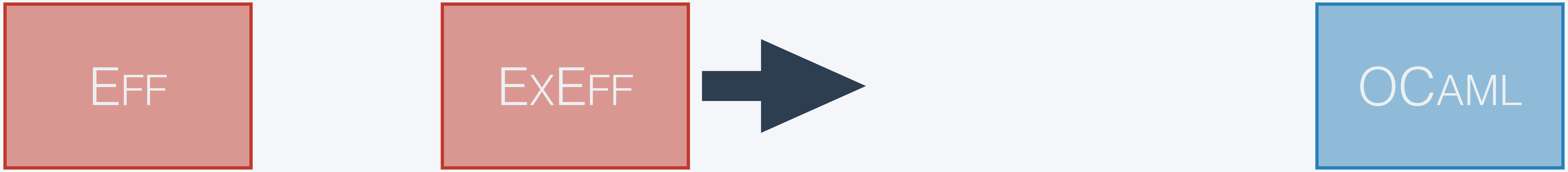
Fig. 14. Relative run-times of Loops example


Eff
OCaml



Eff

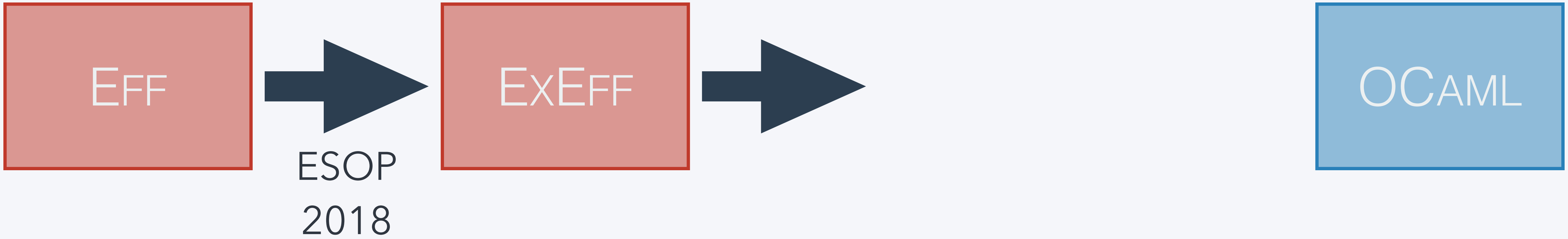
OCaml



Eff

ExEff

OCaml

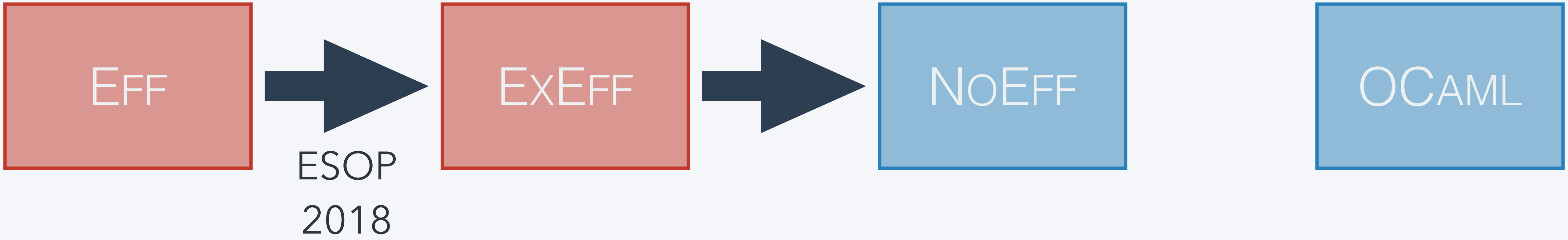


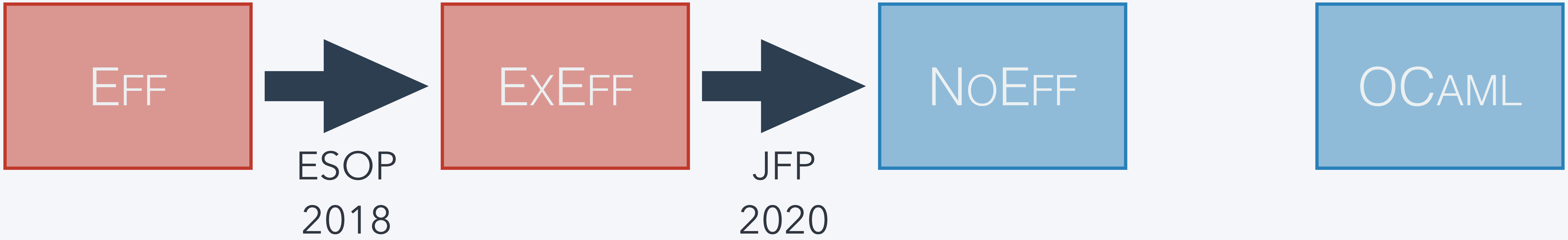
Eff

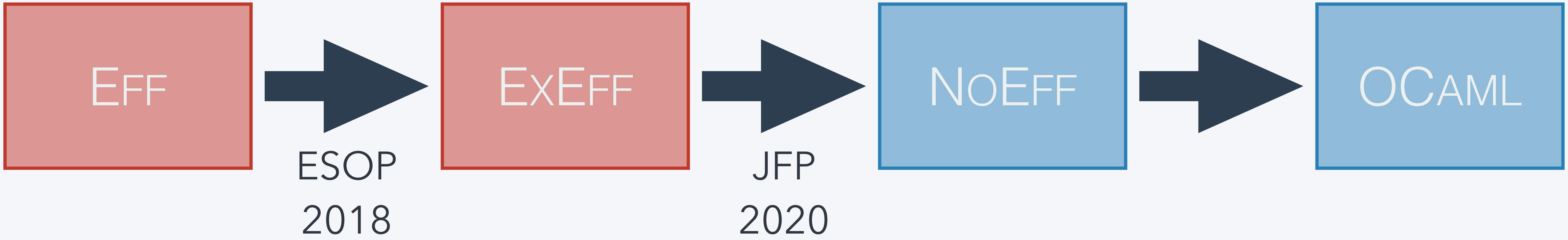
ESOP
2018

ExEff

OCaml


EFF
EXEFF
NOEFF
OCAML
ESOP
2018


EFF
ESOP
2018
EXEFF
JFP
2020
NOEFF
OCAML



Eff

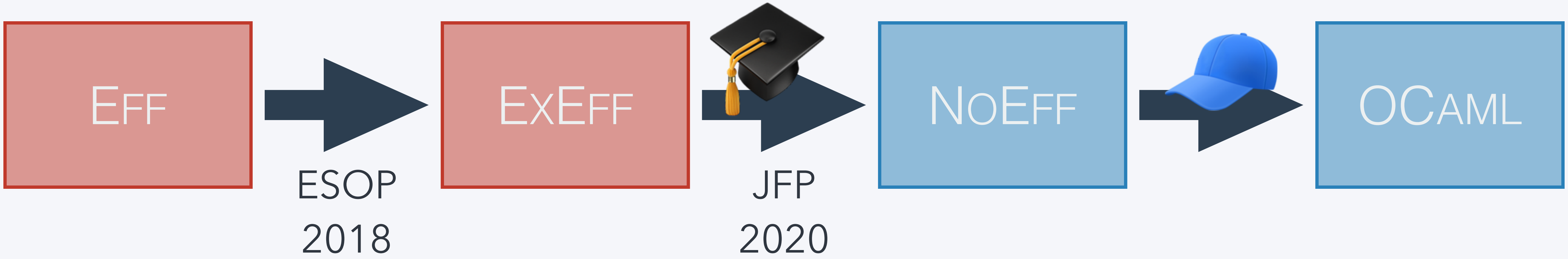
ESOP
2018

ExEff

JFP
2020

NoEff

OCaml



Eff

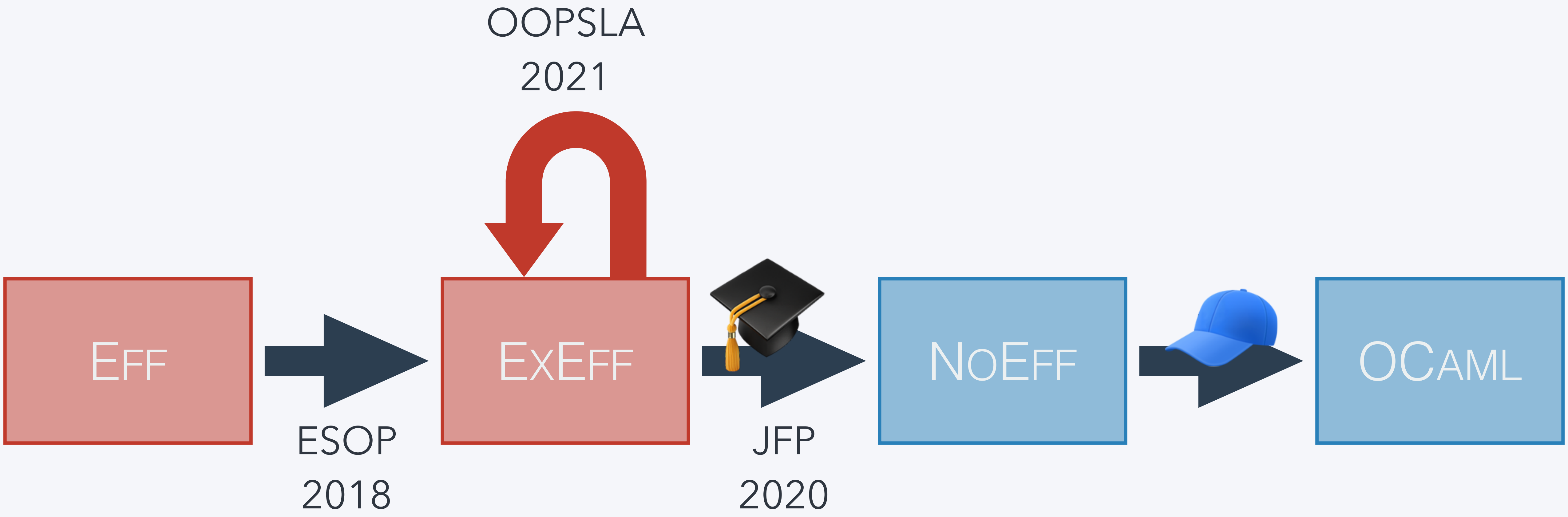
ESOP
2018

ExEff

JFP
2020

NoEff

OCaml



OOPSLA
2021

Eff

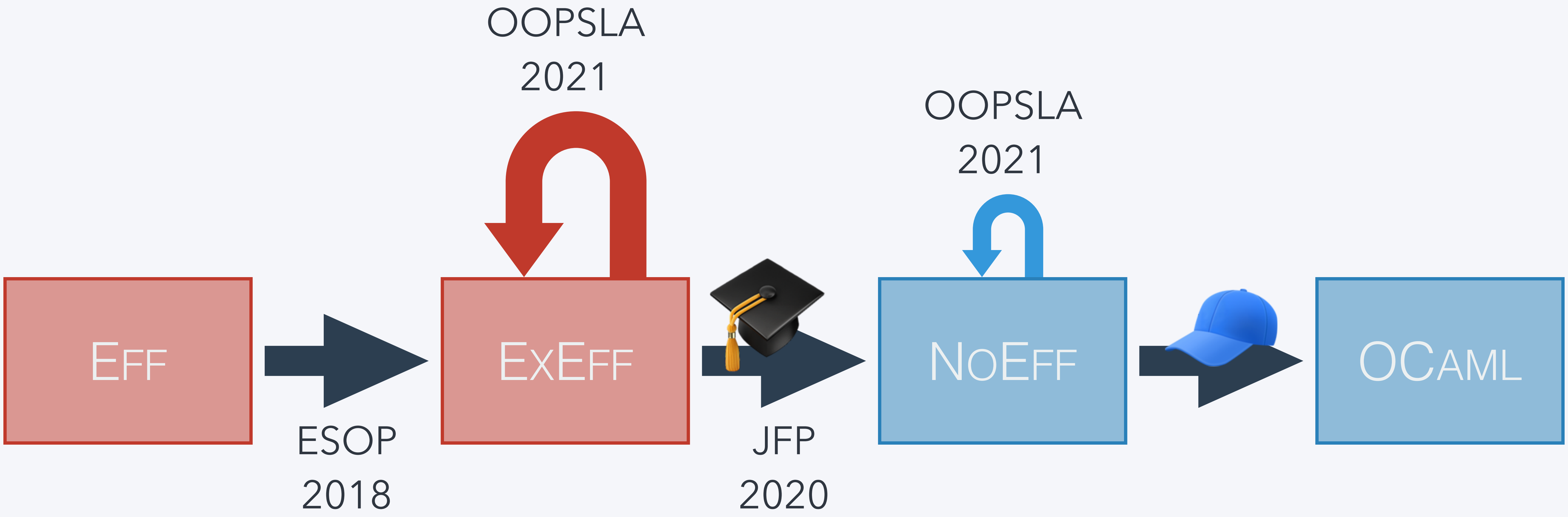
ESOP
2018

ExEff

JFP
2020

NoEff

OCaml



Eff

ESOP 2018

ExEff

OOPSLA 2021

JFP 2020

NoEff

OOPSLA 2021

OCaml

EXEFF & **NO**EFF

ExEff syntax

$$\begin{array}{rl}
\text{value} & v ::= x \mid \mathtt{unit} \mid \mathtt{fun}\ (x : T) \mapsto c \mid h \mid v \rhd \gamma \\
\text{handler} & h ::= \{\mathtt{return}\ (x : T) \mapsto c_r, \mathtt{Op}_1\ x\ k \mapsto c_{\mathtt{Op}_1}, \ldots, \mathtt{Op}_n\ x\ k \mapsto c_{\mathtt{Op}_n}\} \\
\text{computation} & c ::= \mathtt{return}\ v \mid \mathtt{Op}\ v\ (y : T.c) \mid \mathtt{do}\ x \leftarrow c_1 ; c_2 \mid \mathtt{handle}\ c\ \mathtt{with}\ v \\
 & \quad \mid\ v_1\ v_2 \mid \mathtt{let}\ x = v\ \mathtt{in}\ c \mid \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 \mid c \rhd \gamma \\
 & \\
\text{value type} & T ::= \mathtt{Unit} \mid T \to \underline{C} \mid \underline{C}_1 \Rightarrow \underline{C}_2 \\
\text{computation type} & \underline{C} ::= T\ !\ \Delta \\
\text{dirt} & \Delta ::= \emptyset \mid \{\mathtt{Op}\} \cup \Delta \\
\text{coercion type} & \pi ::= T_1 \leqslant T_2 \mid \Delta_1 \leqslant \Delta_2 \mid \underline{C}_1 \leqslant \underline{C}_2 \\
 & \\
\text{coercion} & \gamma ::= \langle \mathtt{Unit} \rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \emptyset_\Delta \mid \{\mathtt{Op}\} \cup \gamma \mid \gamma_1\ !\ \gamma_2
\end{array}$$

NoEff syntax

$$
\begin{array}{rrl}
\text{term} & t & ::= x \mid \mathtt{unit} \mid \mathtt{fun}\ x : A \mapsto t \mid t_1\ t_2 \mid t \rhd \gamma \mid \mathtt{return}\ t \mid h \mid \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \\
 & & \mid\ \mathtt{let\ rec}\ f\ x = t_1\ \mathtt{in}\ t_2 \mid \mathtt{Op}\ t_1\ (y : B.t_2) \mid \mathtt{do}\ x \leftarrow t_1; t_2 \mid \mathtt{handle}\ t_c\ \mathtt{with}\ t_h \\
\text{handler} & h & ::= \{\mathtt{return}\ (x : A) \mapsto t_r, [\mathtt{Op}\, x\, k \mapsto t_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} \\
\text{type} & A, B & ::= \mathtt{Unit} \mid A \to A \mid A \Rightarrow B \mid \mathtt{Comp}\ A \\
\text{coercion type} & \pi & ::= A \leqslant B \\
\text{coercion} & \gamma & ::= \langle \mathtt{Unit} \rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \mathtt{comp}\ \gamma \mid \mathtt{return}\ \gamma \mid \ldots
\end{array}
$$

Translating ExEff to NoEff

$$[\![T \; ! \; \Delta]\!] = \begin{cases} [\![T]\!] & \text{, if } \Delta = \emptyset \\ \mathsf{Comp}\; [\![T]\!] & \text{, if } \Delta \neq \emptyset \end{cases}$$

$$[\![\Gamma \vdash (\mathsf{do}\; x \leftarrow c_1; c2) : B \; ! \; \Delta]\!] = \begin{cases} \mathtt{let}\; x = [\![c_1]\!]\; \mathtt{in}\; [\![c_2]\!] & \text{, if } \Delta = \emptyset \\ \mathsf{do}\; x \leftarrow [\![c_1]\!]; [\![c_2]\!] & \text{, if } \Delta \neq \emptyset \end{cases}$$

$$[\![\gamma_1 \; ! \; \gamma_2]\!] = \begin{cases} [\![\gamma_1]\!] & \text{, if } \gamma_2 : \emptyset \leq \emptyset \\ \mathsf{return}\; [\![\gamma_1]\!] & \text{, if } \gamma_2 : \emptyset \leq \Delta \\ \mathsf{comp}\; [\![\gamma_1]\!] & \text{, if } \gamma_2 : \Delta \leq \Delta' \end{cases}$$

OPTIMIZATION

RULES

ExEff coercion optimizations

$$\frac{\gamma : T \leqslant T}{v \rhd \gamma \rightsquigarrow v}\ \textsc{Elim-Co-Val} \qquad \frac{\gamma : \underline{C} \leqslant \underline{C}}{c \rhd \gamma \rightsquigarrow c}\ \textsc{Elim-Co-Comp}$$

$$\frac{}{(\mathsf{Op}\ v\ (y : T.c)) \rhd \gamma \rightsquigarrow \mathsf{Op}\ v\ (y : T.(c \rhd \gamma))}\ \textsc{Push-Co-Op}$$

$$\frac{c_1 : T}{(\mathsf{do}\ x \leftarrow c_1; c_2) \rhd (\gamma_1 \mathbin{!} \gamma_2) \rightsquigarrow \mathsf{do}\ x \leftarrow (c_1 \rhd \langle T \rangle \mathbin{!} \gamma_2); (c_2 \rhd \gamma_1 \mathbin{!} \gamma_2)}\ \textsc{Push-Co-Do}$$

$$\frac{}{(v_1 \rhd \gamma_1 \to \gamma_2)\ v_2 \rightsquigarrow (v_1\ (v_2 \rhd \gamma_1)) \rhd \gamma_2}\ \textsc{Push-Co-App}$$

$$\frac{}{\mathtt{handle}\ c\ \mathtt{with}\ (v \rhd \gamma_1 \Rightarrow \gamma_2) \rightsquigarrow (\mathtt{handle}\ (c \rhd \gamma_1)\ \mathtt{with}\ v) \rhd \gamma_2}\ \textsc{Push-Co-Handle}$$

ExEff β-reductions

$$\frac{}{(\mathsf{fun}\ (x : T) \mapsto c)\ v \leadsto c[v/x]}\ \textsc{App-Fun} \qquad \frac{}{\mathtt{let}\ x = v\ \mathtt{in}\ c \leadsto c[v/x]}\ \textsc{LetVal}$$

$$\frac{}{(\mathsf{do}\ x \leftarrow ((\mathtt{return}\ v) \rhd (\gamma_{v_1} \mathbin{!} \gamma_{\Delta_1}) \rhd \cdots \rhd (\gamma_{v_n} \mathbin{!} \gamma_{\Delta_n})); c) \leadsto c[(v \rhd \gamma_{v_1} \rhd \cdots \rhd \gamma_{v_n})/x]}\ \textsc{Do-Ret}$$

$$\frac{}{\mathsf{do}\ x \leftarrow (\mathsf{Op}\ v\ (y : T.c_1)); c_2 \leadsto \mathsf{Op}\ v\ (y : T.\mathsf{do}\ x \leftarrow c_1; c_2)}\ \textsc{Do-Op}$$

$$\frac{}{(\mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow c_1; c_2); c_3) \leadsto (\mathsf{do}\ y \leftarrow c_1; (\mathsf{do}\ x \leftarrow c_2; c_3))}\ \textsc{Do-Do}$$

Obvious ExEff handler optimizations

$$\frac{}{\texttt{handle}\ (\texttt{let}\ x = v\ \texttt{in}\ c)\ \texttt{with}\ h \rightsquigarrow \texttt{let}\ x = v\ \texttt{in}\ (\texttt{handle}\ c\ \texttt{with}\ h)}\ \textsc{With-LetVal}$$

$$\frac{}{\texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ h)}\ \textsc{With-LetRec}$$

$$\frac{\texttt{Op} \in \mathcal{O}}{\texttt{handle}\ (\texttt{Op}\ v\ (y : T.c))\ \texttt{with}\ h \rightsquigarrow c_{\texttt{Op}}[v/x, (\texttt{fun}\ (y : T) \mapsto \texttt{handle}\ c\ \texttt{with}\ h)/k]}\ \textsc{With-Handled-Op}$$

$$\frac{\texttt{Op} \notin \mathcal{O}}{\texttt{handle}\ (\texttt{Op}\ v\ (y : T.c))\ \texttt{with}\ h \rightsquigarrow \texttt{Op}\ v\ (y : T.\texttt{handle}\ c\ \texttt{with}\ h)}\ \textsc{With-Unhandled-Op}$$

$$h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$$

Less obvious ExEff handler optimizations

$$\frac{h : T_i \mathrel{!} \Delta_i \Rightarrow T_o \mathrel{!} \Delta_o \qquad c : T \mathrel{!} \Delta \qquad \Delta \cap \mathcal{O} = \emptyset}{\texttt{handle}\ c\ \texttt{with}\ h \rightsquigarrow \texttt{do}\ x \leftarrow (c \rhd \langle T \rangle \mathrel{!} (\Delta \cup \emptyset_{(\Delta_o - \Delta)})); c_r}\ \textsc{With-Pure}$$

$$\frac{h' = \{\texttt{return}\ y \mapsto (\texttt{handle}\ c_2\ \texttt{with}\ h), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ (\texttt{do}\ y \leftarrow c_1; c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c_1\ \texttt{with}\ h'}\ \textsc{With-Do}$$

$$\frac{h' = \{\texttt{return}\ y \mapsto (\texttt{let}\ x = y \rhd \gamma_1\ \texttt{in}\ c_r), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}}{\texttt{handle}\ c \rhd (\gamma_1 \mathrel{!} \gamma_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c\ \texttt{with}\ h'}\ \textsc{With-Cast}$$

$$h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$$

NoEff optimizations

$$\frac{\gamma : A \leqslant A}{t \triangleright \mathtt{return}\ \gamma \rightsquigarrow \mathtt{return}\ t}\ \textsc{Elim-Ret-Co}$$

$$\frac{\gamma : A \leqslant A}{t \triangleright \gamma \rightsquigarrow t}\ \textsc{Elim-Co-Term}$$

$$\frac{}{\mathtt{do}\ x \leftarrow (\mathtt{return}\ t_1); t_2 \rightsquigarrow t_2[t_1/x]}\ \textsc{Do-Ret}$$

$$\frac{}{\mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \rightsquigarrow t_2[t_1/x]}\ \textsc{LetVal}$$

FUNCTION

SPECIALIZATION

```
let rec loop n = …def…

let state_handler = …

let main n =
  (with state_handler handle (loop n)) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    with state_handler handle …def…
  in
  (with state_handler handle (loop n)) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    with state_handler handle …def…
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    with state_handler handle
      if n = 0 then () else
        perform (Put (perform (Get ()) + 1));
        loop (n – 1)
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      with state_handler handle ()
    else
      with state_handler handle
        perform (Put (perform (Get ()) + 1));
        loop (n - 1)
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      fun s -> s
    else
      with state_handler handle
        perform (Put (perform (Get ()) + 1));
        loop (n - 1)
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      fun s -> s
    else
      fun s -> (fun y ->
        with state_handler handle
          perform (Put (y + 1));
          loop (n - 1)
      ) s s
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      fun s -> s
    else
      fun s -> (fun y ->
        (fun _ -> (
           with state_handler handle
             loop (n - 1)
           ) () (y + 1)
      ) s s
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      fun s -> s
    else
      fun s ->
        with state_handler handle
          loop (n - 1) (s + 1)
  in
  (loop’ n) 0
```

```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n =
    if n = 0 then
      fun s -> s
    else
      fun s ->
        loop’ (n - 1) (s + 1)
  in
  (loop’ n) 0
```

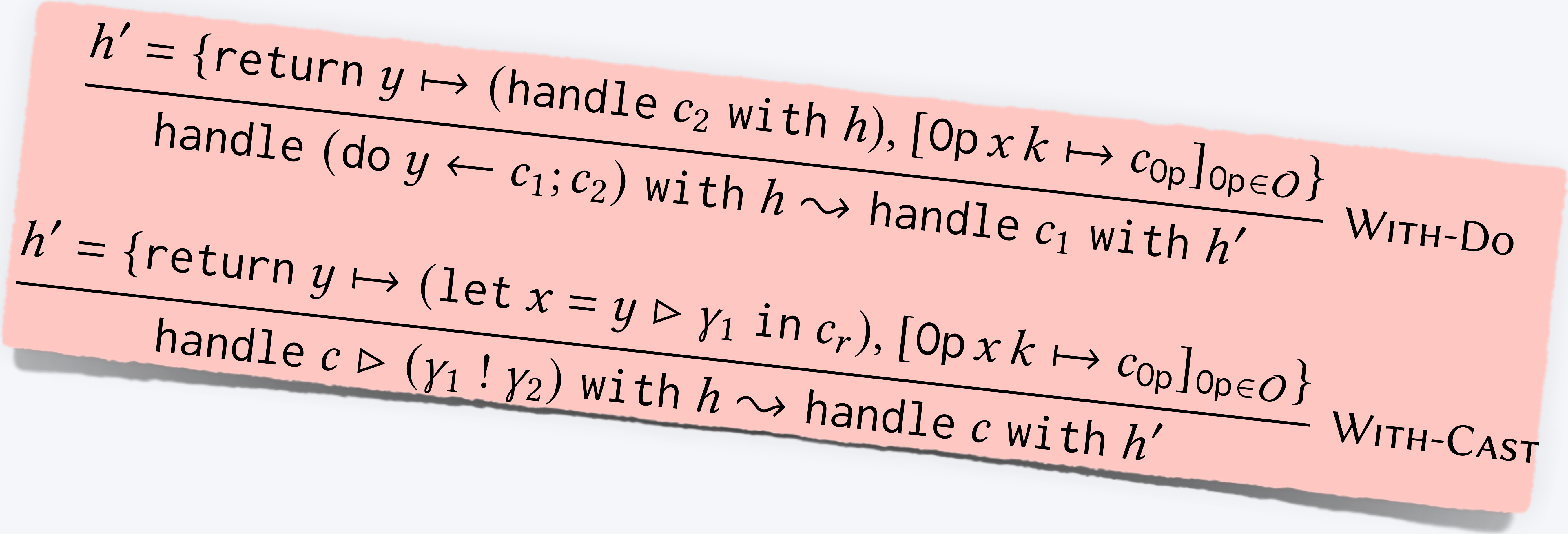
```
let rec loop n = …def…

let state_handler = …

let main n =
  let loop’ n s =
    if n = 0 then
      s
    else
      loop’ (n − 1) (s + 1)
  in
  (loop’ n) 0
```

$$\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$$

$$\texttt{handle}\ f\ v\ \texttt{with}\ h \quad \rightsquigarrow \quad \texttt{let rec}\ f'\ x = \texttt{handle}\ c_f\ \texttt{with}\ h\ \texttt{in}\ f'\ v$$

$$\mathtt{let\ rec}\ f\ x = c_f\ \mathtt{in}\ c$$

$$\mathtt{handle}\ f\ v\ \mathtt{with}\ h \quad \leadsto \quad \mathtt{let\ rec}\ f'\ x = \mathtt{handle}\ c_f\ \mathtt{with}\ h\ \mathtt{in}\ f'\ v$$



$$\frac{h' = \{\mathtt{return}\ y \mapsto (\mathtt{handle}\ c_2\ \mathtt{with}\ h), [\mathsf{Op}\ x\ k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in \mathcal{O}}\}}{\mathtt{handle}\ (\mathtt{do}\ y \leftarrow c_1; c_2)\ \mathtt{with}\ h \leadsto \mathtt{handle}\ c_1\ \mathtt{with}\ h'}\ \textsc{With-Do}$$

$$\frac{h' = \{\mathtt{return}\ y \mapsto (\mathtt{let}\ x = y \rhd \gamma_1\ \mathtt{in}\ c_r), [\mathsf{Op}\ x\ k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in \mathcal{O}}\}}{\mathtt{handle}\ c \rhd (\gamma_1\ !\ \gamma_2)\ \mathtt{with}\ h \leadsto \mathtt{handle}\ c\ \mathtt{with}\ h'}\ \textsc{With-Cast}$$

$$\texttt{let rec}\ f\,x = c_f\ \texttt{in}\ c$$

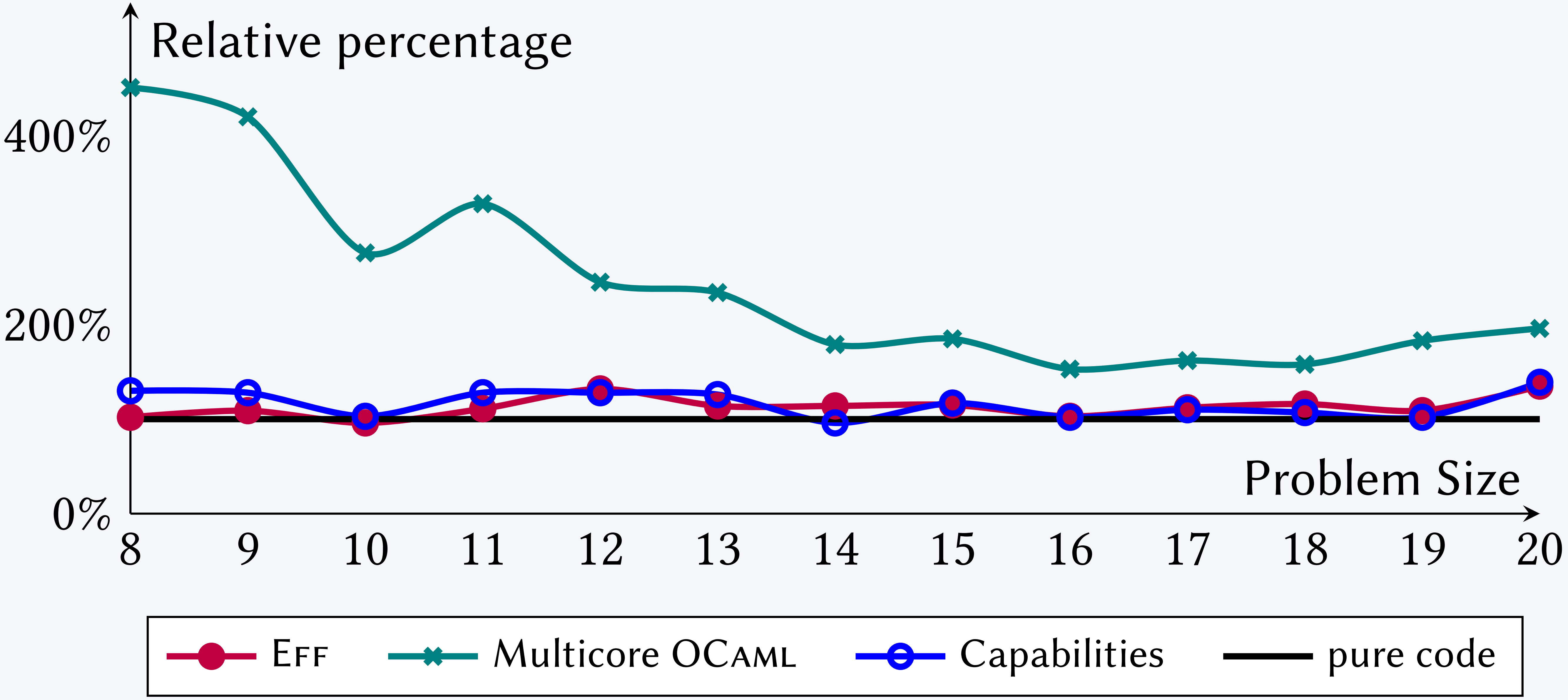
$$\texttt{handle}\ f\,v\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\,x\,k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}$$

$$\rightsquigarrow$$

$$\texttt{let rec}\ f'\,(x, k) = \texttt{handle}\ c_f\ \texttt{with}\ \{\texttt{return}\ x \mapsto k\,x, [\texttt{Op}\,x\,k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}\ \texttt{in}\ f'\,(v, \texttt{fun}\ x \mapsto c_r)$$

BENCHMARKS

Relative speed of a single solution n-Queeens benchmark



Relative speed in other benchmarks

	Eff	Multicore OCaml	Capabilities
one solution of *n*-queens	**135 %**	196 %	139 %
all solutions of *n*-queens	**116 %**	201 %	
stateful counter	**101 %**	6,090 %	556 %
list of generator values	**185 %**	308 %	
stateful sum of generator values	**193 %**	8,695 %	559 %
exceptional arithmetic	145 %	**92 %**	
stateful arithmetic	**140 %**	281 %	
pure tree traversal	**88 %**	422 %	
reader tree traversal	**221 %**	391 %	
stateful tree traversal	**249 %**	367 %	

FUTURE WORK

```
let test_generator n =
  let rec generate (l, u) =
    if l > u then () else
      perform (Yield l); generate (l + 1, u)
  in (
  handle
    handle
      generate (perform (Get ()), n)
    with
    | effect (Yield e) k ->
        (perform (Put (perform (Get ()) + e))); k ()
  with
  | x -> fun s -> s
  | effect (Put s') k -> fun s ->  k () s'
  | effect (Get _) k -> fun s -> k s s
  ) 0
```

```
let test_generator n =
  let rec generate’ (l, u) x =
    if l > u then x
    else generate’ (l + 1, u) (x + l)
  in
  generate’ (0, n) 0
```

∀

QUESTIONS?