Efficient Compilation of Algebraic Effect Handlers

Ningning Xie

UNIVERSITY OF CAMBRIDGE

Can you implement a function,
which takes an integer `i`,
and returns the result of **`42`** divided by `i`?

```haskell
div42 :: Int -> Int
div42 i = 42 / i
```

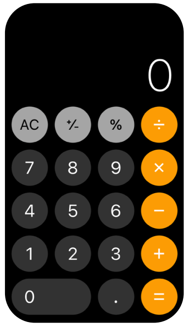
```haskell
div42 :: Int -> Int
div42 i =
  if i == 0
  then error “divided by Zero”
  else 42 / i
```

```
divn :: Int -> Int
divn i =
  n <- getUserInput ()
  if i == 0
  then error “divided by Zero”
  else n / i
```

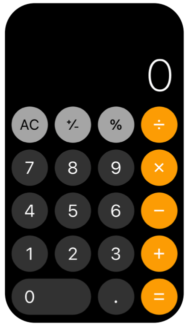
```
divn :: Int -> Int
divn i =
  n <- getUserInput ()
  if i == 0
  then error “divided by Zero”
  else writeLog “success”
       n / i
```

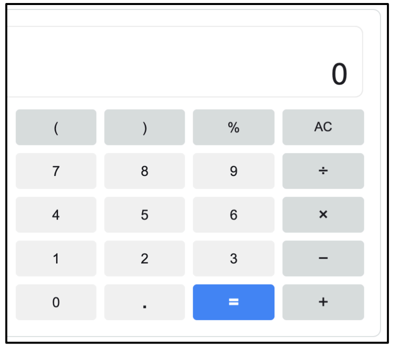
```
divn :: Int -> Int
divn i =
  n <- getUserInput ()
  if i == 0
  then error “divided by Zero”
  else writeLog “success”
       count += 1
       n / i
```

calculator



calculator

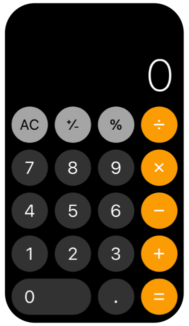


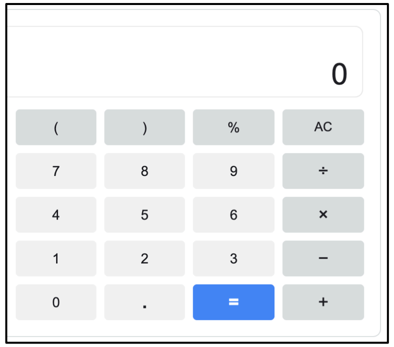


Coq proof assistant

calculator

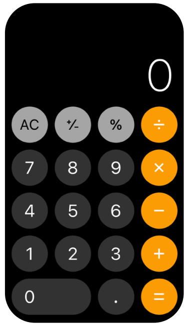
0 / 0

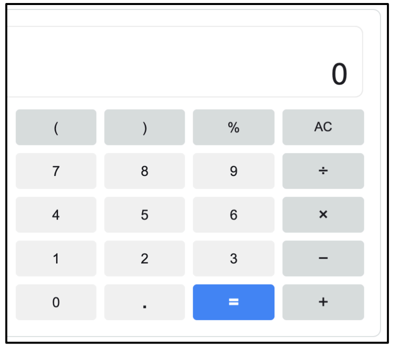




Coq proof assistant

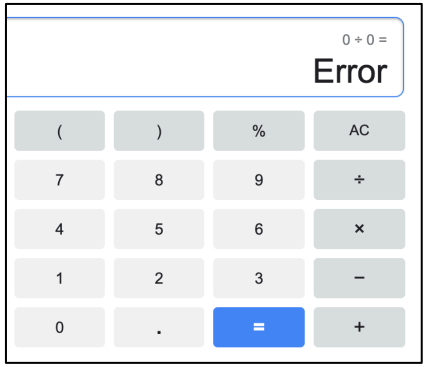
calculator





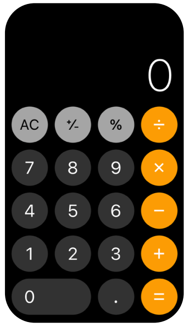
Coq proof assistant

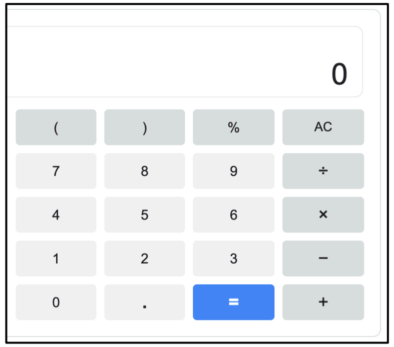
0 / 0



0

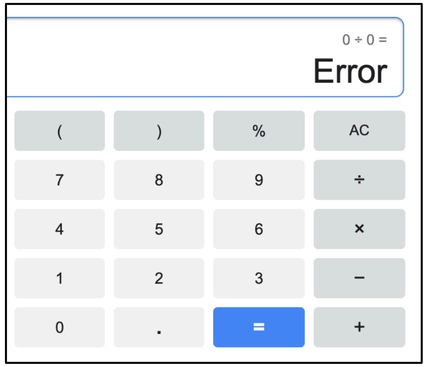
calculator





Coq proof assistant

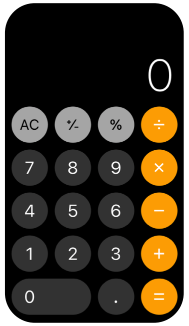
0 / 0

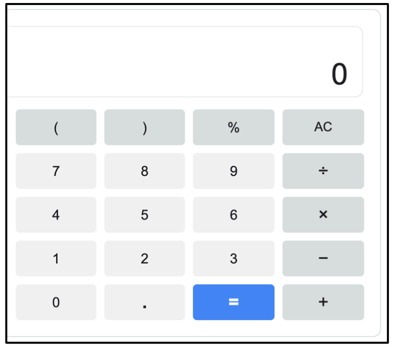


0

1 / 0

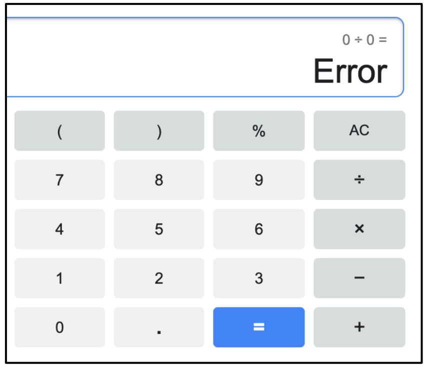
calculator





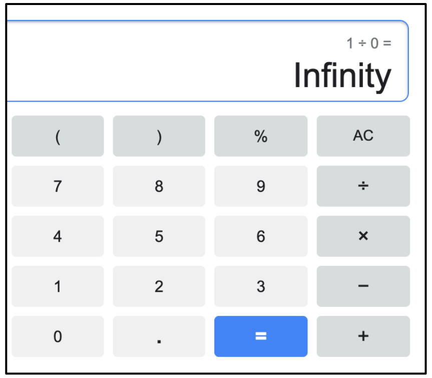
Coq proof assistant

0 / 0



0

1 / 0



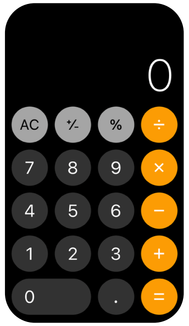
0

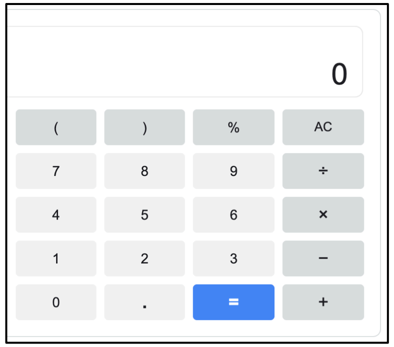
calculator

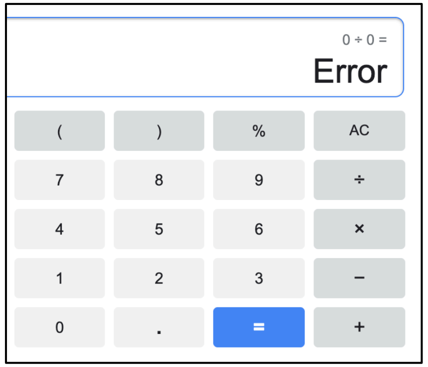
0 / 0

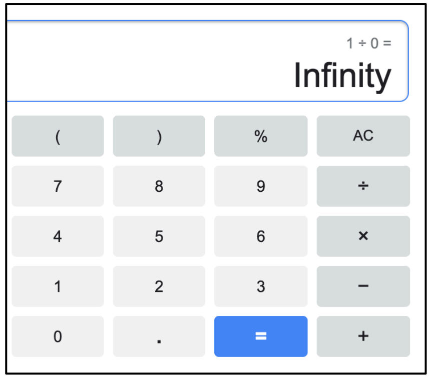
1 / 0

1 + (1 / 0)







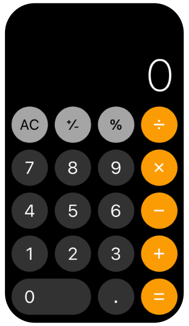


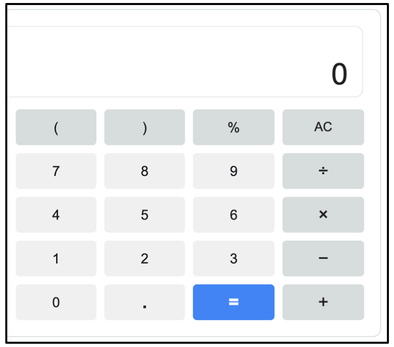
Coq proof assistant

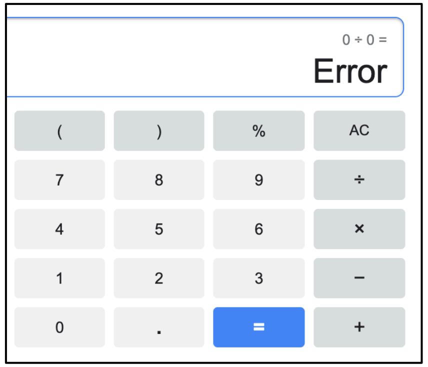
0

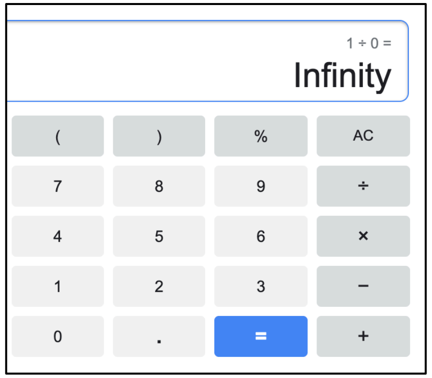
0

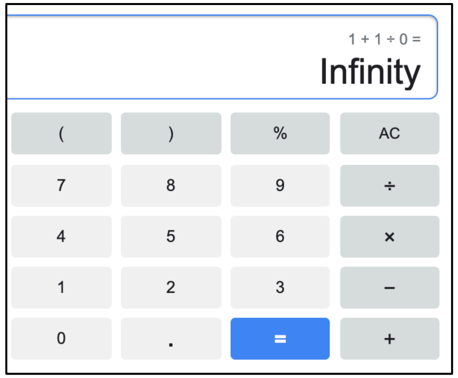
calculator | **0 / 0** | **1 / 0** | **1 + (1 / 0)**







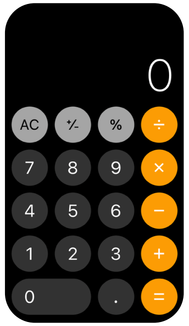


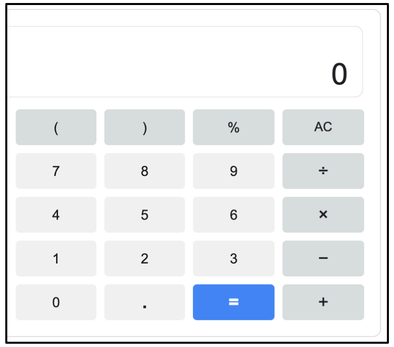


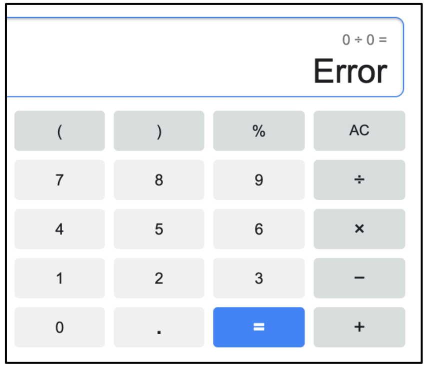
Coq proof assistant

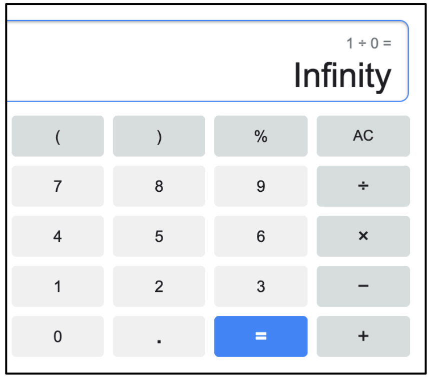
0 | **0** | **1**

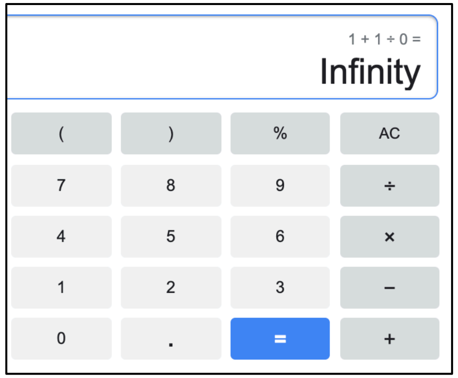
calculator	0 / 0	1 / 0	1 + (1 / 0)





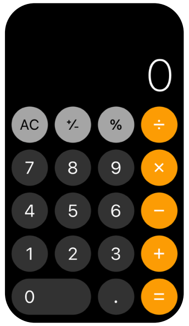
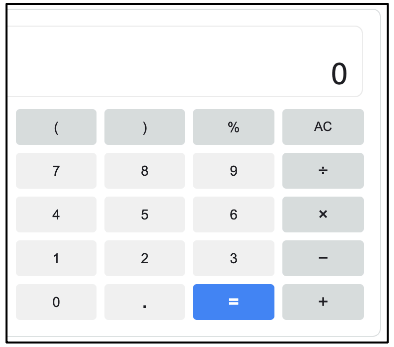
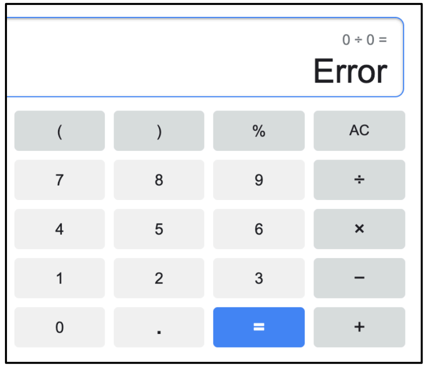
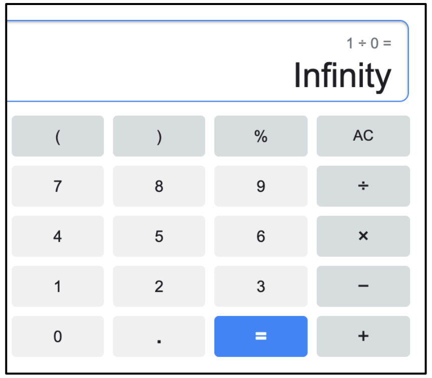
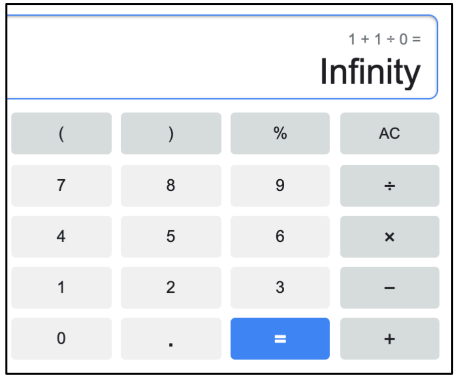






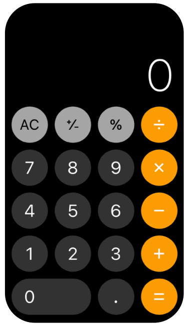
Coq proof assistant

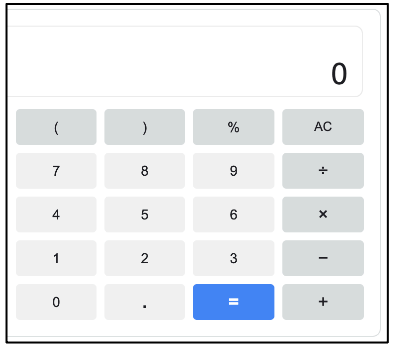
	0 / 0	1 / 0	1 + (1 / 0)
Coq proof assistant	**0**	**0**	**1**

calculator	0 / 0	1 / 0	1 + (1 / 0)
			
			
Coq proof assistant	**0**	**0**	**1**

calculator	0 / 0	1 / 0	1 + (1 / 0)

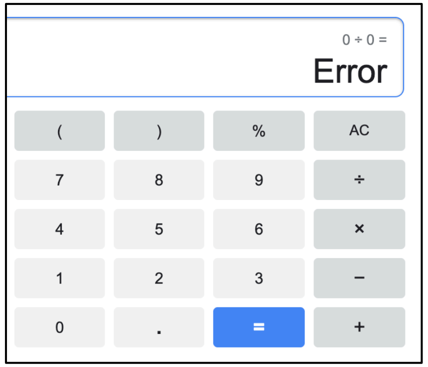
calculator





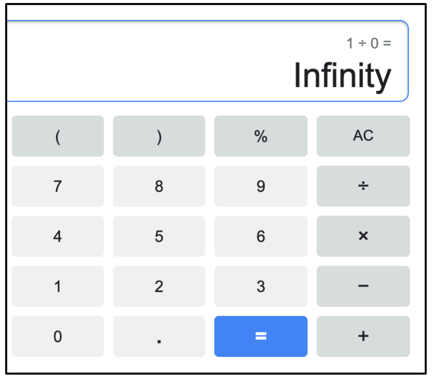
Coq proof assistant

0 / 0



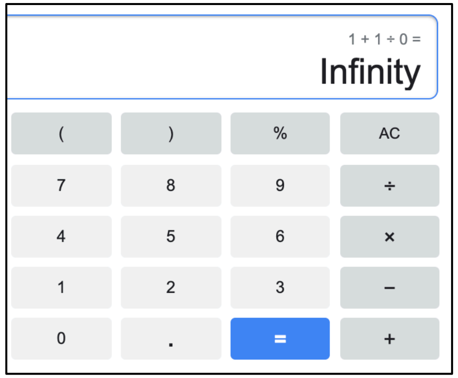
0

1 / 0



0

1 + (1 / 0)



1

1. How to compose computational effects?
2. How to handle effects according to applications?

Algebraic effects and handlers

Composable and modular computational effects

Algebraic effects and handlers

Composable and modular computational effects

Algebraic effects and handlers

Composable and modular computational effects

algebraic effects

define a family of operations

Algebraic effects and handlers

Composable and modular computational effects

algebraic effects

define a family of operations

effect handlers

give semantics to operations

Algebraic effects and handlers

Composable and modular computational effects

algebraic effects

define a family of operations

Algebraic Operations and Generic Effects

Applied Categorical Structures 2003

Gordon Plotkin and John Power *

Division of Informatics, University of Edinburgh, King's Buildings,
Edinburgh EH9 3JZ, Scotland

effect handlers

give semantics to operations

Algebraic effects and handlers

Composable and modular computational effects

algebraic effects

define a family of operations

Algebraic Operations and Generic Effects

Applied Categorical Structures 2003

Gordon Plotkin and John Power *

Division of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland

effect handlers

give semantics to operations

Handlers of Algebraic Effects

ESOP 2019

Gordon Plotkin * and Matija Pretnar **

Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Scotland

HANDLING ALGEBRAIC EFFECTS *

Logical Methods in Computer Science 2013

GORDON D. PLOTKIN [a] AND MATIJA PRETNAR [b]

Eff

Programming with algebraic effects and handlers

Journal of Logical and Algebraic Methods in Programming 2015

Andrej Bauer, Matija Pretnar *

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Eff

Programming with algebraic effects and handlers

Journal of Logical and Algebraic Methods in Programming 2015

Andrej Bauer, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types

Mathematically Structured Functional Programming 2014

Daan Leijen
Microsoft Research
daan@microsoft.com

Eff

Programming with algebraic effects and handlers

Journal of Logical and Algebraic Methods in Programming 2015

Andrej Bauer, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types

Mathematically Structured Functional Programming 2014

Daan Leijen
Microsoft Research
daan@microsoft.com

Links

Row-based Effect Types for Database Integration

TLDI 2012

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

James Cheney
The University of Edinburgh
jcheney@inf.ed.ac.uk

Eff

Programming with algebraic effects and handlers

Journal of Logical and Algebraic Methods in Programming 2015

Andrej Bauer, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types

Mathematically Structured Functional Programming 2014

Daan Leijen

Microsoft Research

daan@microsoft.com

Links

Row-based Effect Types for Database Integration

TLDI 2012

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

James Cheney
The University of Edinburgh
jcheney@inf.ed.ac.uk

Frank

Do Be Do Be Do

POPL 2017

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk

Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk

Craig McLaughlin
The University of Edinburgh, UK
craig.mclaughlin@ed.ac.uk

Eff

Programming with algebraic effects and handlers

Journal of Logical and Algebraic Methods in Programming 2015

Andrej Bauer, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types

Mathematically Structured Functional Programming 2014

Daan Leijen
Microsoft Research
daan@microsoft.com

Links

Row-based Effect Types for Database Integration

TLDI 2012

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

James Cheney
The University of Edinburgh
jcheney@inf.ed.ac.uk

Frank

Do Be Do Be Do

POPL 2017

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk

Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk

Craig McLaughlin
The University of Edinburgh, UK
craig.mclaughlin@ed.ac.uk

Effekt

Effekt: Lightweight Effect Polymorphism for Handlers

OOPSLA 2020

JONATHAN IMMANUEL BRACHTHÄUSER, EPFL, Switzerland
PHILIPP SCHUSTER, University of Tübingen, Germany
KLAUS OSTERMANN, University of Tübingen, Germany

OCaml

Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

OCaml

Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

https://discuss.ocaml.org/t/multicore-ocaml-september-2021-effect-handlers-will-be-in-ocaml-5-0/8554

Multicore OCaml: September 2021, effect handlers will be in OCaml 5.0!

Community multicore, multicore-monthly

avsm Maintainer 19d

Welcome to the September 2021 Multicore OCaml 27 monthly report! This month's update along with the previous updates 2 have been compiled by me, **@ctk21**, **@kayceesrk** and **@shakthimaan**. The team has been working over the past few months to finish the last few features 18 necessary to reach feature parity with stock OCaml. We also worked closely with the core OCaml team to develop the timeline for upstreaming Multicore OCaml to stock OCaml, and have now agreed that:

OCaml 5.0 will support shared-memory parallelism through domains *and* direct-style concurrency through effect handlers (without syntactic support).

OCaml

Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

https://discuss.ocaml.org/t/multicore-ocaml-september-2021-effect-handlers-will-be-in-ocaml-5-0/8554

Multicore OCaml: September 2021, effect handlers will be in OCaml 5.0!

Community multicore, multicore-monthly

avsm Maintainer 19d

Welcome to the September 2021 Multicore OCaml 27 monthly report! This month's update along with the previous updates 2 have been compiled by me, **@ctk21**, **@kayceesrk** and **@shakthimaan**. The team has been working over the past few months to finish the last few features 18 necessary to reach feature parity with stock OCaml. We also worked closely with the core OCaml team to develop the timeline for upstreaming Multicore OCaml to stock OCaml, and have now agreed that:

OCaml 5.0 will support shared-memory parallelism through domains *and* direct-style concurrency through effect handlers (without syntactic support).

WEBASSEMBLY

https://github.com/WebAssembly/design/issues/1359

WebAssembly / **design** Public

Code | Issues 176 | Pull requests 10 | Actions | Projects | Security

Typed continuations to model stacks #1359

Open **rossberg** opened this issue on Jul 29, 2020 · 68 comments

OCaml

Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

https://discuss.ocaml.org/t/multicore-ocaml-september-2021-effect-handlers-will-be-in-ocaml-5-0/8554

Multicore OCaml: September 2021, effect handlers will be in OCaml 5.0!

Community multicore, multicore-monthly

avsm Maintainer 19d

Welcome to the September 2021 Multicore OCaml 27 monthly report! This month's update along with the previous updates 2 have been compiled by me, @ctk21, @kayceesrk and @shakthimaan. The team has been working over the past few months to finish the last few features 18 necessary to reach feature parity with stock OCaml. We also worked closely with the core OCaml team to develop the timeline for upstreaming Multicore OCaml to stock OCaml, and have now agreed that:

OCaml 5.0 will support shared-memory parallelism through domains *and* direct-style concurrency through effect handlers (without syntactic support).

WA

WEBASSEMBLY

https://github.com/WebAssembly/design/issues/1359

WebAssembly / design Public

Code Issues 176 Pull requests 10 Actions Projects Security

Typed continuations to model stacks #1359

Open rossberg opened this issue on Jul 29, 2020 · 68 comments

One specific way of typing continuations and the values communicated back and forth is by following the approach taken by so-called *effect handlers*, one modern way of representing delimited continuations,…

React: A JavaScript library for building user interfaces

https://reesew.io/posts/react-algebraic-effects/

Article — JavaScript

Algebraic Effects for React Developers

Reese Williams
01 Nov 2020 • 11 min read

PYRO: Deep Universal Probabilistic Programming Language

https://docs.pyro.ai/en/dev/poutine.html

Poutine (Effect handlers)

Beneath the built-in inference algorithms, Pyro has a library of composable effect handlers for creating new inference algorithms and working with probabilistic programs. Pyro's inference algorithms are all built by applying these handlers to stochastic functions. In order to get a general understanding what effect handlers are and what problem they solve, read An Introduction to Algebraic Effects and Handlers by Matija Pretnar.

April 22 – 27 , 2018, Dagstuhl Seminar 18172

Algebraic Effect Handlers go Mainstream

Organizers

Sivaramakrishnan Krishnamoorthy Chandrasekaran (University of Cambridge, GB)

Daan Leijen (Microsoft Research – Redmond, US)

Matija Pretnar (University of Ljubljana, SI)

Tom Schrijvers (KU Leuven, BE)

Agenda

- Algebraic effects 101
- Examples, and more examples
- Efficient compilation of algebraic effects
- Koka: algebraic effects via evidence-passing semantics

Algebraic effects 101

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 signature

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

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

Algebraic effects 101

effect signature

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

operation

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

Algebraic effects 101

effect signature

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

operation

effect handler

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

Algebraic effects 101

effect signature

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

operation

effect handler

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

implementation

Algebraic effects 101

effect signature

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

operation

argument

effect handler

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

implementation

Algebraic effects 101

effect signature

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

operation

argument

resumption

effect handler

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

implementation

Algebraic effects 101

effect signature

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

operation

argument

resumption

effect handler

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

implementation

computation

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

Algebraic effects 101

effect signature

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

operation

argument

resumption

effect handler

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

implementation

computation

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

perform an effect

Algebraic effects 101

effect signature

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

operation

argument

resumption

effect handler

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

implementation

computation

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

perform an effect

Exception

```
effect exn {
  throw : () -> a
}
```

Exception

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

Exception

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
handler {
  throw x k -> Nothing
} (\_.
  Just (div 42 2)
) // Just 21
```

```
handler {
  throw x k -> Nothing
} (\_.
 Just (div 42 0)
) // Nothing
```

Exception

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
  div 42 2
) // Just 21
```

```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
 div 42 0
) // Nothing
```

Exception

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
handler {
  throw x k -> []
  return v  -> [v]
} (\_.
  div 42 2
) // Just 21
```

```
handler {
  throw x k -> []
  return v  -> [v]
} (\_.
 div 42 0
) // Nothing
```

```
2 * (1 + 20)
```

```
2 * (1 + 20)
```

2 *

1 +

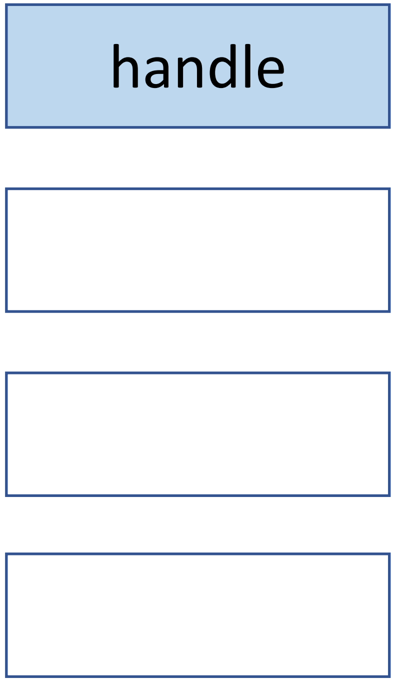
20

```
2 * (1 + 20)        2 * 21
```

2 *	2 *
1 +	21
20	

`2 * (1 + 20)`   `2 * 21`   `42`

2 *	2 *	42
1 +	21	
20		


handle

```
return x -> e1
```

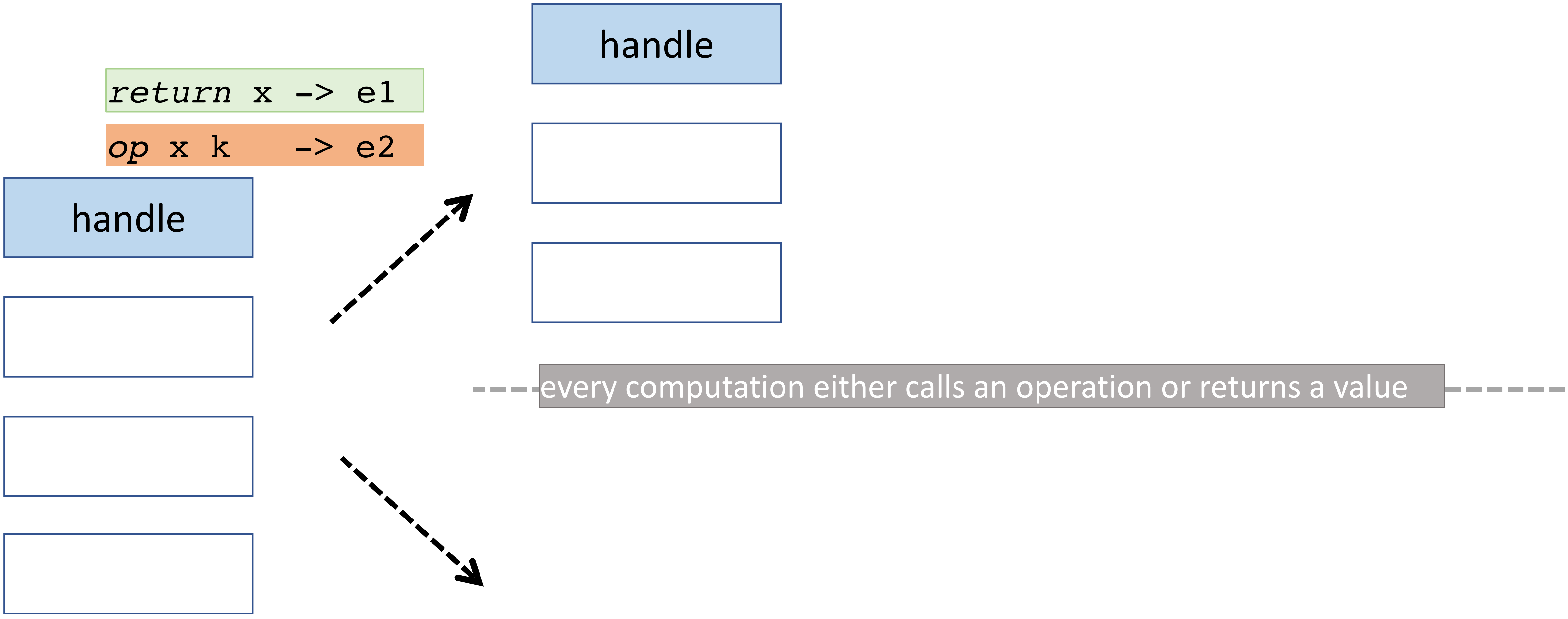
handle

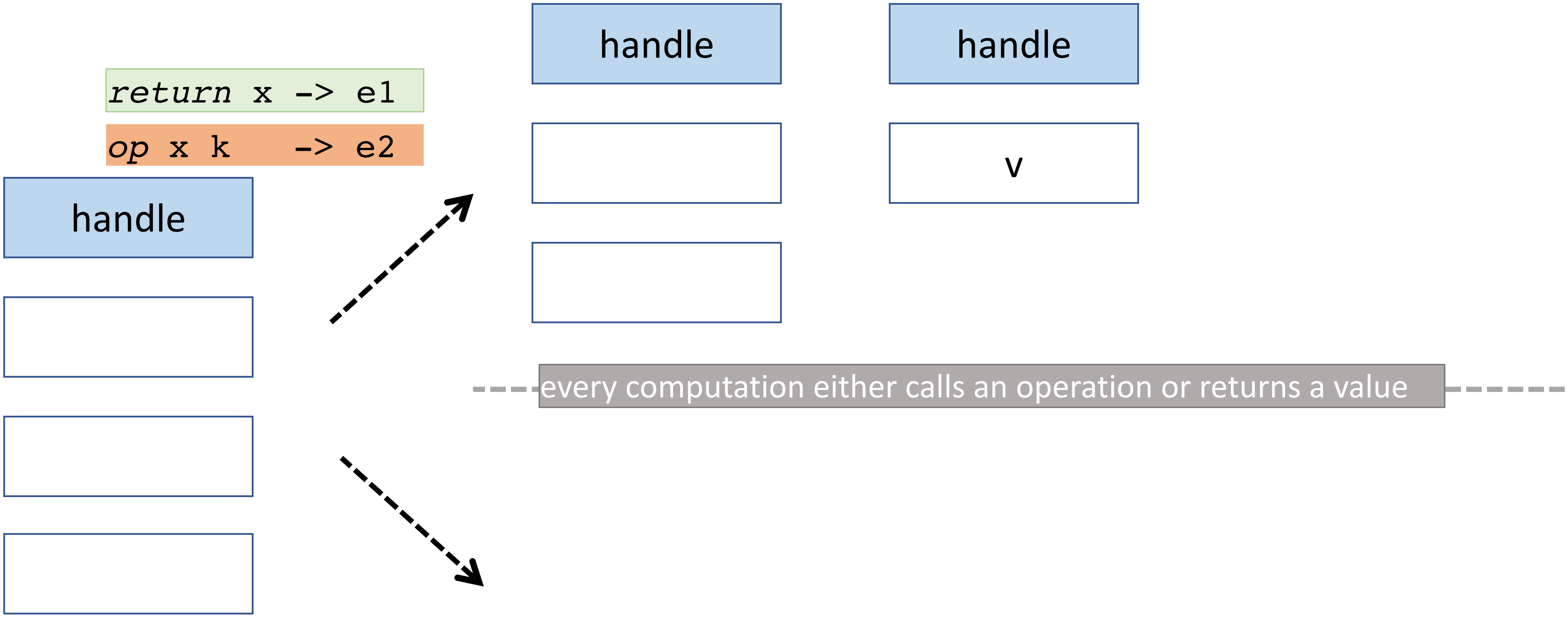
```
return x -> e1
op x k   -> e2
```

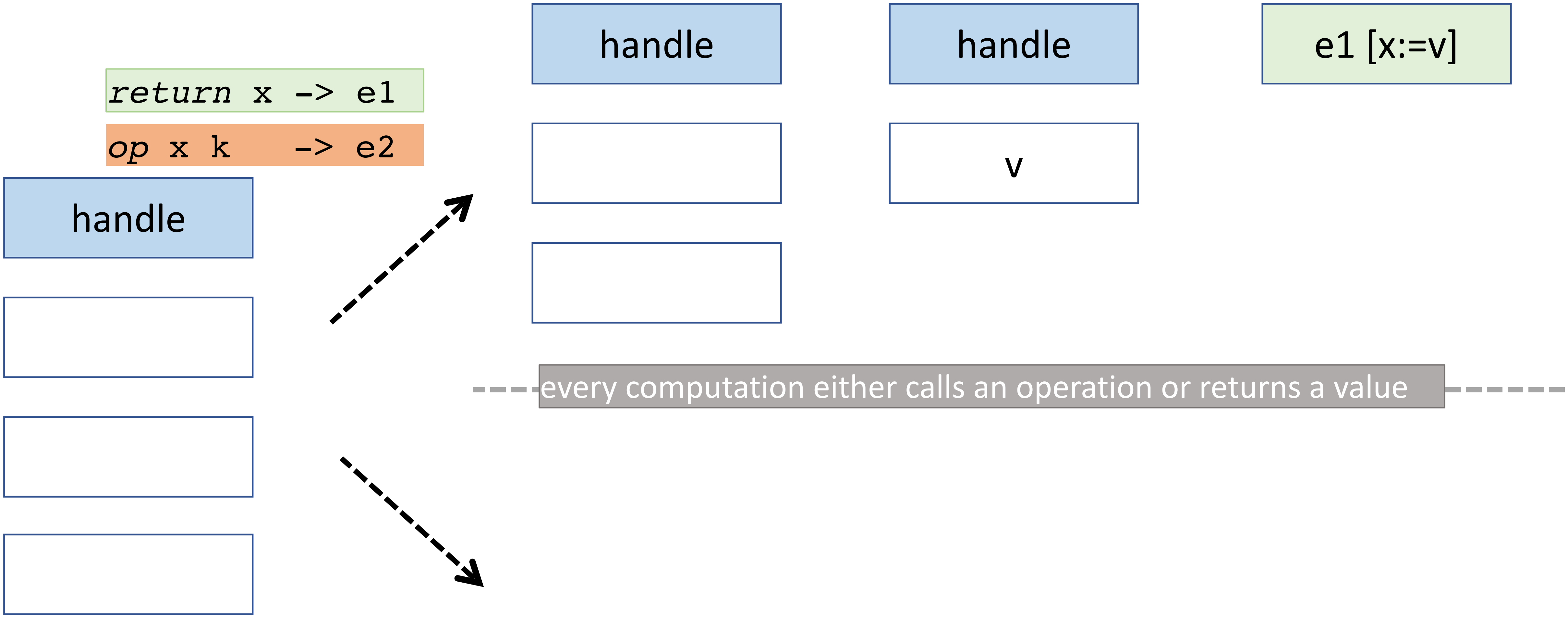
handle

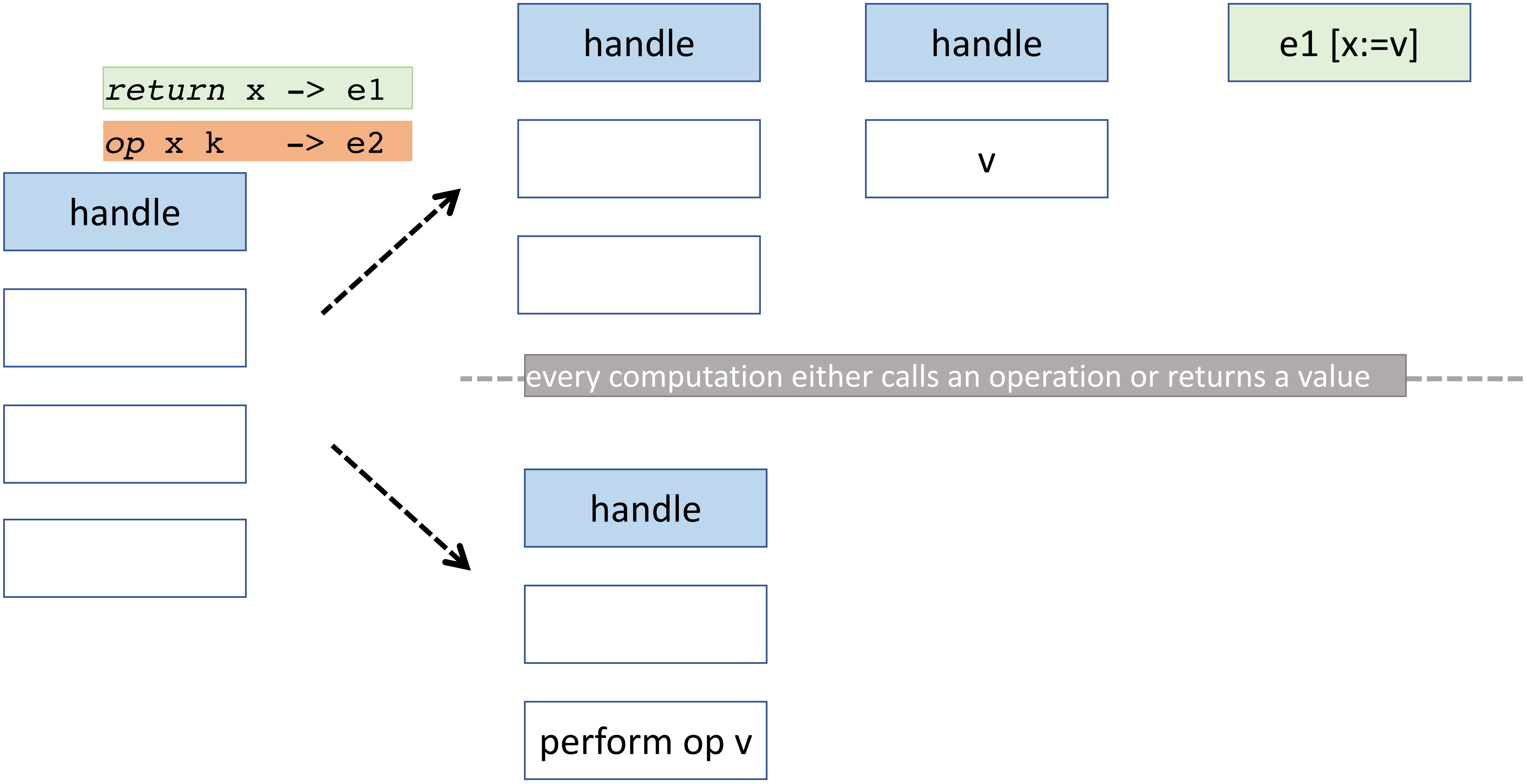

return x -> e1
op x k -> e2
handle

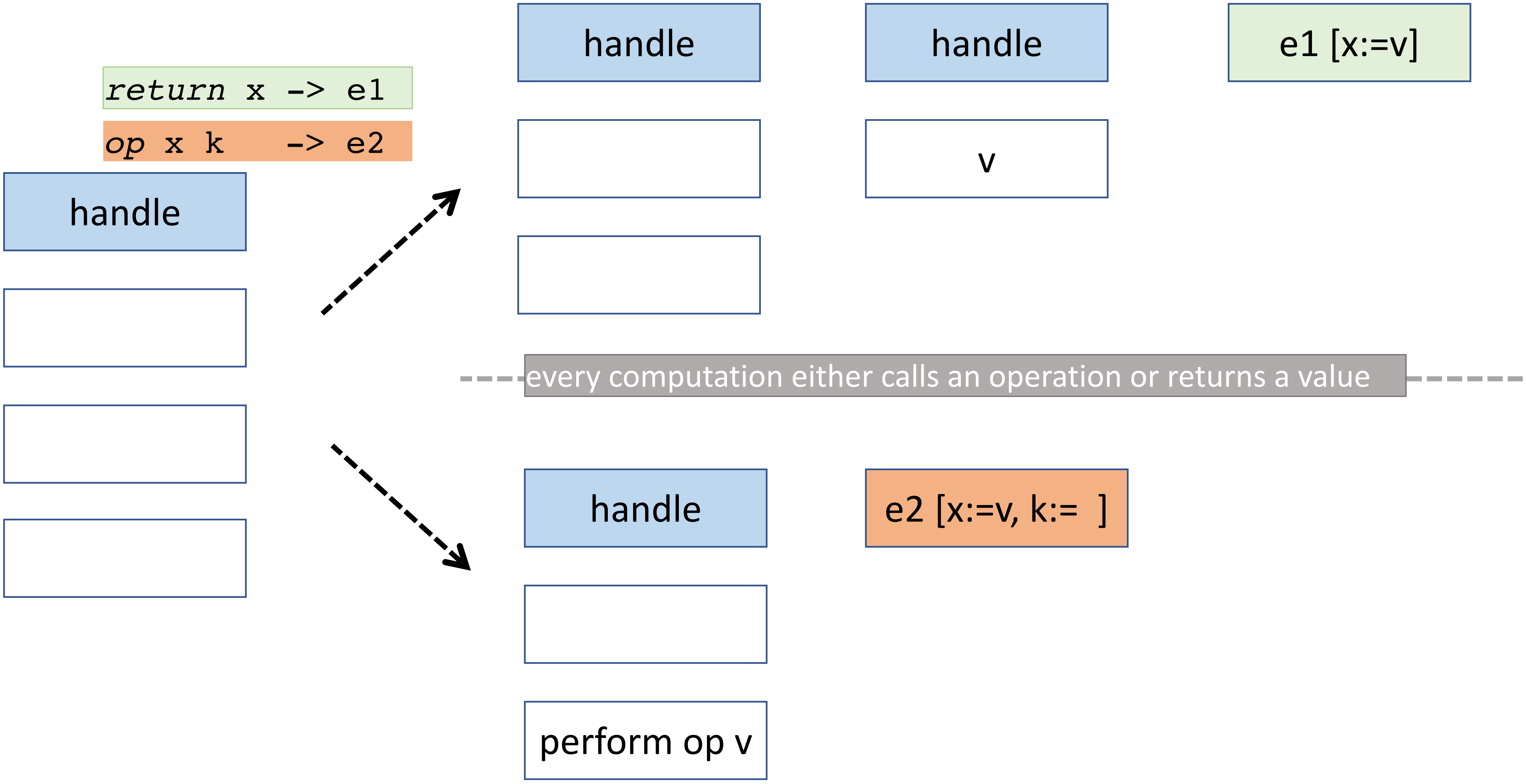

return x -> e1
op x k -> e2
handle
every computation either calls an operation or returns a value


return x -> e1
op x k -> e2
handle
handle
every computation either calls an operation or returns a value

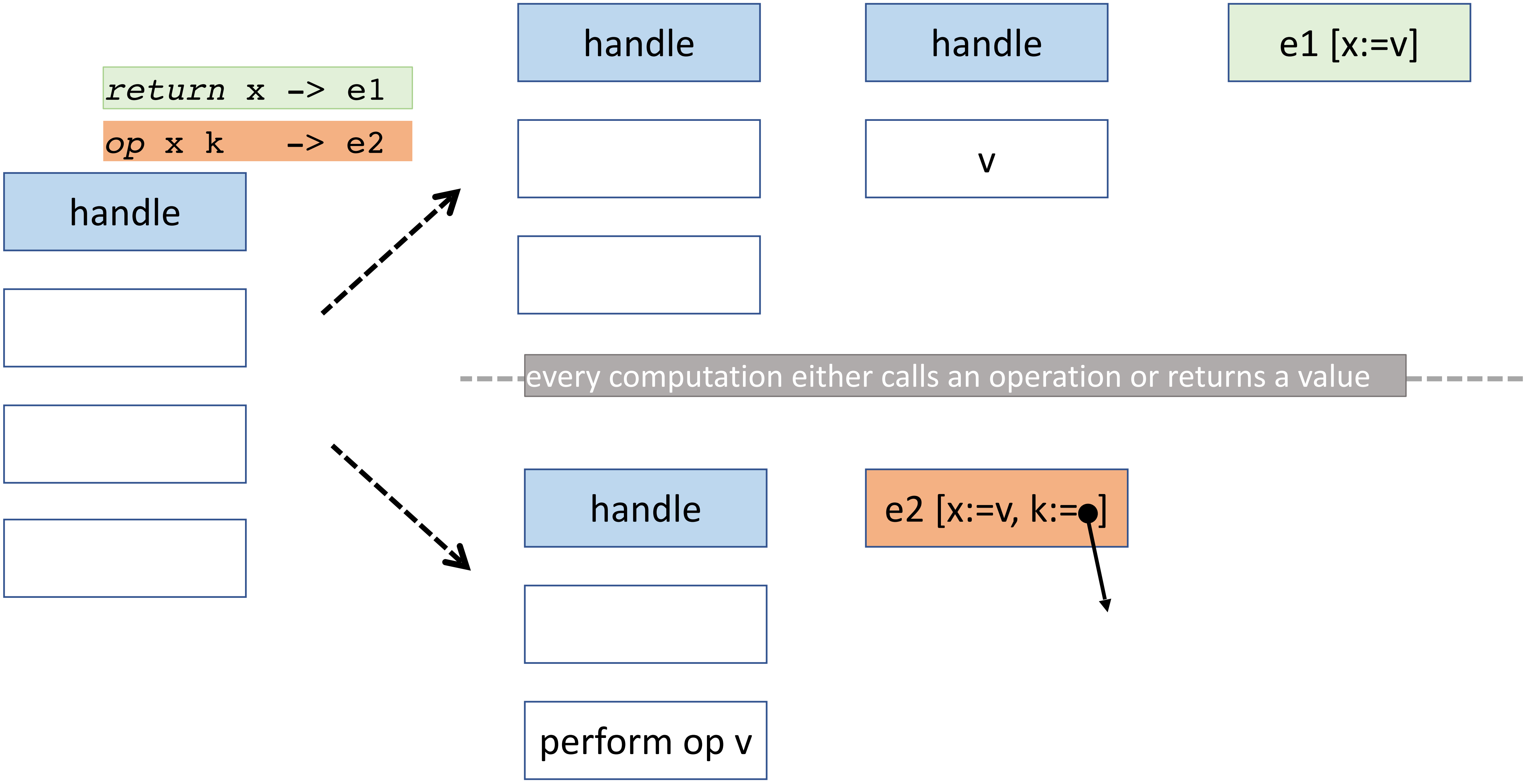

return x -> e1
op x k -> e2
handle
handle
handle
v
every computation either calls an operation or returns a value


return x -> e1
op x k -> e2
handle
handle
handle
v
e1 [x:=v]
every computation either calls an operation or returns a value

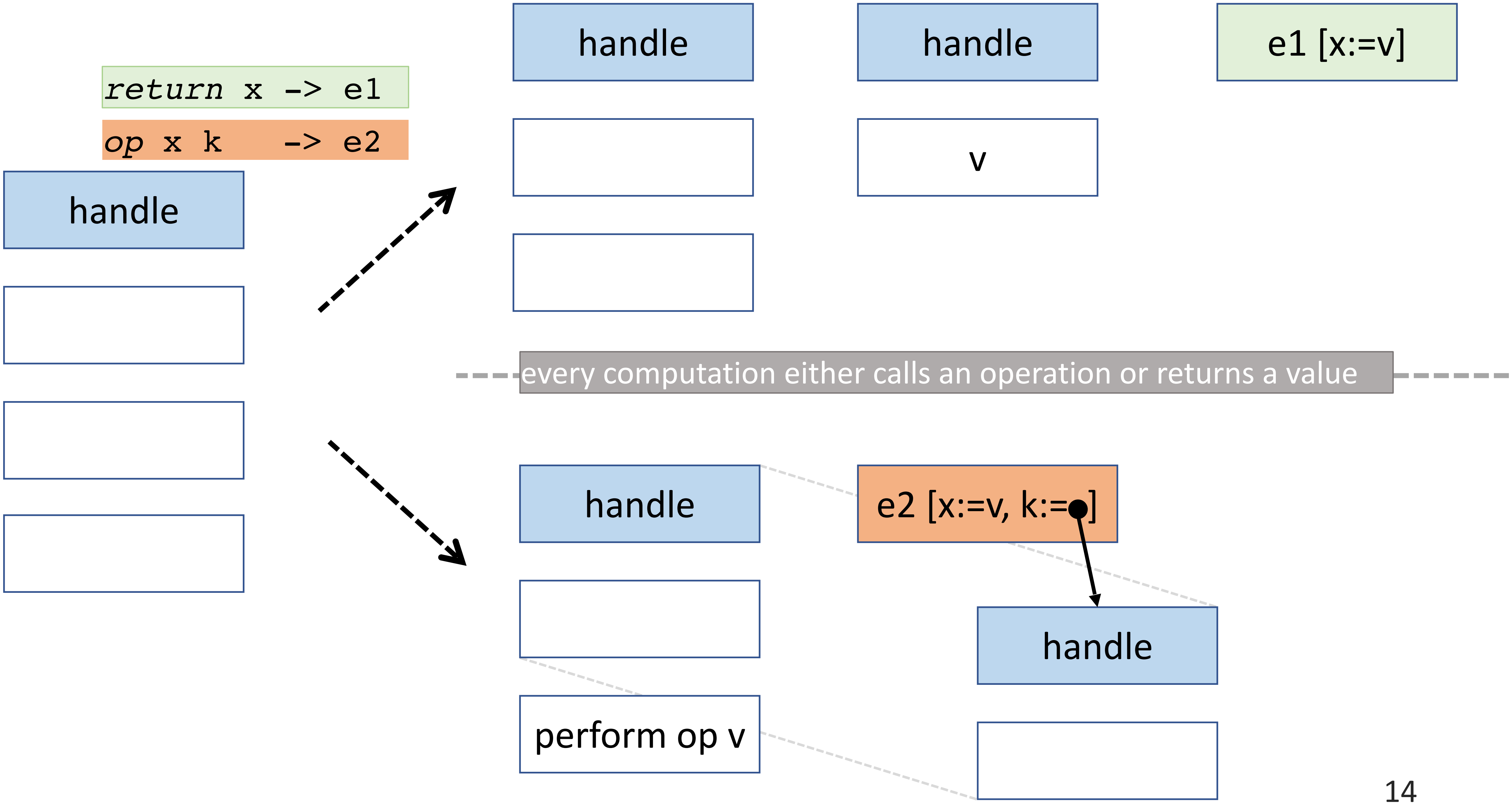

return x -> e1
op x k -> e2
handle
handle
handle
v
e1 [x:=v]
every computation either calls an operation or returns a value
handle
perform op v

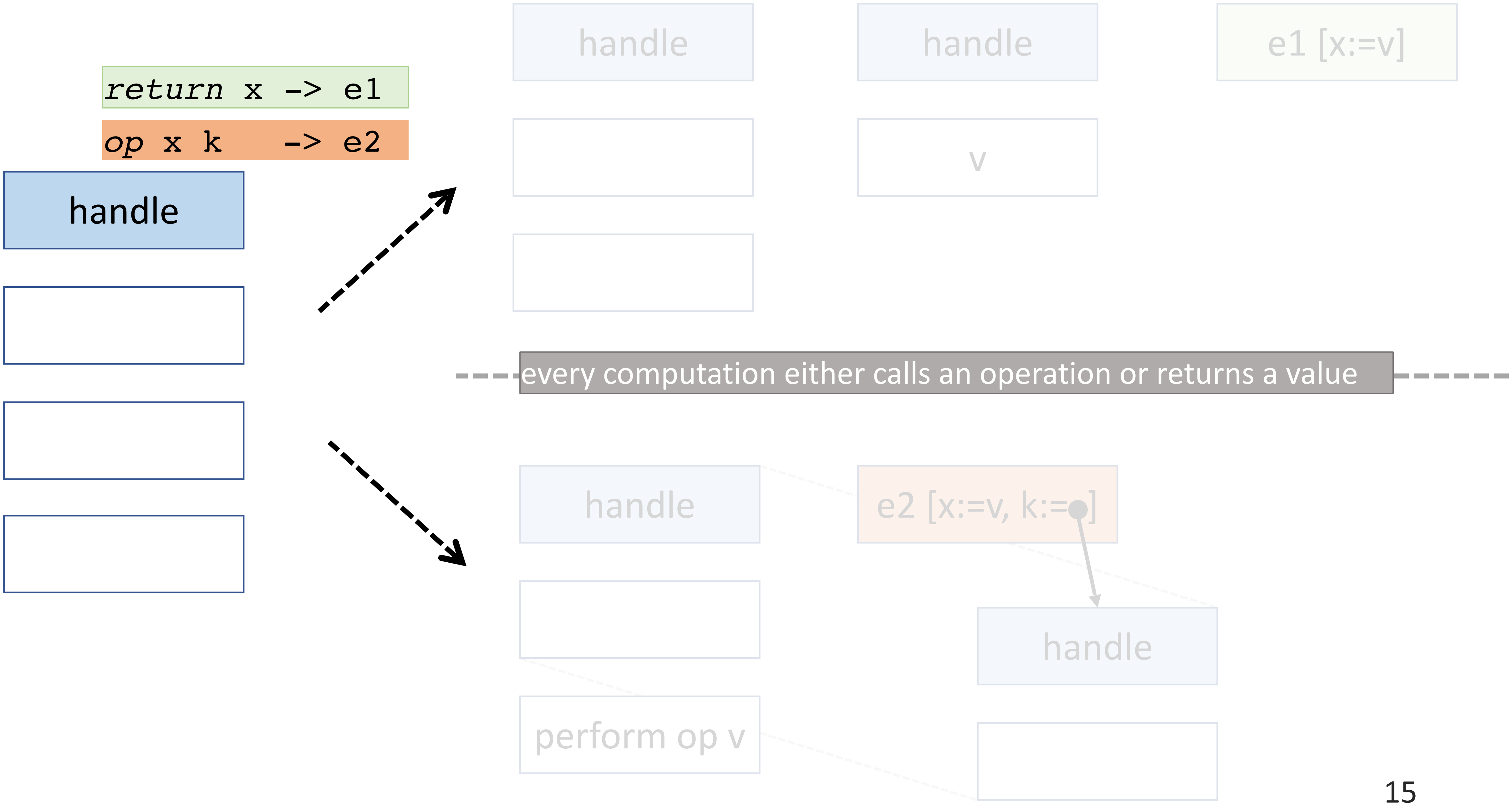


return x -> e1
op x k -> e2
handle
handle
handle
e1 [x:=v]
v
every computation either calls an operation or returns a value
handle
e2 [x:=v, k:=]
perform op v



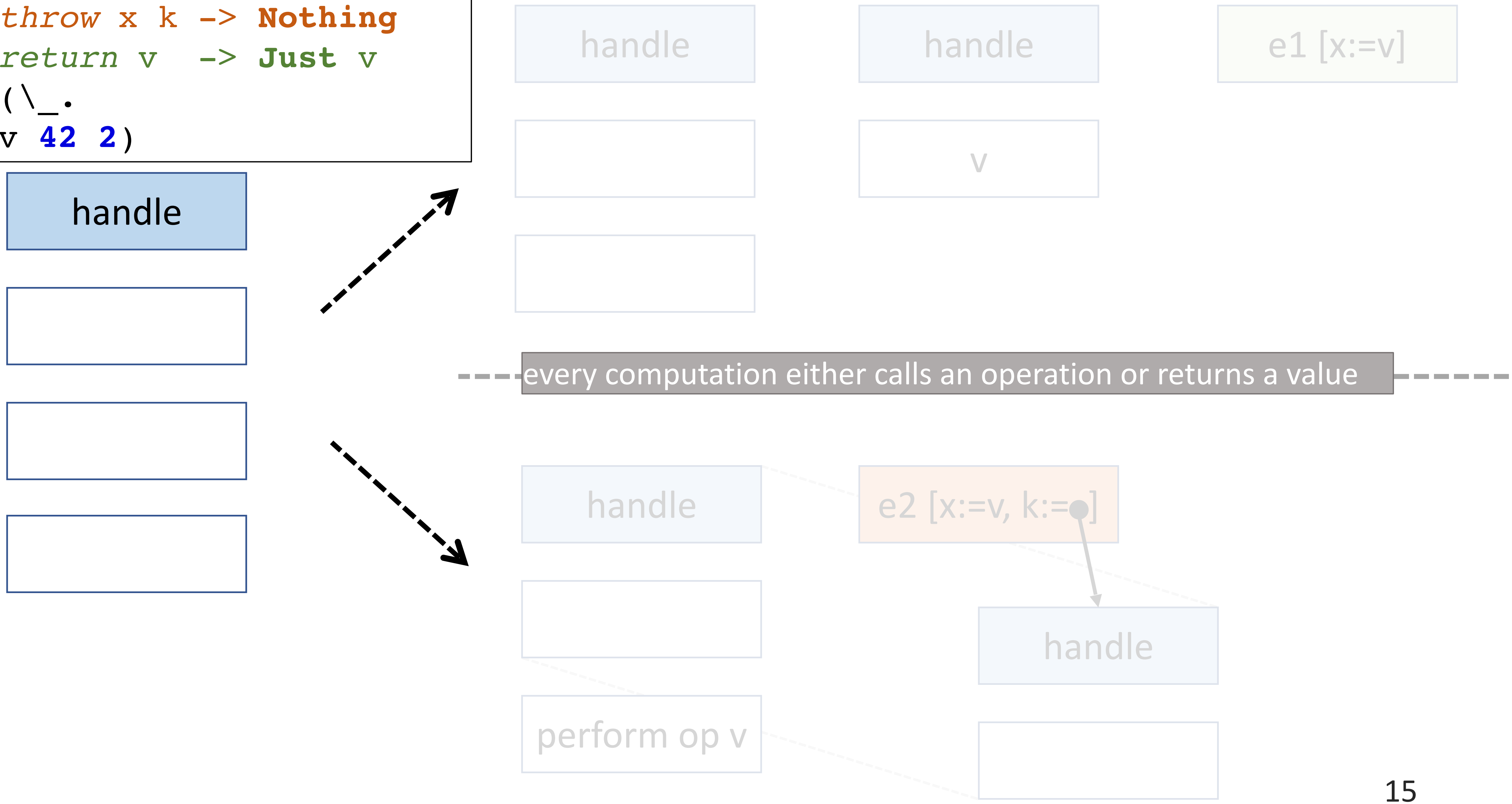
return x -> e1
op x k -> e2
handle
handle
handle
e1 [x:=v]
v
every computation either calls an operation or returns a value
handle
e2 [x:=v, k:=]
perform op v


return x -> e1
op x k -> e2
handle
handle
handle
e1 [x:=v]
v
every computation either calls an operation or returns a value
handle
e2 [x:=v, k:=●]
handle
perform op v

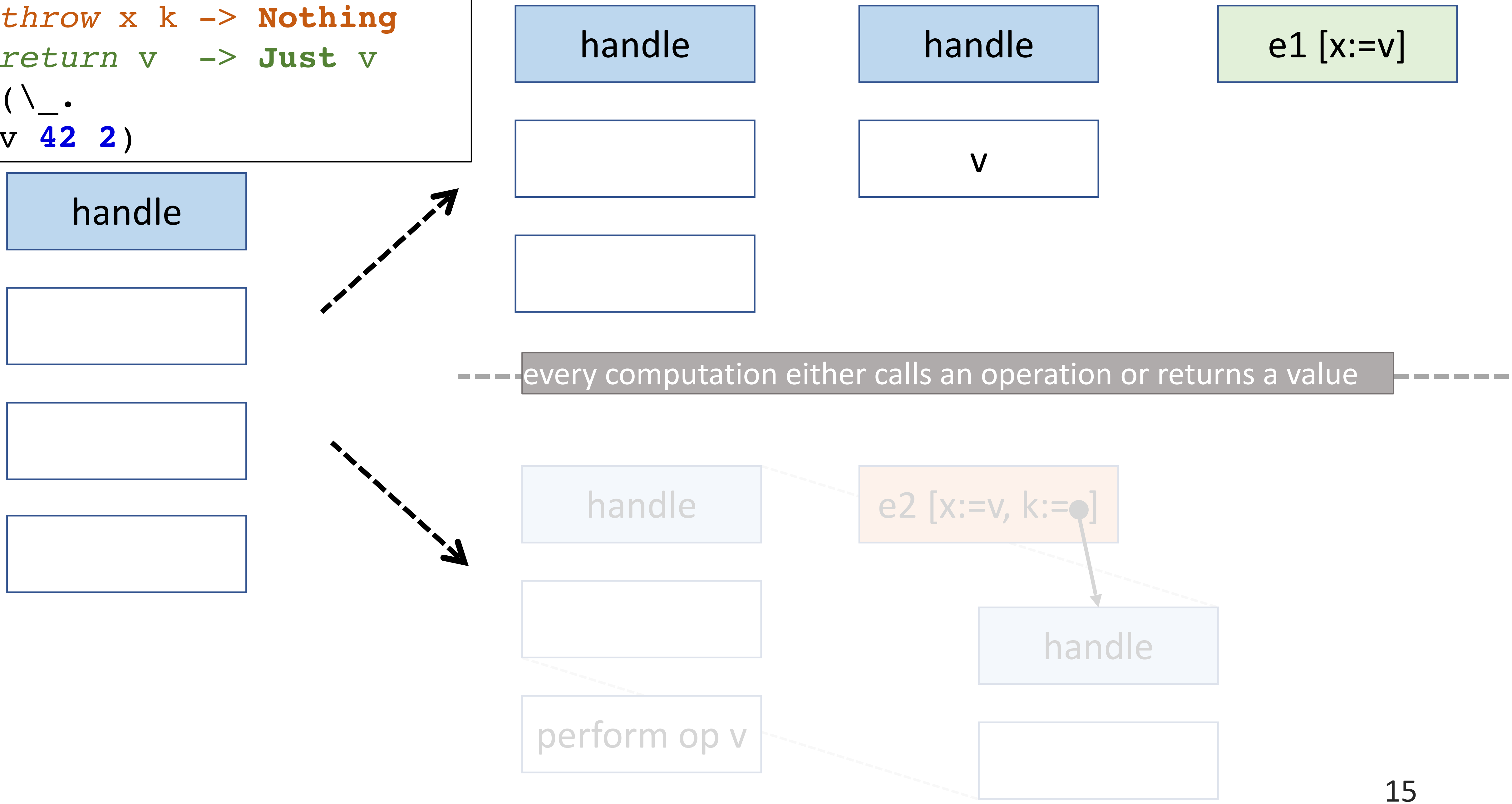


return x -> e1
op x k -> e2
handle
handle
handle
e1 [x:=v]
v
every computation either calls an operation or returns a value
handle
e2 [x:=v, k:=]
handle
perform op v

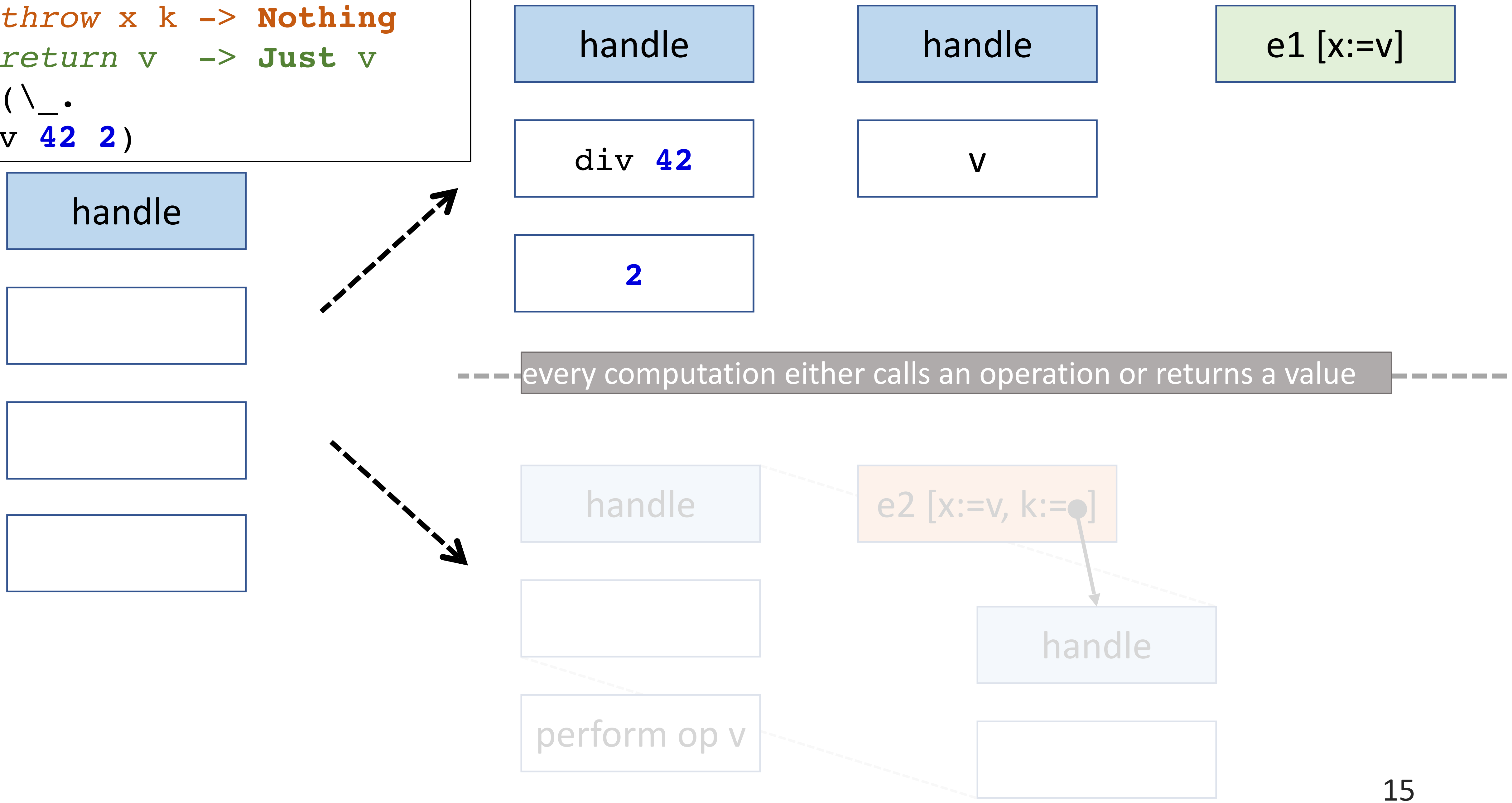
```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
div 42 2)
```



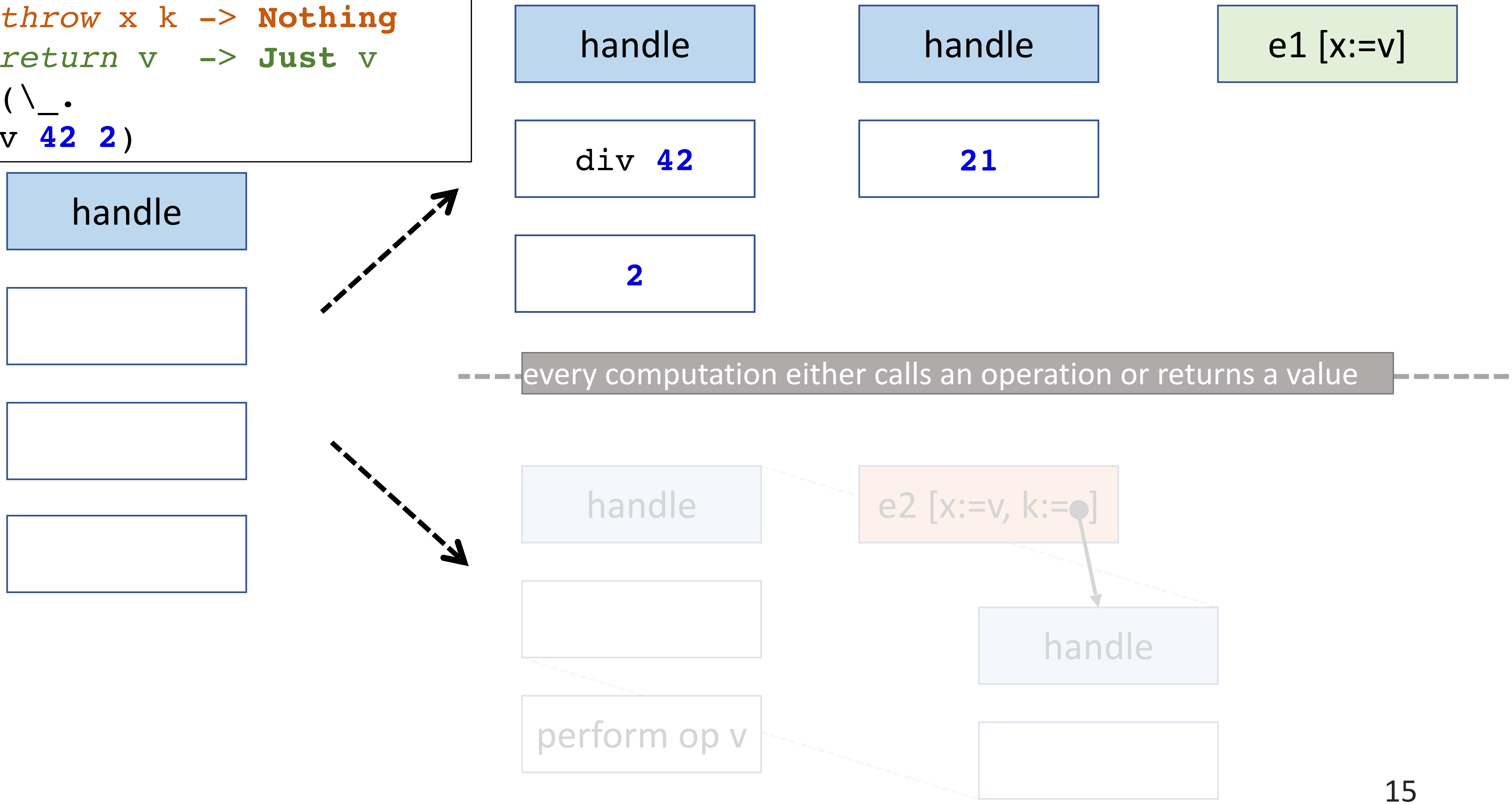
```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
div 42 2)
```



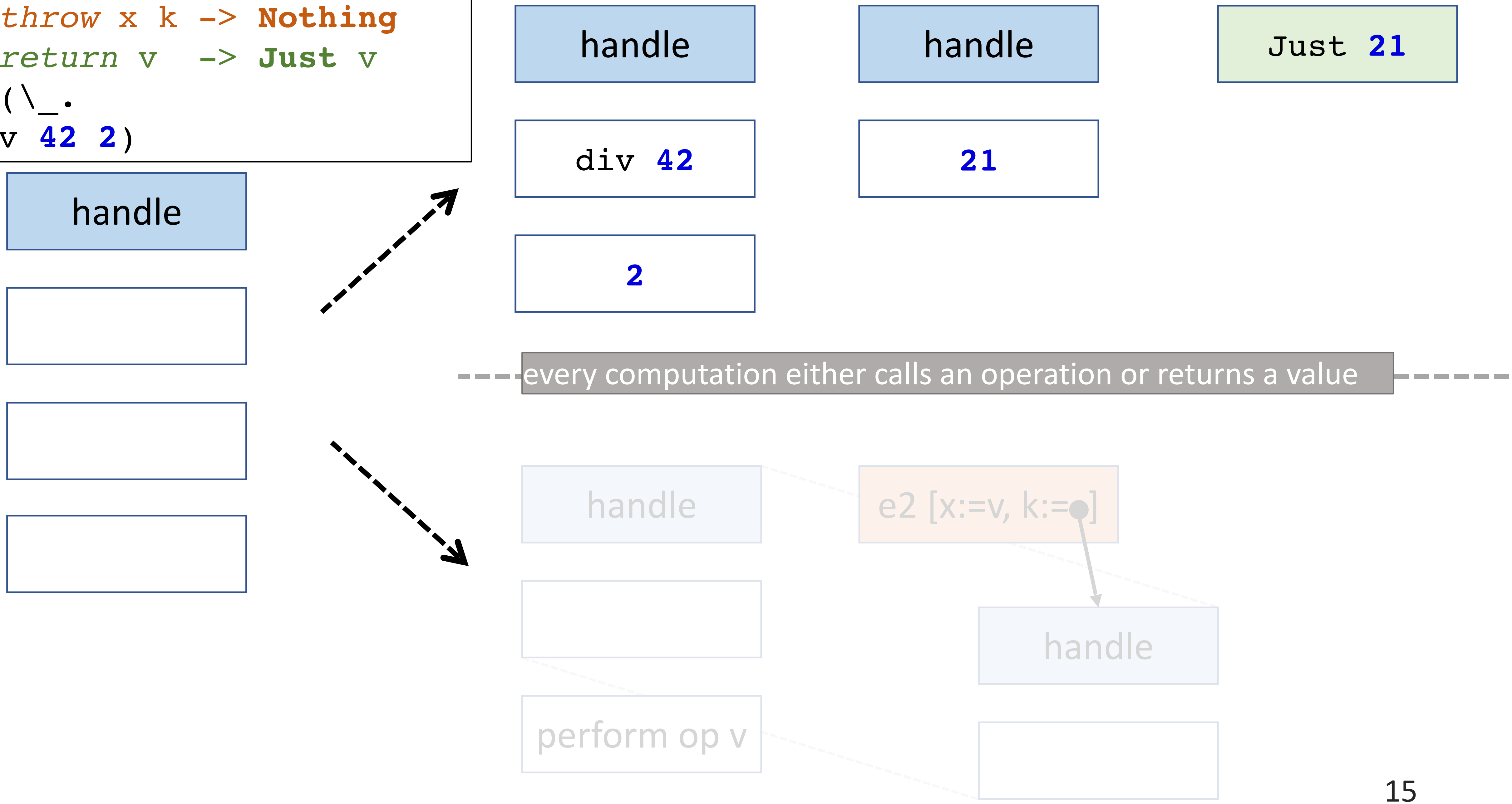
```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
div 42 2)
```

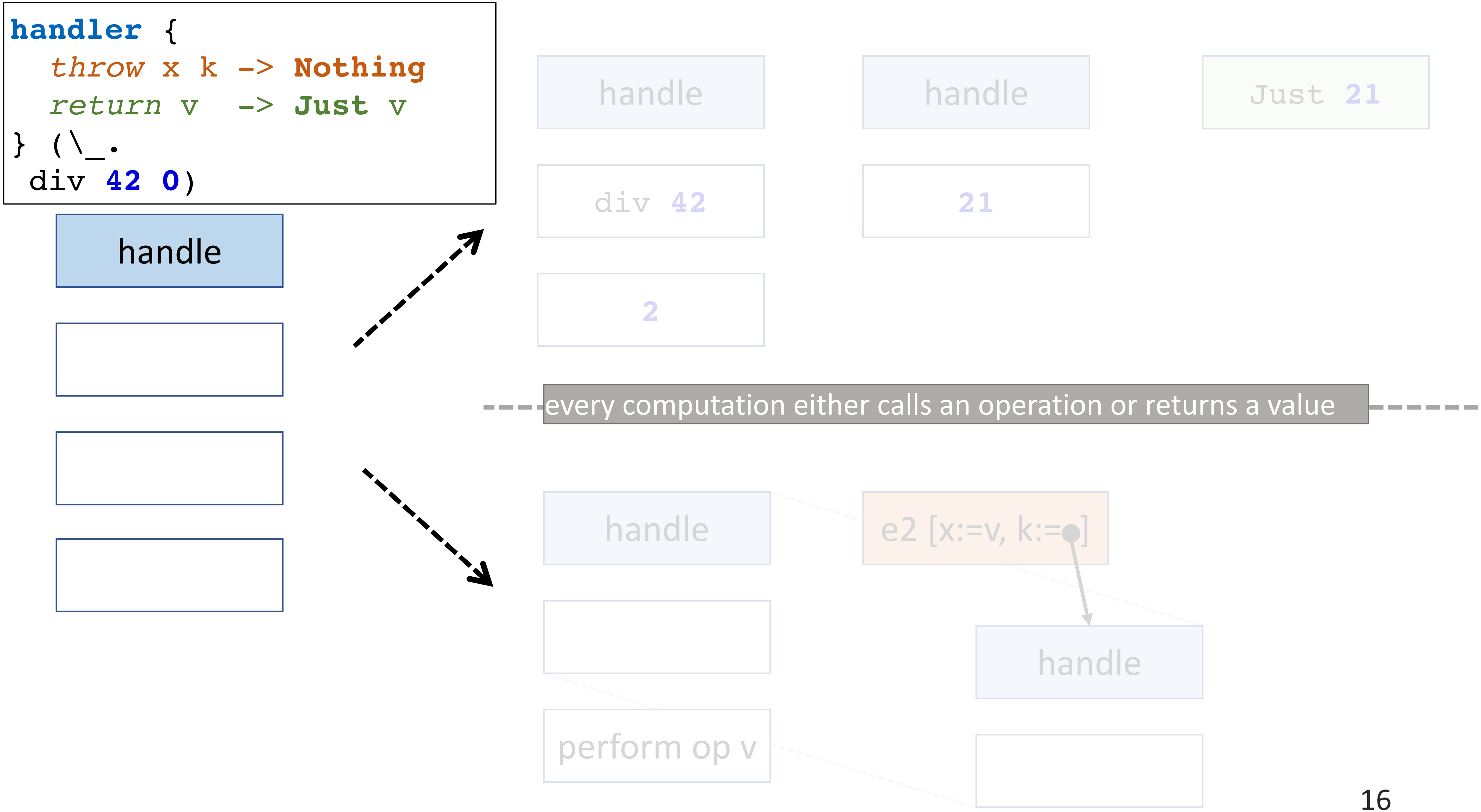


```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
div 42 2)
```

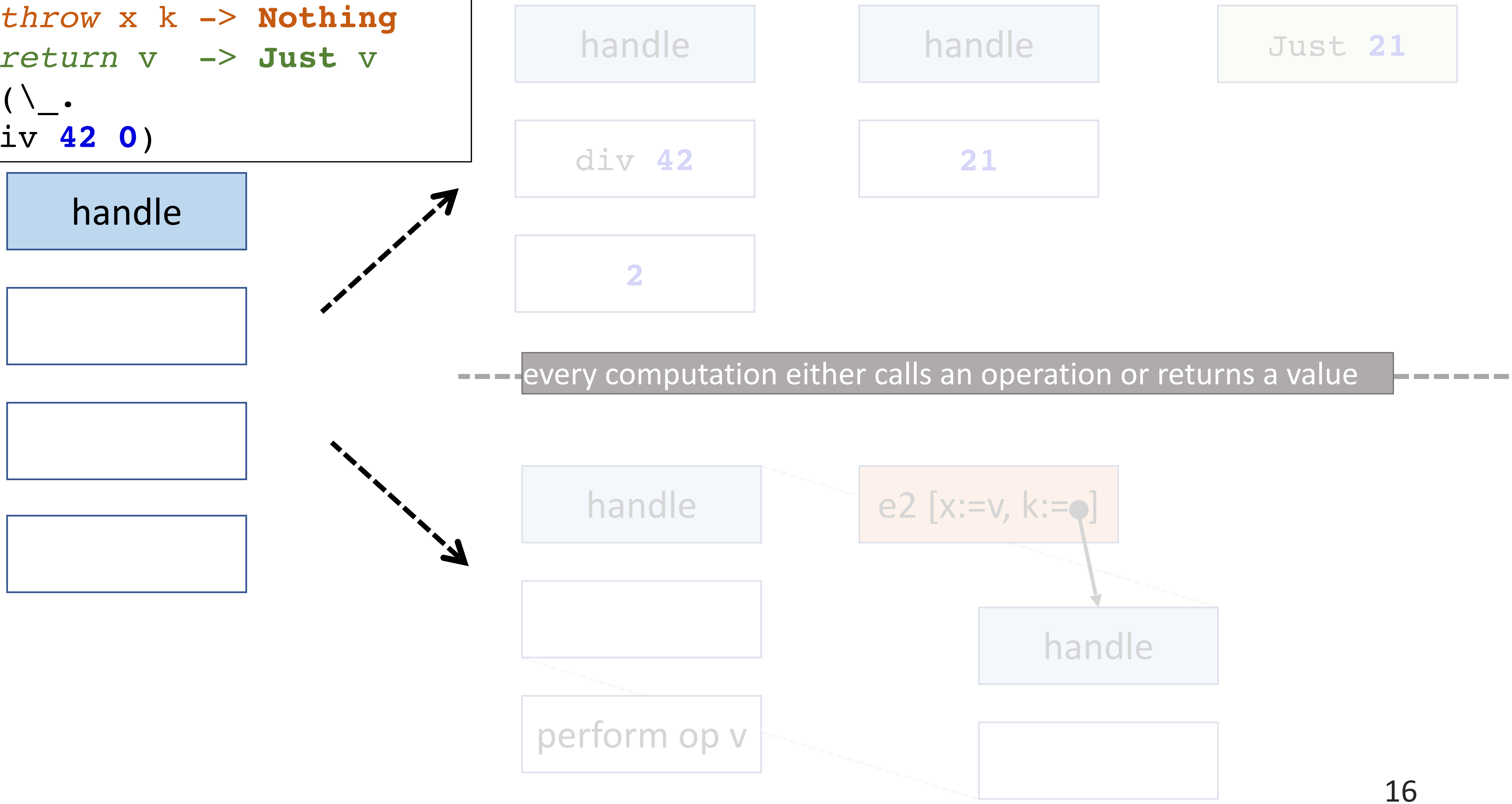


```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
div 42 2)
```

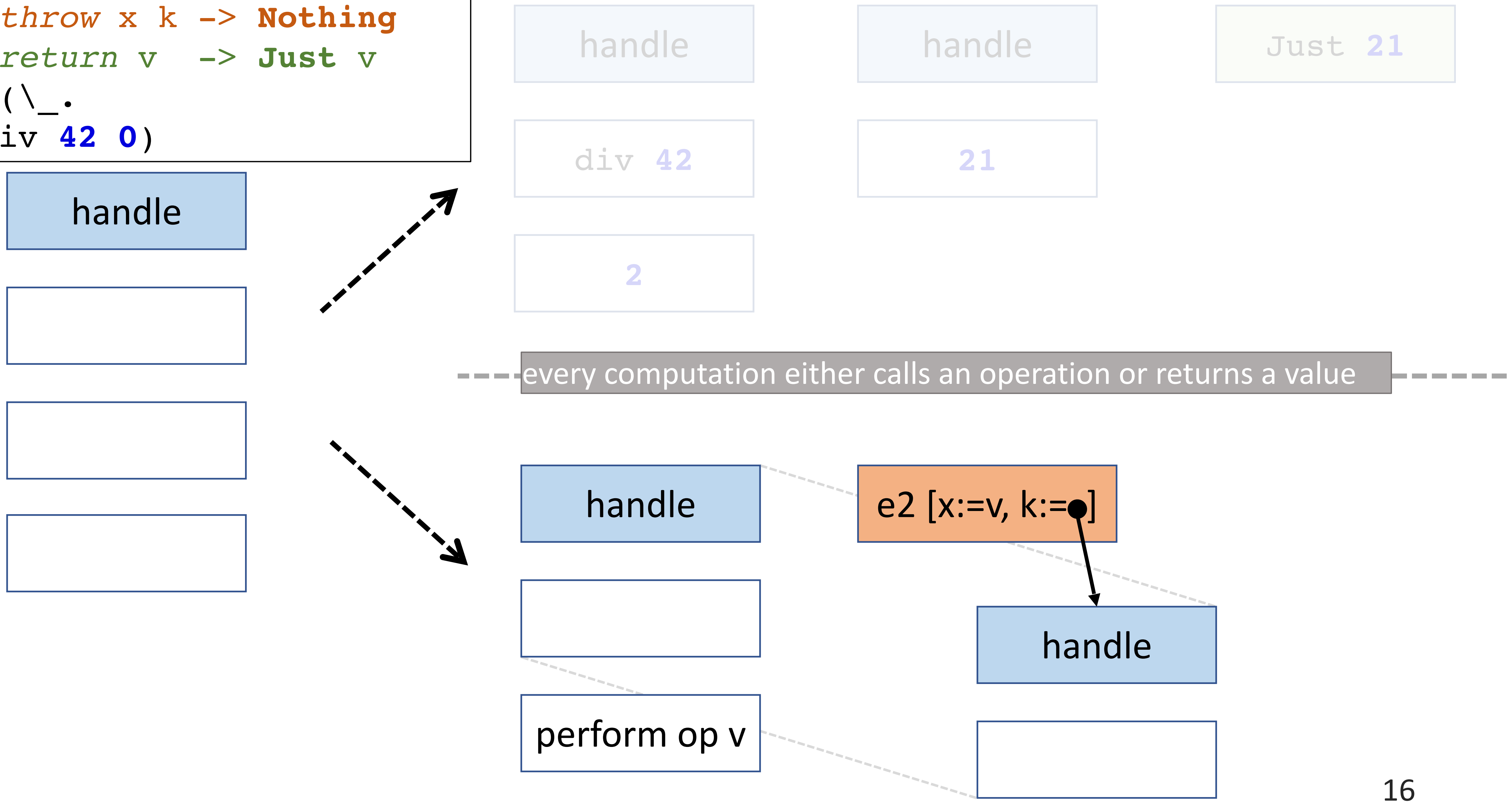



handler {
throw x k -> Nothing
return v -> Just v
} (_.
div 42 0)
handle
handle
handle
Just 21
div 42
21
2
every computation either calls an operation or returns a value
handle
e2 [x:=v, k:=•]
handle
perform op v

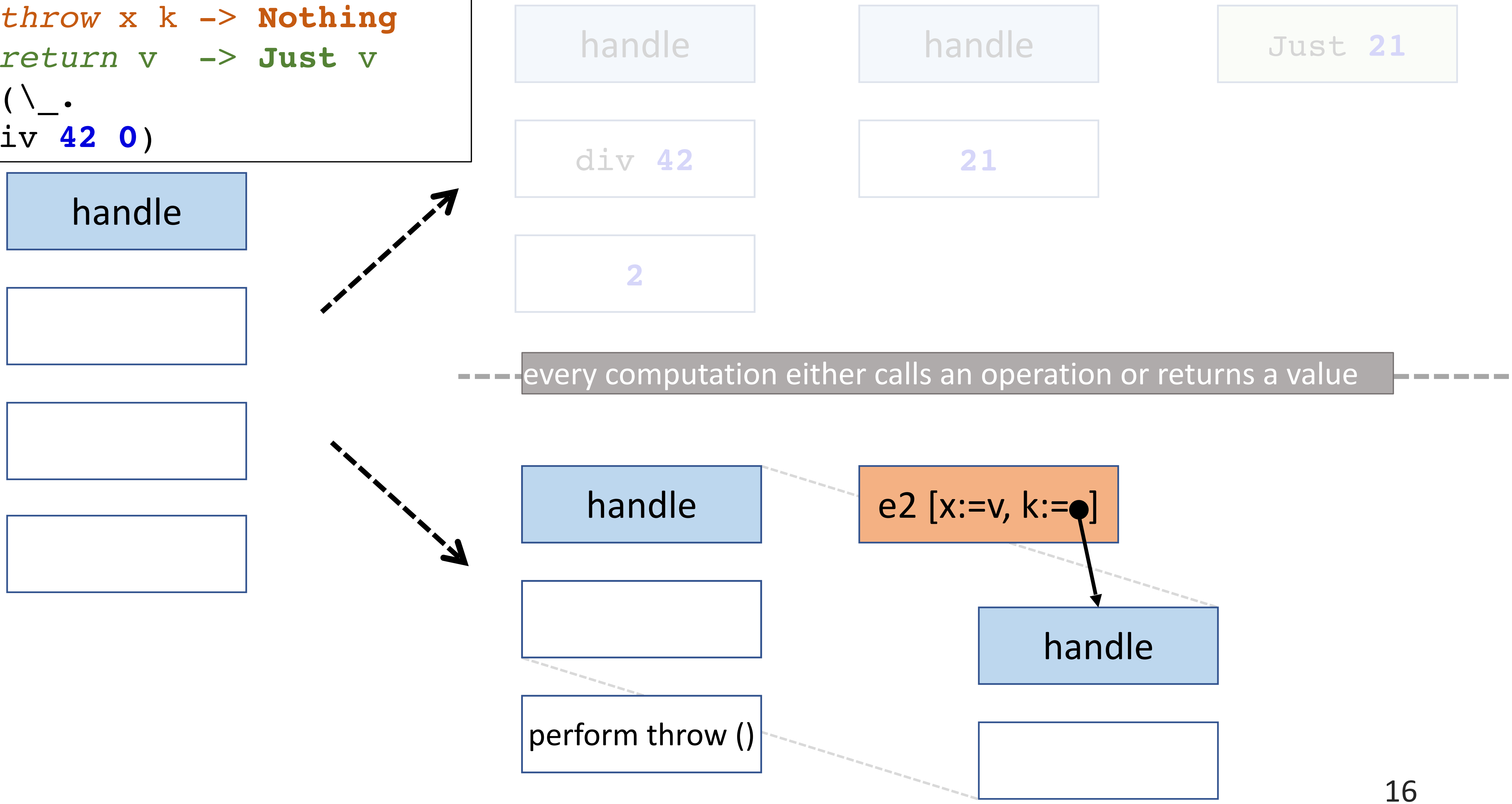
```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
 div 42 0)
```



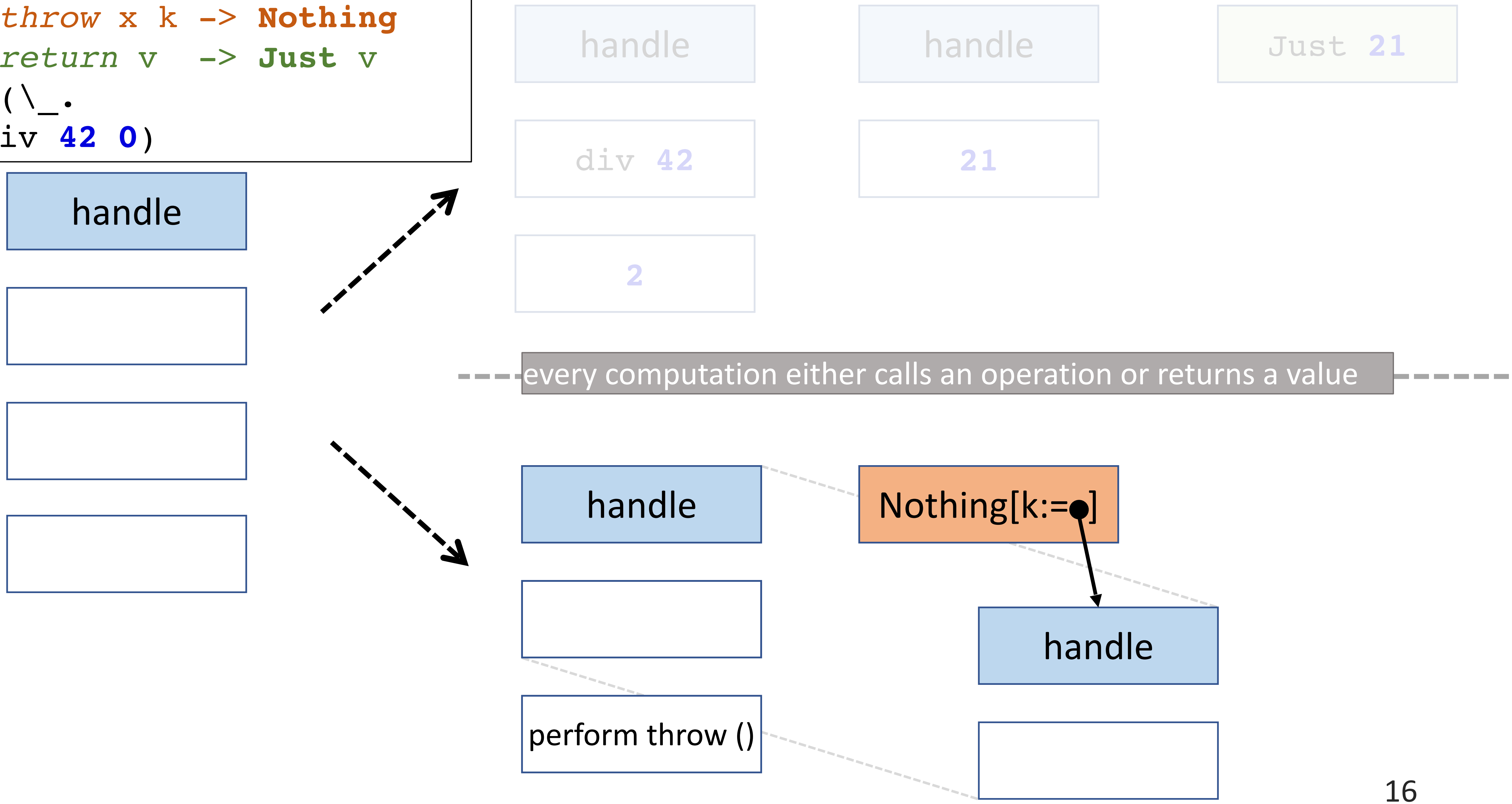
```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
 div 42 0)
```

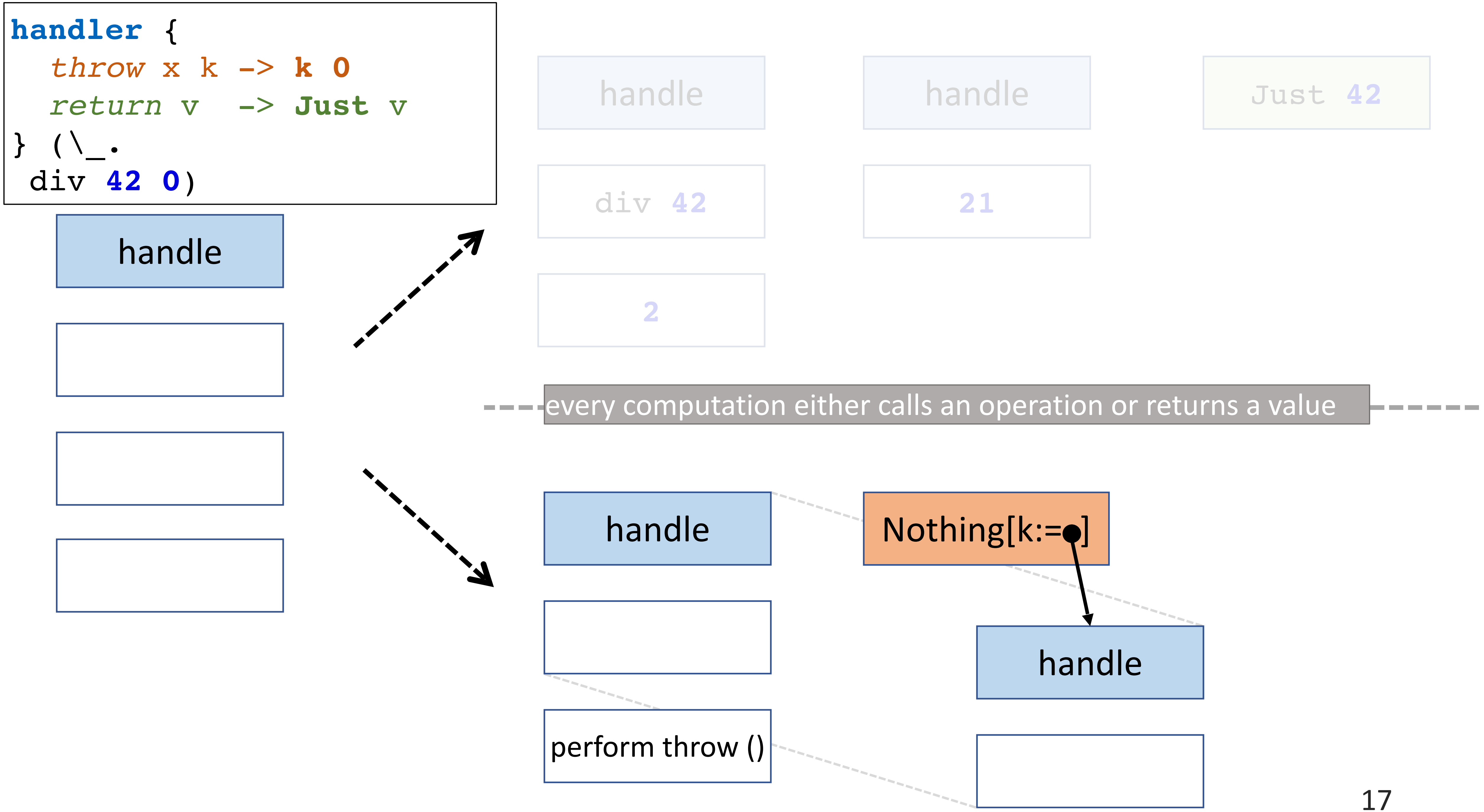


```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
 div 42 0)
```

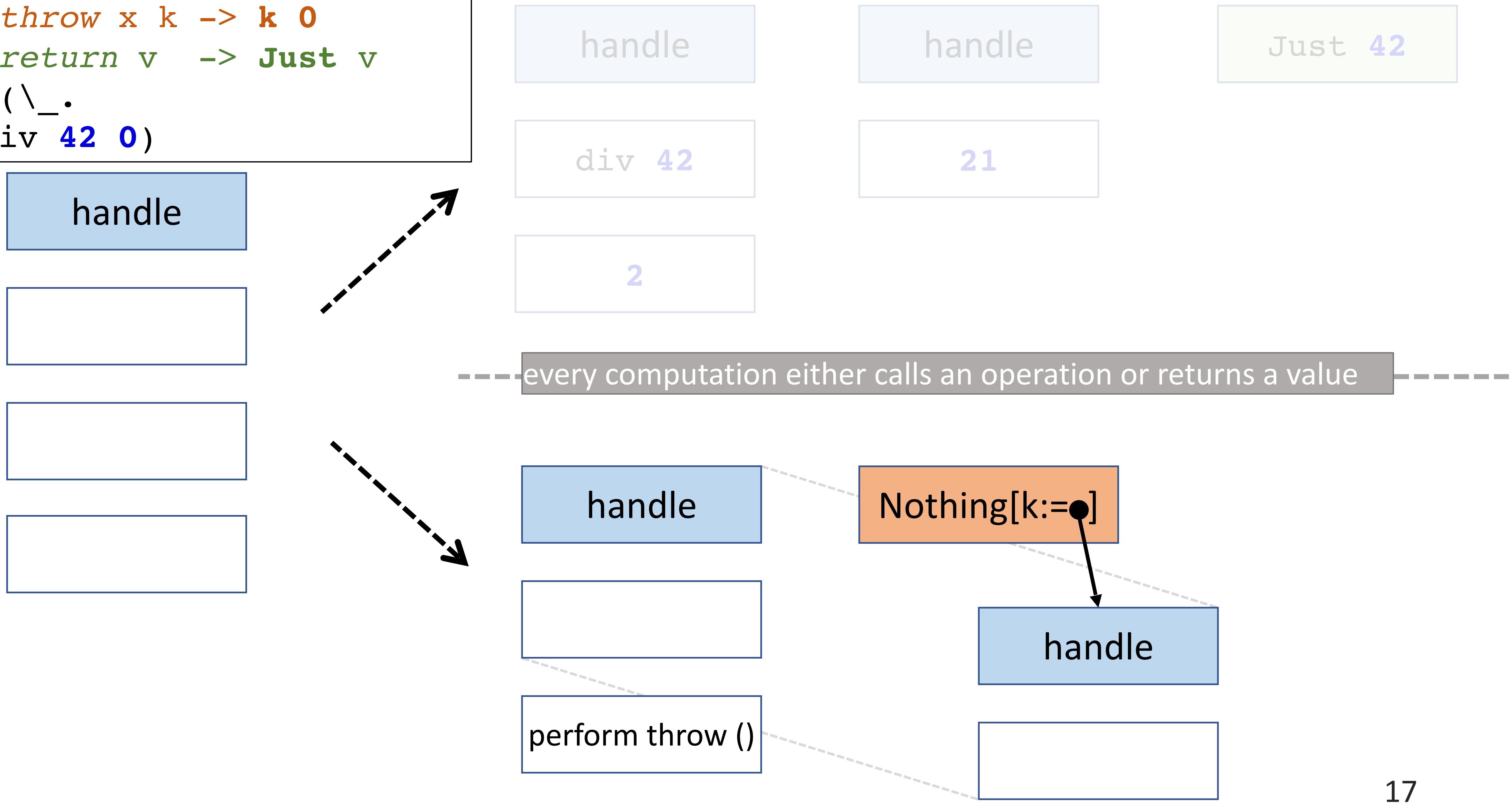


```
handler {
  throw x k -> Nothing
  return v  -> Just v
} (\_.
 div 42 0)
```

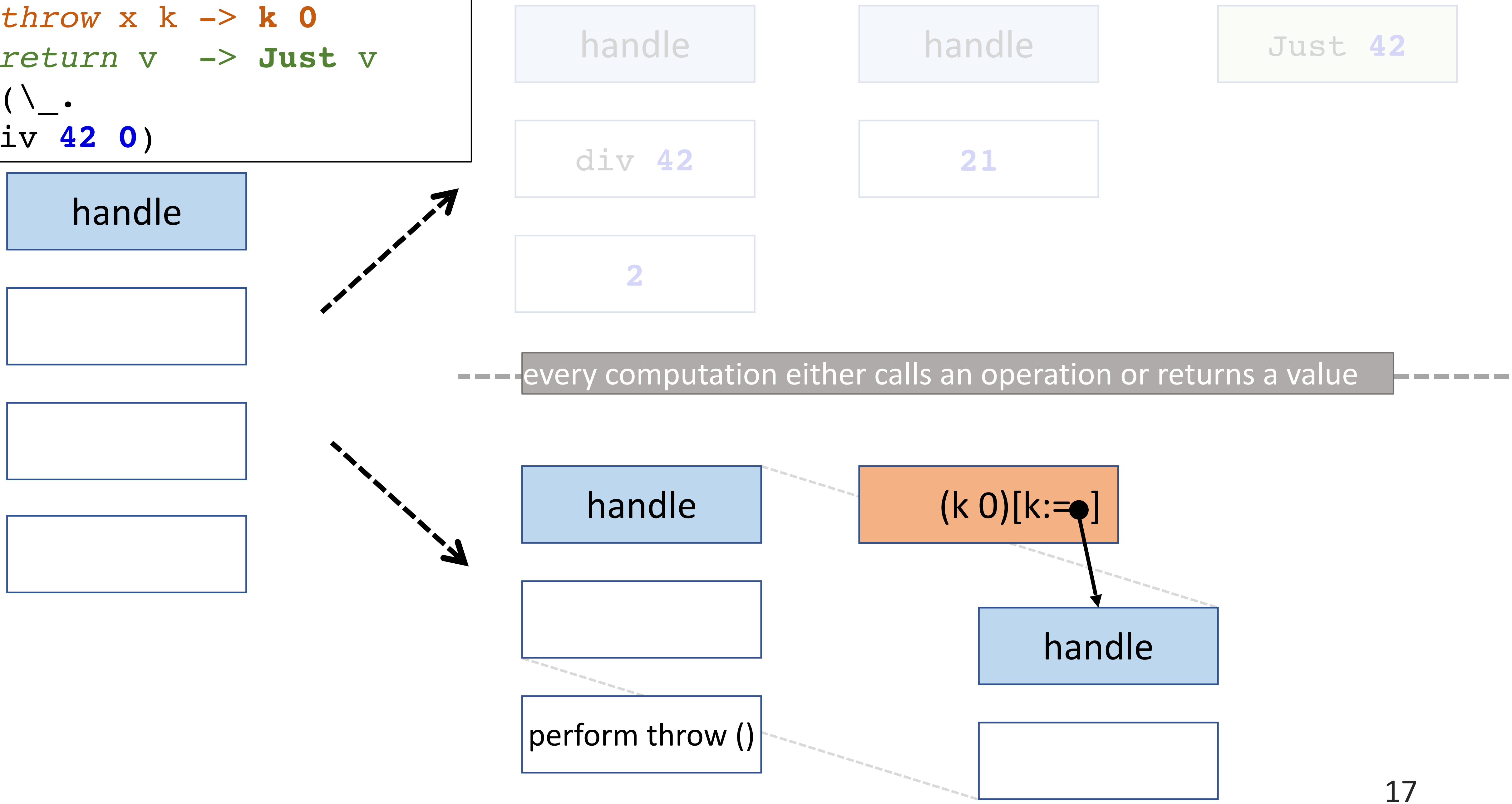



handler {
throw x k -> k 0
return v -> Just v
} (_.
div 42 0)
handle
handle
handle
Just 42
div 42
21
2
every computation either calls an operation or returns a value
handle
Nothing[k:=●]
perform throw ()
handle

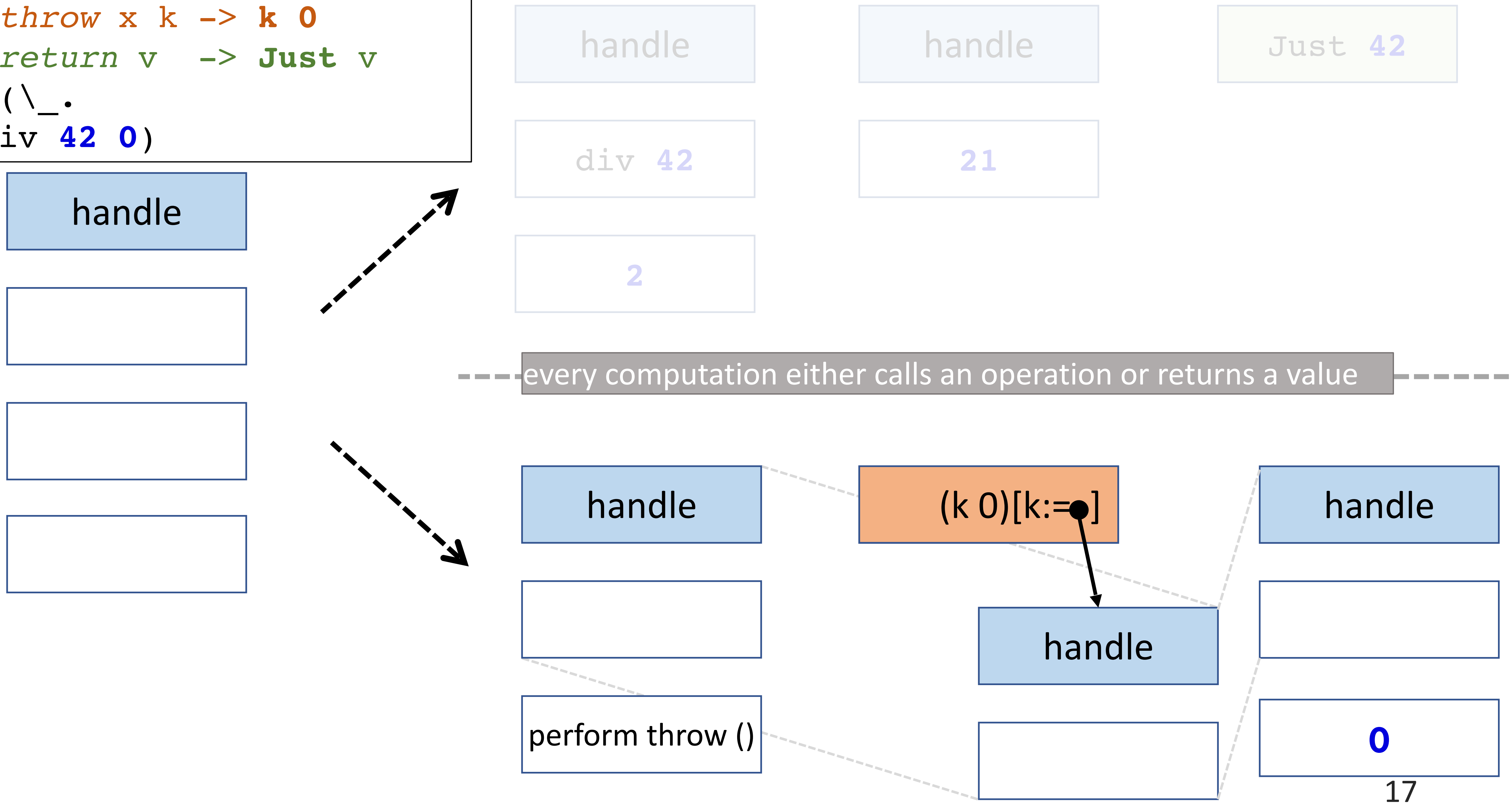
```
handler {
  throw x k -> k 0
  return v  -> Just v
} (\_.
 div 42 0)
```



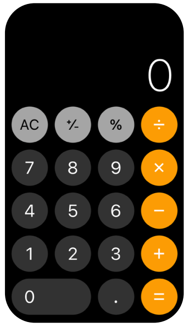
```
handler {
  throw x k -> k 0
  return v  -> Just v
} (\_.
 div 42 0)
```

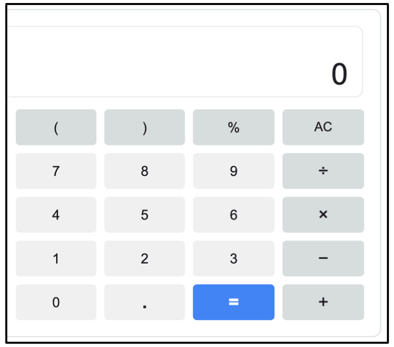


```
handler {
  throw x k -> k 0
  return v  -> Just v
} (\_.
 div 42 0)
```



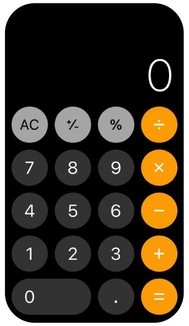
calculator

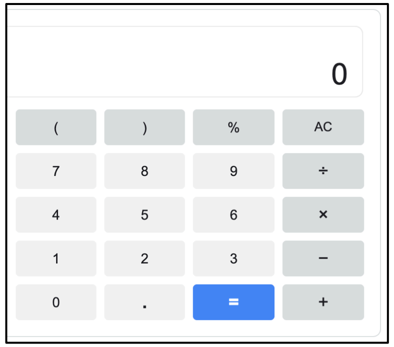




```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

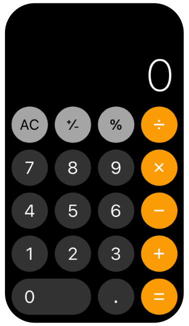
calculator

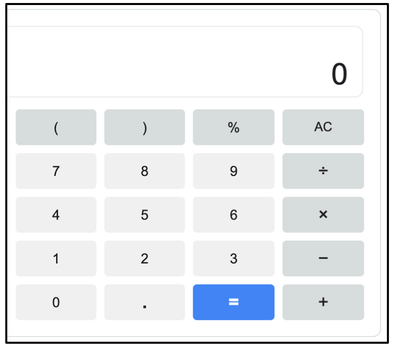




```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator





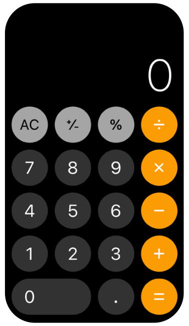
```
ios_div m n =
  handle {

  } (div m n)
```

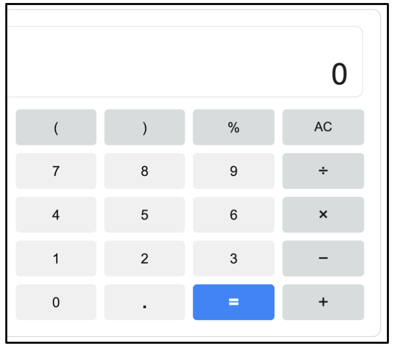
```
google_div m n =
  handle {




  } (div m n)
```

```
coq_div m n =
  handle {

  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator





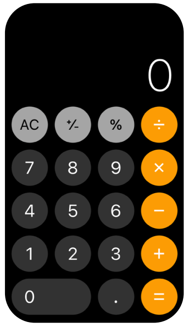
```
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)
```

```
google_div m n =
  handle {



  } (div m n)
```

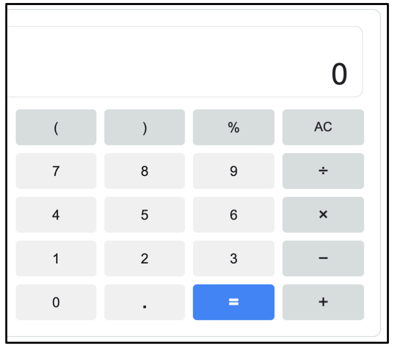
```
coq_div m n =
  handle {

  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator




```
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)
```

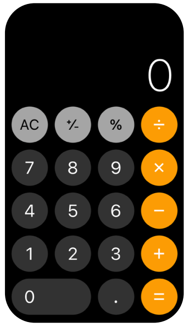


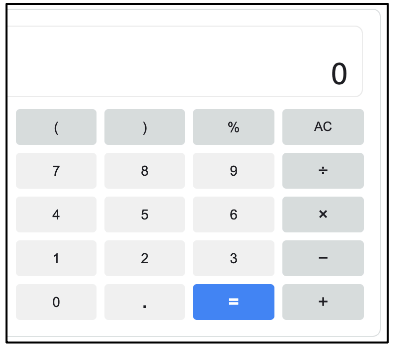

```
google_div m n =
  handle {
    divByZero x k ->
      if x == 0 then Error
      else Infinity
  } (div m n)
```

```
coq_div m n =
  handle {

  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator





```
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)
```

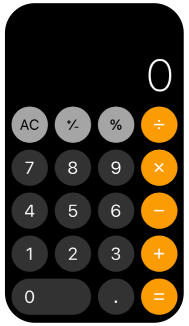
argument

```
google_div m n =
  handle {
    divByZero x k ->
      if x == 0 then Error
      else Infinity
  } (div m n)
```

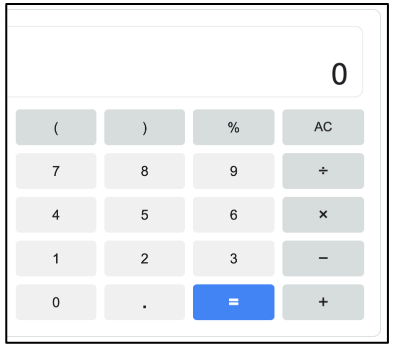
```
coq_div m n =
  handle {

  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator




```
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)
```



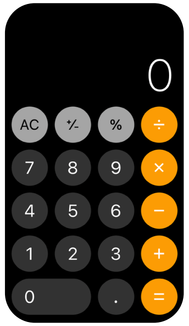

```
google_div m n =
  handle {
    divByZero x k ->
      if x == 0 then Error
      else Infinity
  } (div m n)
```

argument

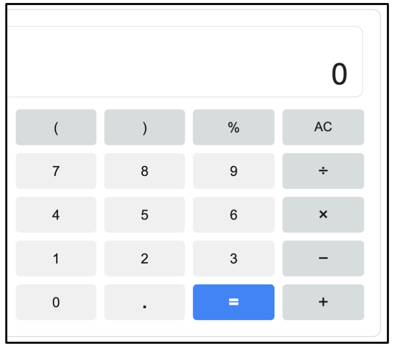
```
coq_div m n =
  handle {
    divByZero x k -> k 0
  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

calculator




```
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)
```



argument

```
google_div m n =
  handle {
    divByZero x k ->
      if x == 0 then Error
      else Infinity
  } (div m n)
```

resume with default value

```
coq_div m n =
  handle {
    divByZero x k -> k 0
  } (div m n)
```

```
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
```

State

State

```
effect st<a> {
  get : () -> a
  set : a -> ()
}
```

State

```
effect st<a> {
  get : () -> a
  set : a -> ()
}
```

```
(handler {
  get x k -> (\y. k y y)
  set x k -> (\y. k () x)
  return x -> (\_. x)
} (\_.
 perform set 21; w <- perform get (); w + w))
0
```

State

```
effect st<a> {
  get : () -> a
  set : a -> ()
}
```

```
(handler {
  get x k -> (\y. k y y)
  set x k -> (\y. k () x)
  return x -> (\_. x)
} (\_.
 perform set 21; w <- perform get (); w + w))
0
```

State

```
effect st<a> {
  get : () -> a
  set : a -> ()
}
```

```
(handler {
  get x k -> (\y. k y y)
  set x k -> (\y. k () x)
  return x -> (\_. x)
} (\_.
 perform set 21; w <- perform get (); w + w))
0
// 42
```

Choice

Choice

```
effect choice {
  flip : () -> bool
}
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
x <- perform flip ()
y <- perform flip ()
x && y
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

```
x   True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

```
x   True
y   True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

```
// [True
x   True
y   True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True
x   True  True
y   True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

```
// [True
x    True   True
y    True  False
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)
```

```
// [True, False
x    True   True
y    True  False
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False
```

```
x    True   True    False
y    True  False
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False

x   True  True   False  False

y   True  False
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False
```

```
x    True   True    False   False
y    True  False    True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False, False
```

```
x    True   True   False   False
y    True  False   True
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False, False
x    True   True    False   False
y    True  False    True    False
```

Choice

```
effect choice {
  flip : () -> bool
}
```

```
handler {
  flip    x k -> k True ++ k False
  return x    -> [x]
} (\_.
 x <- perform flip ()
 y <- perform flip ()
 x && y
)

// [True, False, False, False]
```

x	True	True	False	False
y	True	False	True	False

Choice and Exception

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {


} (\_.
handler {


} (\_.
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
// [Just True
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
// [Just True, Just False
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
// [Just True, Just False, Nothing]
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
```

Choice and Exception

```
effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```

```
handler {
  throw  x k -> Nothing
  return x   -> Just x
} (\_.
handler {
  flip   x k -> k True ++ k False
  return x   -> [x]
} (\_.
 x <- perform flip ()
 if x then
    perform flip ()
 else
    perform throw ()
))
// Nothing
```

Select

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

```
x <- perform select [1..15]
y <- perform select [1..15]
z <- perform select [1..15]
if x * x + y * y == z * z
then (x,y,z)
else failed
```

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

```
handler {
  select xs k -> concatMap k xs
  return x    -> [x]
} (\_.
 x <- perform select [1..15]
 y <- perform select [1..15]
 z <- perform select [1..15]
 if x * x + y * y == z * z
 then (x,y,z)
 else failed
)
```

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

```
handler {
  select xs k -> concatMap k xs
  return x    -> [x]
} (\_.
 x <- perform select [1..15]
 y <- perform select [1..15]
 z <- perform select [1..15]
 if x * x + y * y == z * z
 then (x,y,z)
 else failed
)
//  [(3,4,5),(4,3,5),(5,12,13),(6,8,10)
//  ,(8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

```
handler {
  select xs k ->
    let f ys = case ys of
      []      -> Nothing
      y':ys' -> case k y' of Nothing -> f ys'
      r                          Just v  -> Just v
    in f xs
  return x    -> Just x
)
 x <- perform select [1..15]
 y <- perform select [1..15]
 z <- perform select [1..15]
 if x * x + y * y == z * z
 then (x,y,z)
 else failed
)
```

Select

```
effect select<a> {
  select : [a] -> a
}

failed = perform select []
```

```
handler {
  select xs k ->
    let f ys = case ys of
      []      -> Nothing
      y':ys' -> case k y' of Nothing -> f ys'
      r                       Just v  -> Just v
    in f xs
  return x    -> Just x
)
 x <- perform select [1..15]
 y <- perform select [1..15]
 z <- perform select [1..15]
 if x * x + y * y == z * z
 then (x,y,z)
 else failed
)
// Just (3,4,5)
```

N-Queens

```
effect select<a> {
  select : [a] -> bool
}

failed = perform select []
```

N-Queens

```
effect select<a> {
  select : [a] -> bool
}

failed = perform select []
```

```
nQueens n = fold f [] [1..n] where
    f rows col = row <- perform select [1..n]
                 if (safeAddition rows row 1)
                 then (row : rows)
                 else failed

// is it safe to add the new queen?
safeAddition rows r i =
  case rows of
    []        -> True
    (r:rows) ->
      row /= r &&
      abs (row - r) /= i &&
      safeAddition rows row (i + 1)
```

Cooperative multi-threading

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print "A"; perform fork (\_. print "B"; perform yield (); print "E");
  print "C"; perform fork (\_. print "D"; perform yield (); print "G"); print "F"
)
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print “A”; perform fork (\_. print “B”; perform yield (); print “E”);
  print “C”; perform fork (\_. print “D”; perform yield (); print “G”); print “F”
) // A
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print "A"; perform fork (\_. print "B"; perform yield (); print "E");
  print "C"; perform fork (\_. print "D"; perform yield (); print "G"); print "F"
) // A B
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print “A”; perform fork (\_. print “B”; perform yield (); print “E”);
  print “C”; perform fork (\_. print “D”; perform yield (); print “G”); print “F”
) // A B C
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print “A”; perform fork (\_. print “B”; perform yield (); print “E”);
  print “C”; perform fork (\_. print “D”; perform yield (); print “G”); print “F”
) // A B C D
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print “A”; perform fork (\_. print “B”; perform yield (); print “E”);
  print “C”; perform fork (\_. print “D”; perform yield (); print “G”); print “F”
) // A B C D E
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
 }
 f
```

```
scheduler (\_.
  print “A”; perform fork (\_. print “B”; perform yield (); print “E”);
  print “C”; perform fork (\_. print “D”; perform yield (); print “G”); print “F”
) // A B C D E F
```

Cooperative multi-threading

```
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork  : (() -> ()) -> ()
}
```

```
scheduler f =
  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f
```

```
scheduler (\_.
  print "A"; perform fork (\_. print "B"; perform yield (); print "E");
  print "C"; perform fork (\_. print "D"; perform yield (); print "G"); print "F"
) // A B C D E F G
```

Algebraic effects Summary

Composable and modular computational effects

Algebraic effects Summary

Composable and modular computational effects

Key ideas:

Algebraic effects Summary

Composable and modular computational effects

Key ideas:

1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value

Algebraic effects Summary

Composable and modular computational effects

Key ideas:

1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value

Examples:

Algebraic effects Summary

Composable and modular computational effects

Key ideas:

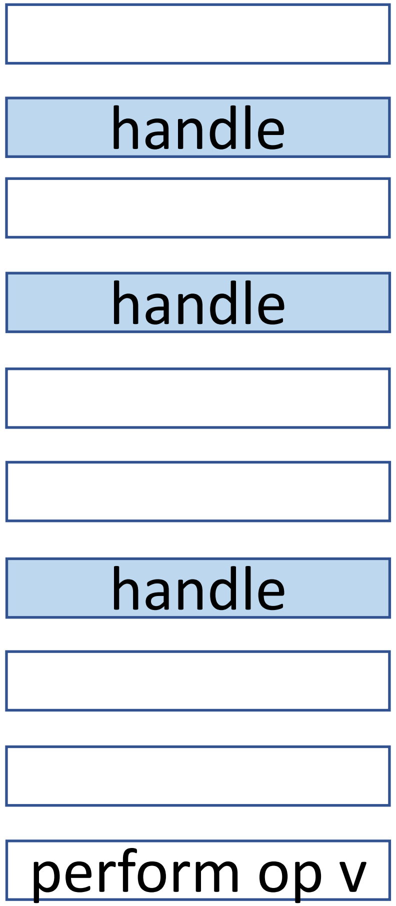
1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value

Examples:

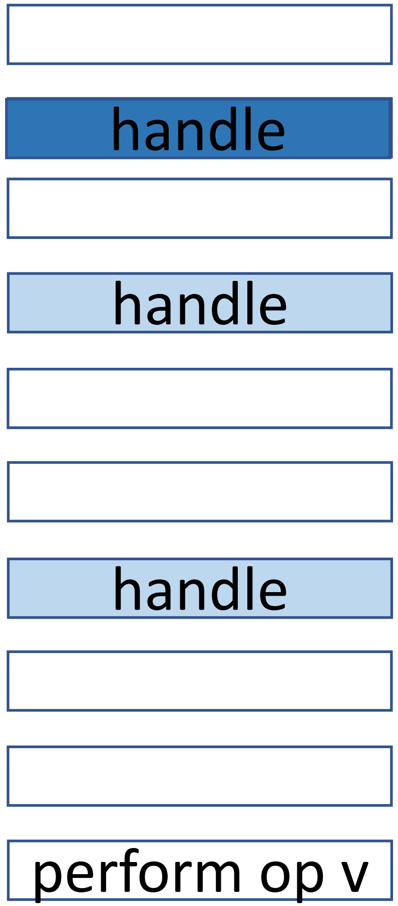
```
read, exn, state, choice, select, coop, …
```

Challenges

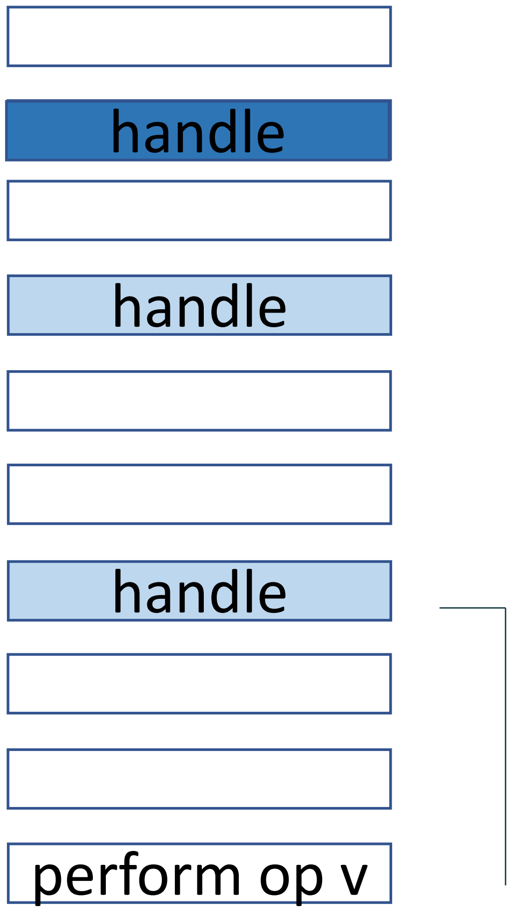
Challenges



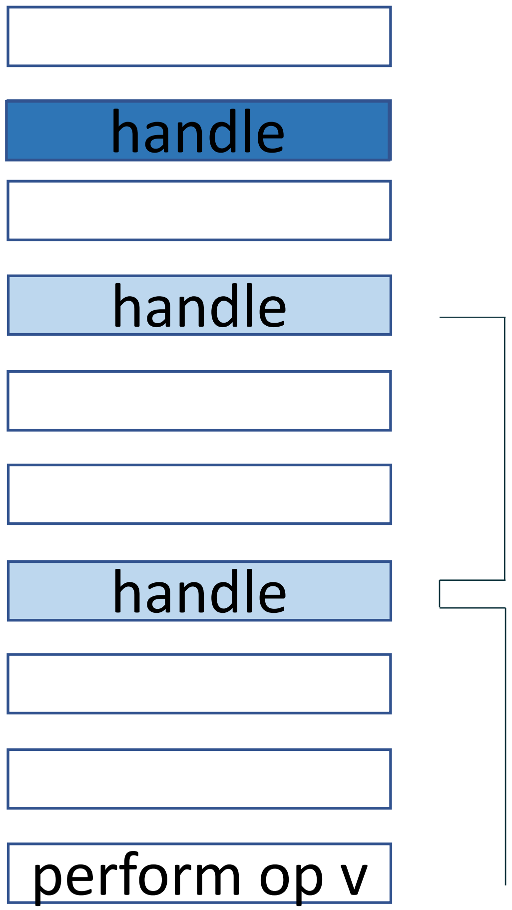
Challenges



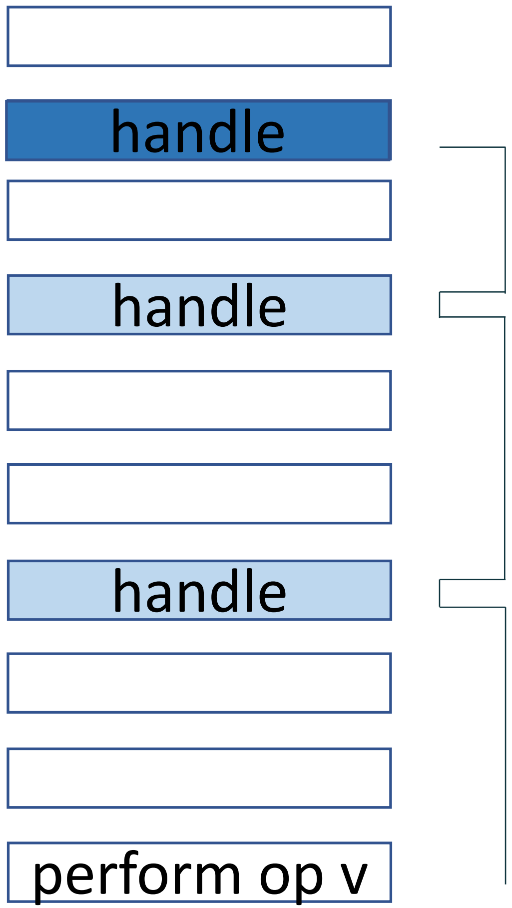
Challenges



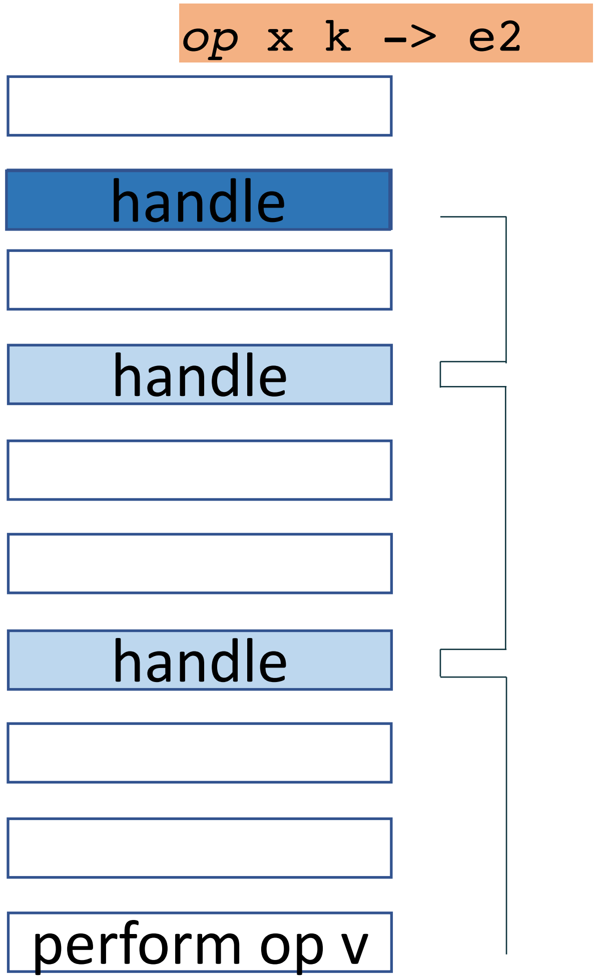
Challenges



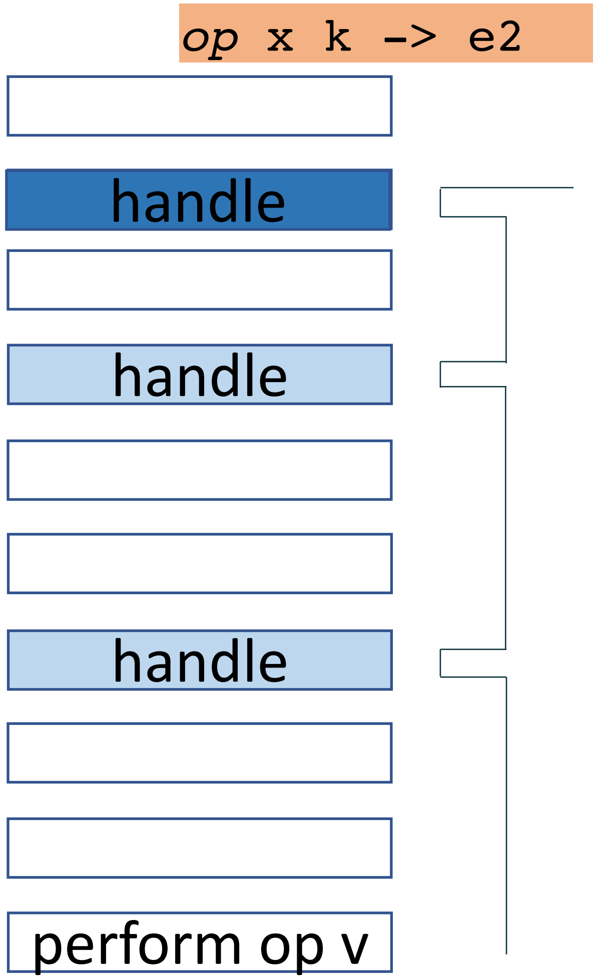
Challenges



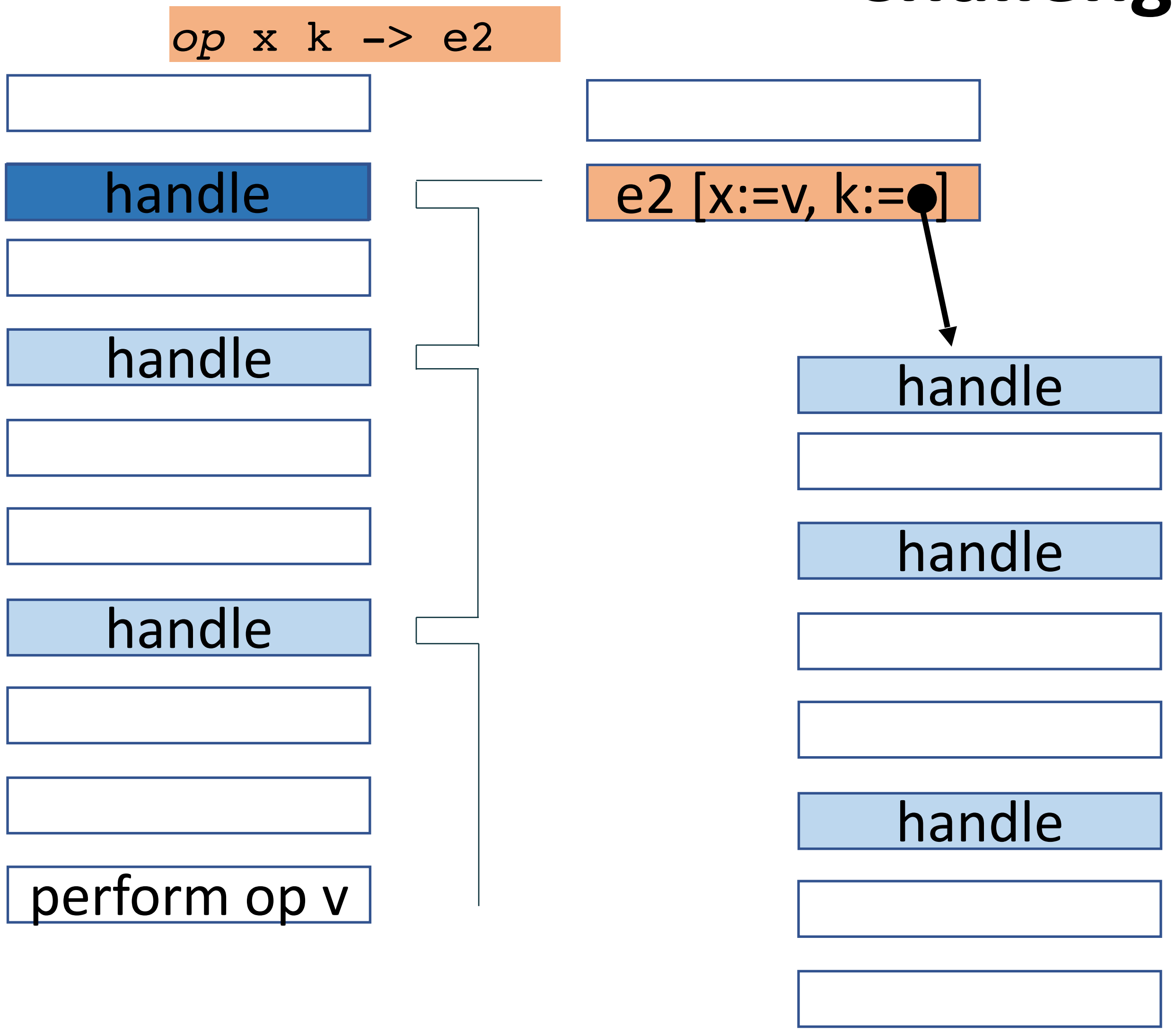
Challenges



Challenges

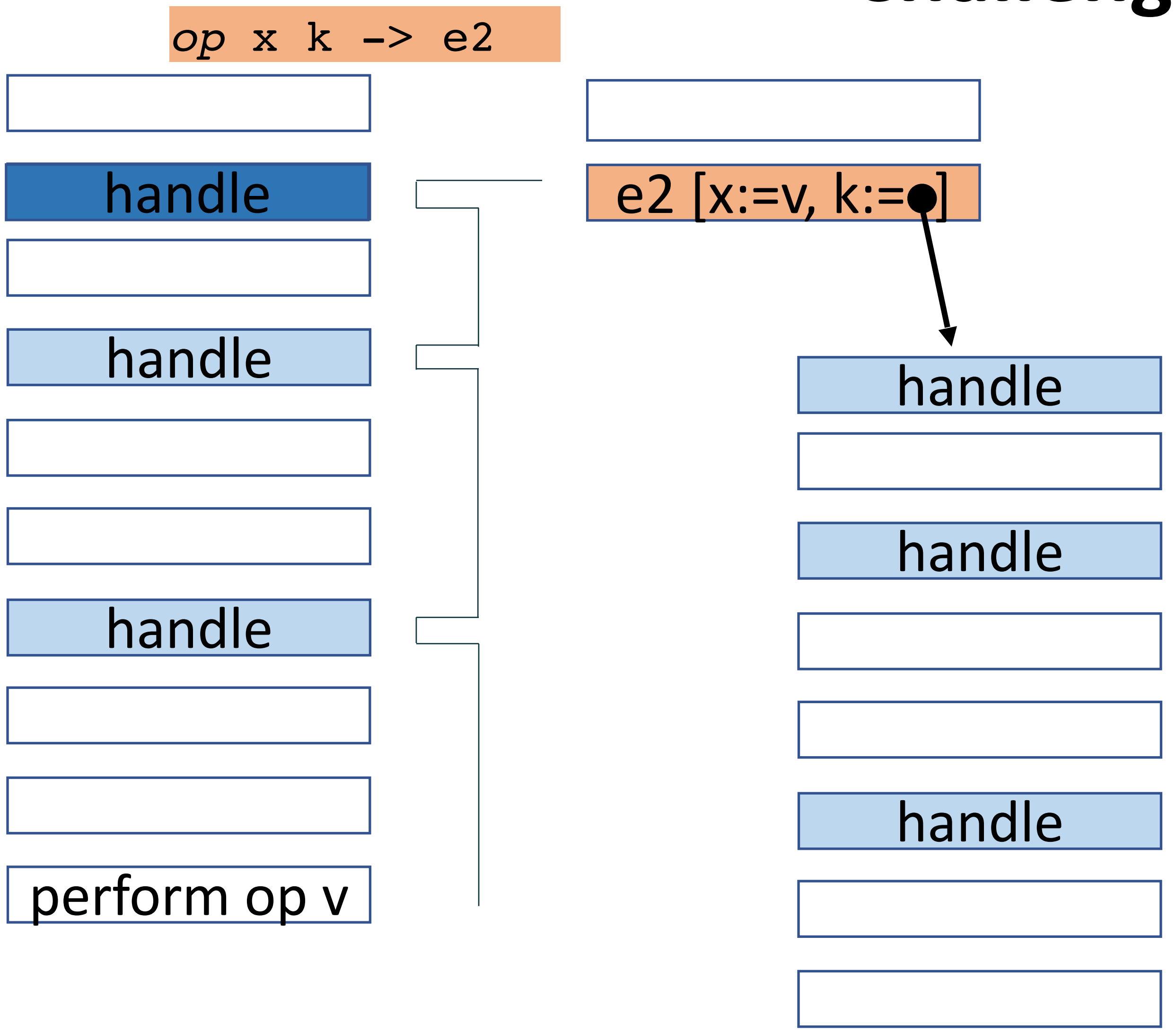


Challenges



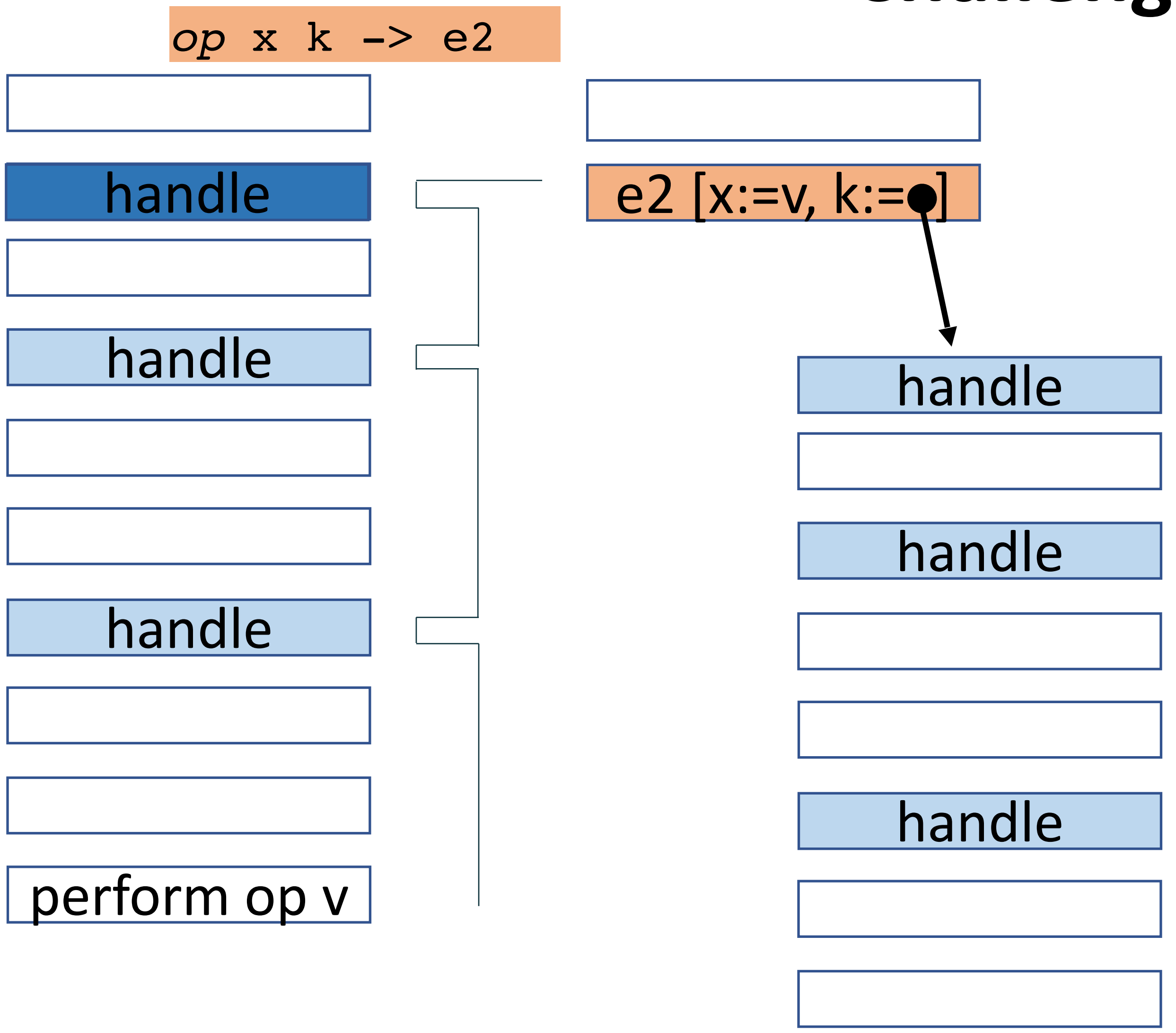
op x k -> e2
handle
handle
handle
perform op v
e2 [x:=v, k:=●]
handle
handle
handle

Challenges



1. **Searching**
   a *linear* search through the current evaluation context

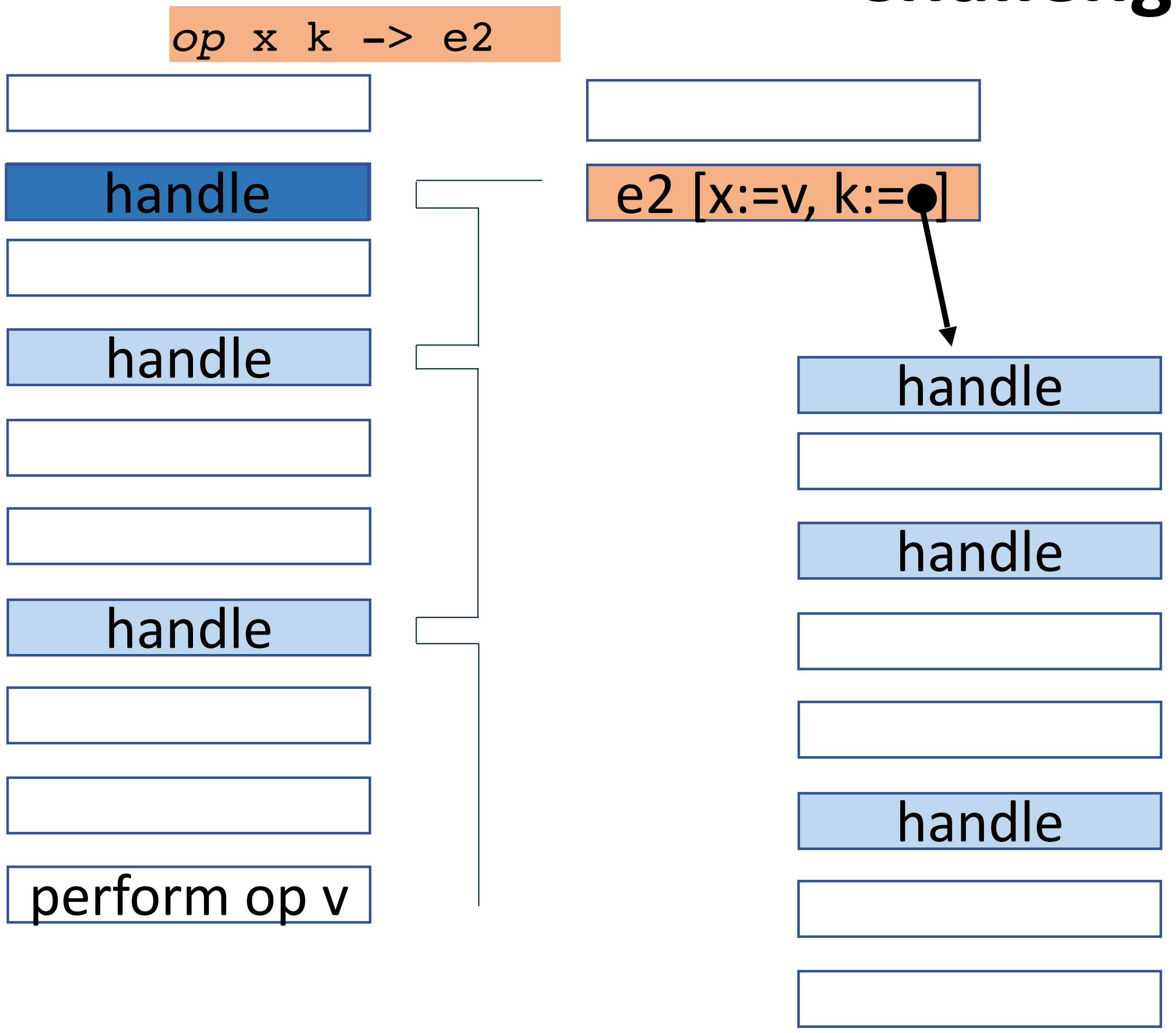
Challenges



1. **Searching**
   a *linear* search through the current evaluation context

2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

Challenges



1. **Searching**
   a *linear* search through the current evaluation context

2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

Can we implement algebraic effects efficiently?

Continuation-passing style

Links Hillerström et al 2017, 2020

Leijen 2017
Schuster et al 2020
……

Segmented Stacks

OCaml

Dolan et al 2014, 2015
Sivaramakrishnan et al 2021
……

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020
……

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

Continuation-passing style

Closure allocation cost

Segmented Stacks

OCaml

Dolan et al 2014, 2015
Sivaramakrishnan et al 2021
……

Capability-passing style

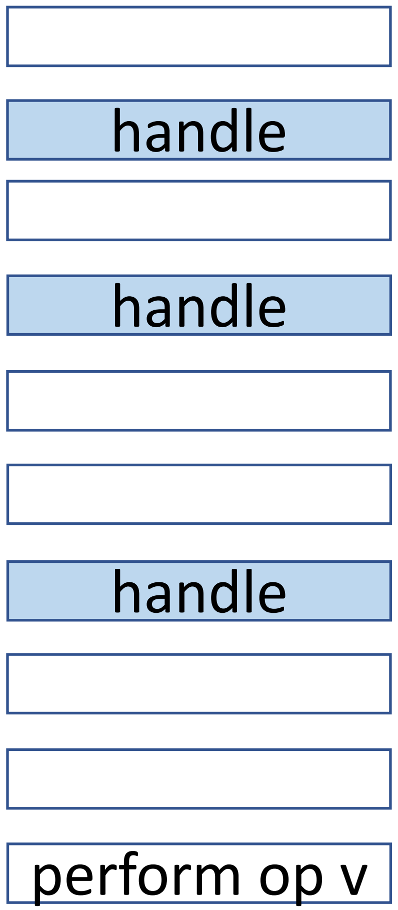
Effekt

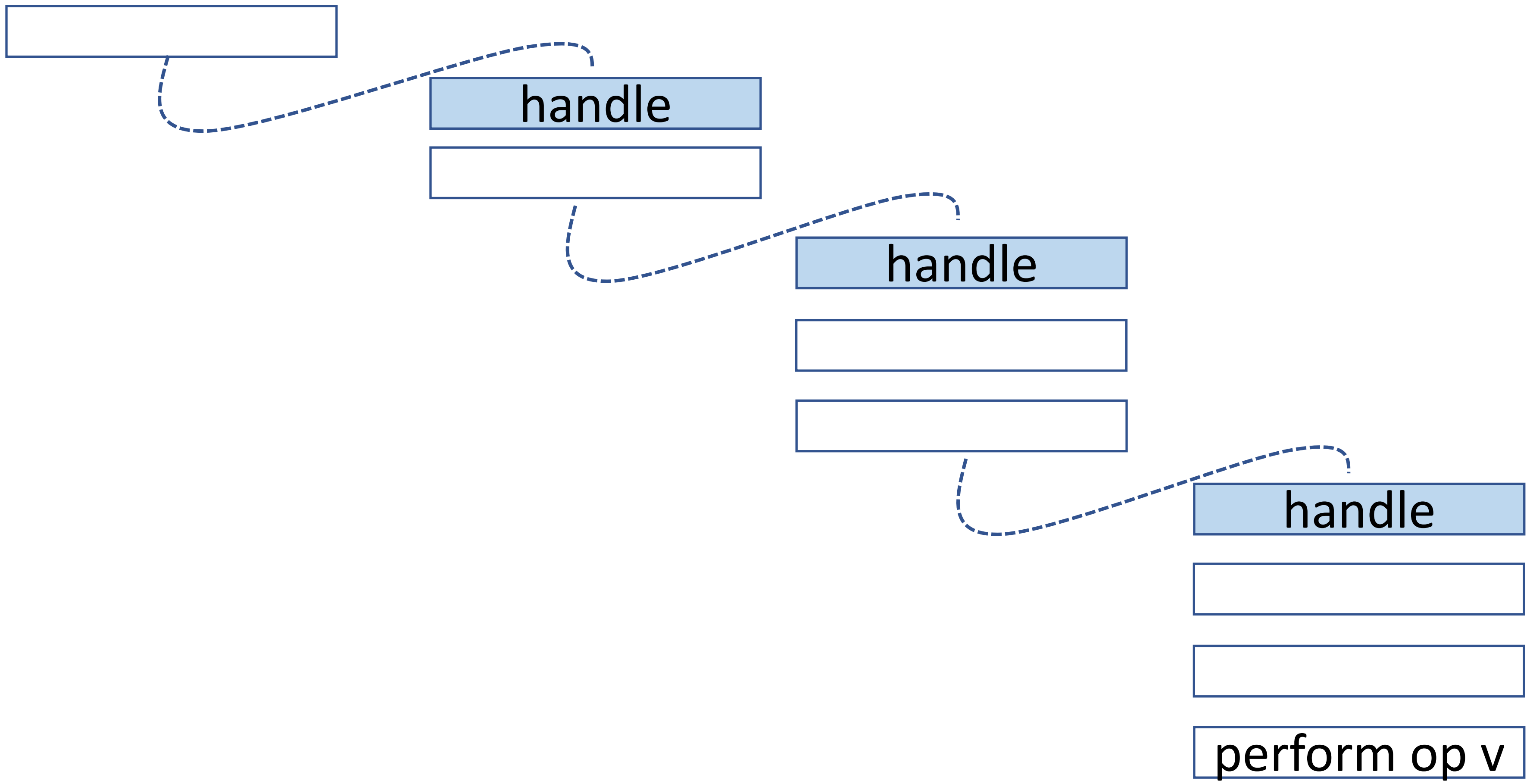
Schuster et al 2020
Brachthäuser et al 2020
……

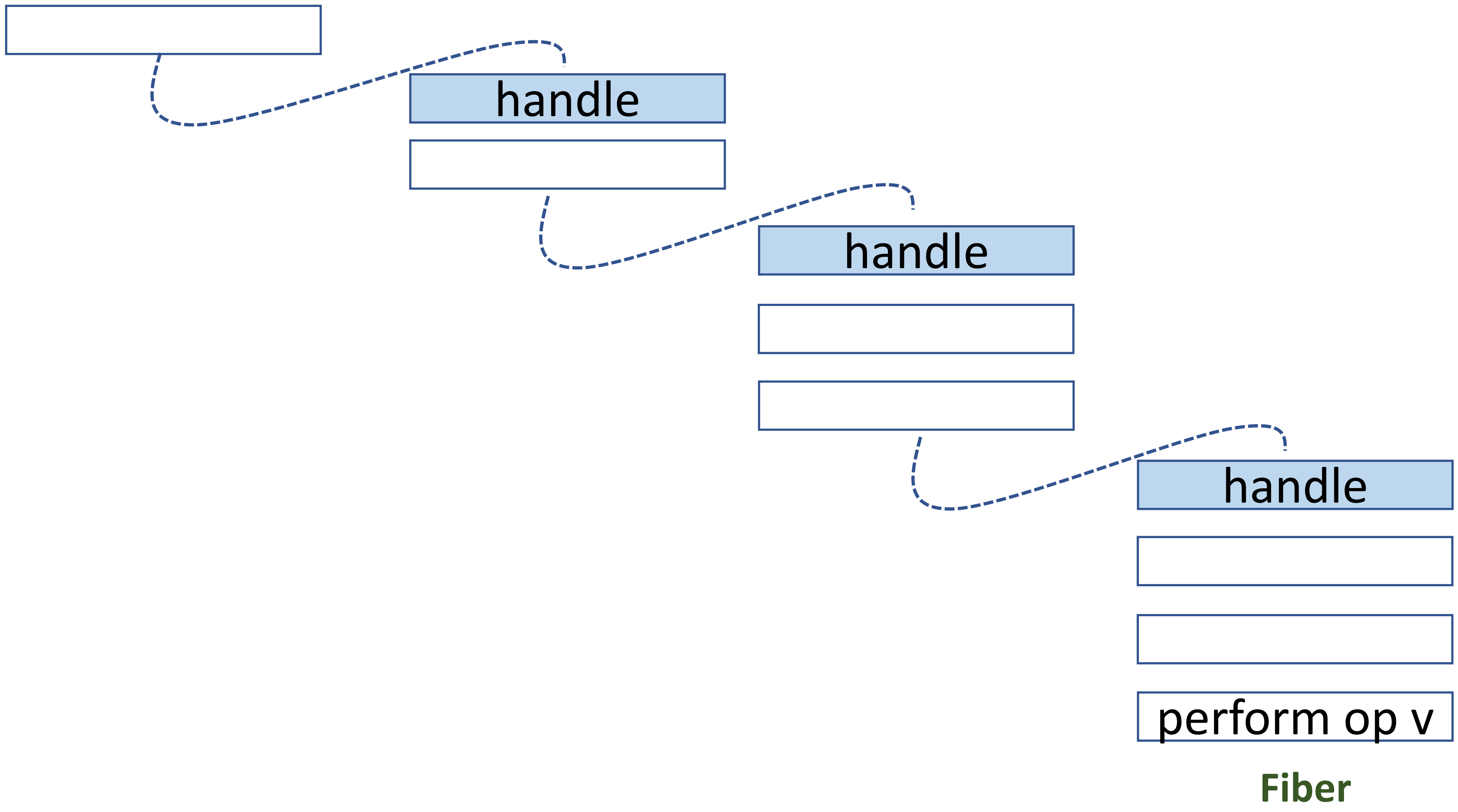
Rewriting

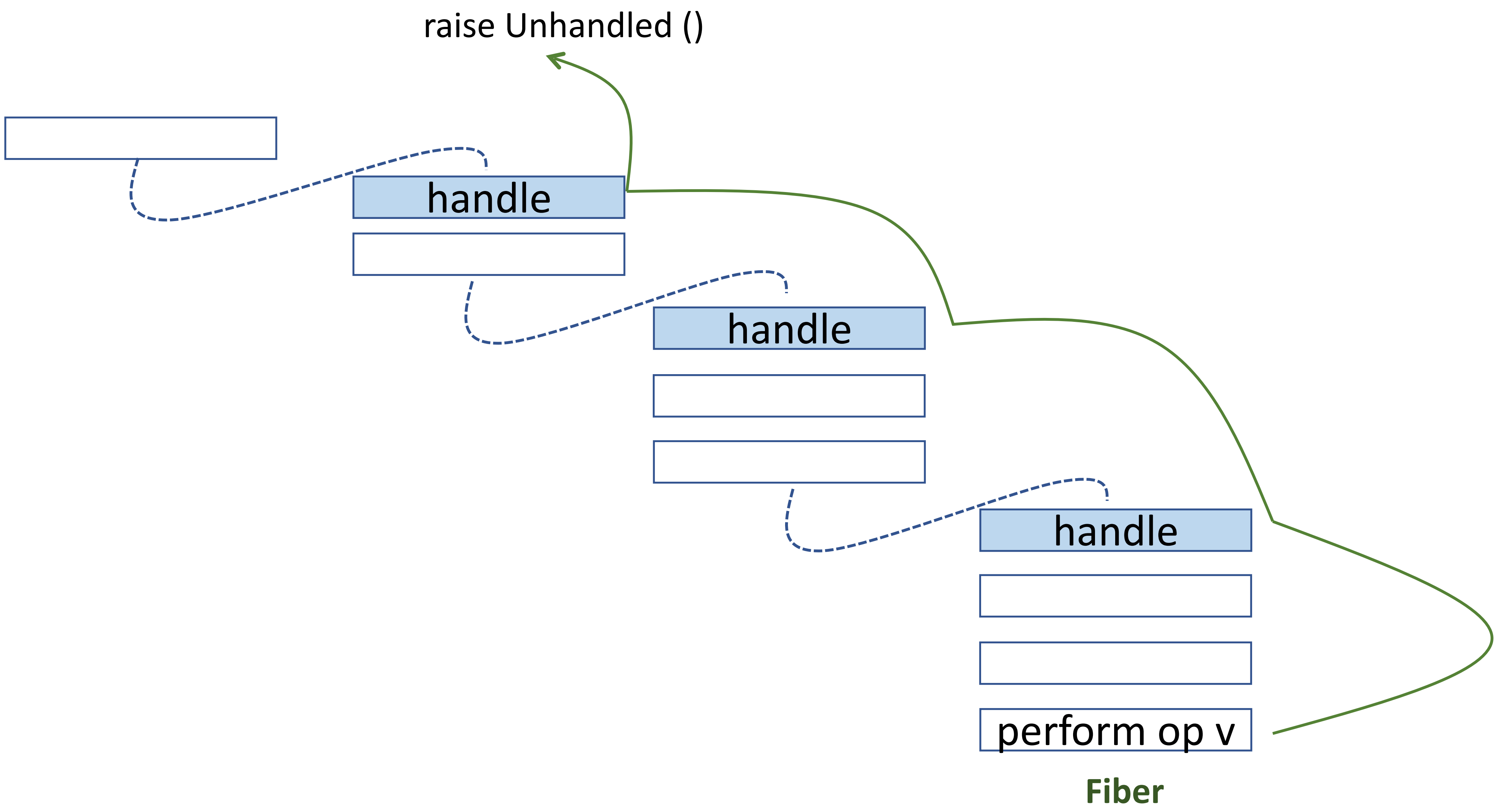
Eff

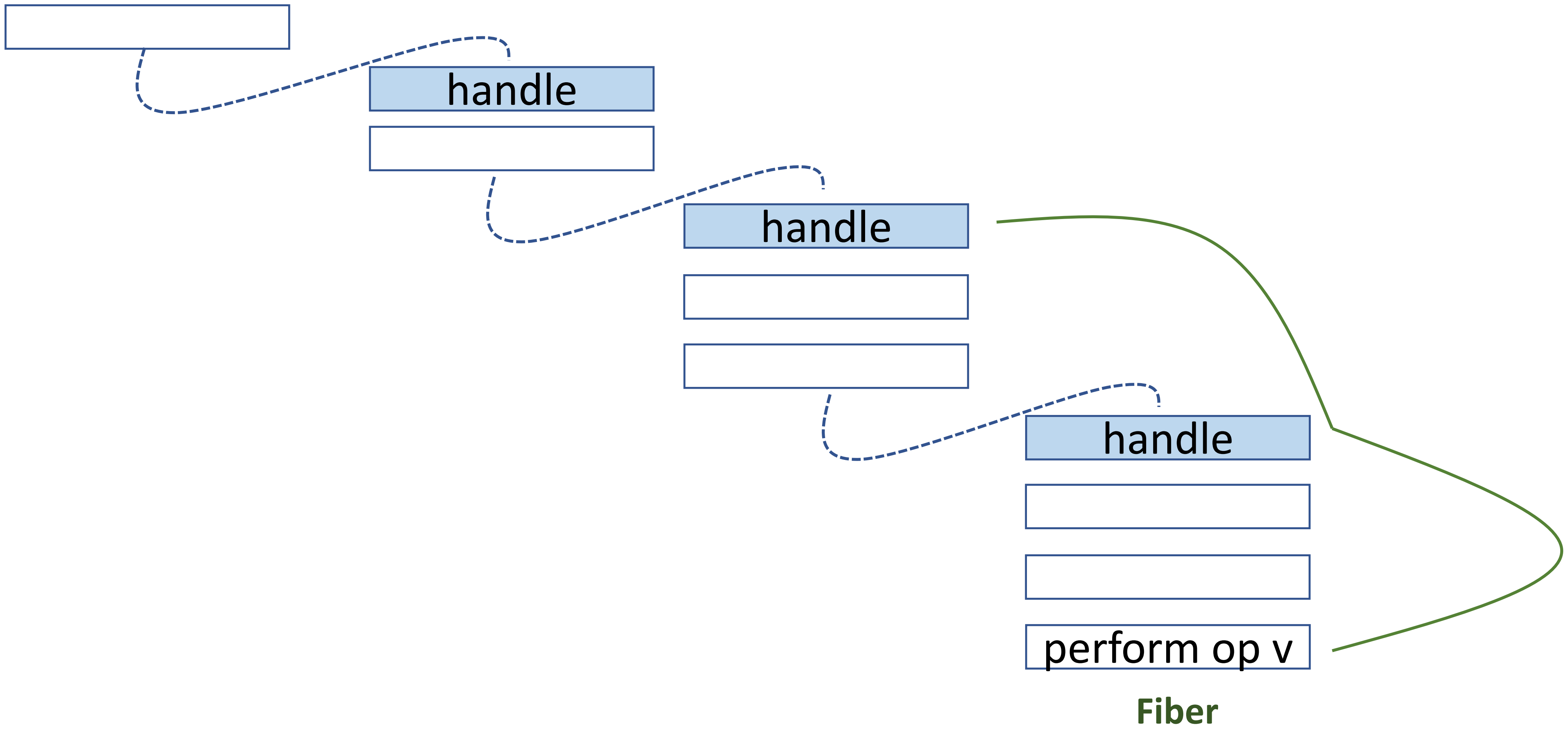
Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

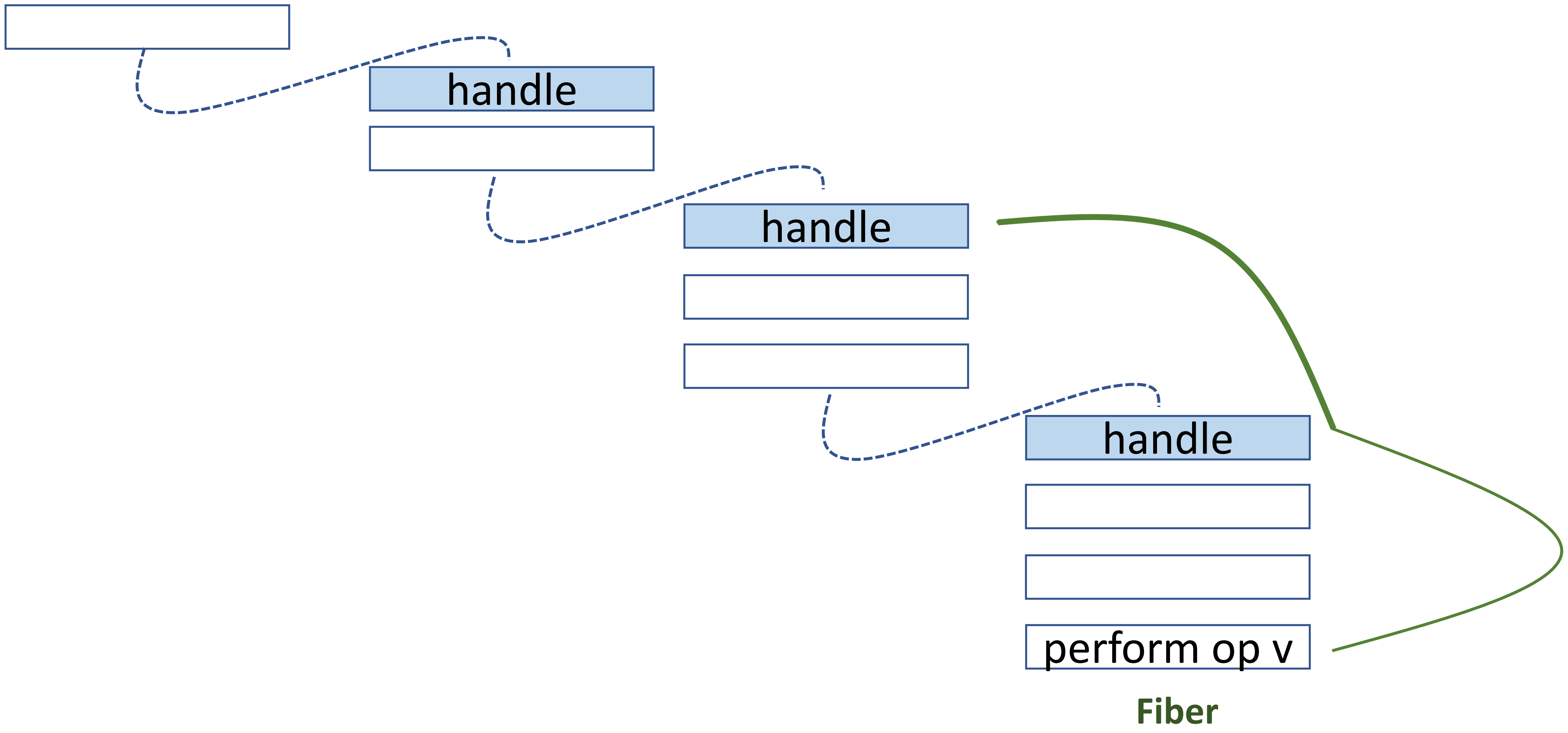

handle
handle
handle
perform op v

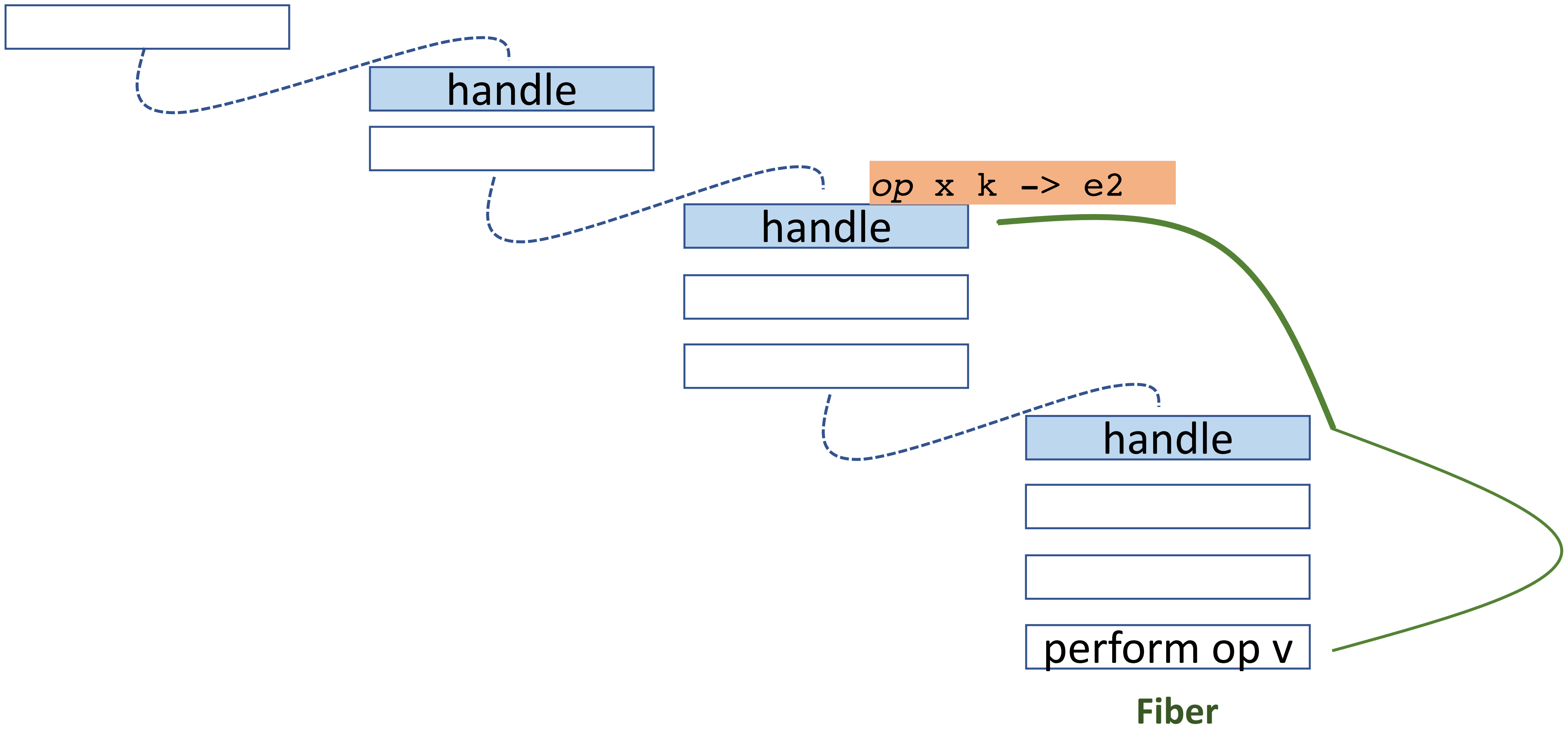

handle
handle
handle
perform op v

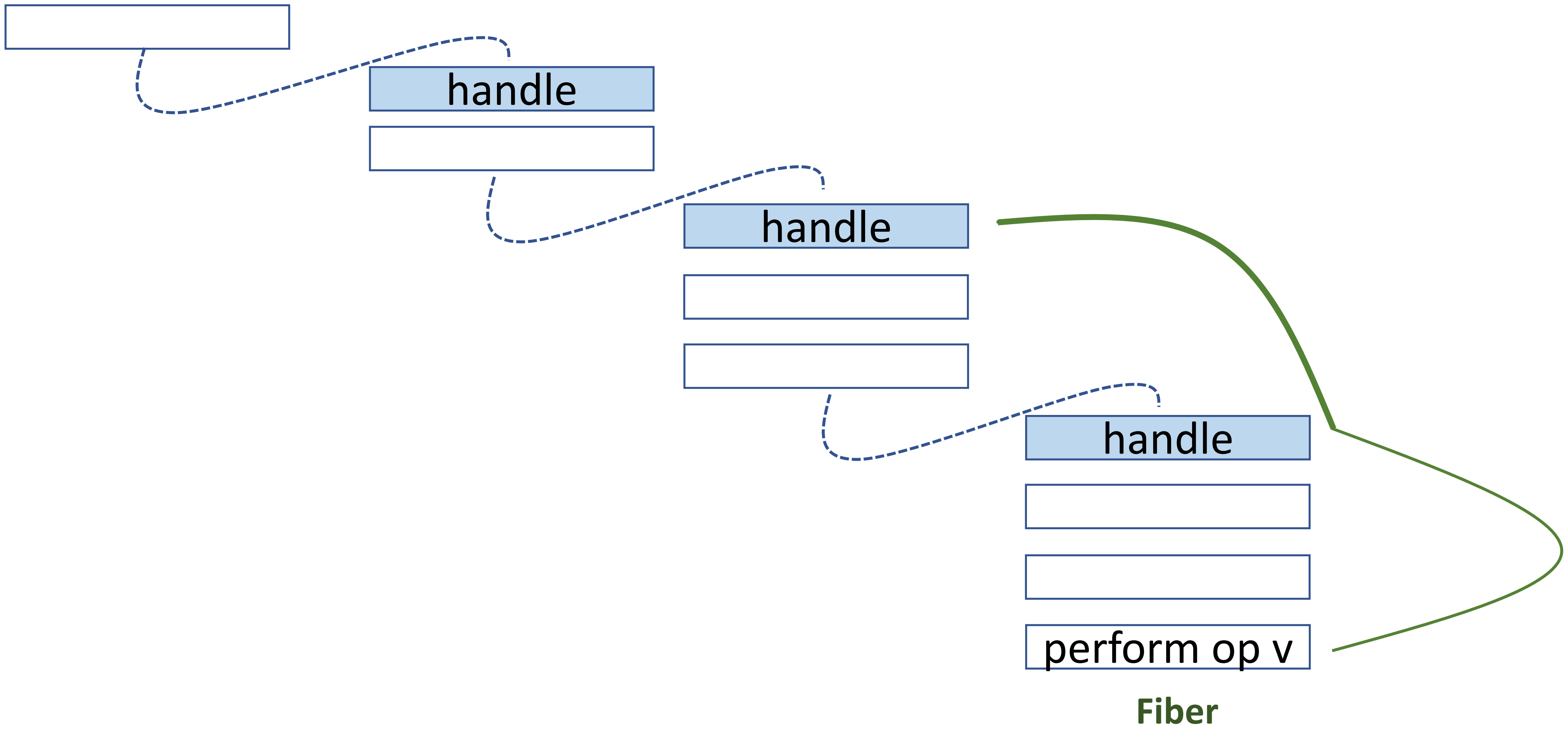

handle
handle
handle
perform op v
Fiber

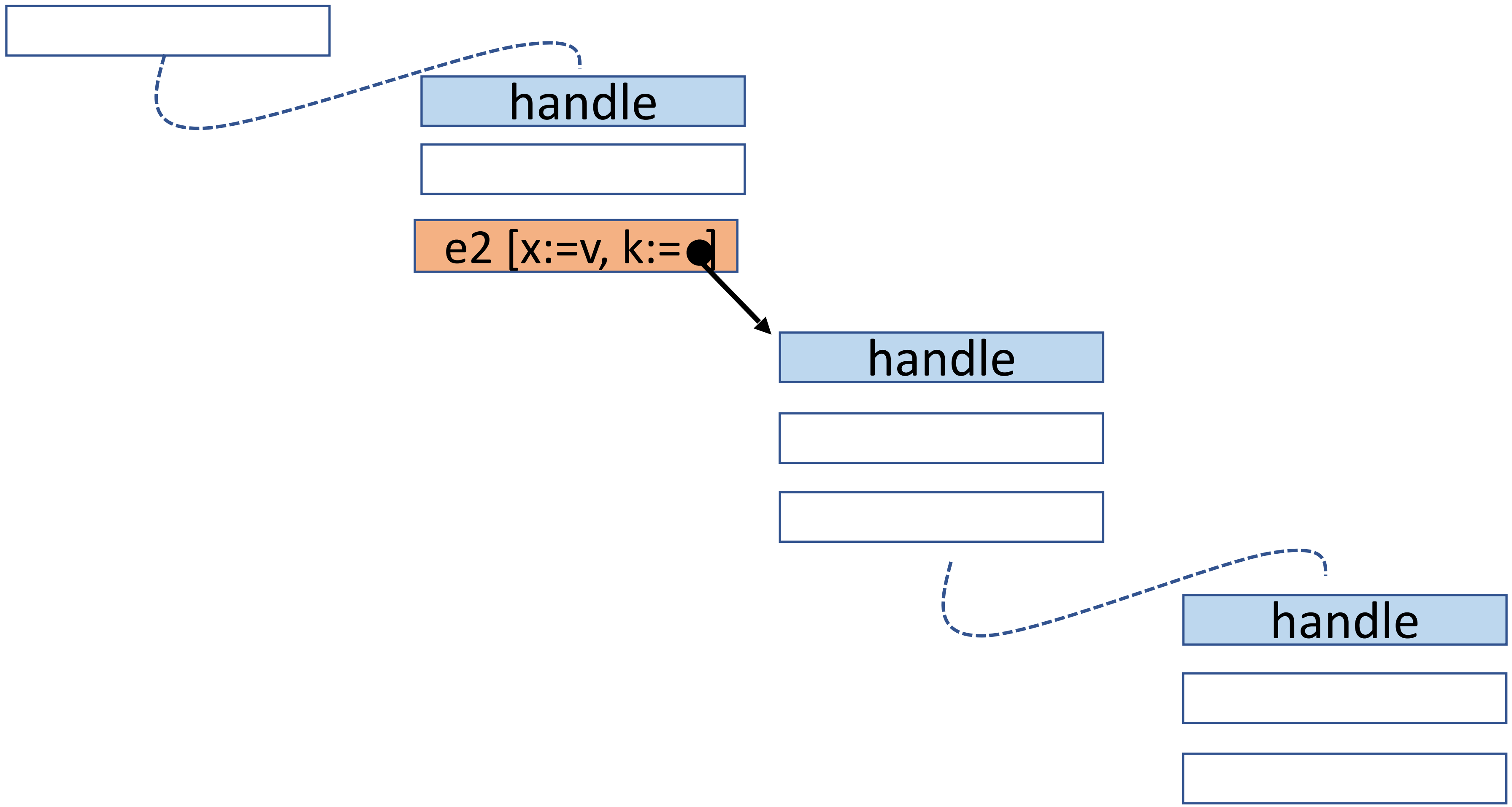

raise Unhandled ()
handle
handle
handle
perform op v
Fiber

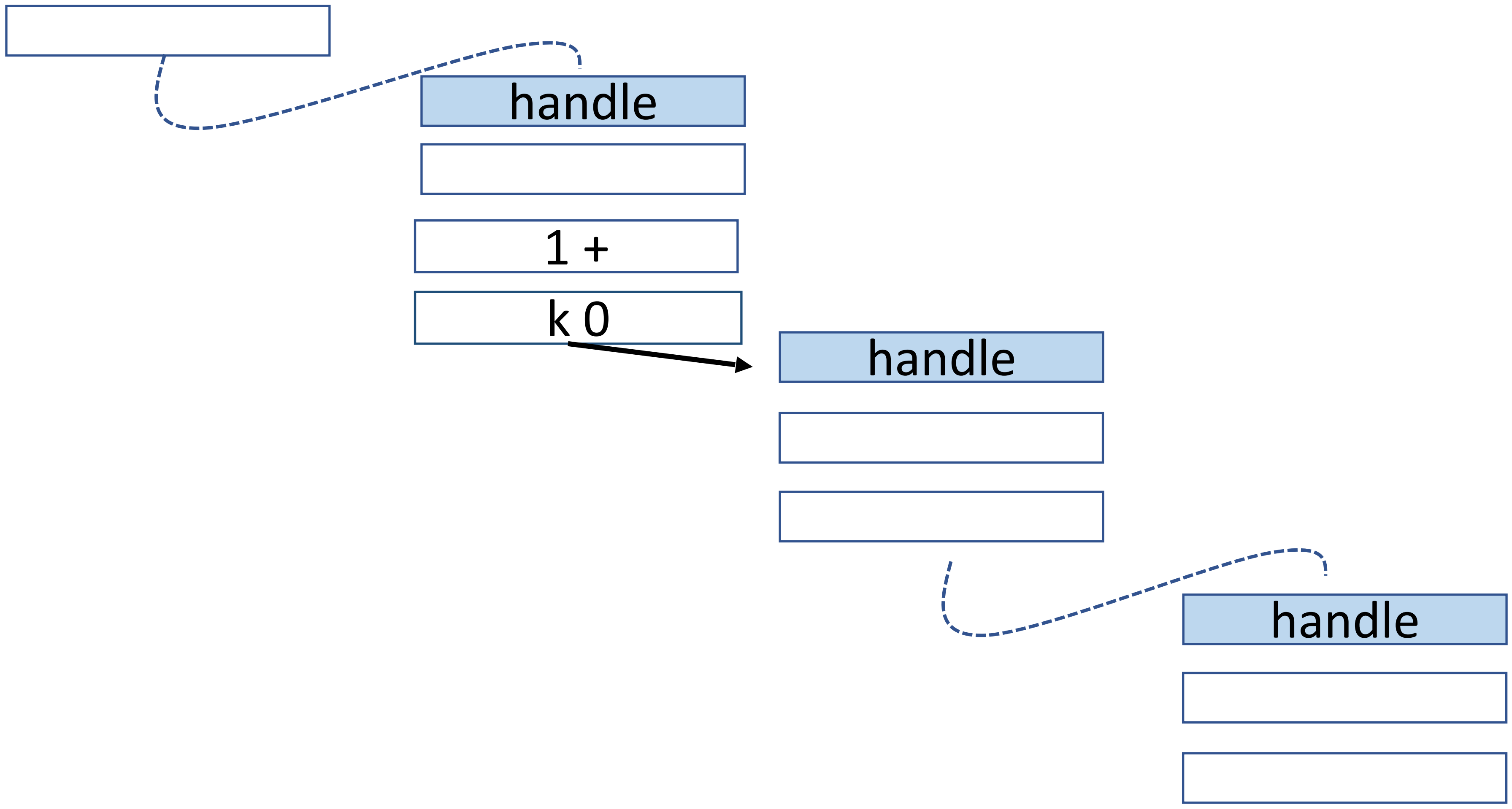

handle
handle
handle
perform op v
Fiber

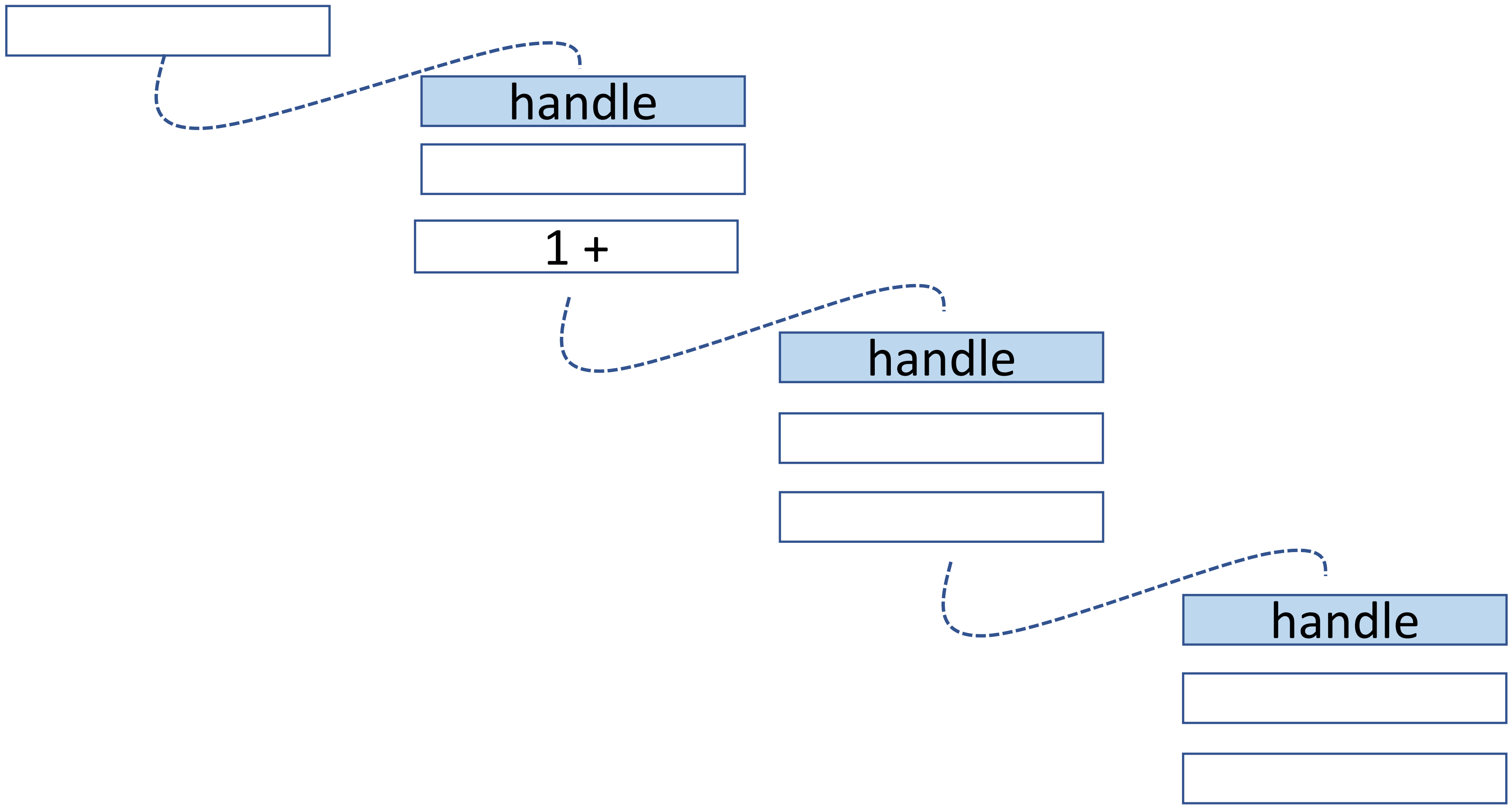

handle
handle
handle
perform op v
Fiber

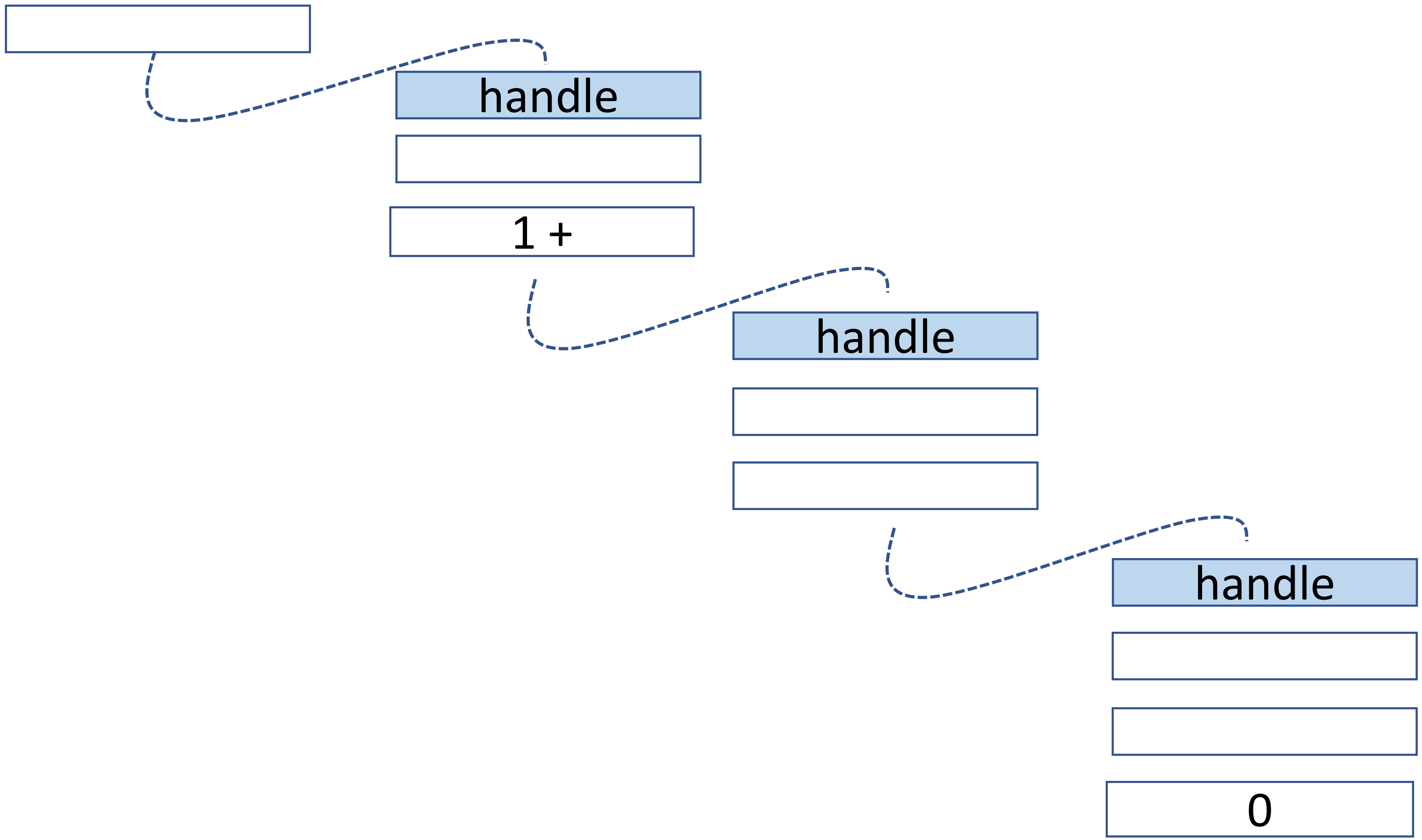

handle
op x k -> e2
handle
handle
perform op v
Fiber


handle
handle
handle
perform op v
Fiber


handle
e2 [x:=v, k:=●]
handle
handle


handle
1 +
k 0
handle
handle


handle
1 +
handle
handle


handle
1 +
handle
handle
0

Continuation-passing style

Closure allocation cost

Segmented Stacks

OCaml

Dolan et al 2014, 2015
Sivaramakrishnan et al 2021
……

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020
……

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020
……

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
div m n throw
  = if n == 0
    then perform throw ()
    else m / n
```

```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

```
handler {
  throw x k -> Nothing
} (\_.
 div 42 0
) // Nothing
```

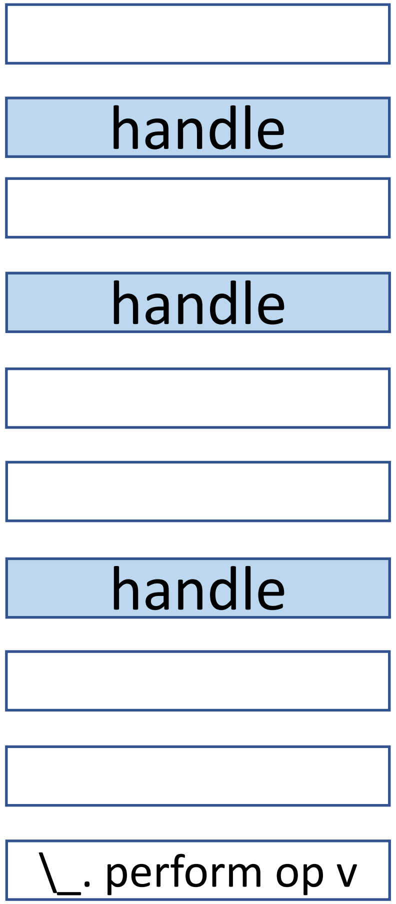
```
div m n throw
  = if n == 0
    then perform throw ()
    else m / n
```

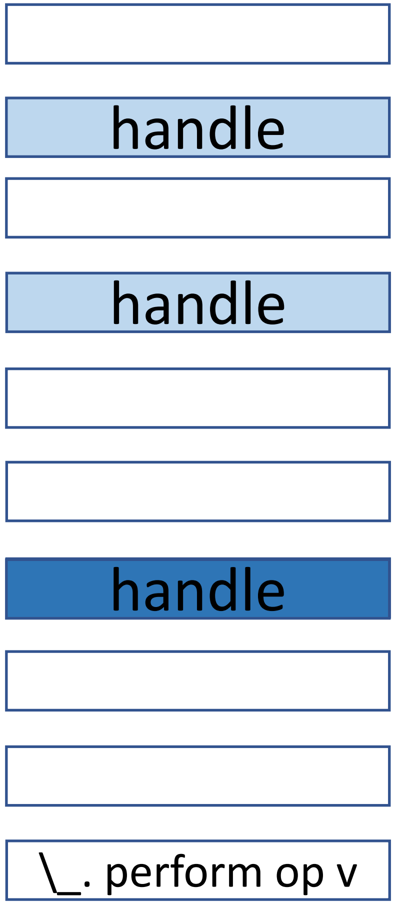
```
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n
```

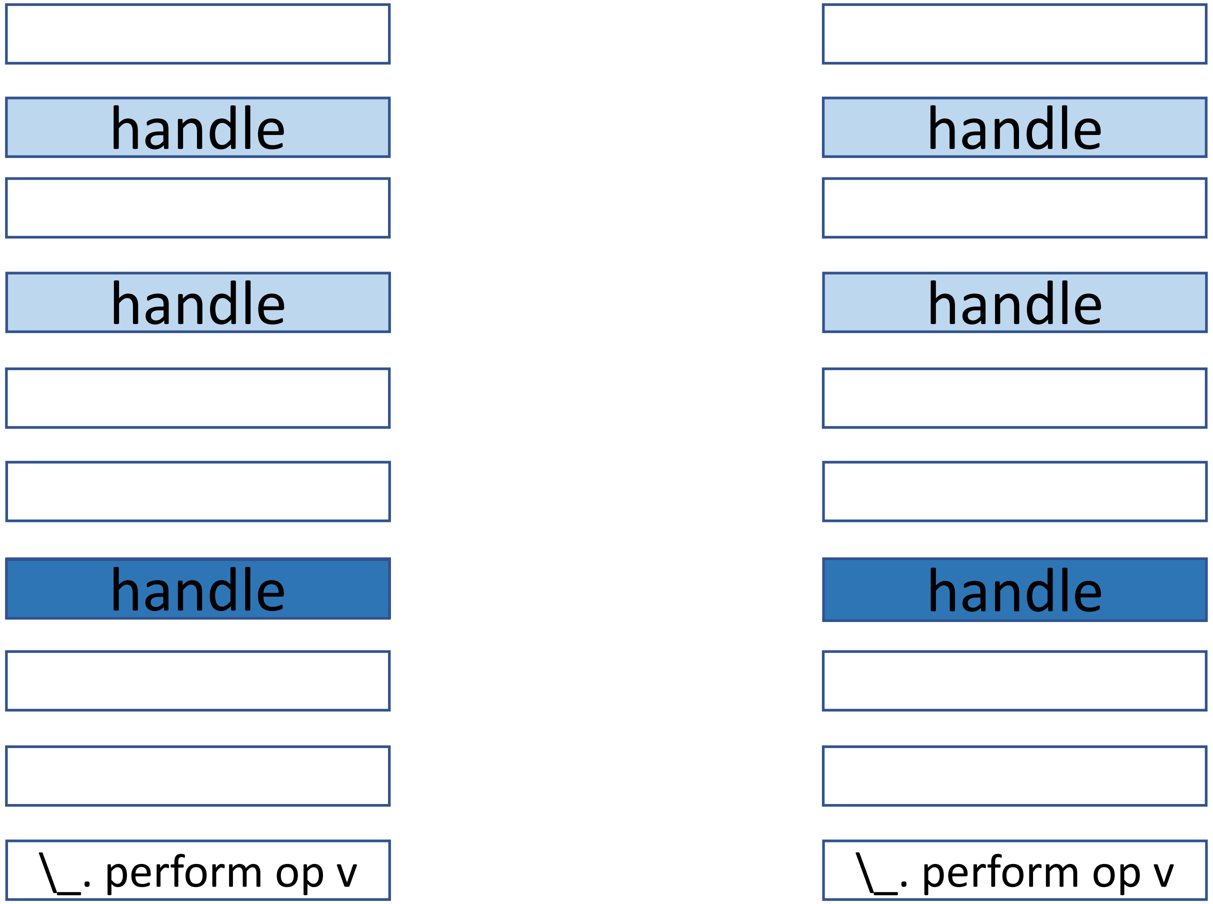
```
div m n throw
  = if n == 0
    then perform throw ()
    else m / n
```

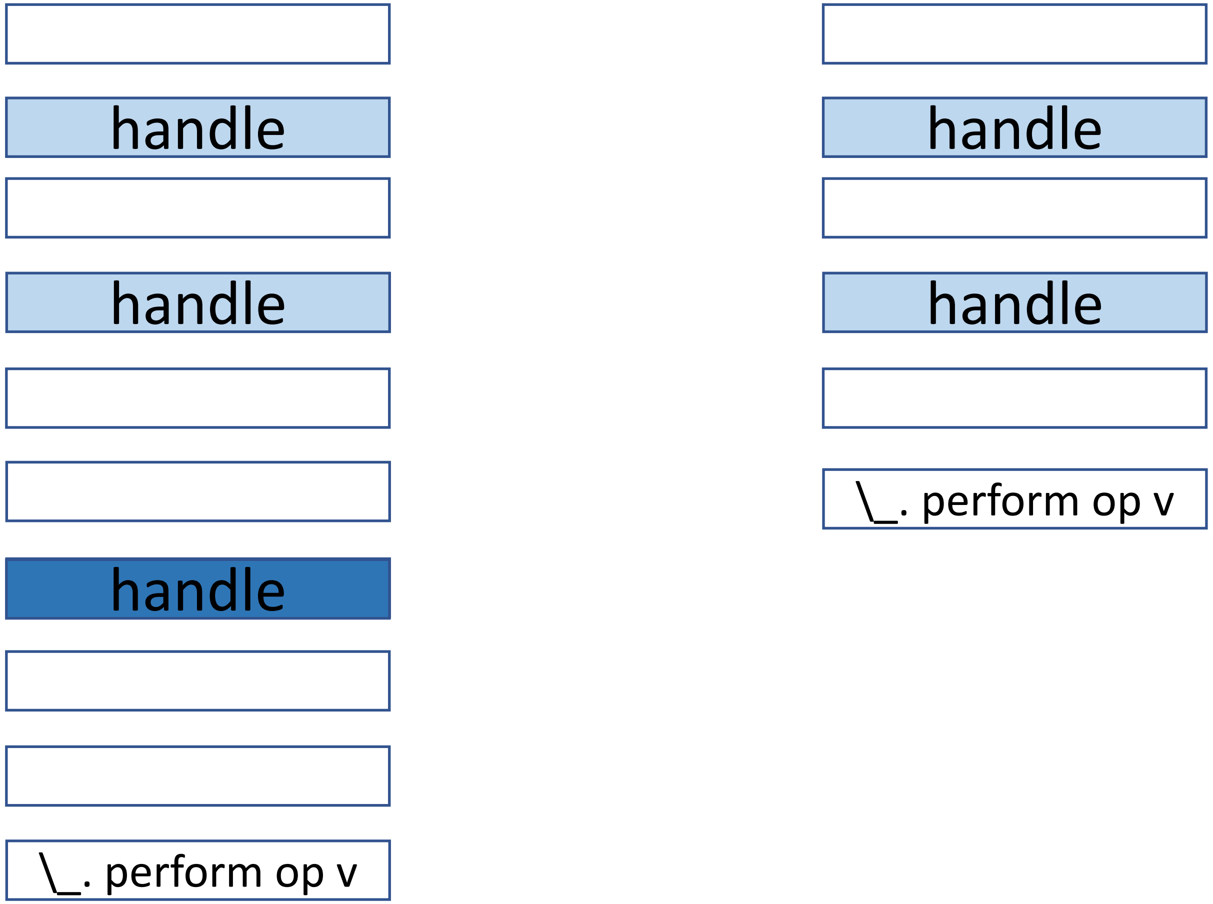
```
handler {
  throw x k -> Nothing
} (\_.
 div 42 0
) // Nothing
```

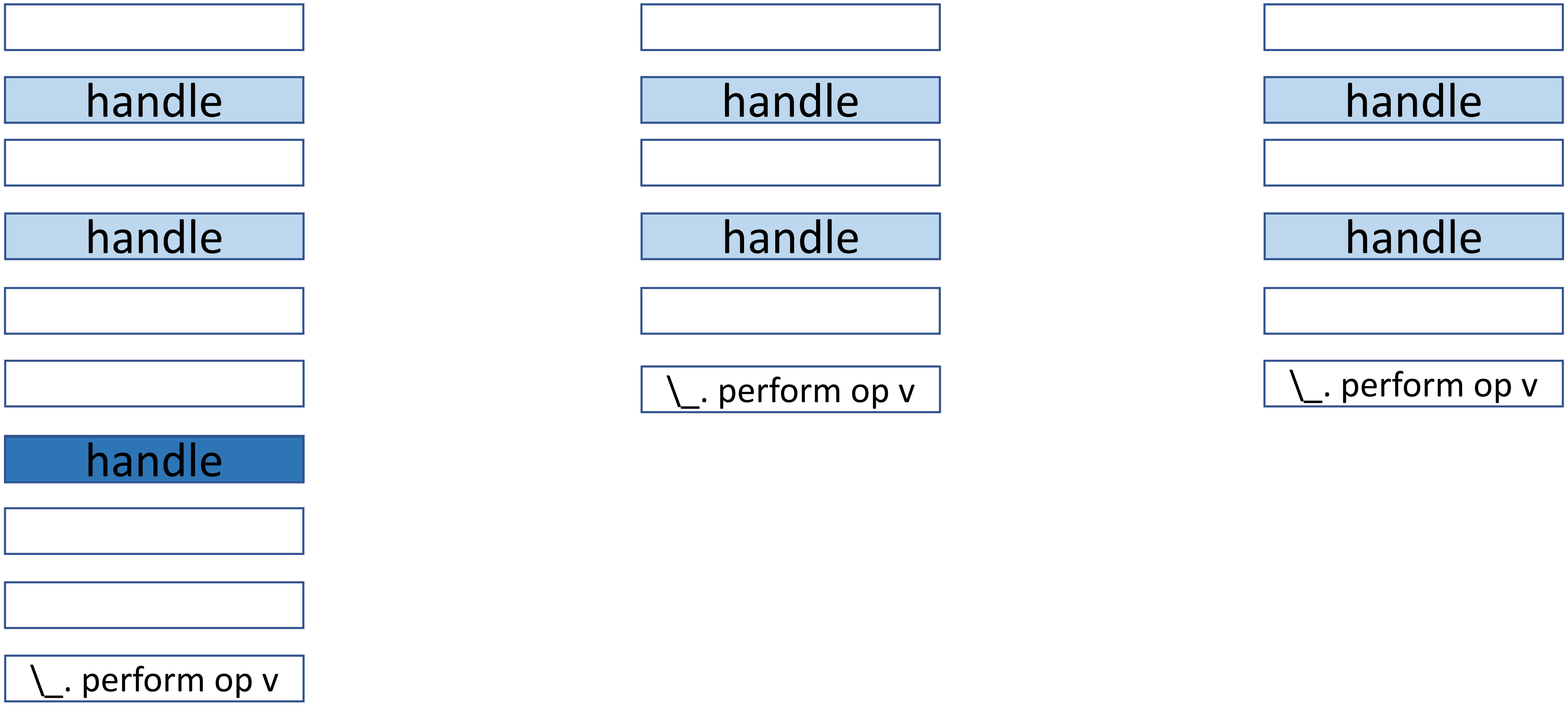
```
handle {
  throw x k -> Nothing
}
(div 42 0 throw)
```

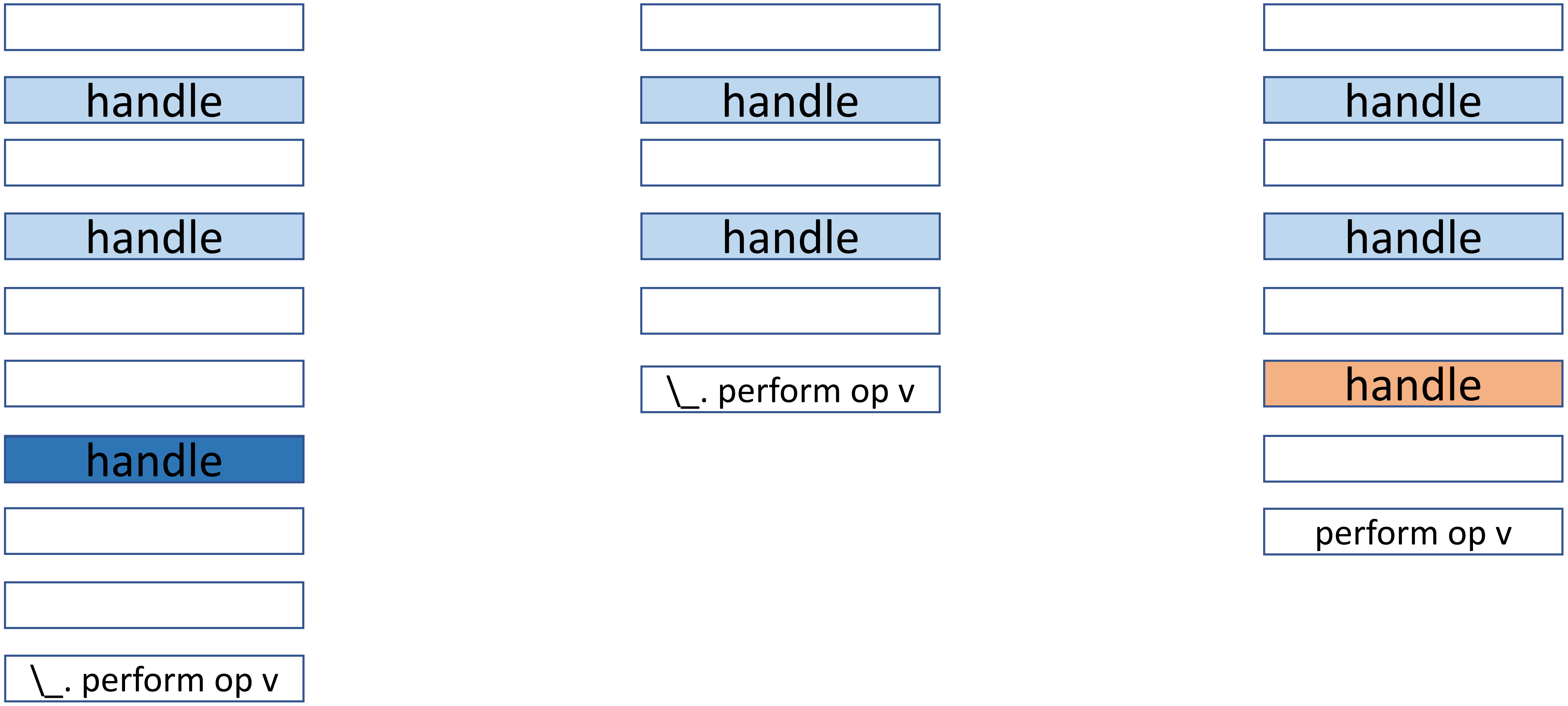

handle
handle
handle
_. perform op v


handle
handle
handle
_. perform op v


handle
handle
handle
_. perform op v
handle
handle
handle
_. perform op v


handle
handle
handle
_. perform op v
handle
handle
_. perform op v


handle
handle
handle
_. perform op v
handle
handle
_. perform op v
handle
handle
_. perform op v


handle
handle
handle
_. perform op v
handle
handle
_. perform op v
handle
handle
handle
perform op v

Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020
……

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Efficient lexically scoped handlers

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
……

Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Efficient lexically scoped handlers

Rewriting

Source-to-source transformations

Algebraic effects and evidence-passing semantics in Koka

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

Algebraic effects and evidence-passing semantics in Koka

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

Koka: Programming with Row Polymorphic Effect Types
Leijen, **MSFP 2014**

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, **POPL 2017**

Implementing Algebraic Effects in C
Leijen, **APLAS 2017**

Algebraic effects and evidence-passing semantics in Koka

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

Koka: Programming with Row Polymorphic Effect Types
Leijen, **MSFP 2014**

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, **POPL 2017**

Implementing Algebraic Effects in C
Leijen, **APLAS 2017**

Algebraic effects and evidence-passing semantics in Koka

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

Koka: Programming with Row Polymorphic Effect Types
Leijen, **MSFP 2014**

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, **POPL 2017**

Implementing Algebraic Effects in C
Leijen, **APLAS 2017**

Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, **ICFP 2020**

Effect Handlers in Haskell, Evidently
Xie and Leijen, **Haskell 2020**

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, **ICFP 2021**

Algebraic effects and evidence-passing semantics in Koka

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

Koka: Programming with Row Polymorphic Effect Types
Leijen, **MSFP 2014**

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, **POPL 2017**

Implementing Algebraic Effects in C
Leijen, **APLAS 2017**

Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, **ICFP 2020**

Effect Handlers in Haskell, Evidently
Xie and Leijen, **Haskell 2020**

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, **ICFP 2021**

Perceus: Garbage Free Reference Counting with Reuse
Reinking*, Xie*, de Moura and Leijen, **PLDI 2021**

Algebraic effects and evidence-passing semantics in Koka

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

Koka: Programming with Row Polymorphic Effect Types
Leijen, **MSFP 2014**

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, **POPL 2017**

Implementing Algebraic Effects in C
Leijen, **APLAS 2017**

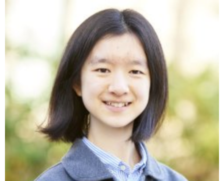
Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, **ICFP 2020**

Effect Handlers in Haskell, Evidently
Xie and Leijen, **Haskell 2020**

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, **ICFP 2021**

Perceus: Garbage Free Reference Counting with Reuse
Reinking*, Xie*, de Moura and Leijen, **PLDI 2021**

First-class Handler Names
Xie, Cong and Leijen, **HOPE 2021**



Evidence-passing semantics

Evidence-passing semantics

Algebraic effects

Efficient C (with no special runtime support)

Evidence-passing semantics

Algebraic effects

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

Efficient C (with no special runtime support)

Evidence-passing semantics

Algebraic effects

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

Evidence-passing semantics

Efficient C (with no special runtime support)

Evidence-passing semantics

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)

Evidence-passing semantics

- 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)

Evidence-passing semantics

- 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)

Evidence-passing semantics

- 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)

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

Evidence-passing semantics

- 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)

PLDI 2021

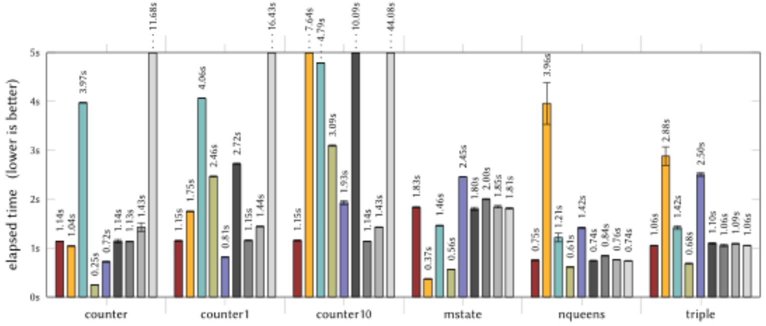
Perceus: Garbage Free Reference Counting with Reuse

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

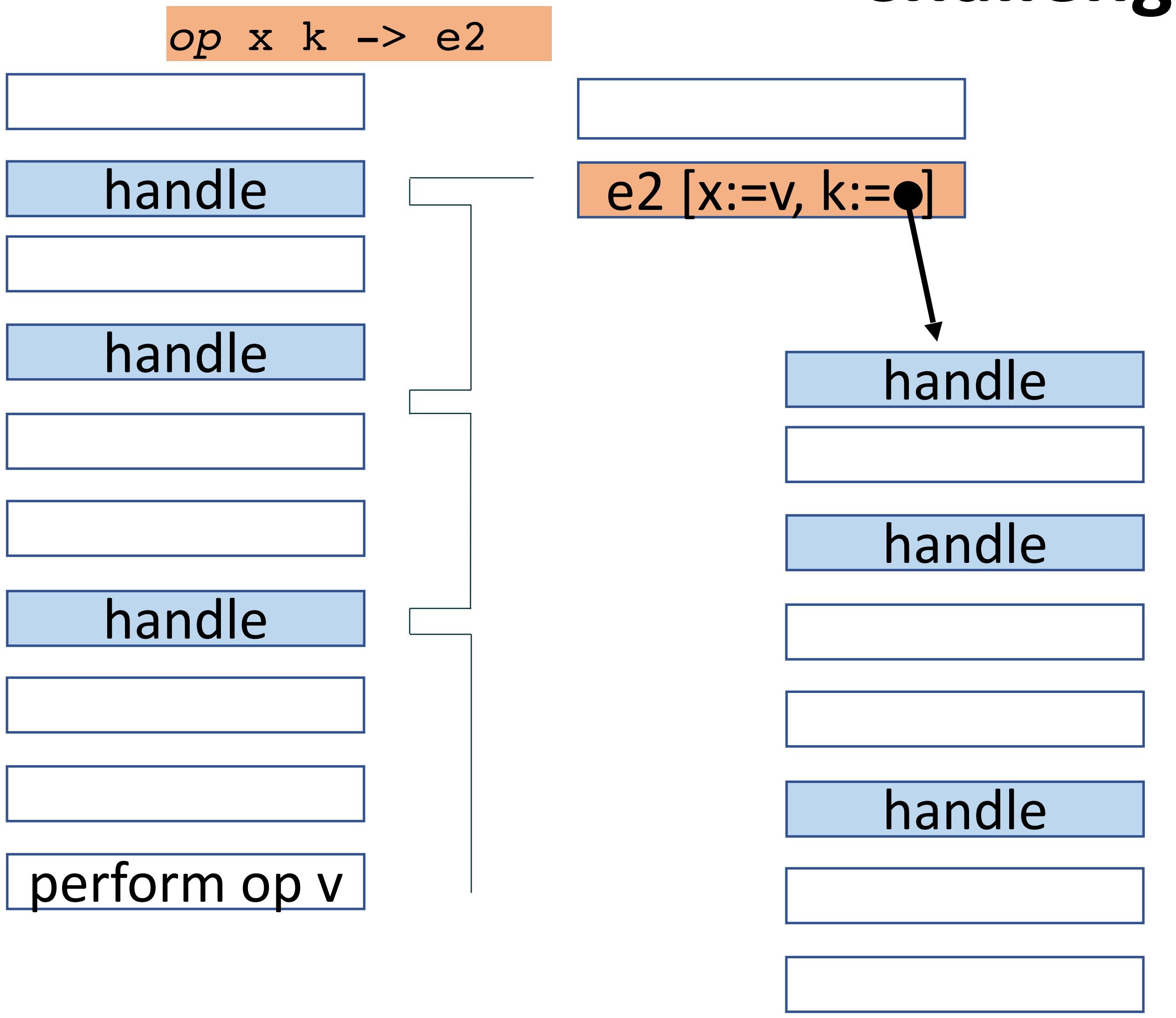
Ningning Xie*
University of Hong Kong
Hong Kong, China
nnxie@cs.hku.hk

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

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



Challenge



1. **Searching**
   a *linear* search through the current evaluation context

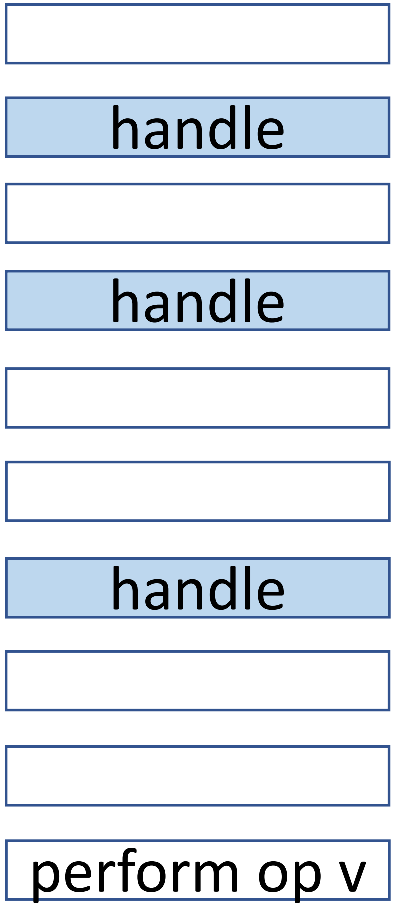
2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

Multi-prompt semantics

separating searching from capturing

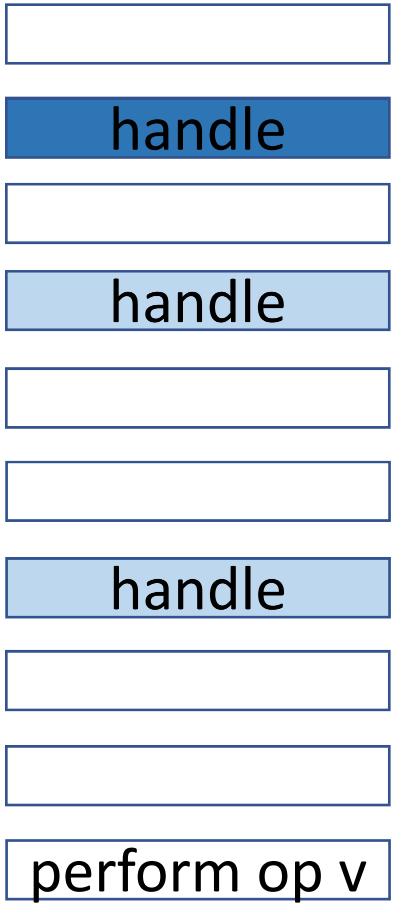
Multi-prompt semantics

separating searching from capturing



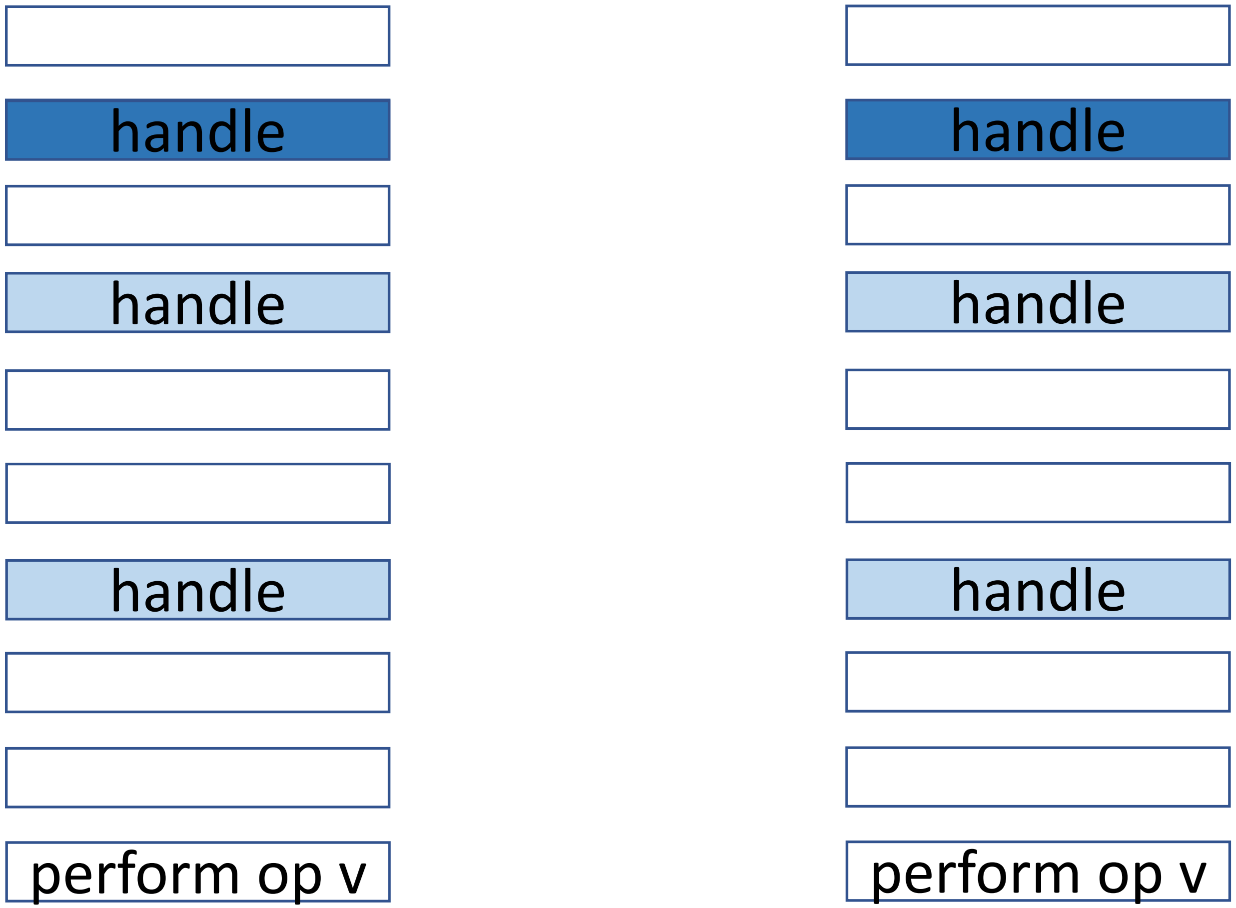
Multi-prompt semantics

separating searching from capturing



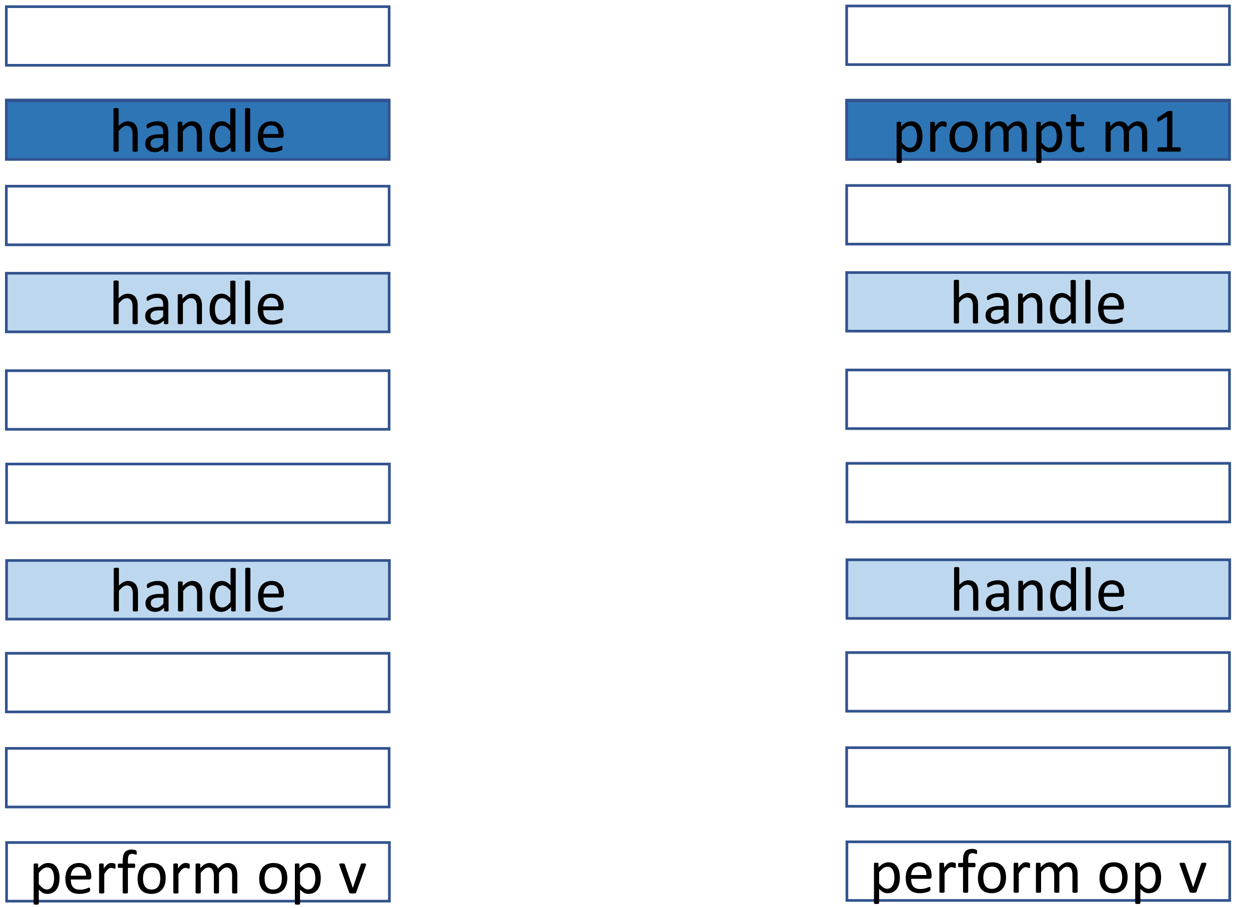
Multi-prompt semantics

separating searching from capturing



Multi-prompt semantics

separating searching from capturing



Multi-prompt semantics

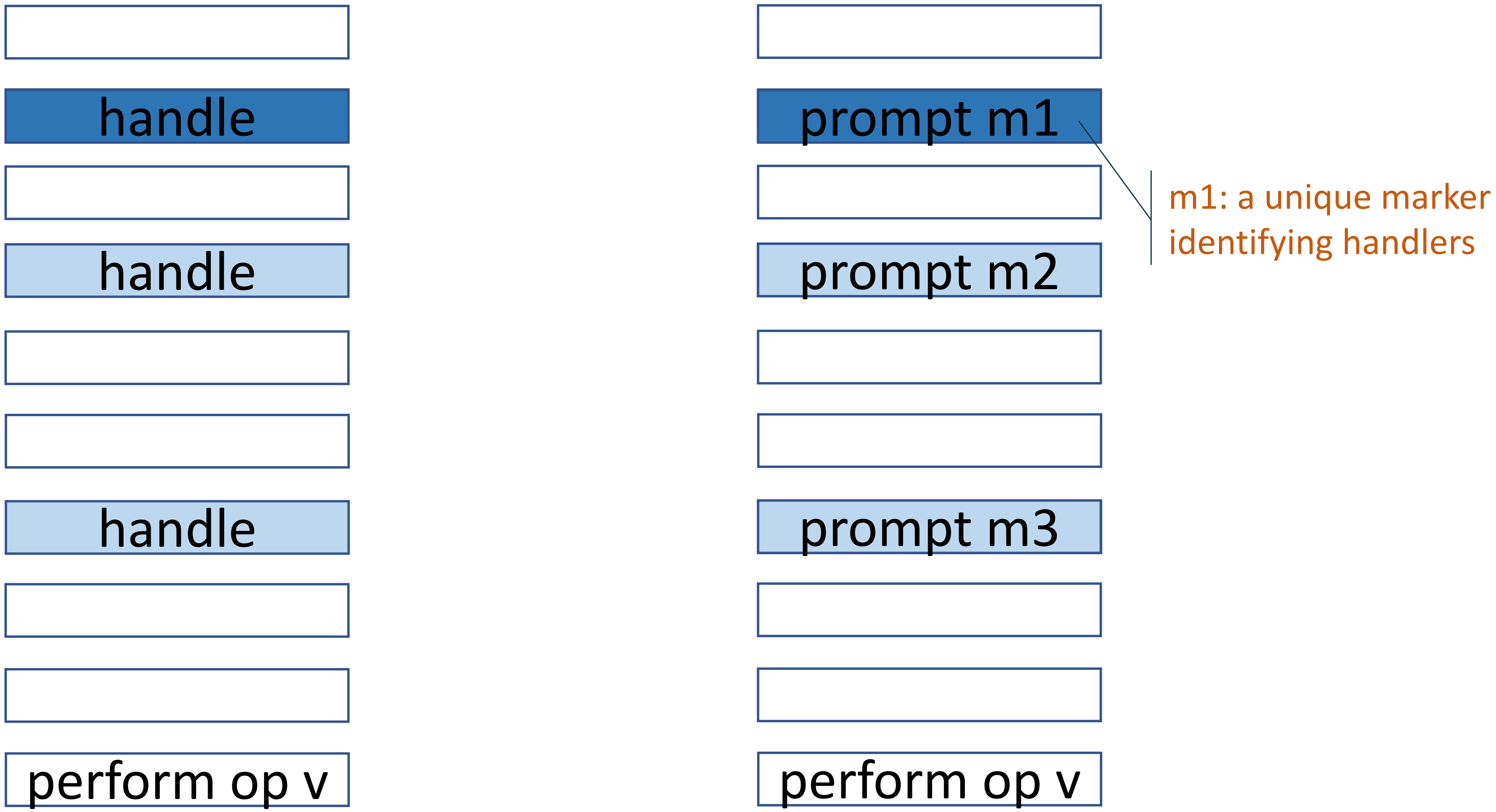
separating searching from capturing

Multi-prompt semantics

separating searching from capturing

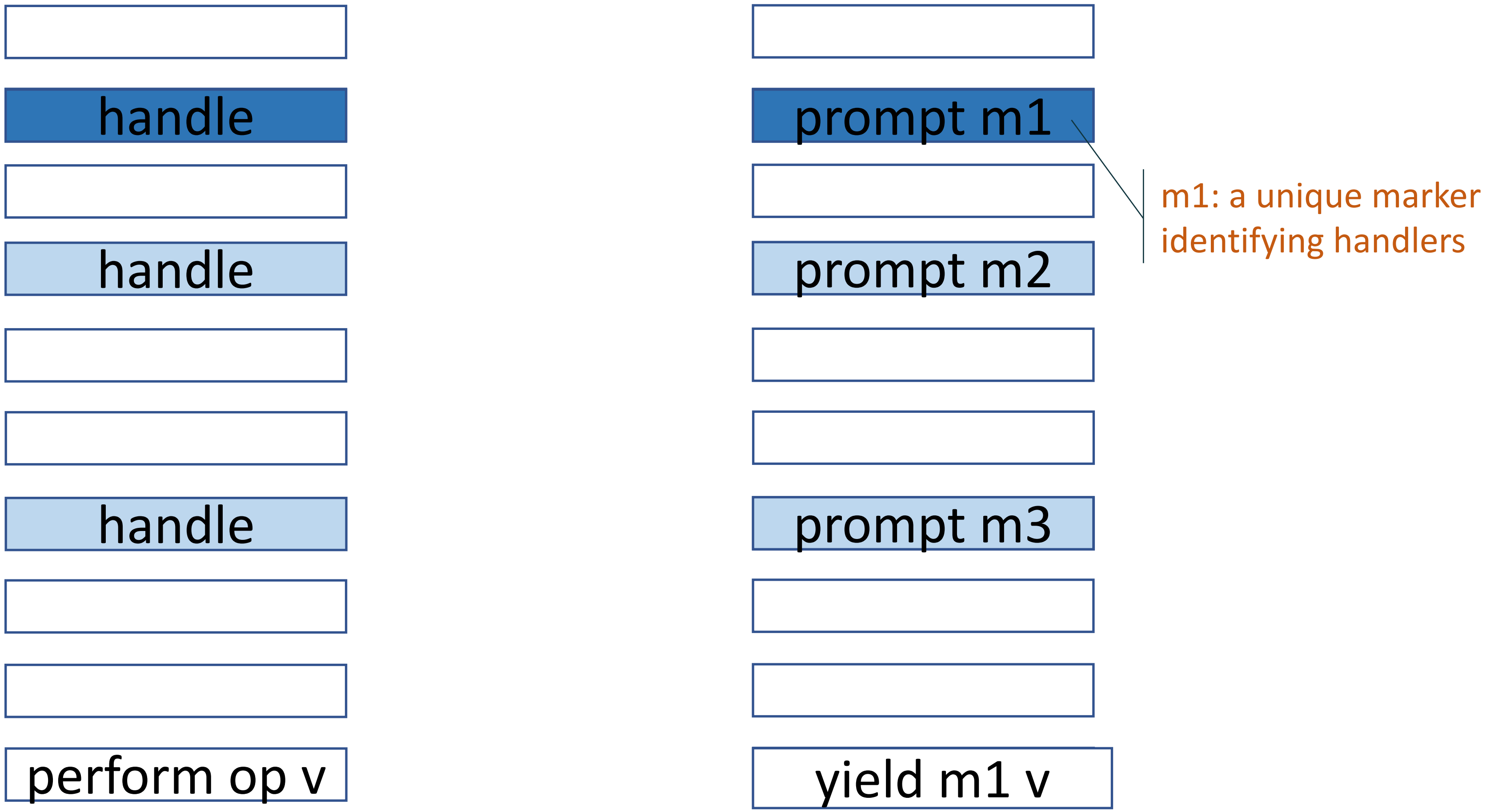
Multi-prompt semantics

separating searching from capturing



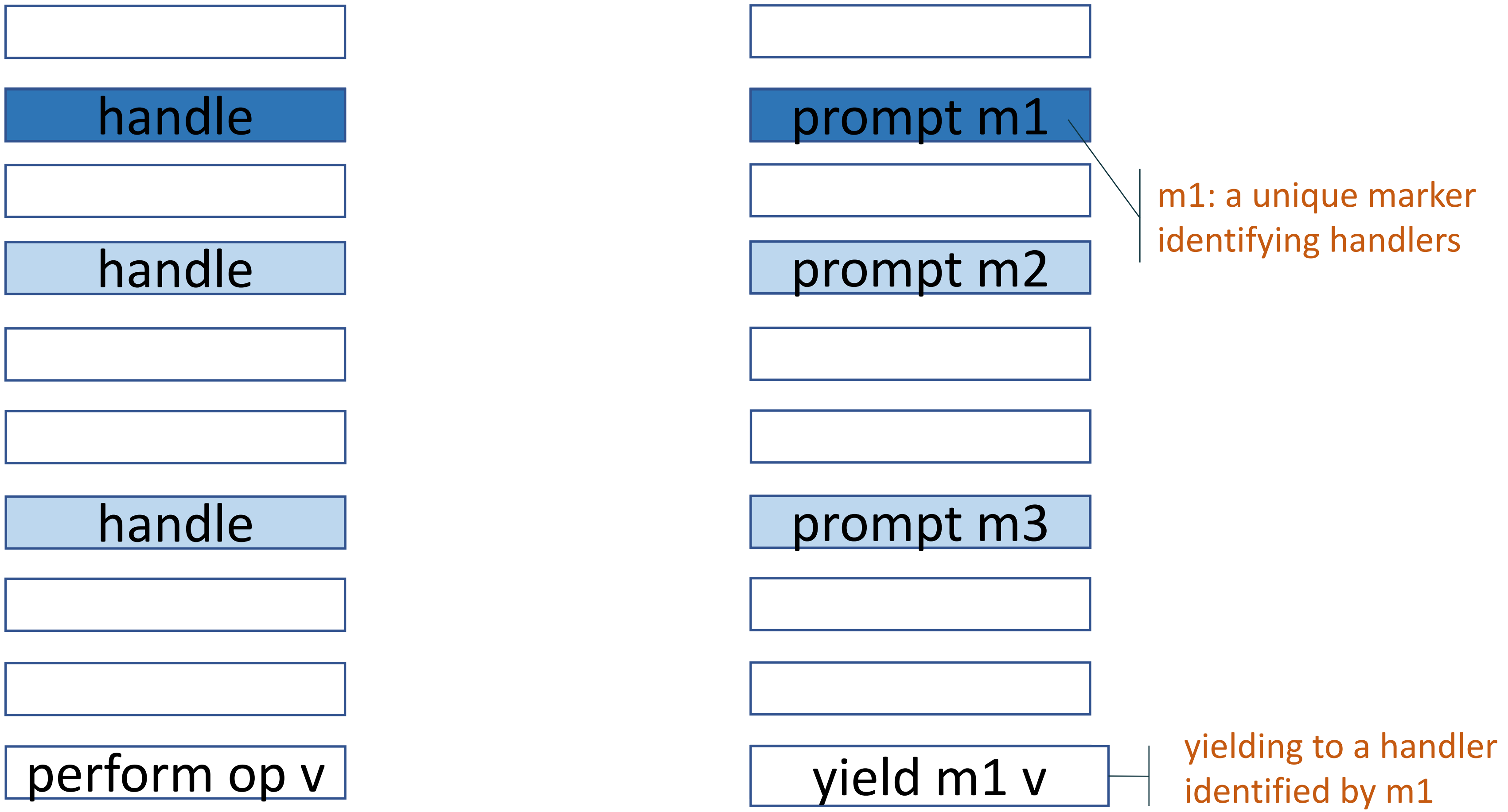
Multi-prompt semantics

separating searching from capturing



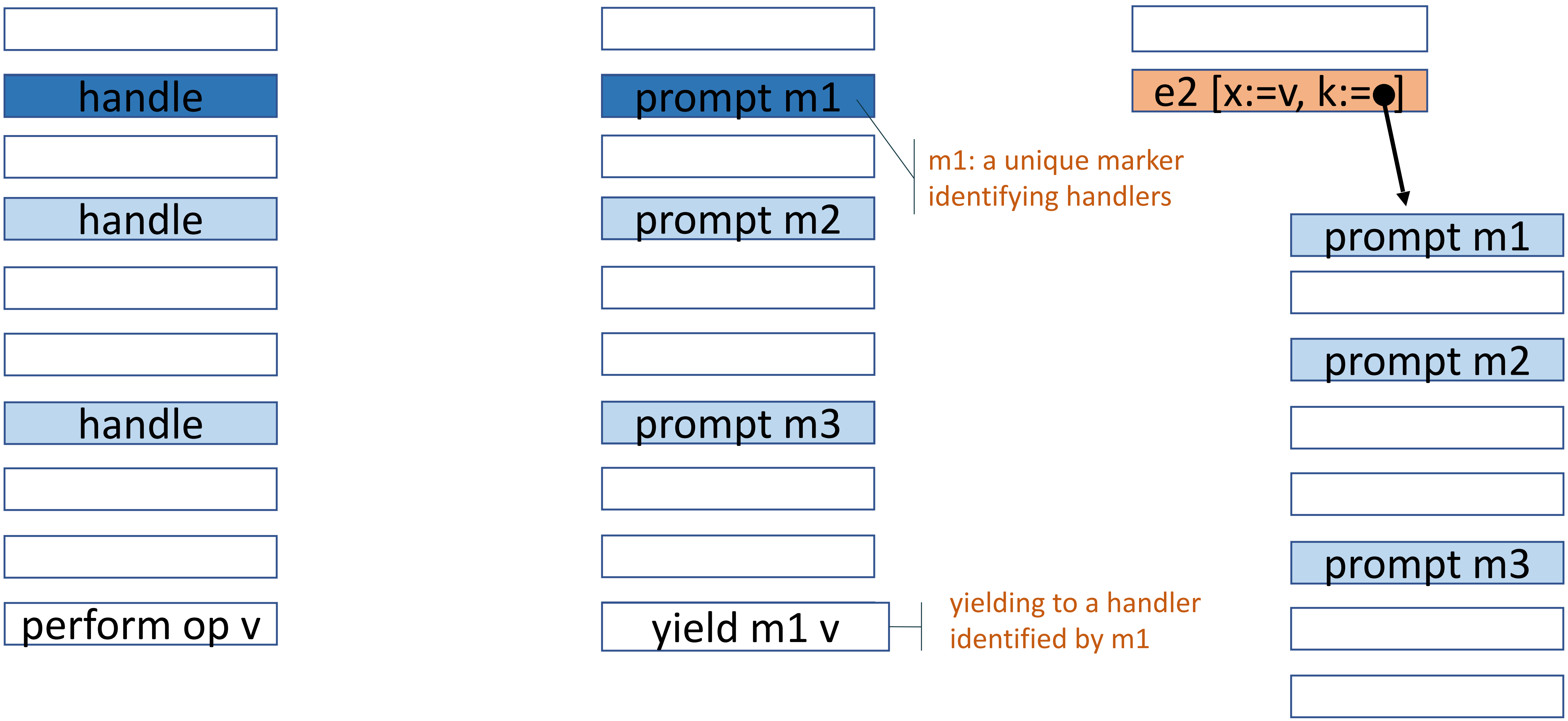
Multi-prompt semantics

separating searching from capturing



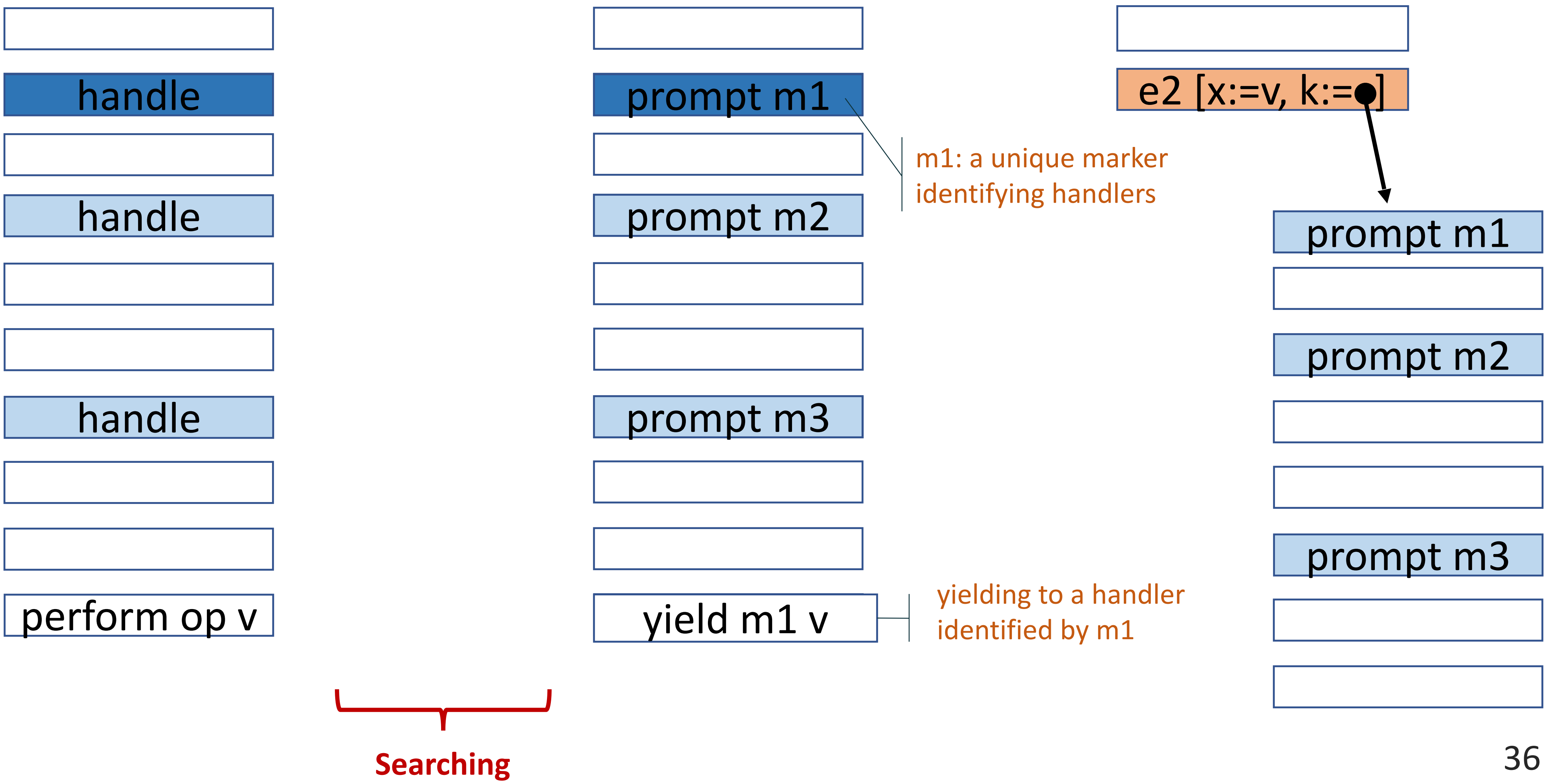
Multi-prompt semantics

separating searching from capturing



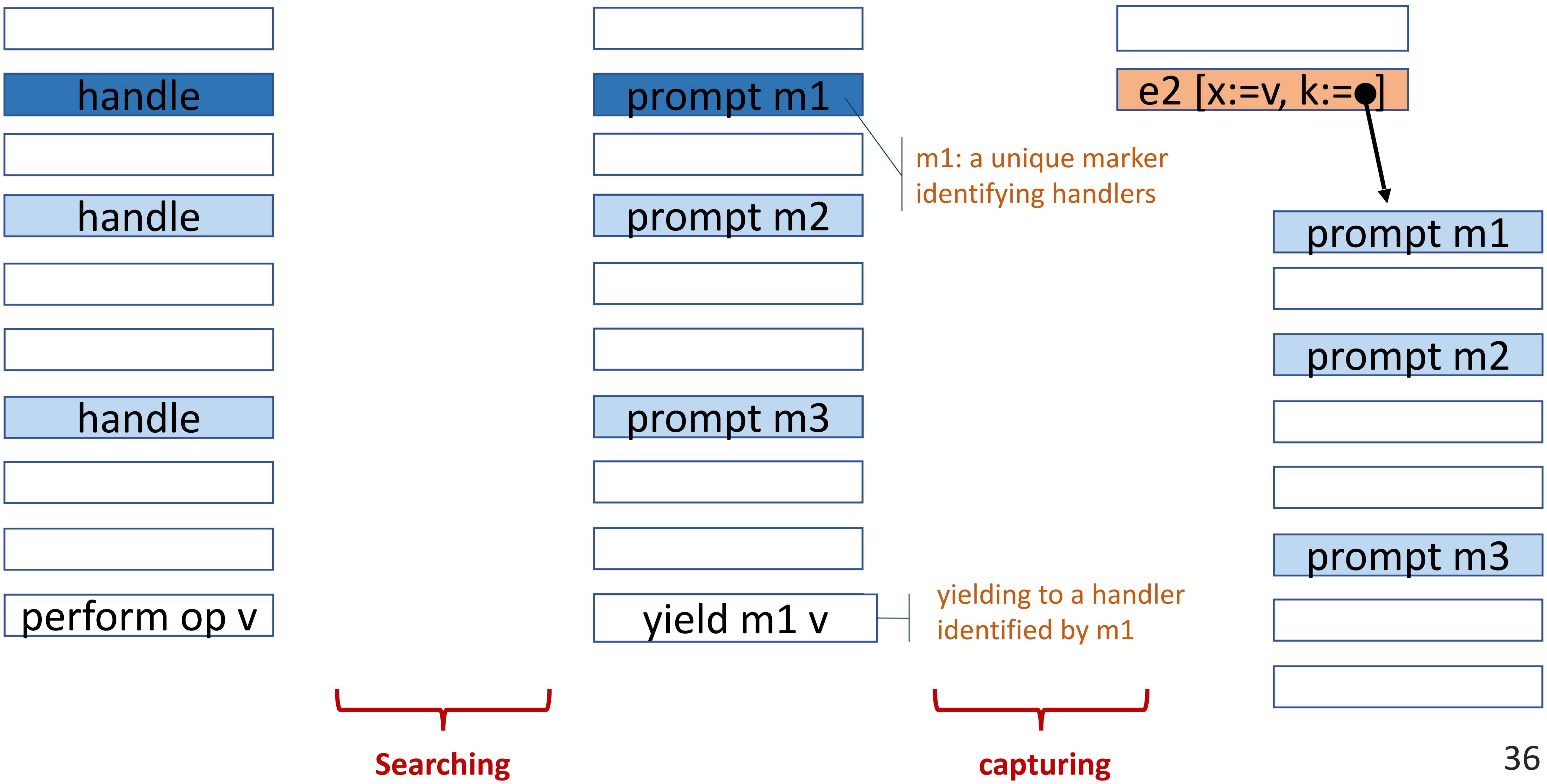
Multi-prompt semantics

separating searching from capturing



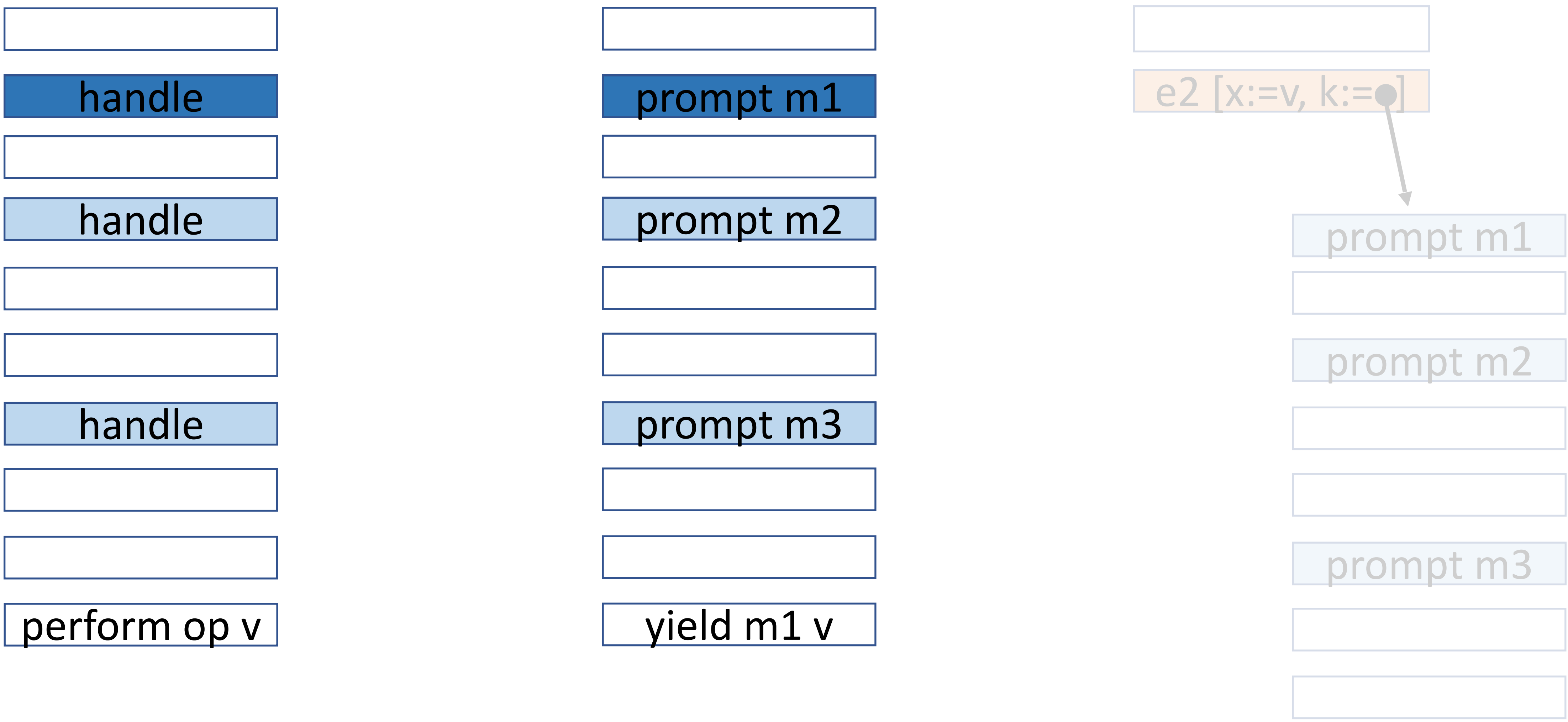
Multi-prompt semantics

separating searching from capturing



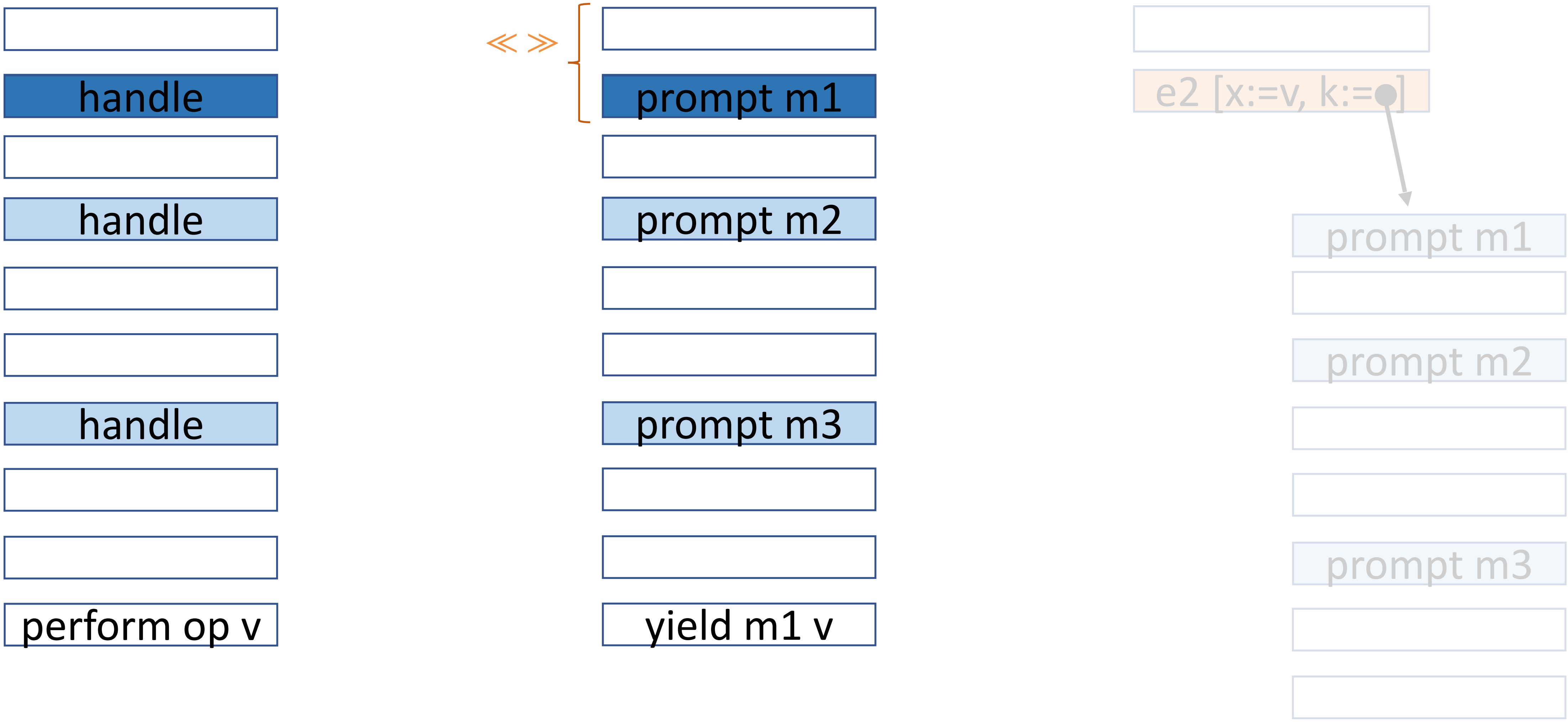
Evidence-passing semantics

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



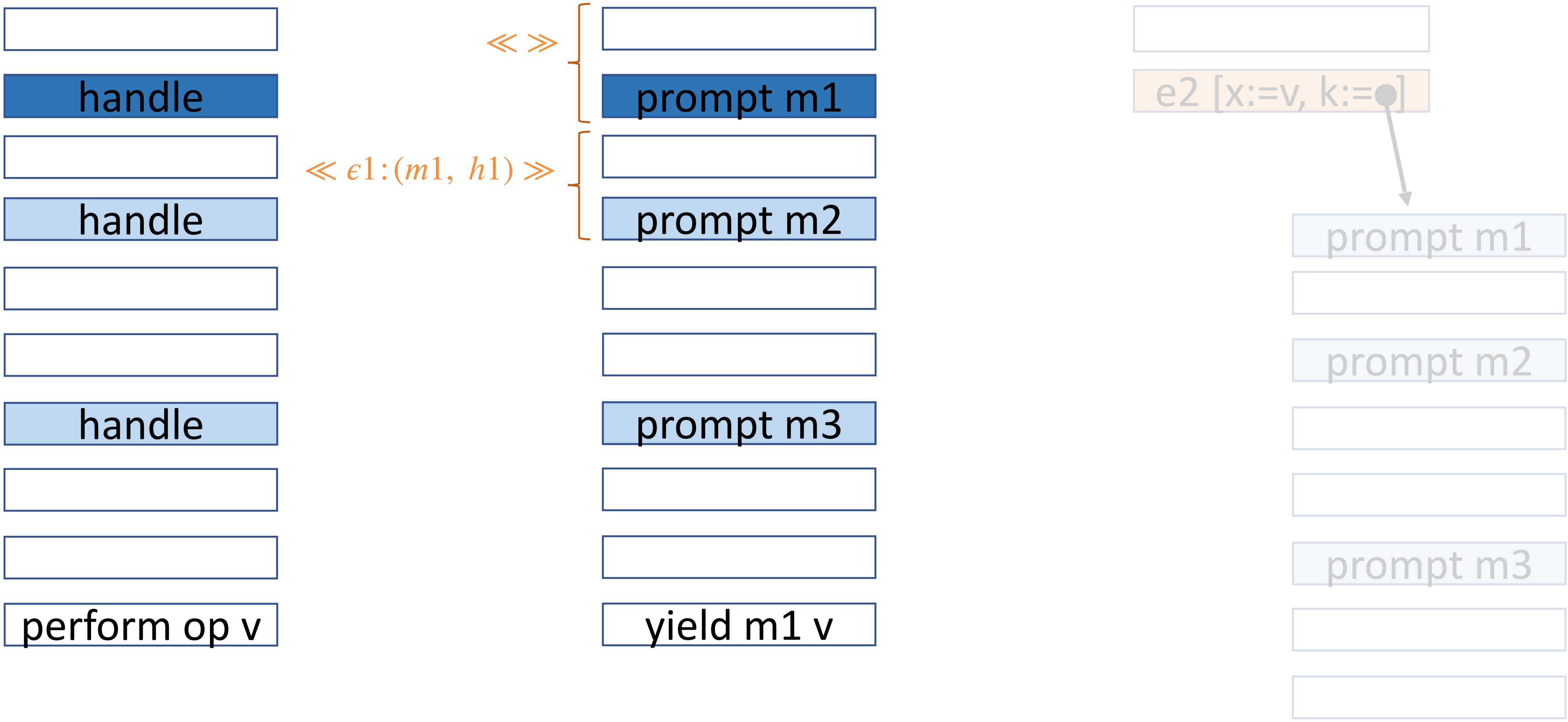
Evidence-passing semantics

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



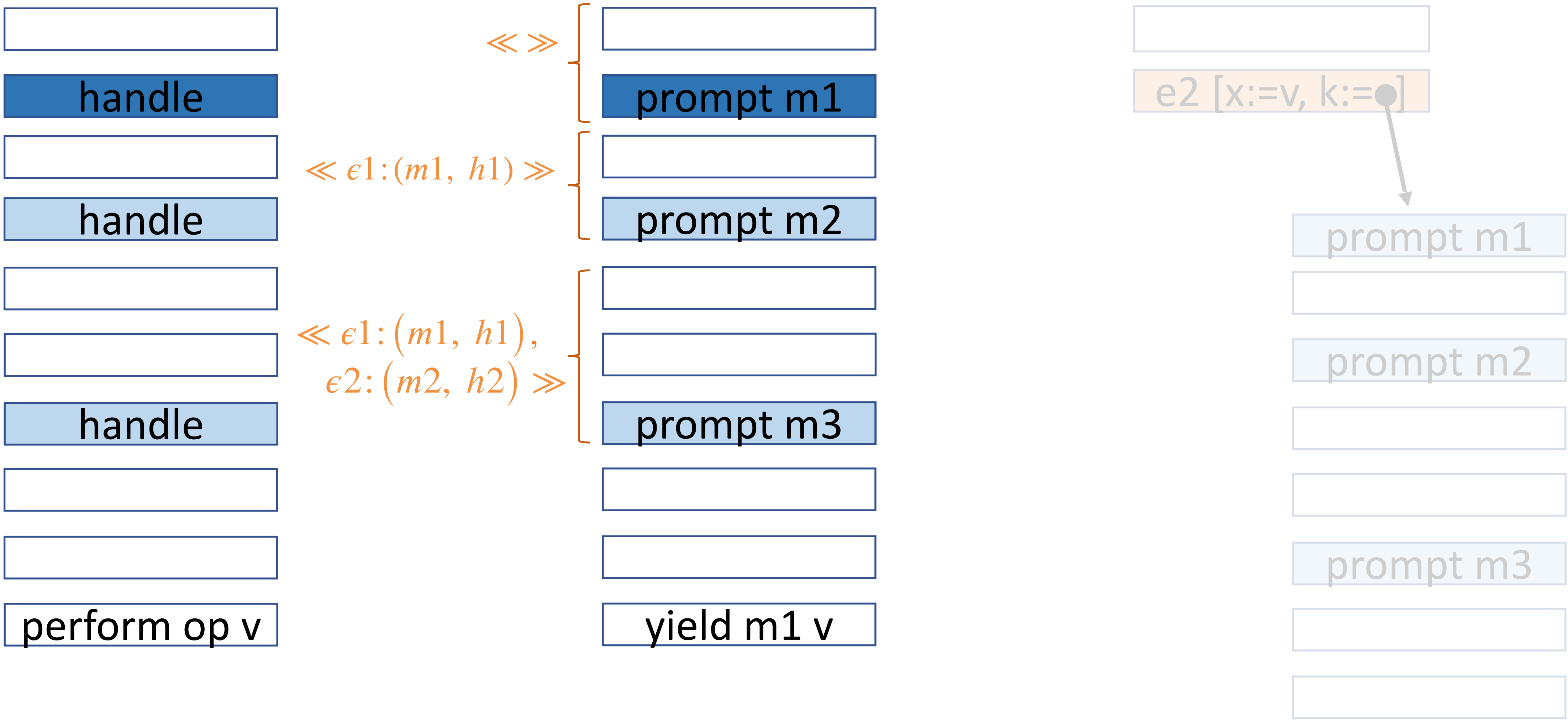
Evidence-passing semantics

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



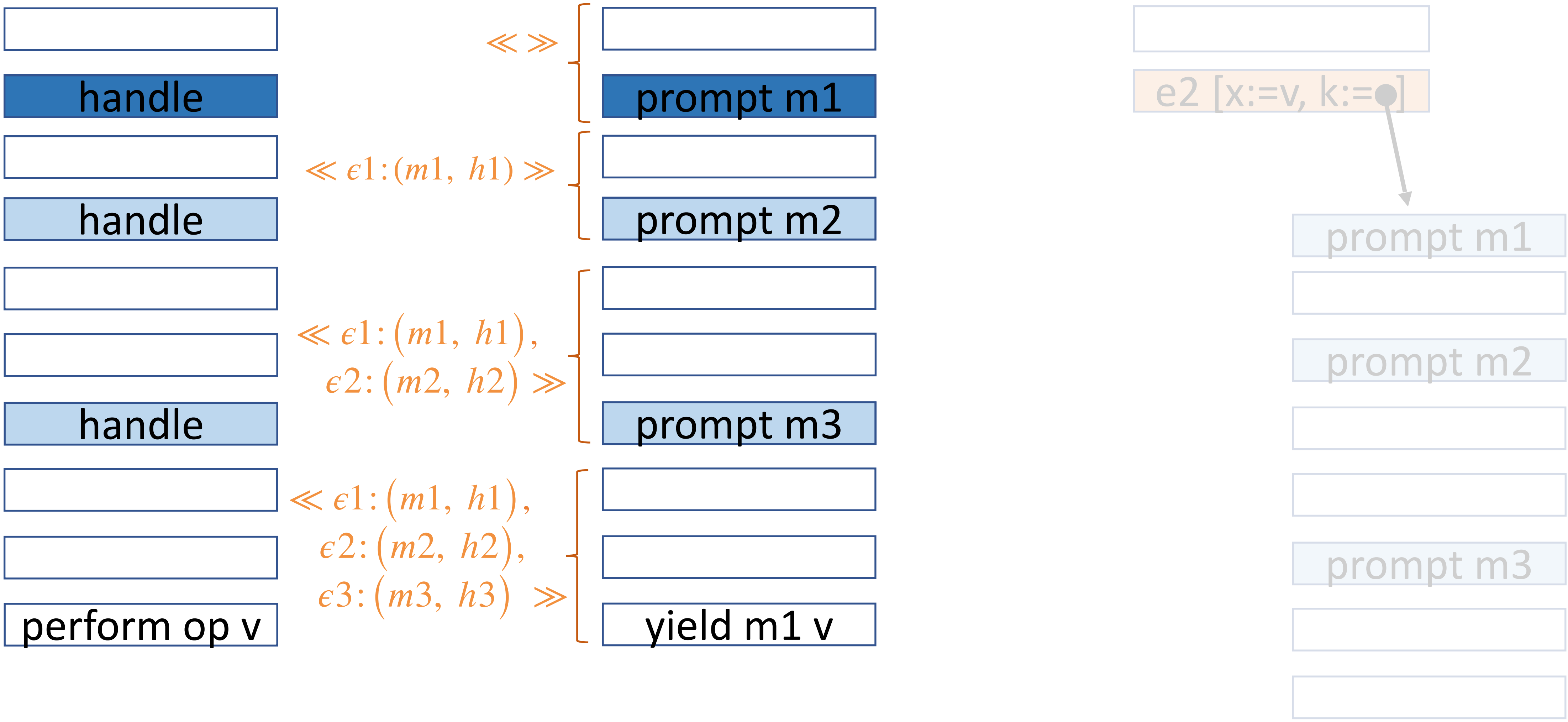
Evidence-passing semantics

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



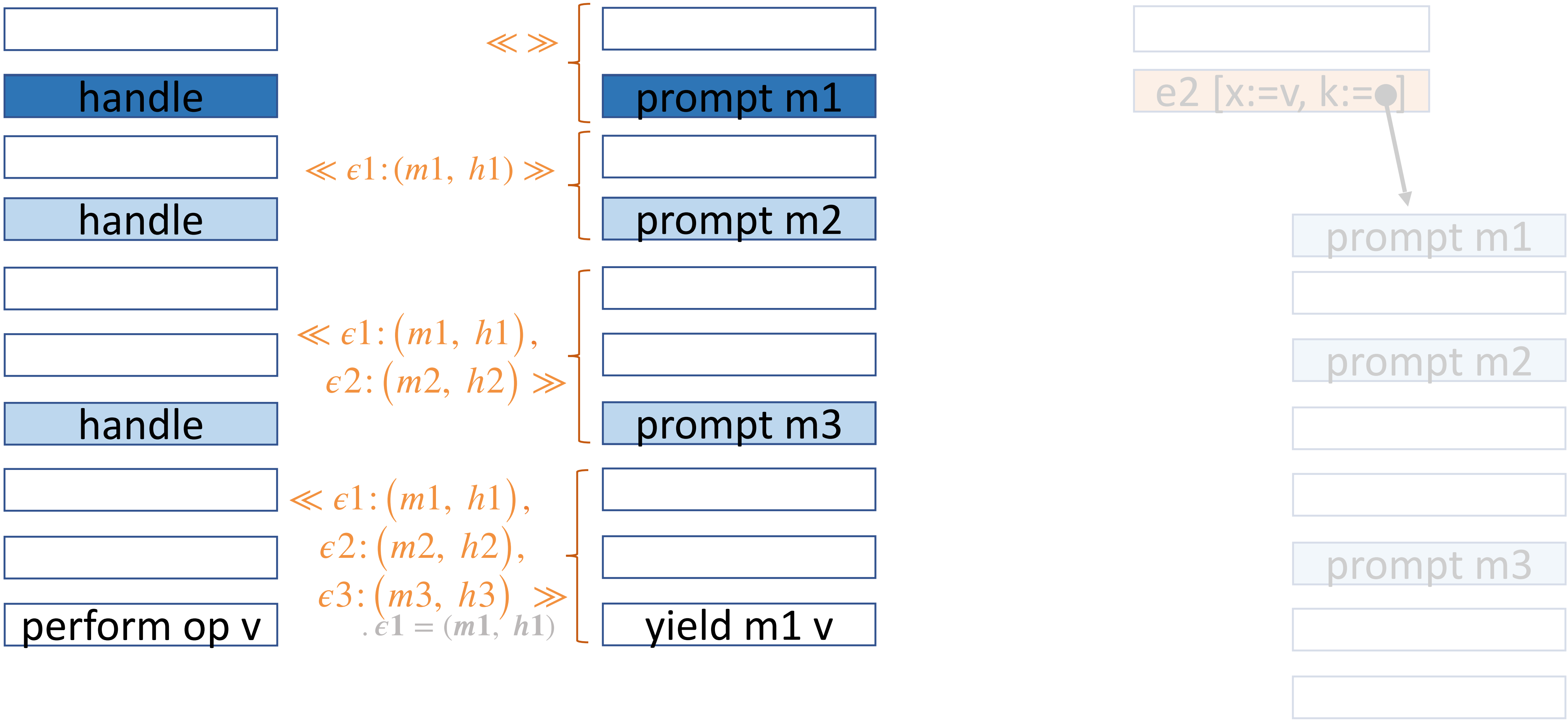
Evidence-passing semantics

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



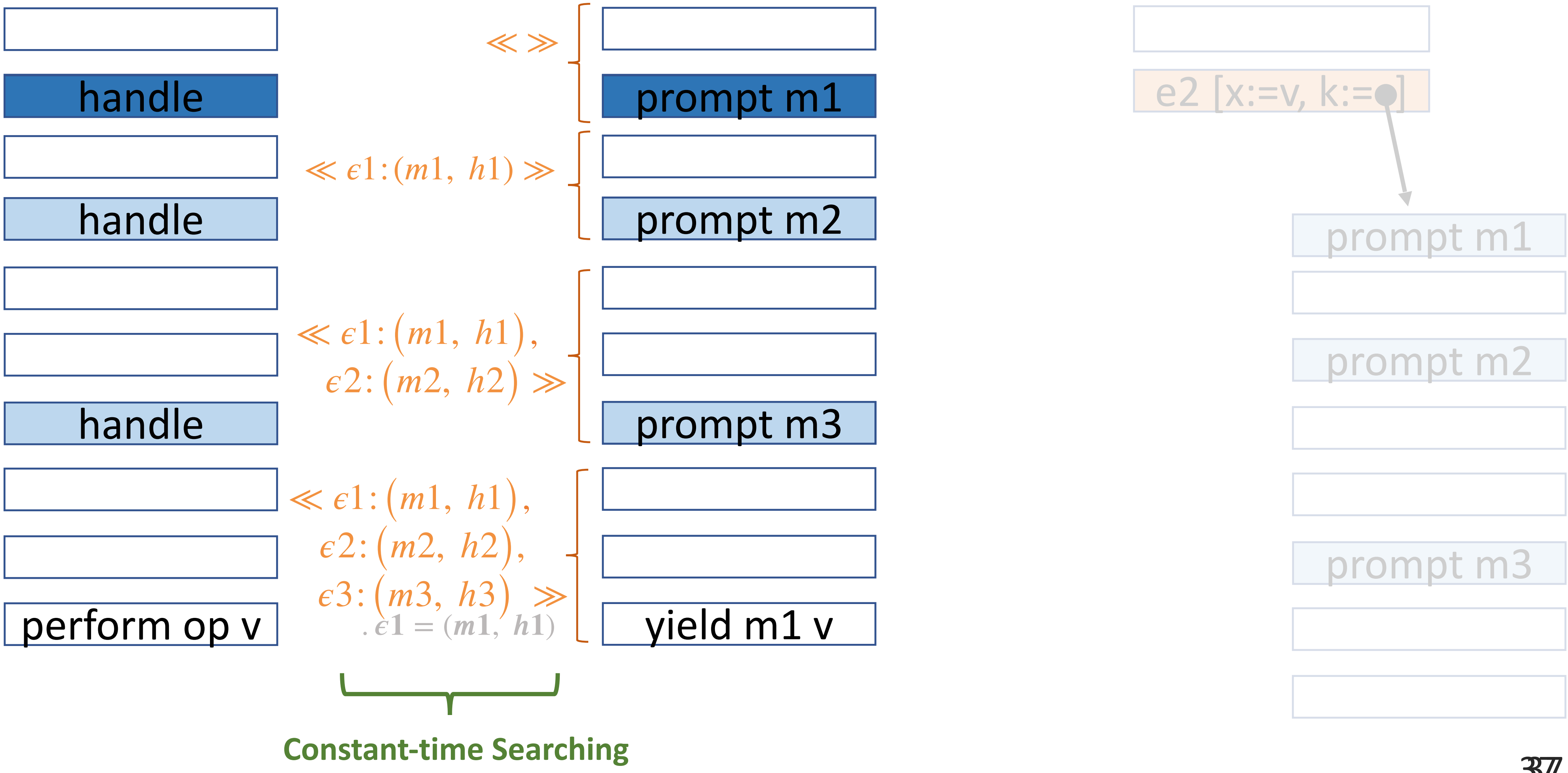
Evidence-passing semantics

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



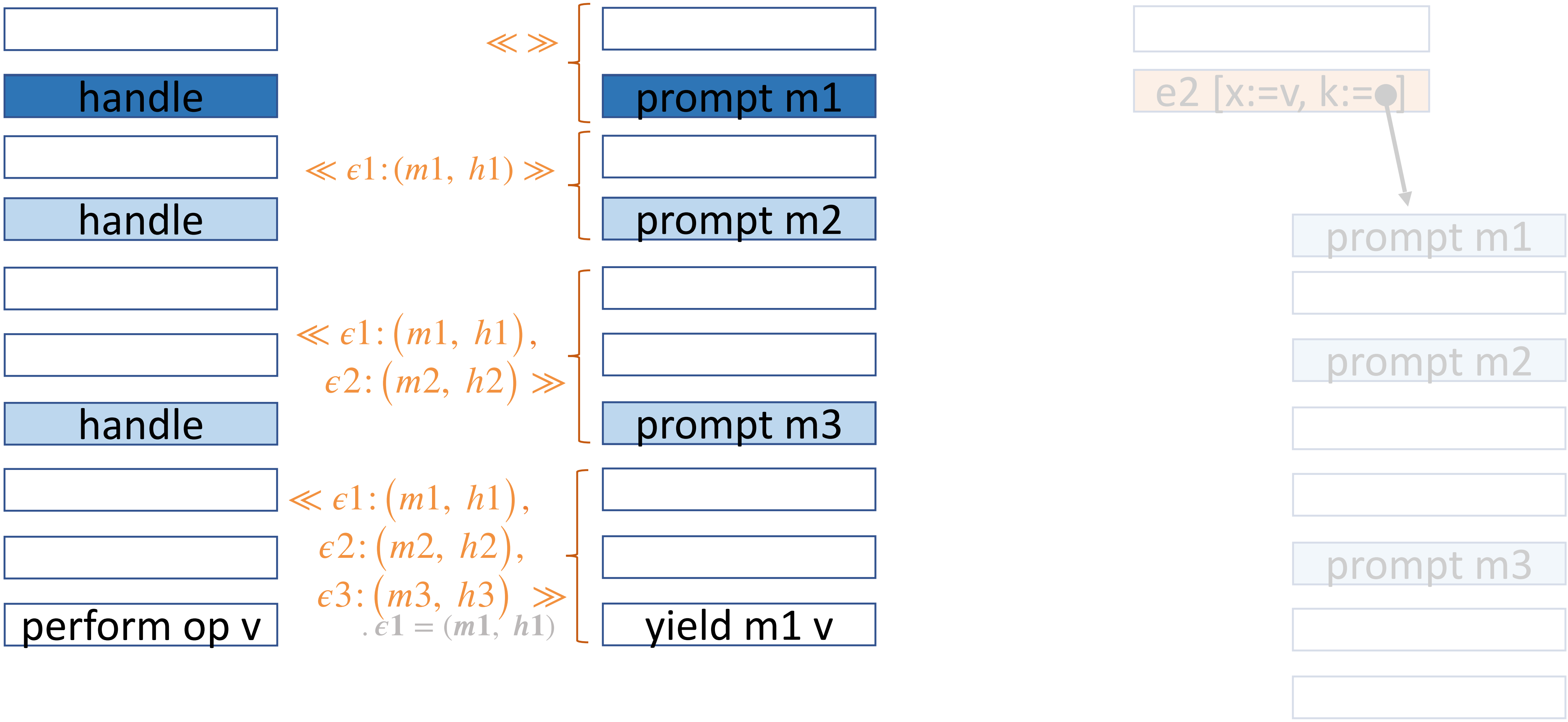
Evidence-passing semantics

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



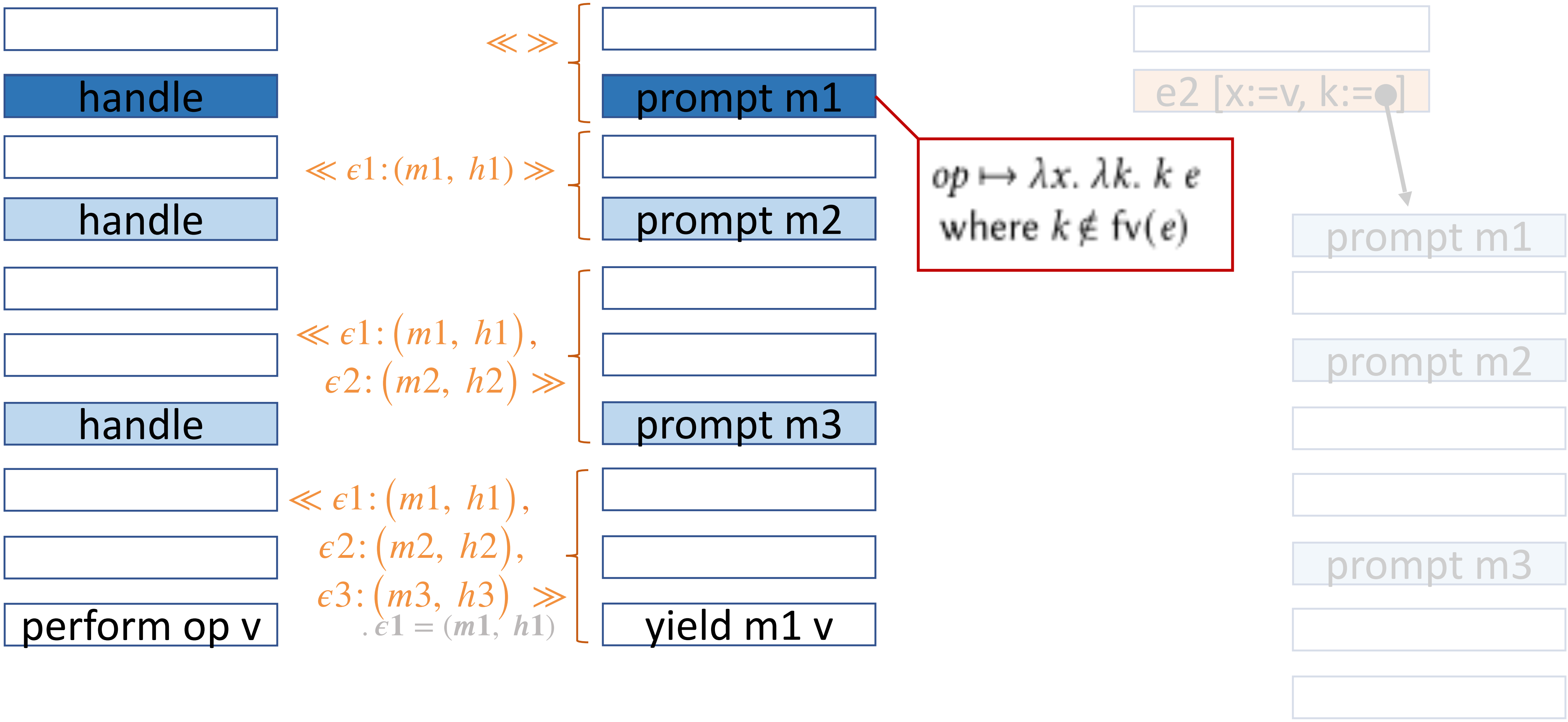
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place



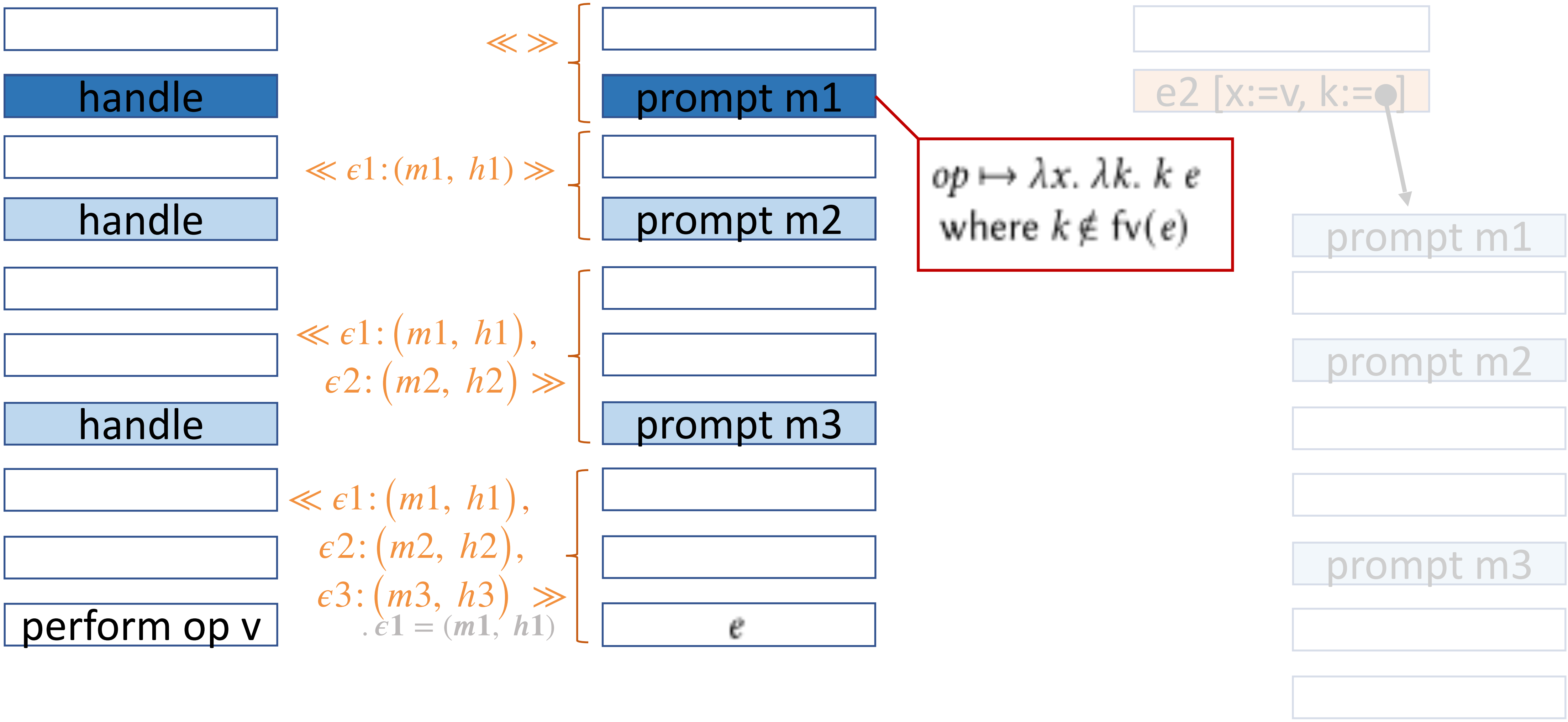
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place



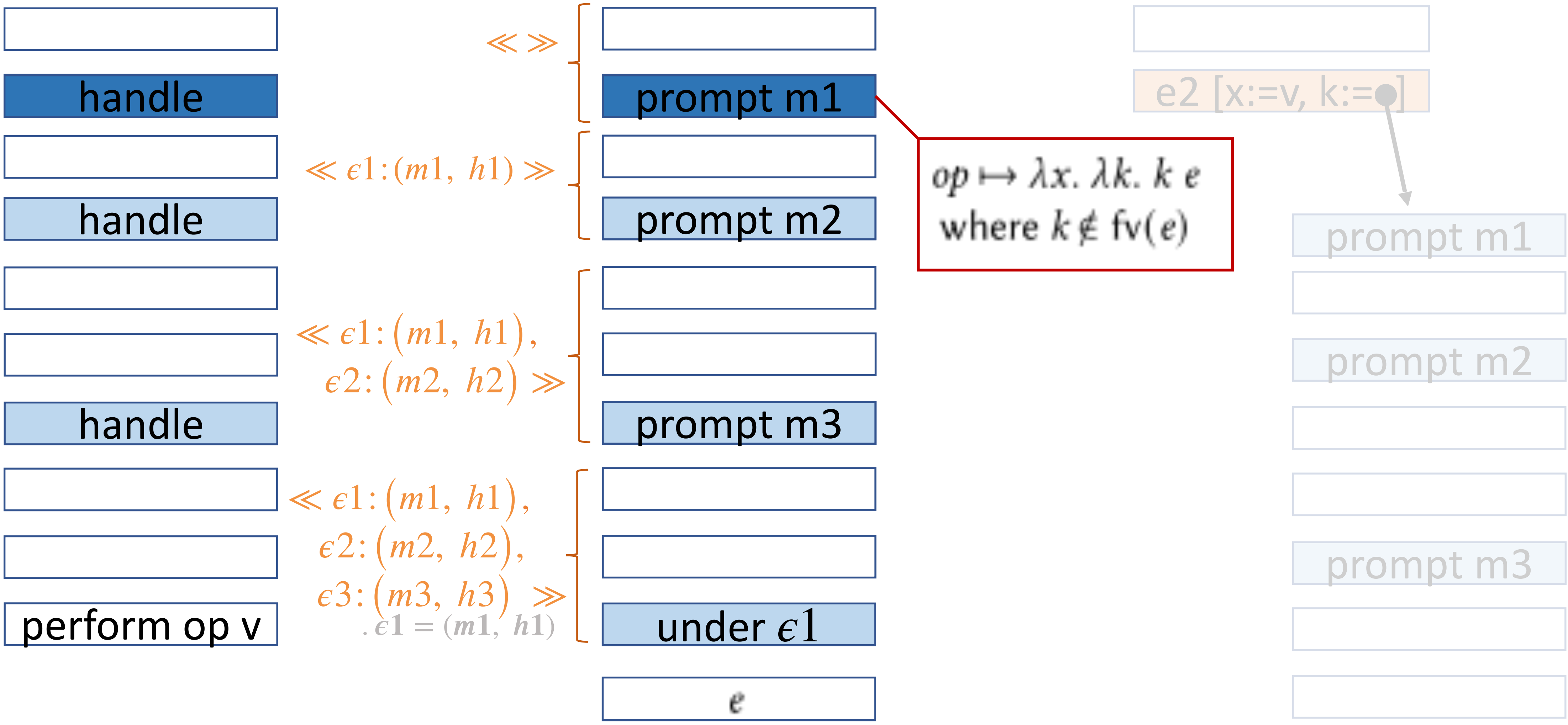
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place



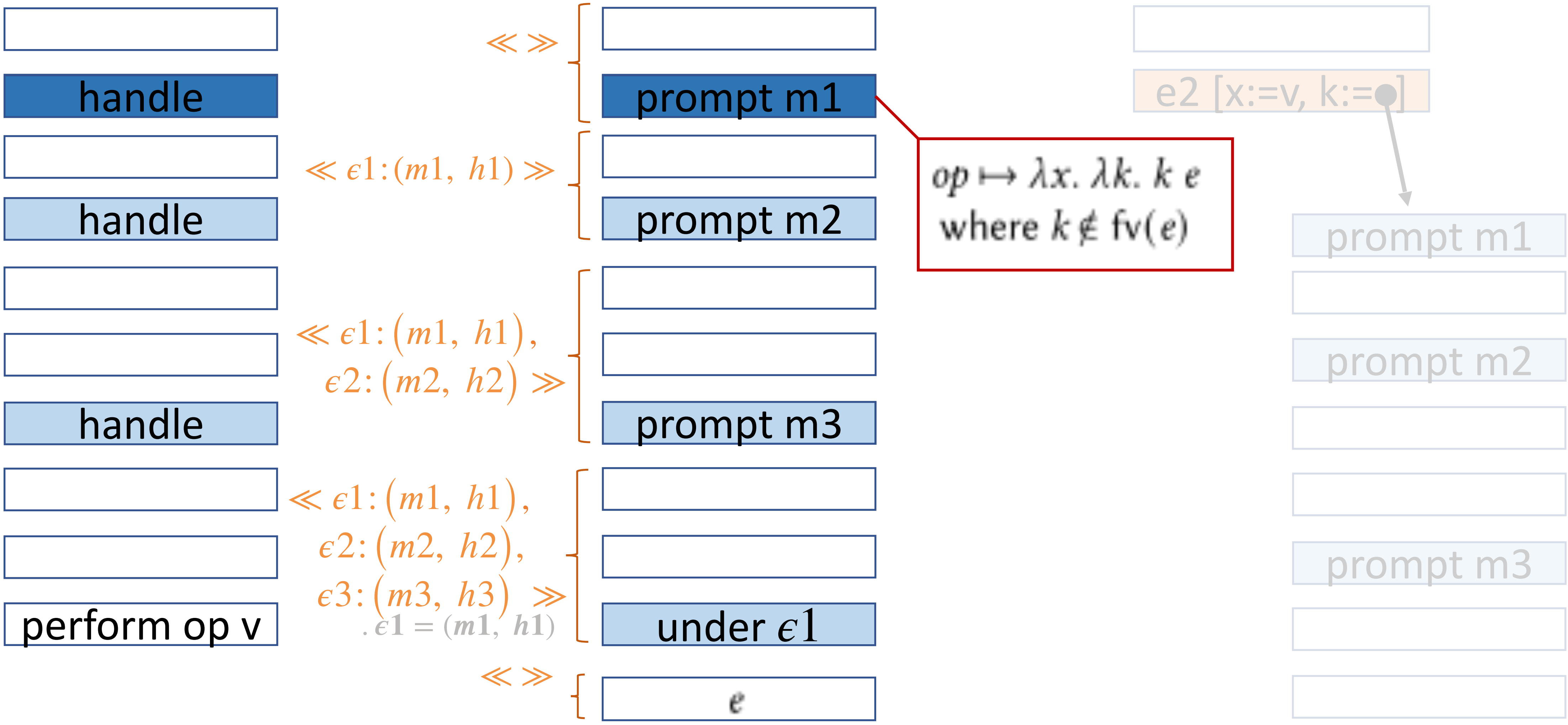
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place



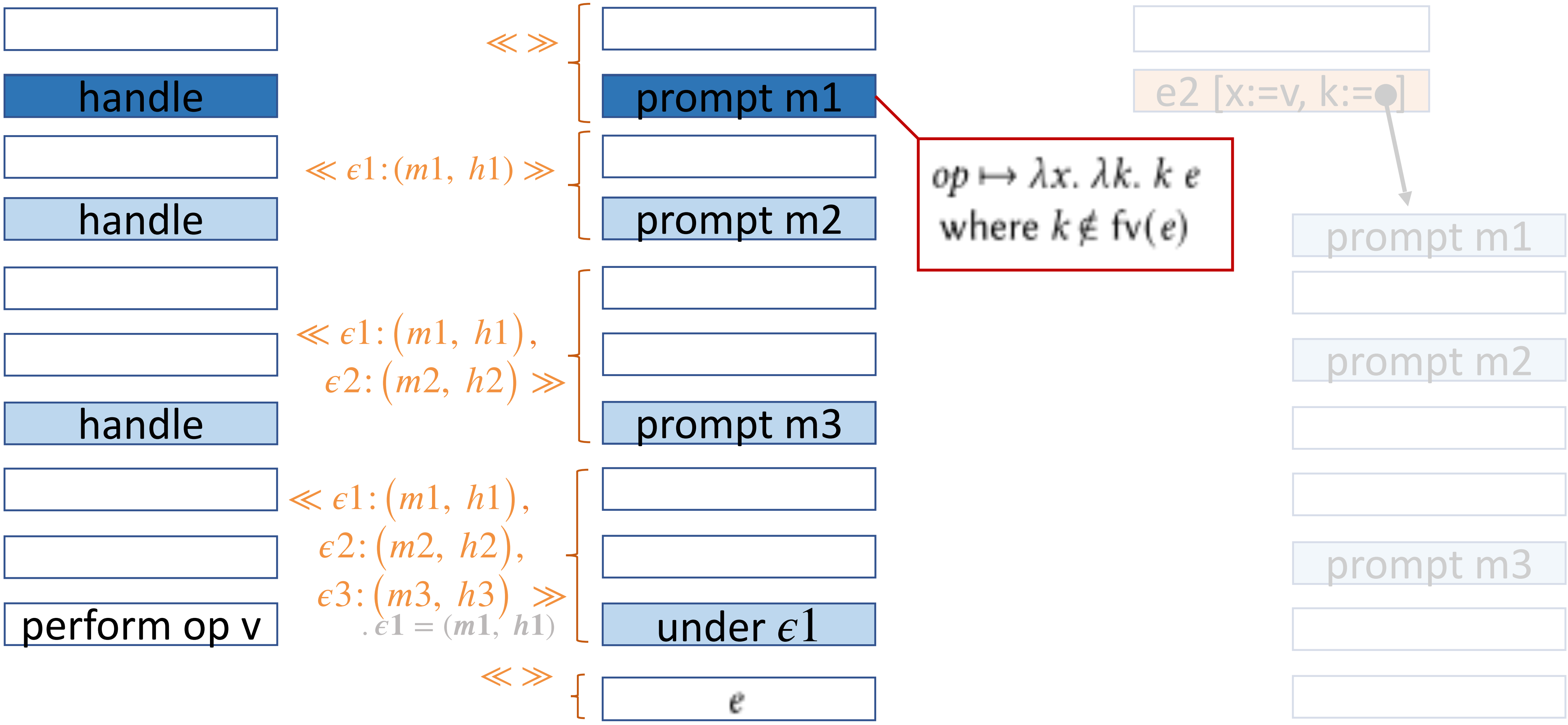
Optimization of tail-resumptive operations

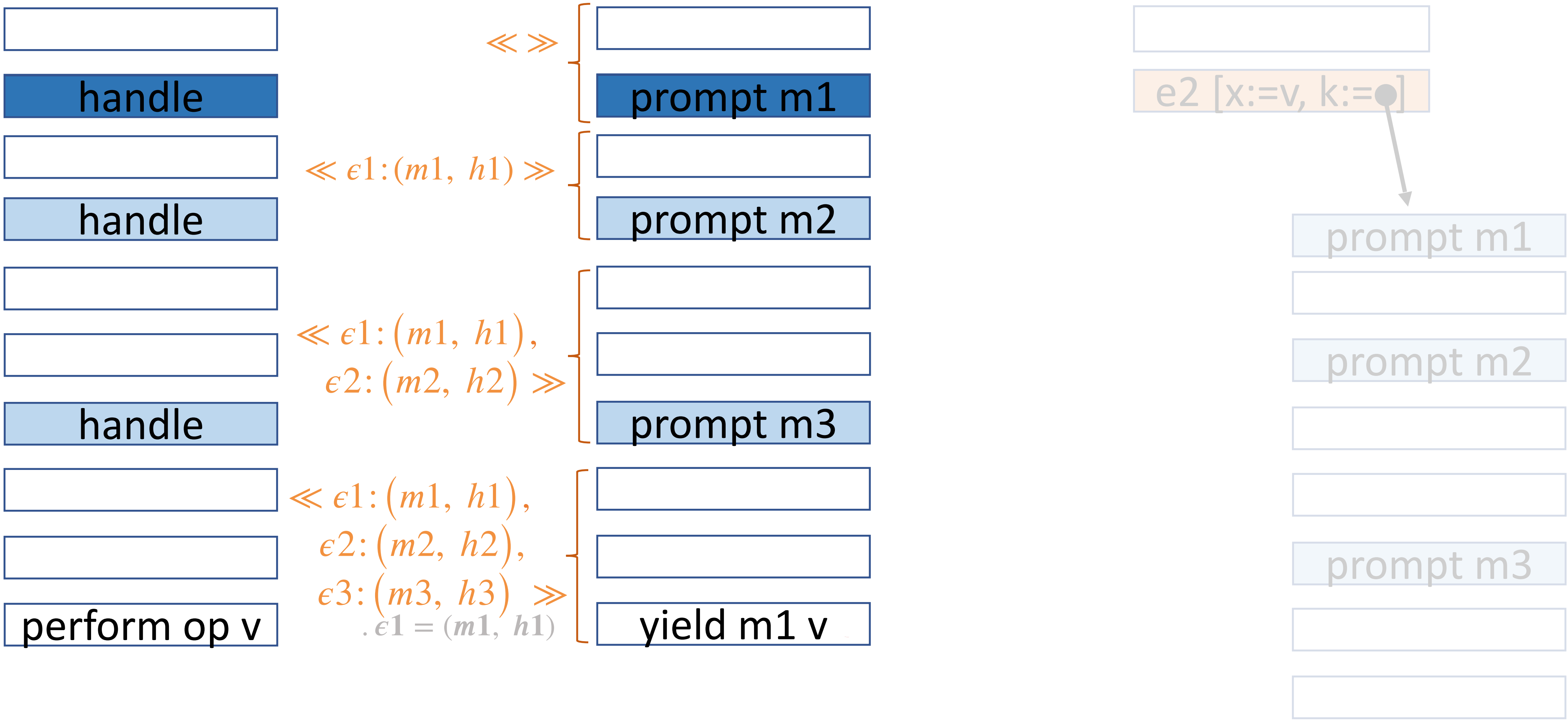
avoid yields: evaluate tail-resumptive operations in-place

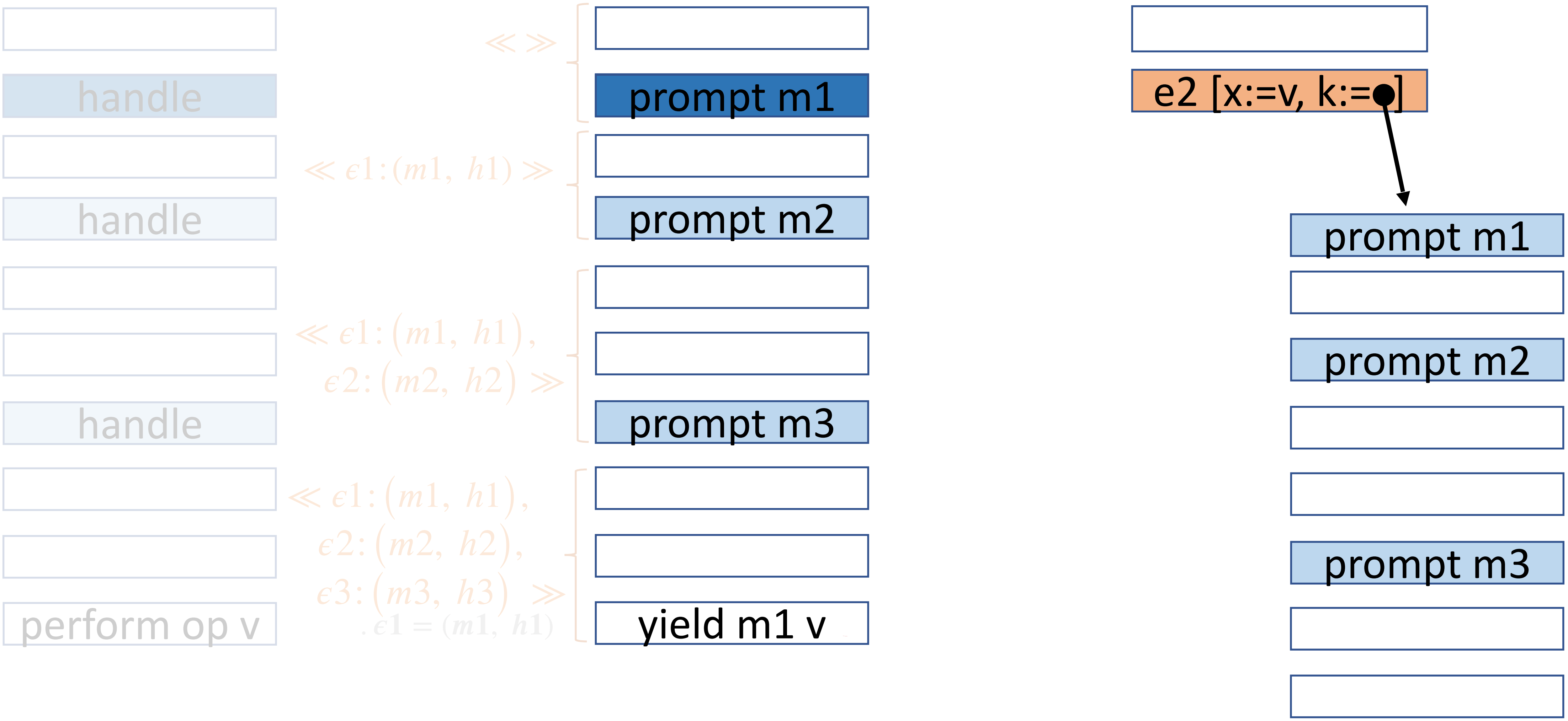


Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

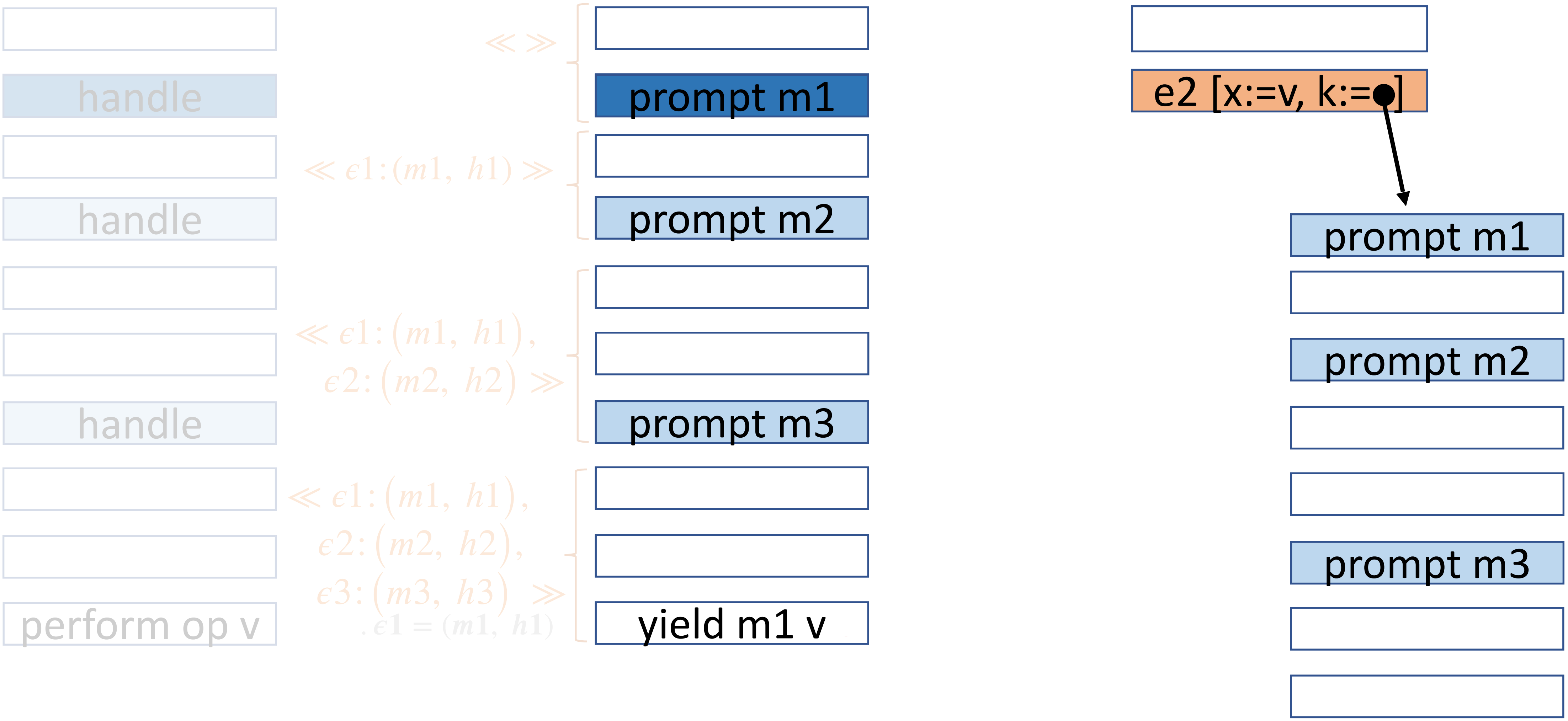



handle
handle
handle
perform op v
≪ ≫
≪ ϵ1:(m1, h1) ≫
≪ ϵ1:(m1, h1), ϵ2:(m2, h2) ≫
≪ ϵ1:(m1, h1), ϵ2:(m2, h2), ϵ3:(m3, h3) ≫
. ϵ1 = (m1, h1)
prompt m1
prompt m2
prompt m3
yield m1 v
e2 [x:=v, k:=●]
prompt m1
prompt m2
prompt m3


handle
handle
handle
perform op v
≪ ≫
≪ ϵ1 : (m1, h1) ≫
≪ ϵ1 : (m1, h1), ϵ2 : (m2, h2) ≫
≪ ϵ1 : (m1, h1), ϵ2 : (m2, h2), ϵ3 : (m3, h3) ≫
. ϵ1 = (m1, h1)
prompt m1
prompt m2
prompt m3
yield m1 v
e2 [x:=v, k:=●]
prompt m1
prompt m2
prompt m3

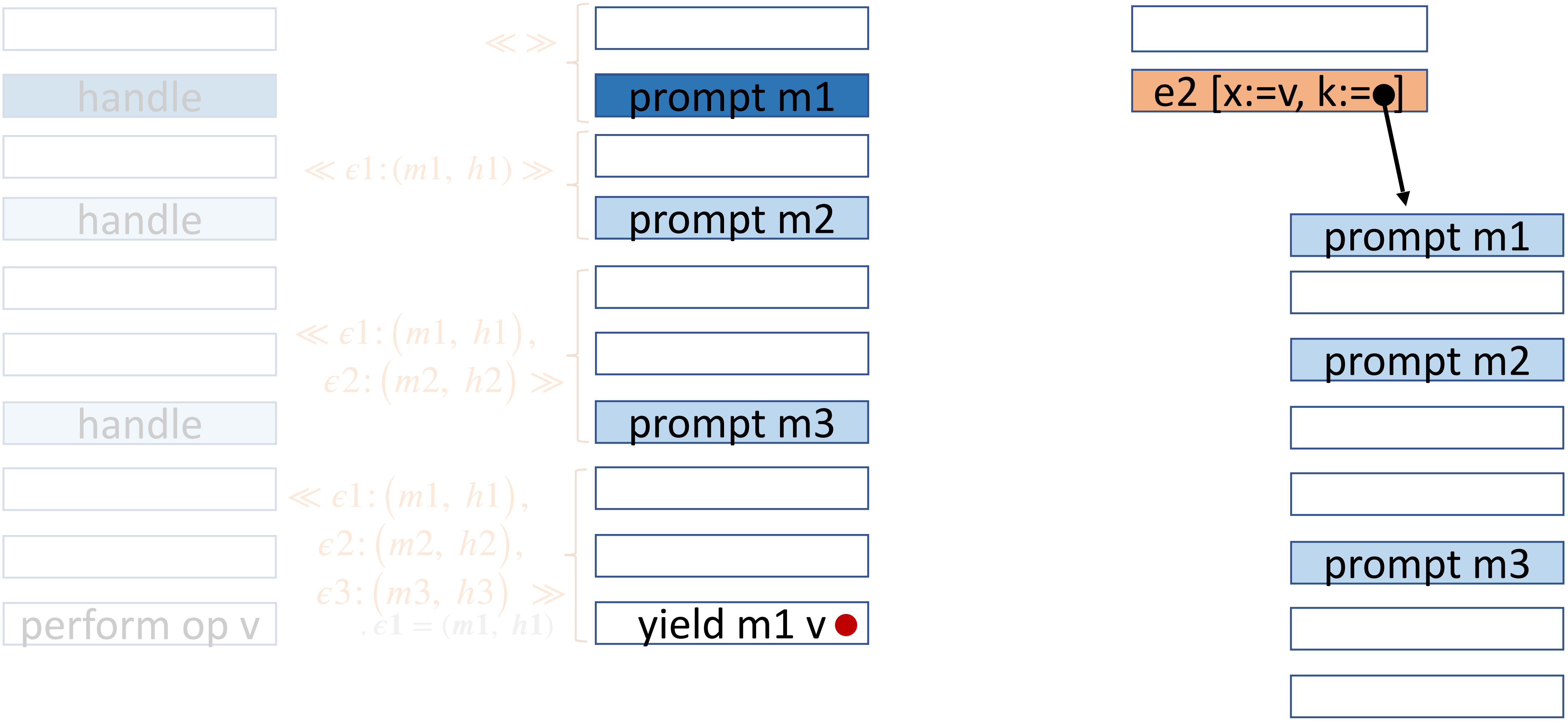
Bubbling yields

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



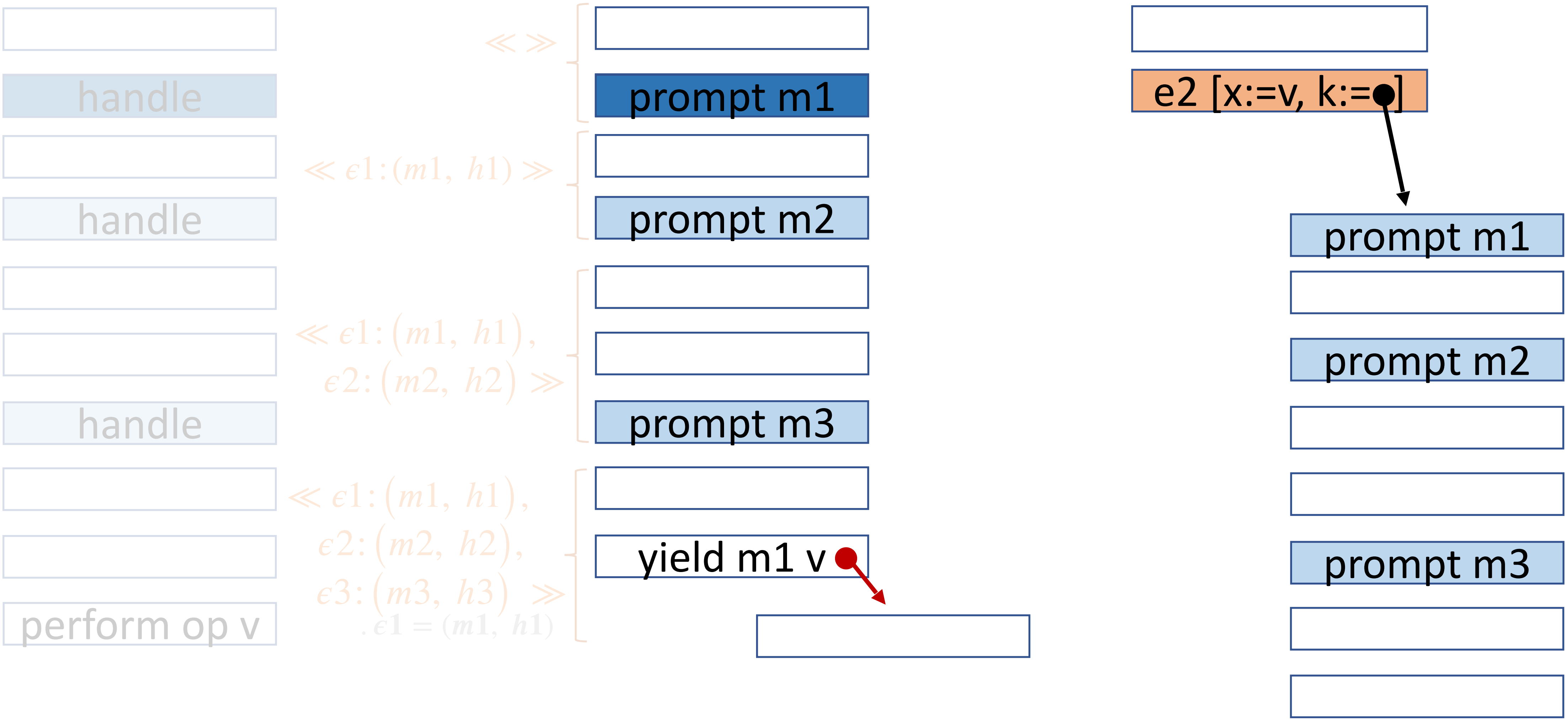
Bubbling yields

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



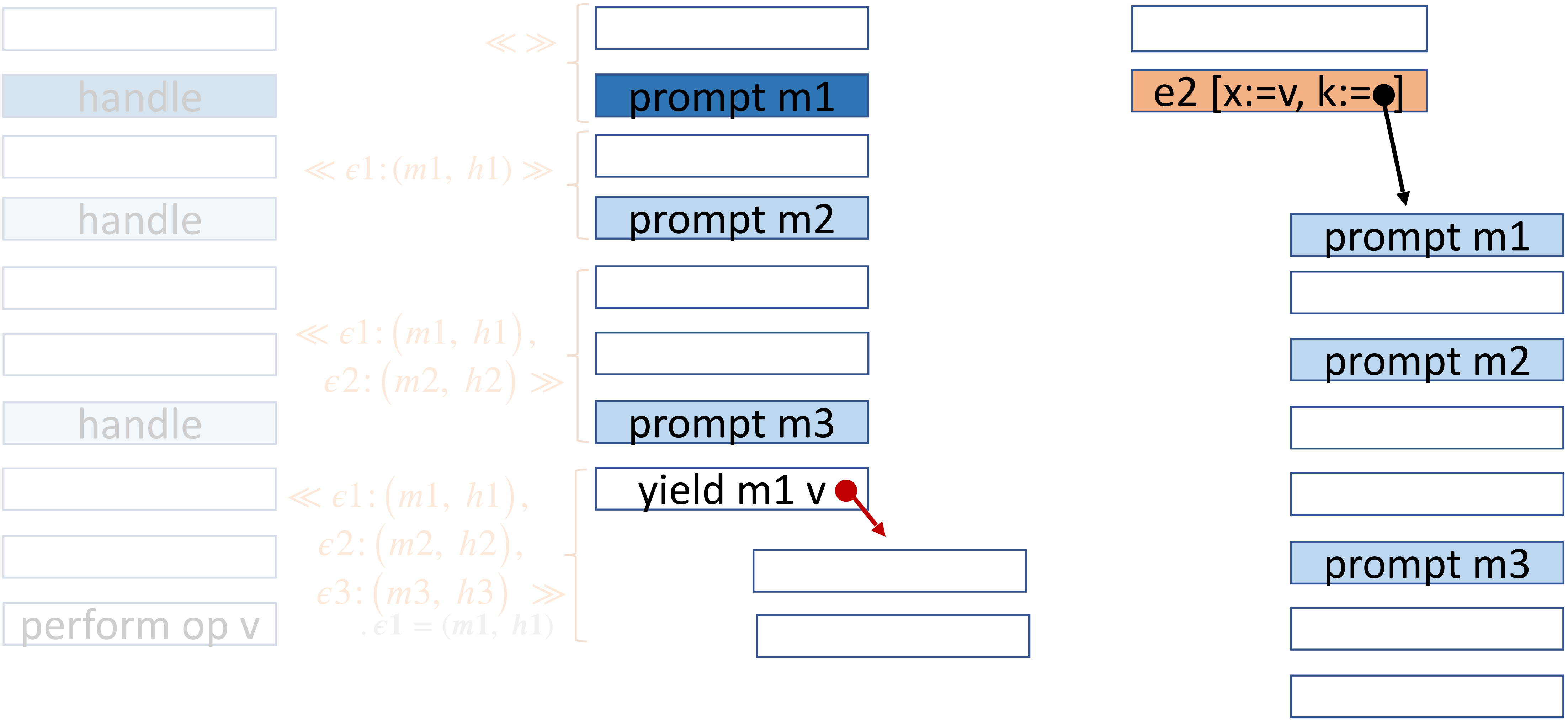
Bubbling yields

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



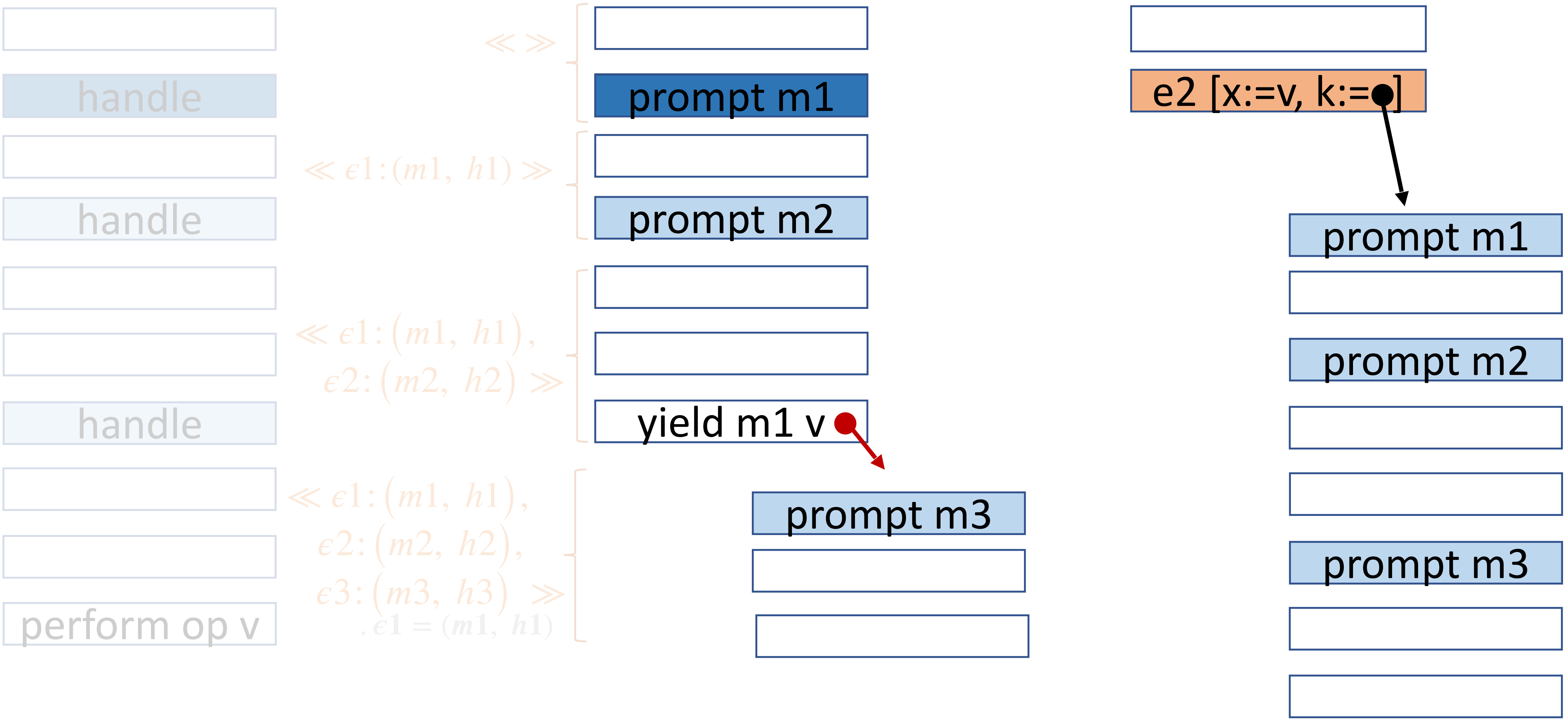
Bubbling yields

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



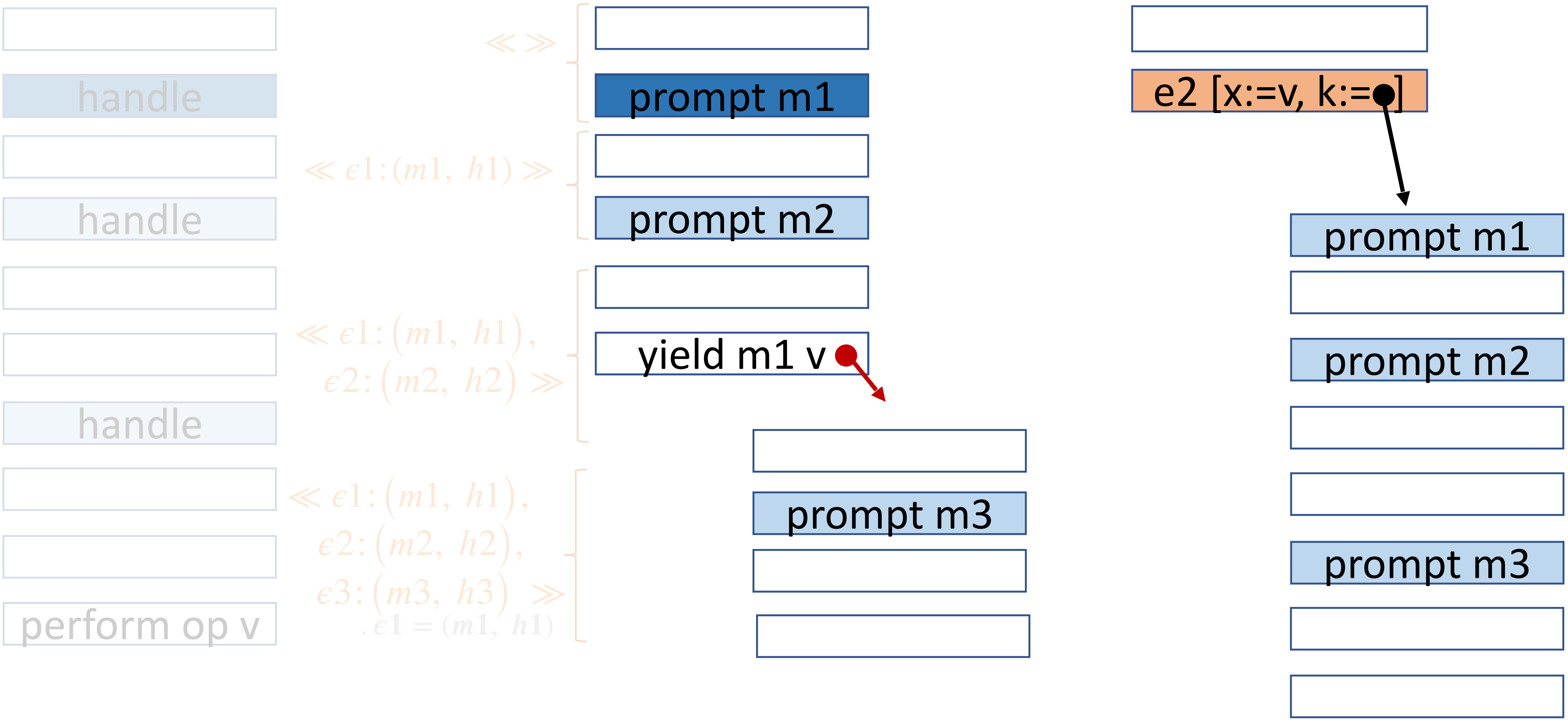
Bubbling yields

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



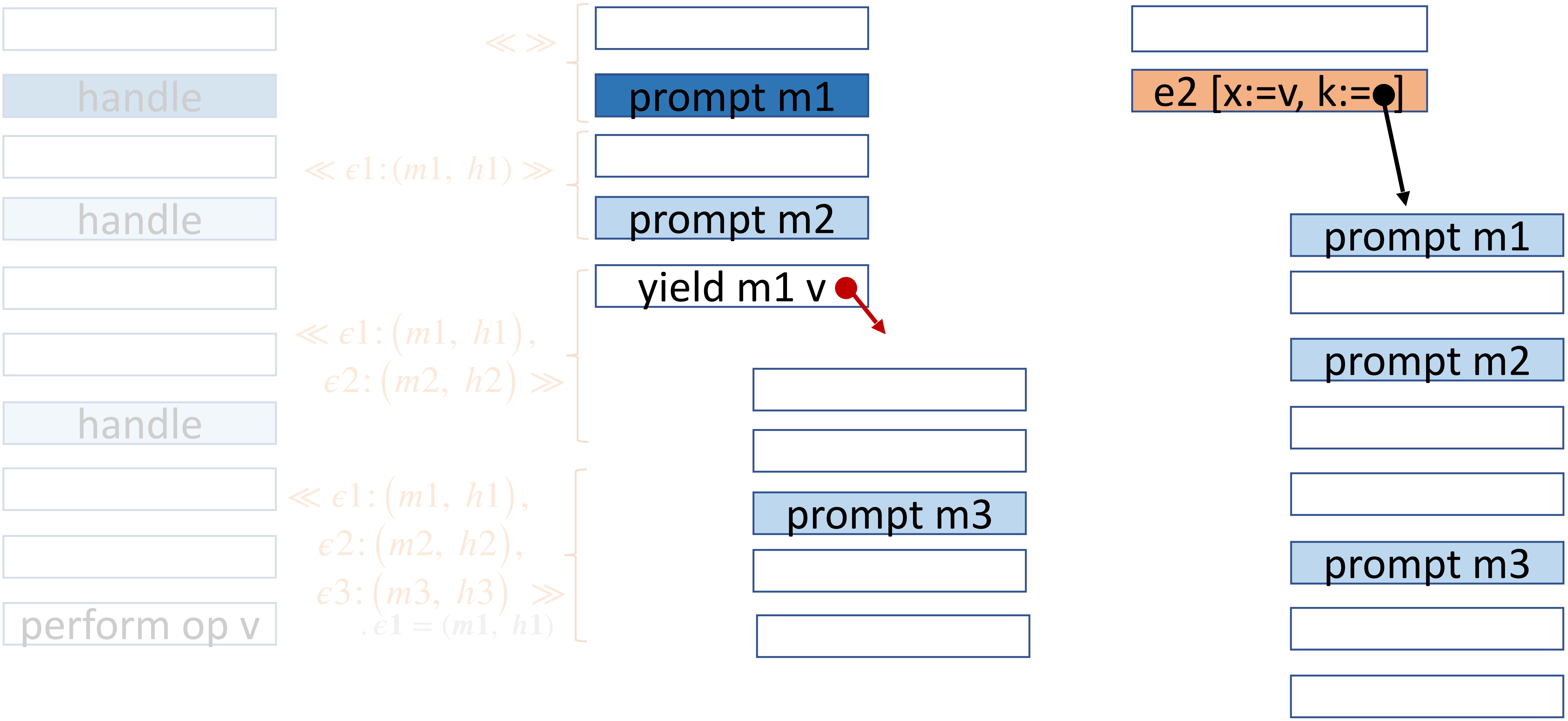
Bubbling yields

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



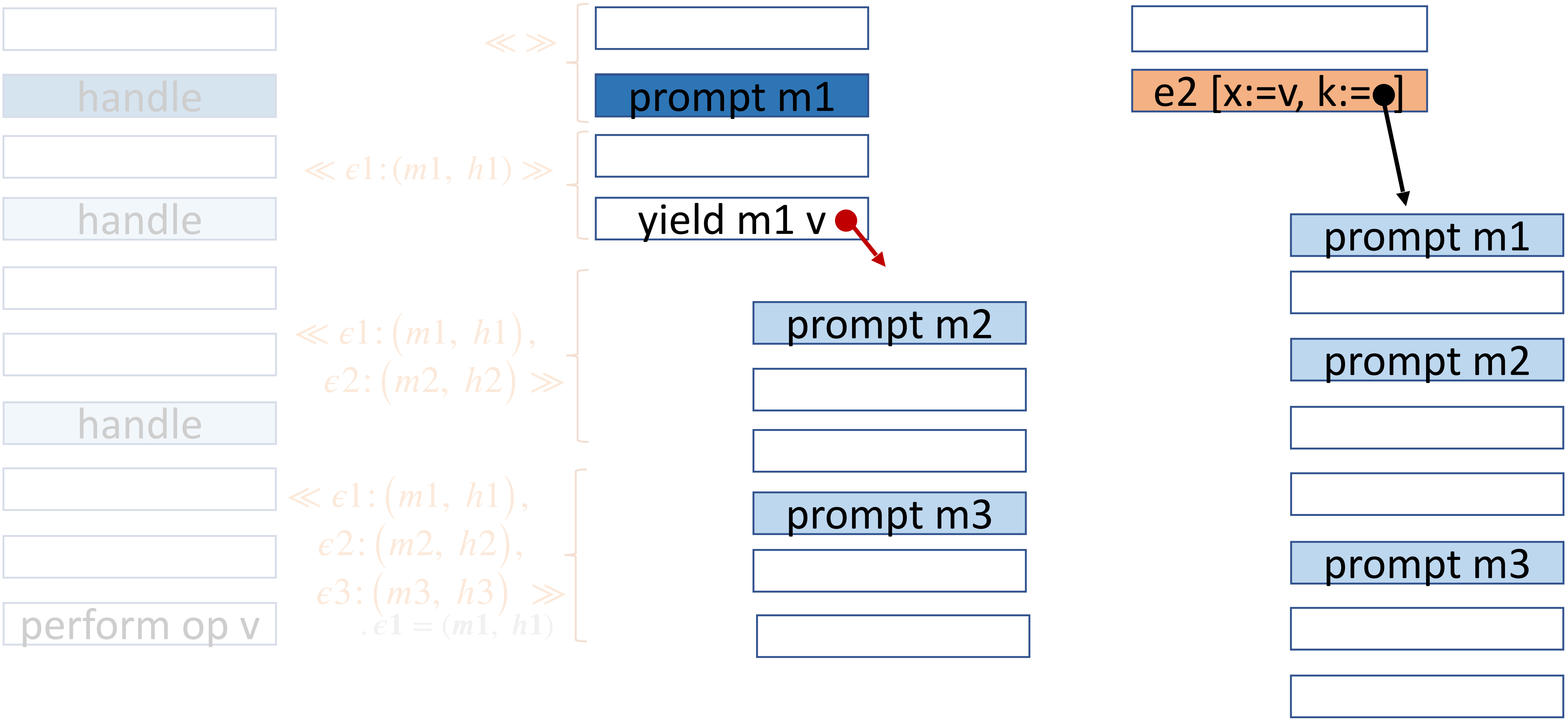
Bubbling yields

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



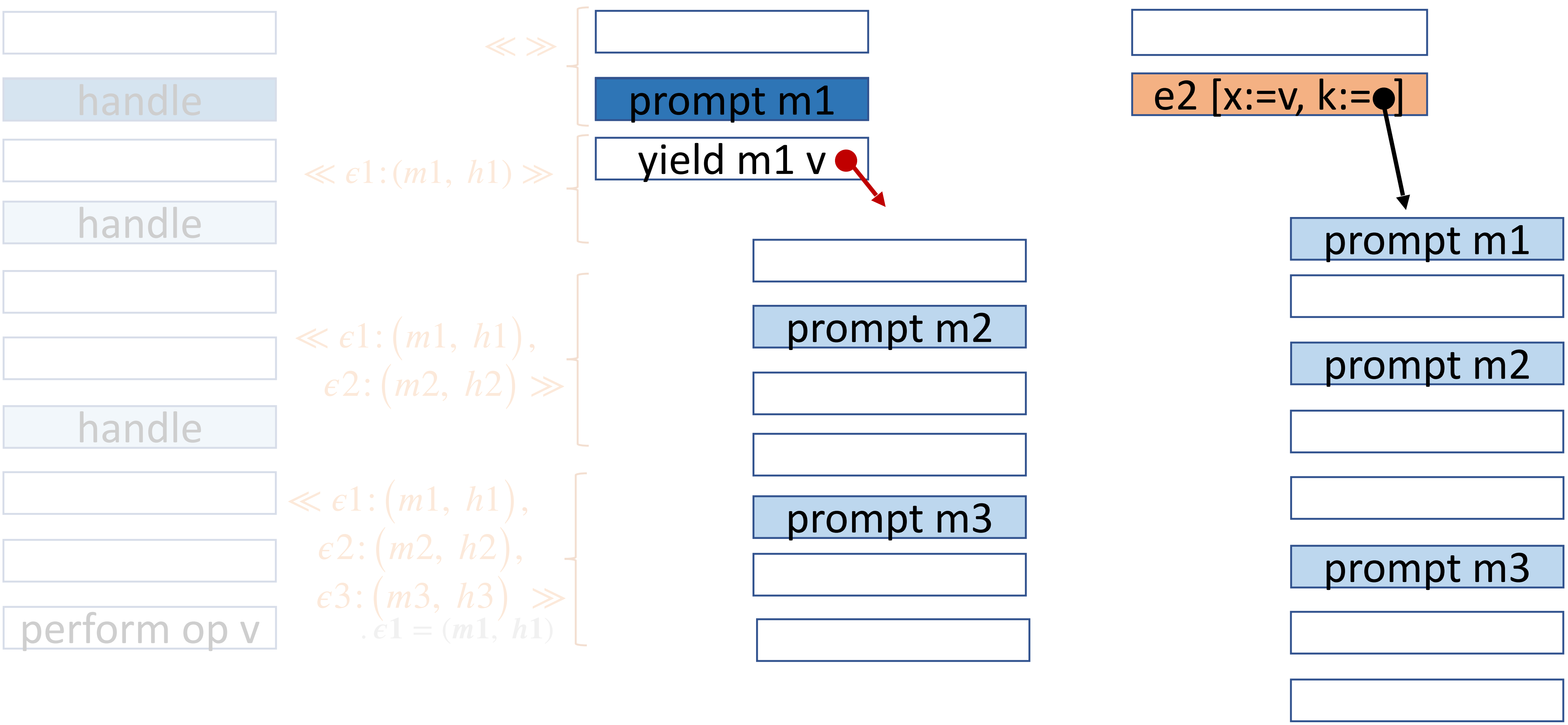
Bubbling yields

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



Bubbling yields

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



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 ())
```

$\rightsquigarrow$

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

A evidence-passing multi-prompt delimited control monad

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

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

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 ())
```

$\rightsquigarrow$

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

A evidence-passing multi-prompt delimited control monad

evidence passing

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

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

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

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

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

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

pass the result and the current evidence

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

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

control monad

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

pass the result and the current evidence

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

bubbling

Compiling to C

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

⇝

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

```
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); }
```

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); }
```

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); }
```

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); }
```

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); }
```

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

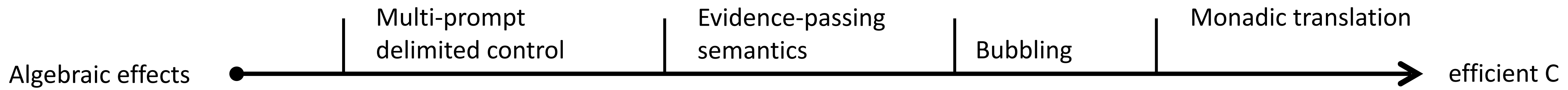
```
int expr( unit_t u, context_t* ctx) {
  int x = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; }
  int y = perform_ask( ctx→w[0], unit, ctx );
  if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
  return (x+y); }
```

control monad

bubbling

Metatheory

ICFP 2021

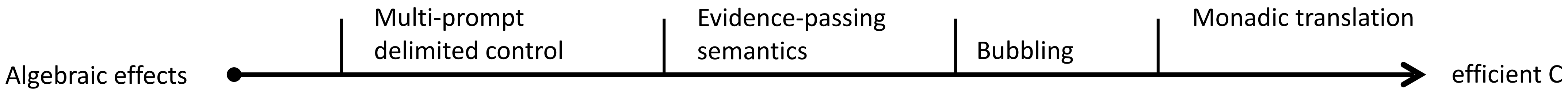


Theorem 7. (*Semantics Preserving*). Given $\varnothing \vdash e : int \mid \langle\rangle \rightsquigarrow e'$, if $e \longmapsto^* n$ in F^ϵ, then $e'\ \langle\!\langle\rangle\!\rangle \longmapsto^*$ Pure $\langle\rangle$ *int* n, in the polymorphic lambda calculus and if $e \Uparrow$ in F^ϵ, then $e'\ \langle\!\langle\rangle\!\rangle \Uparrow$ in the polymorphic lambda calculus.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\varnothing \vdash e : \sigma \mid \epsilon$, then $e \cong_{ctx} e$.

Metatheory

ICFP 2021

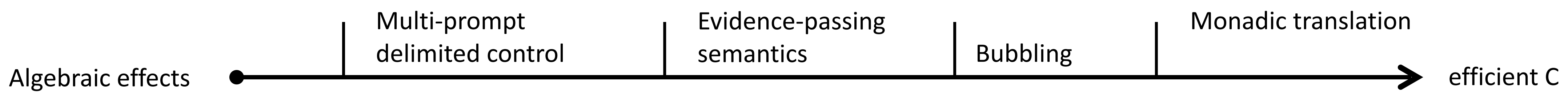


Theorem 7. (*Semantics Preserving*). Given $\varnothing \vdash e : int \mid \langle\rangle \rightsquigarrow e'$, if $e \longmapsto^* n$ in F^{ϵ}, then $e' \, \langle\!\langle\rangle\!\rangle \longmapsto^*$ Pure $\langle\rangle$ int n, in the polymorphic lambda calculus and if $e \Uparrow$ in F^{ϵ}, then $e' \, \langle\!\langle\rangle\!\rangle \Uparrow$ in the polymorphic lambda calculus.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\varnothing \vdash e : \sigma \mid \epsilon$, then $e \cong_{ctx} e$.

Metatheory

ICFP 2021

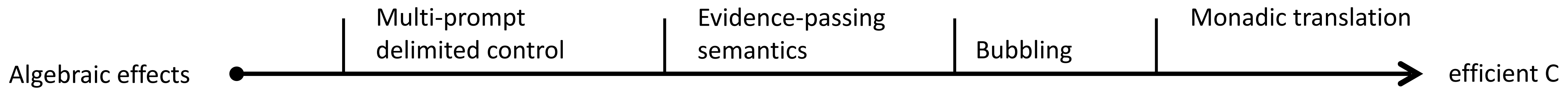


Theorem 7. (*Semantics Preserving*). Given $\varnothing \vdash e : int \mid \langle\rangle \rightsquigarrow e'$, if $e \longmapsto^* n$ in F^ϵ, then $e' \langle\!\langle\rangle\!\rangle \longmapsto^*$ Pure $\langle\rangle$ $int\ n$, in the polymorphic lambda calculus and if $e \Uparrow$ in F^ϵ, then $e' \langle\!\langle\rangle\!\rangle \Uparrow$ in the polymorphic lambda calculus.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\varnothing \vdash e : \sigma \mid \epsilon$, then $e \cong_{ctx} e$.

Metatheory

ICFP 2021

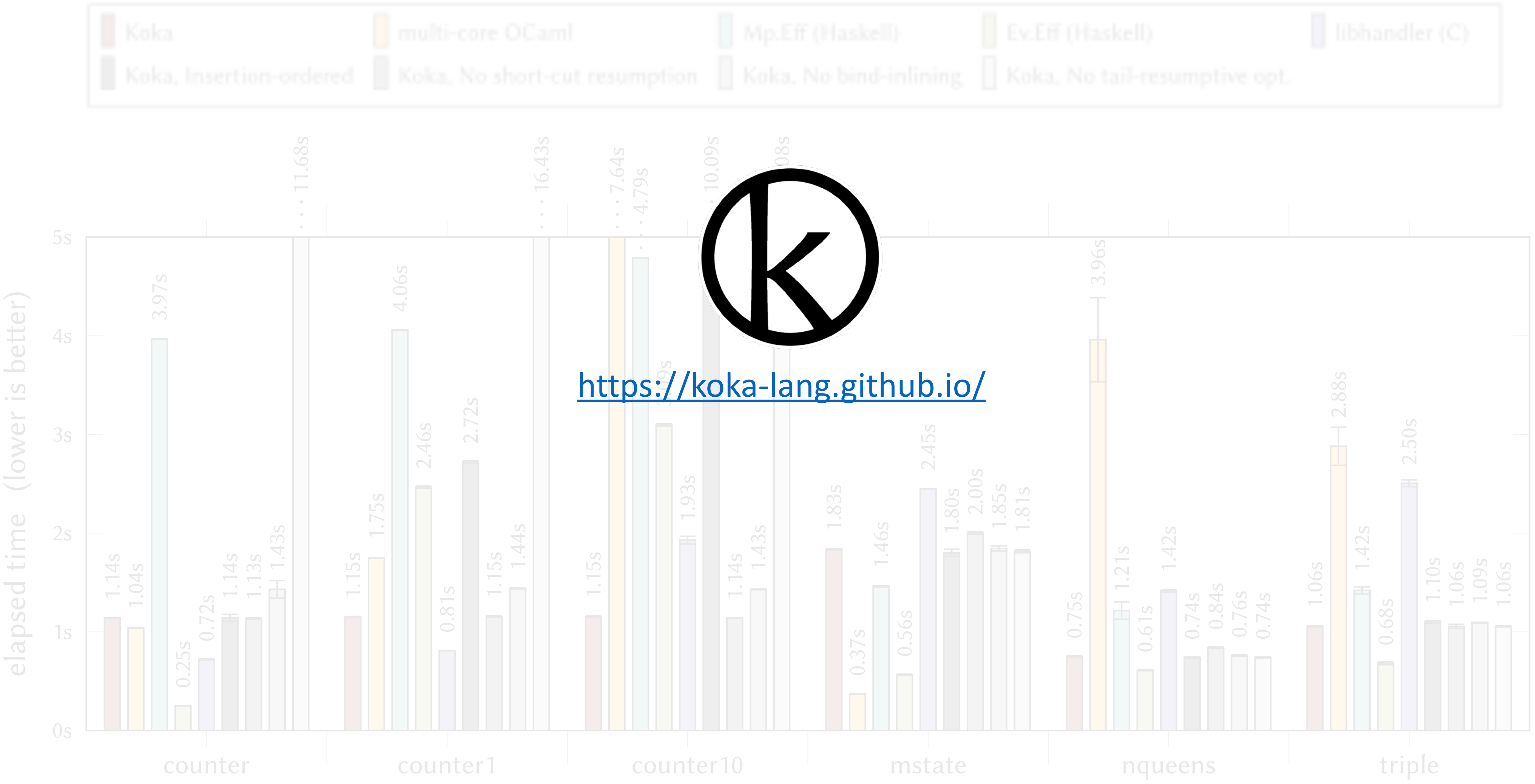


Theorem 7. (*Semantics Preserving*). Given $\varnothing \vdash e : int \mid \langle\rangle \rightsquigarrow e'$, if $e \longmapsto^* n$ in F^ϵ, then $e' \, \langle\!\langle\rangle\!\rangle \longmapsto^*$ Pure $\langle\rangle$ int n, in the polymorphic lambda calculus and if $e \Uparrow$ in F^ϵ, then $e' \, \langle\!\langle\rangle\!\rangle \Uparrow$ in the polymorphic lambda calculus.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\varnothing \vdash e : \sigma \mid \epsilon$, then $e \cong_{ctx} e$.

Benchmarks

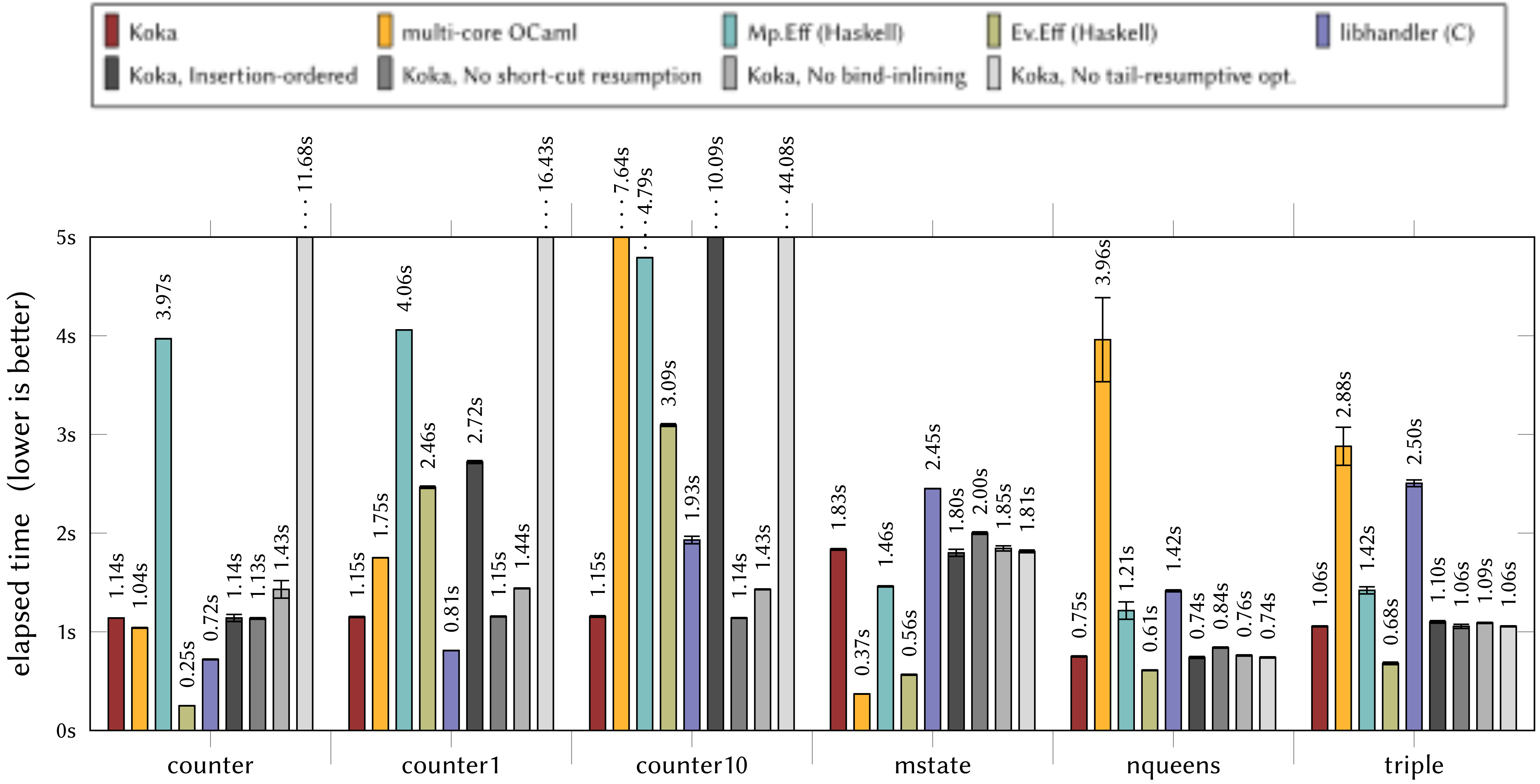
ICFP 2021



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

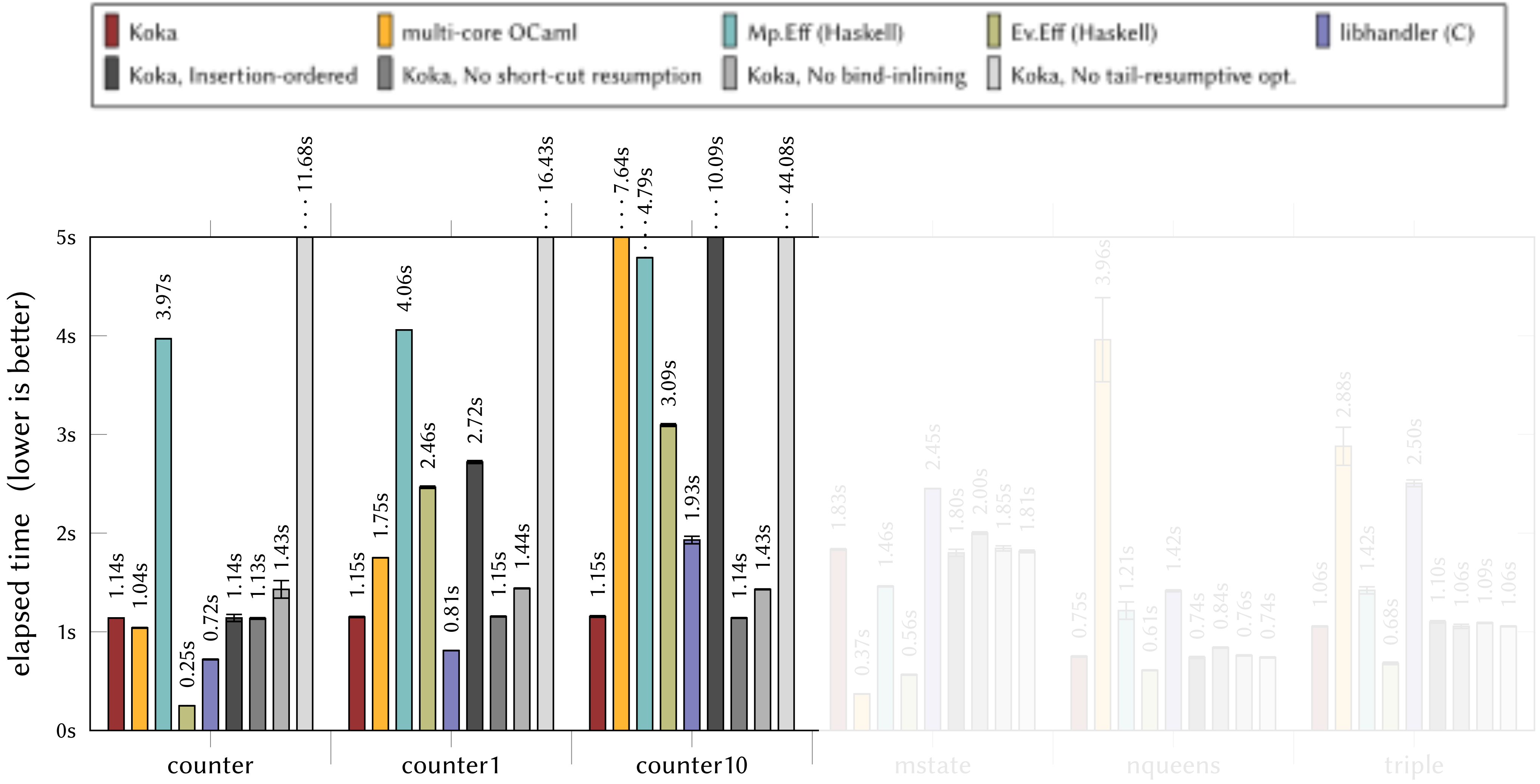
Benchmarks

ICFP 2021



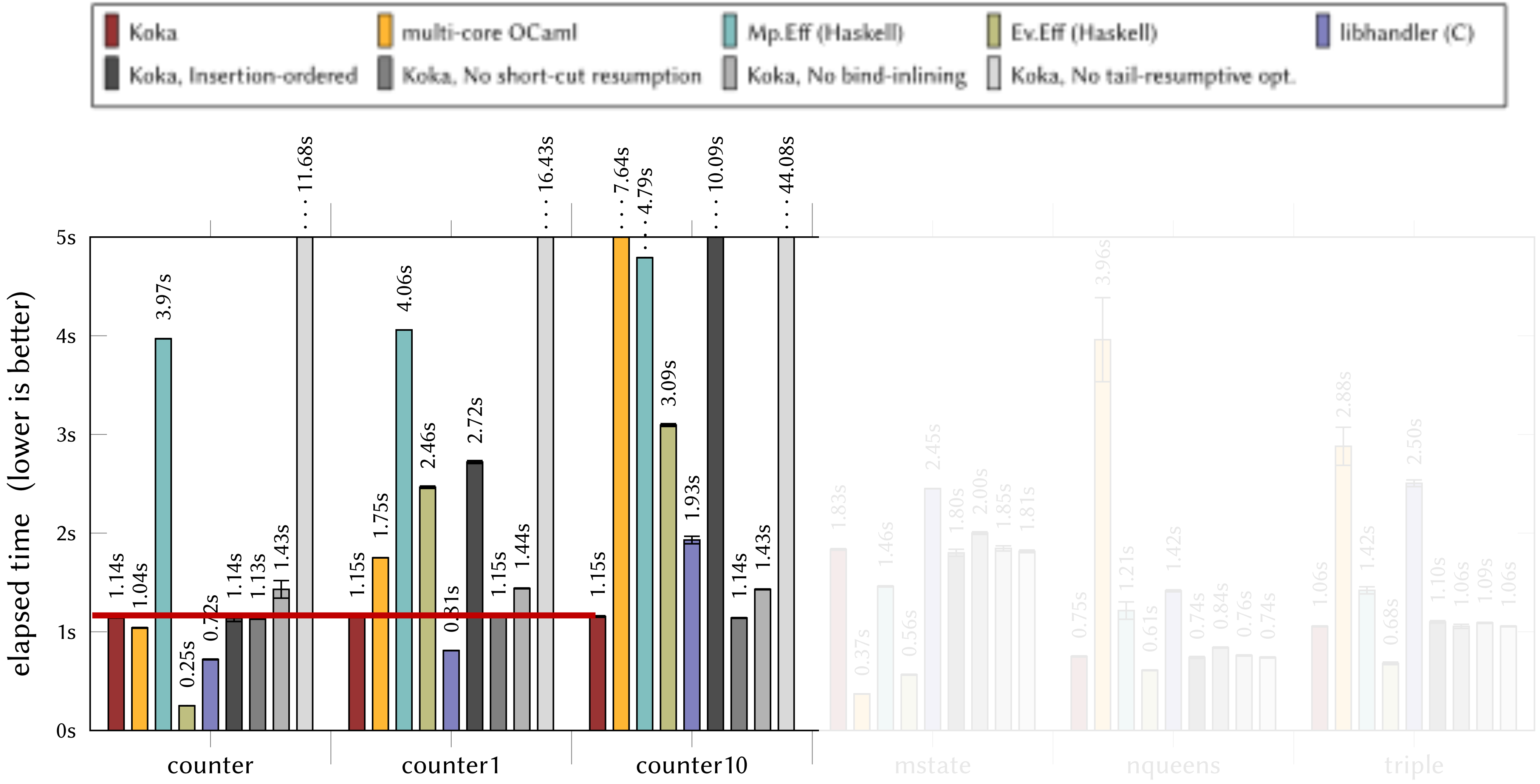
Benchmarks

ICFP 2021



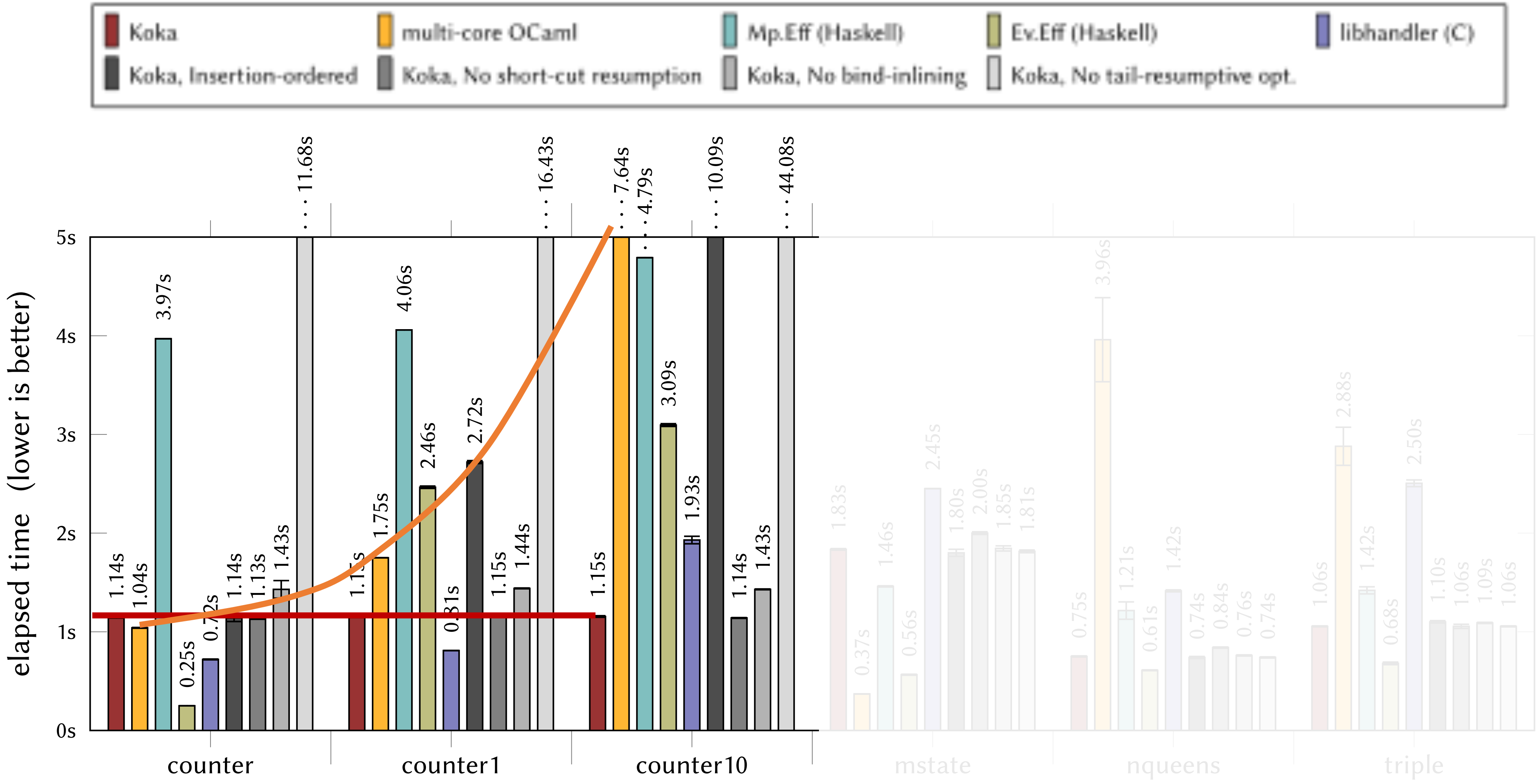
Benchmarks

ICFP 2021



Benchmarks

ICFP 2021



Take-aways

Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

Algebraic effects and handlers: composable and modular computational effects

Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

   Algebraic effects and handlers: composable and modular computational effects

3. Can we implement algebraic effects efficiently?

Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

Algebraic effects and handlers: composable and modular computational effects

3. Can we implement algebraic effects efficiently?

Evidence-passing semantics

Take-aways

April 22 – 27 , 2018, Dagstuhl Seminar 18172

Algebraic Effect Handlers go Mainstream

Organizers

Sivaramakrishnan Krishnamoorthy Chandrasekaran (University of Cambridge, GB)

Daan Leijen (Microsoft Research – Redmond, US)

Matija Pretnar (University of Ljubljana, SI)

Tom Schrijvers (KU Leuven, BE)

Efficient Compilation of Algebraic Effect Handlers

Ningning Xie

UNIVERSITY OF CAMBRIDGE