Lexical Effect Handlers: Fast by Design, Correct by Proof

Cong Ma
University of Waterloo

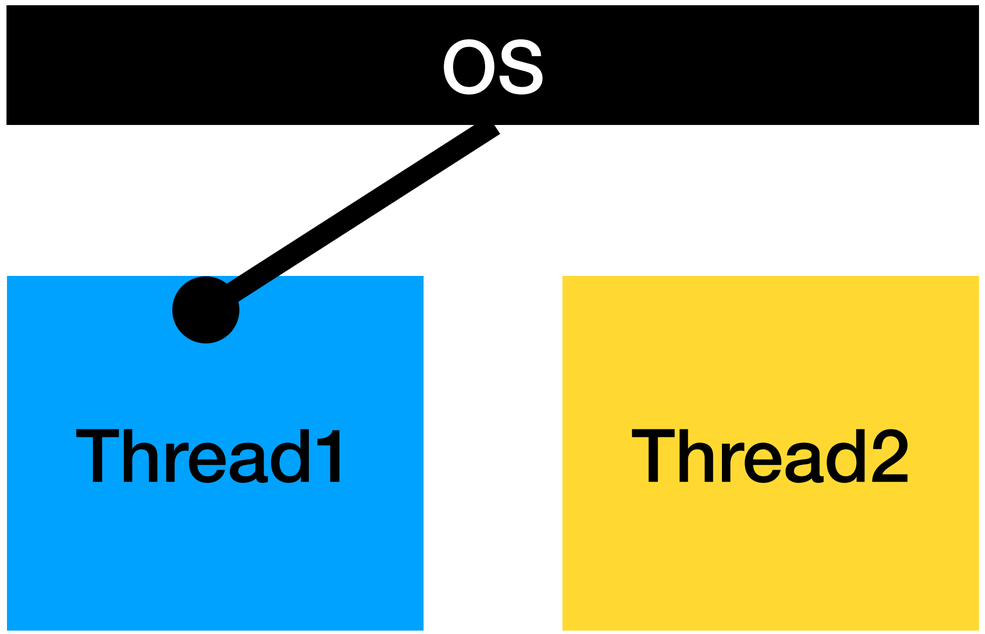
Effect handler

A powerful language feature

Effect handlers subsume many control flow features: async/await, coroutine, generator…

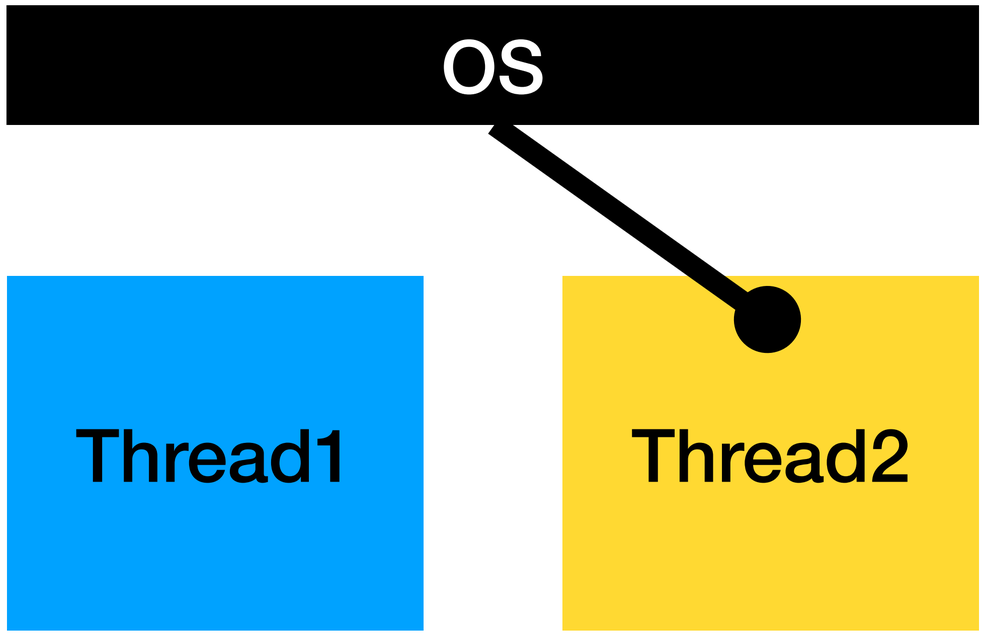
Effect handler

Scheduler inside OS



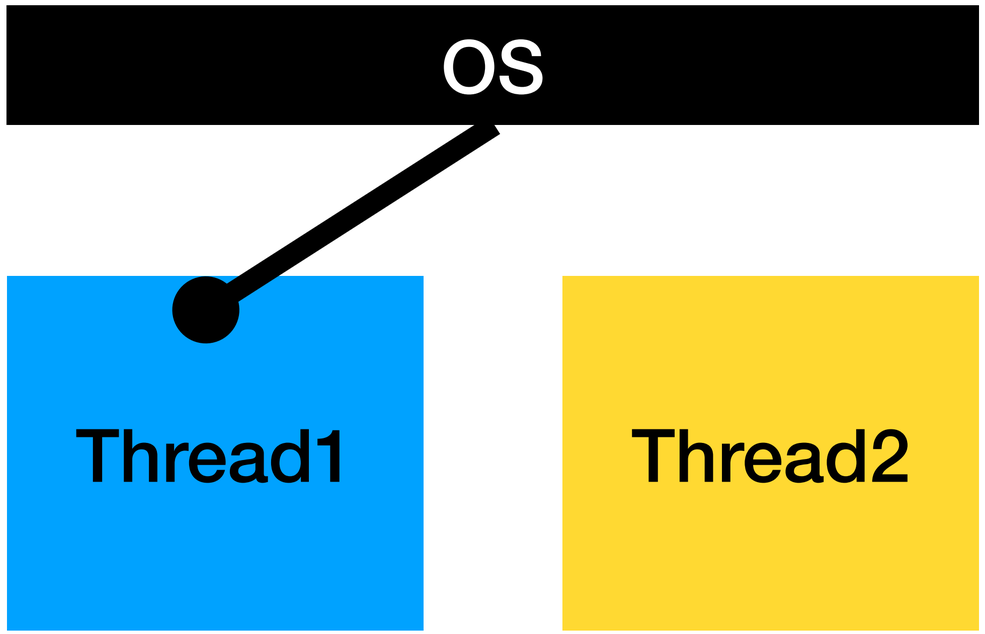
Effect handler

Scheduler inside OS



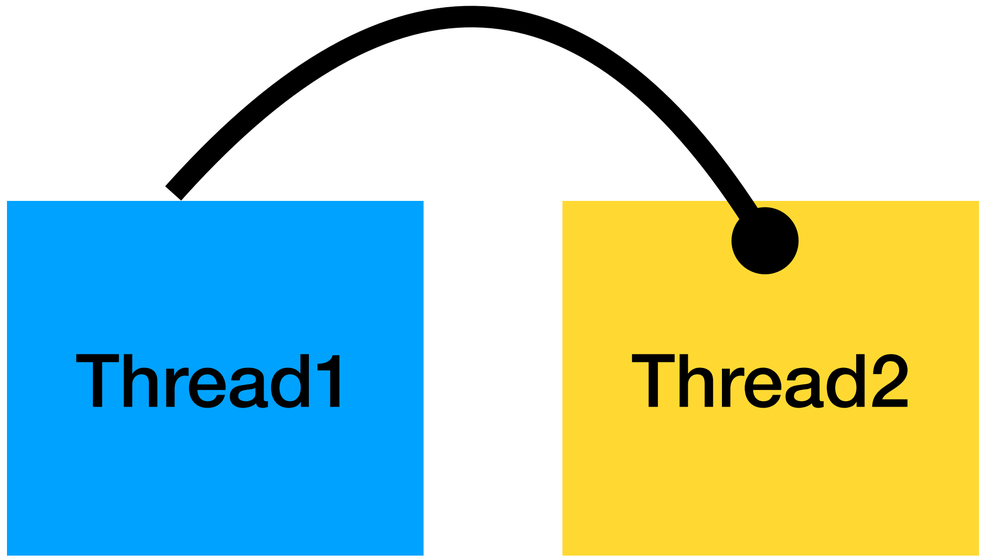
Effect handler

Scheduler inside OS



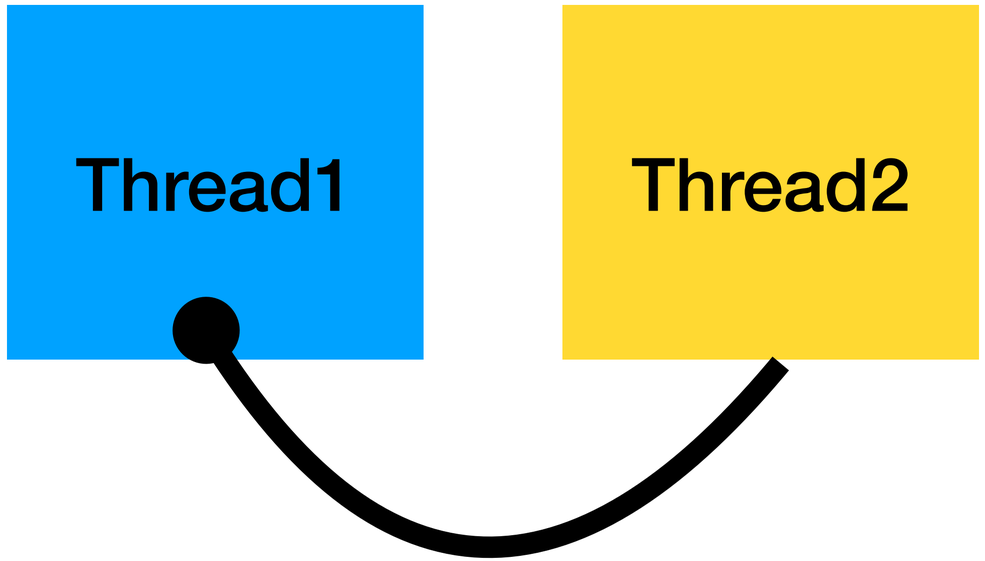
Effect handler

User-level scheduling using effect handler



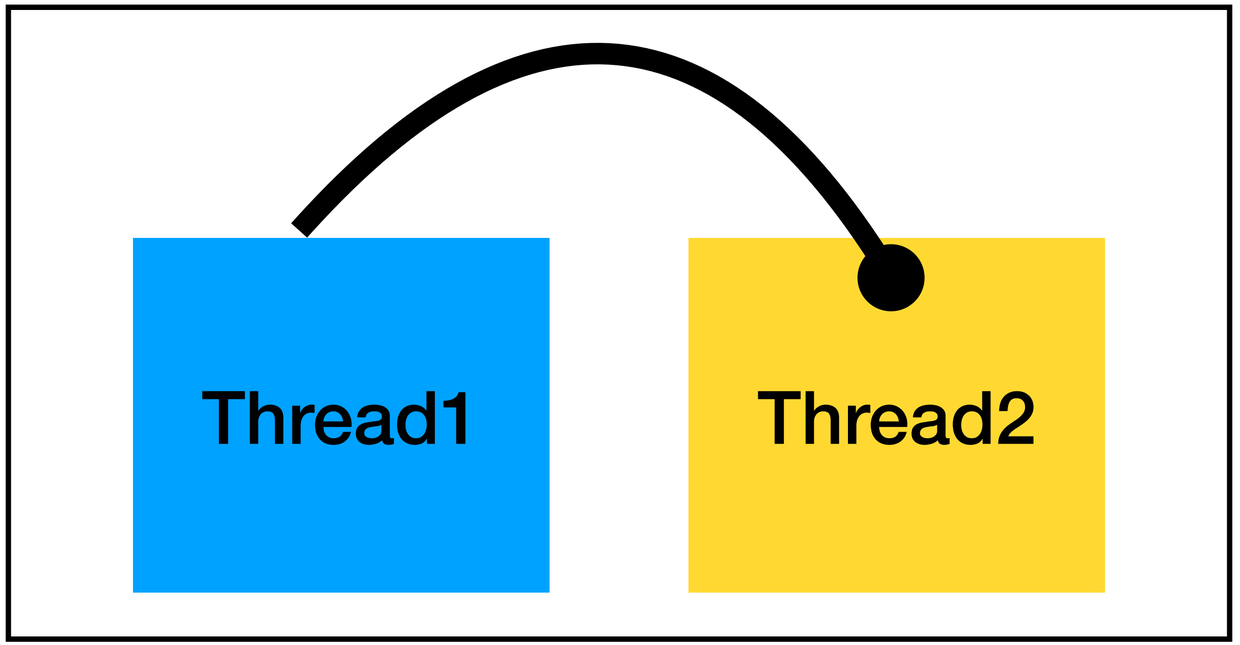
Effect handler

User-level scheduling using effect handler



Effect handler

User-level scheduling using effect handler

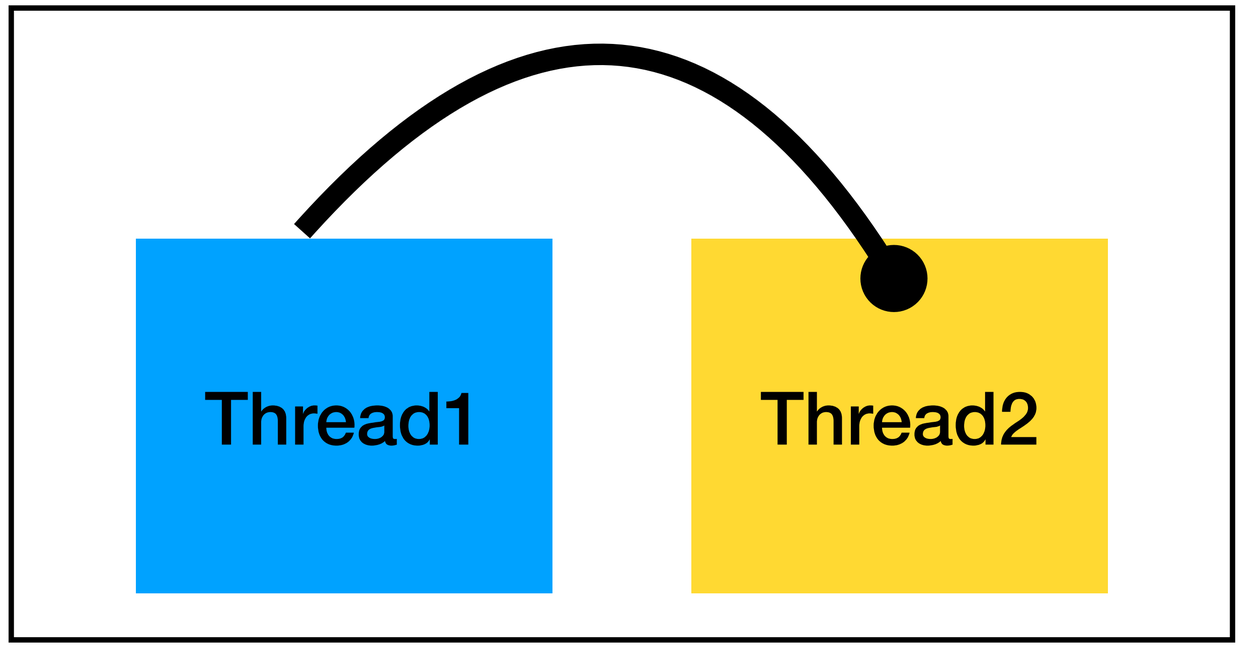


Within the programming language, without runtime support

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

Common language features:

```
val
def
while
if…then…
```

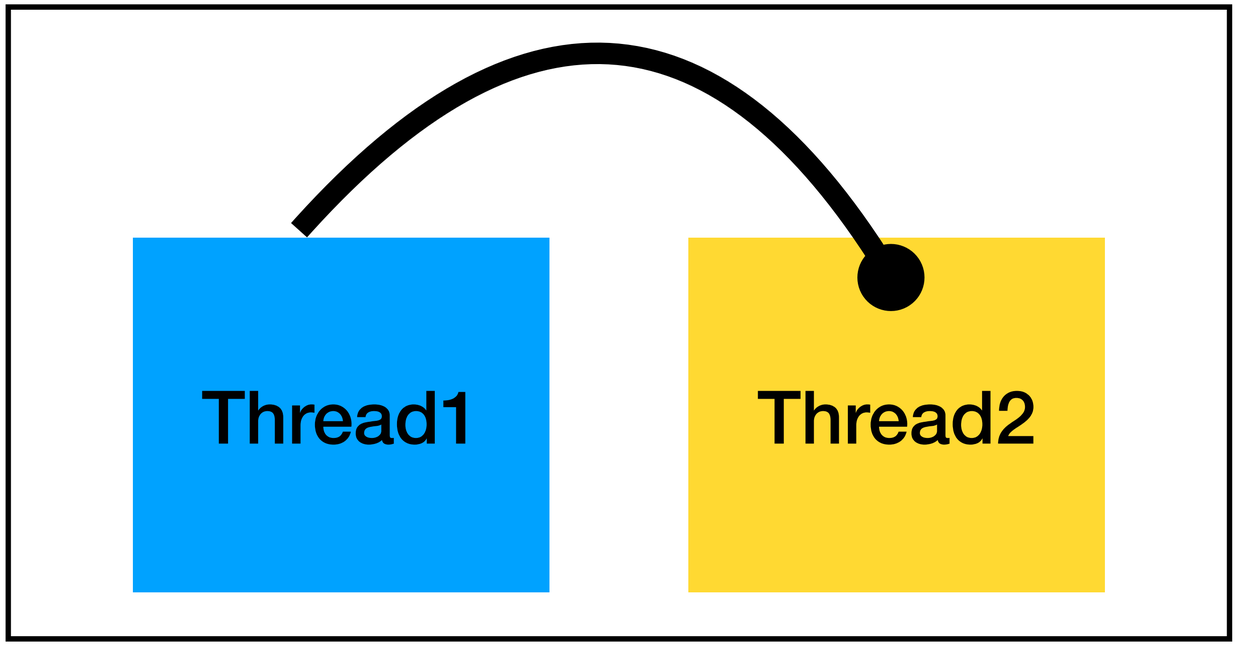
New language features:

```
handle…with…
raise
resume
```

Effect handler

A powerful language feature

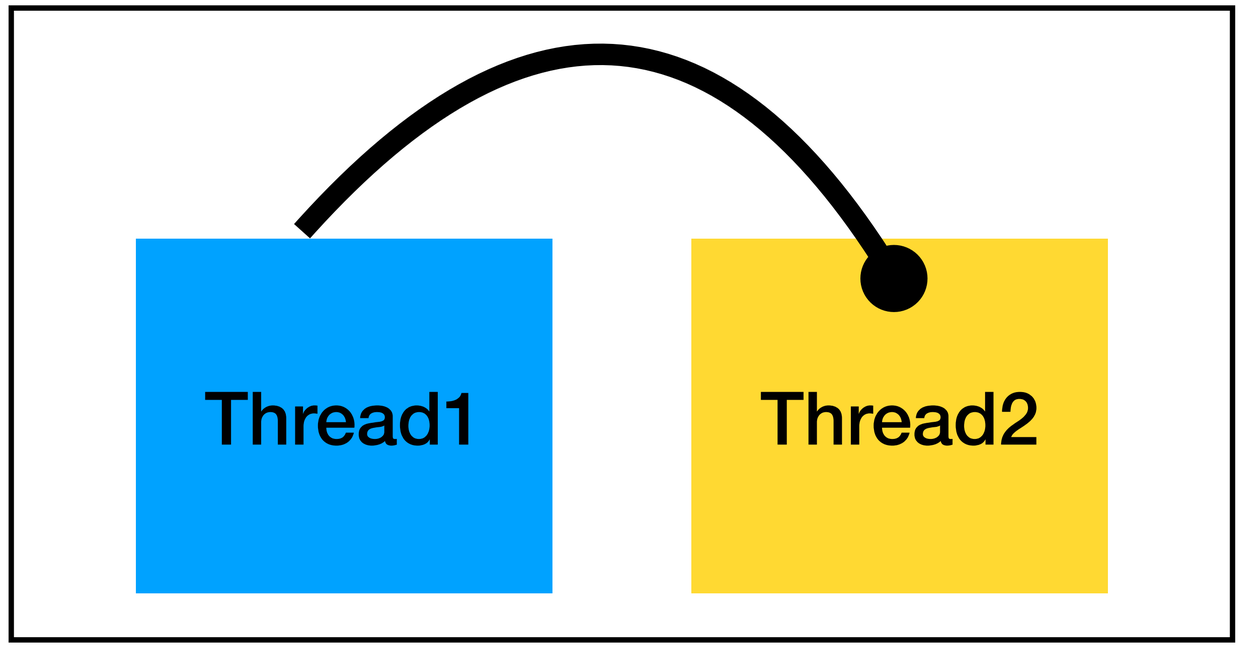
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

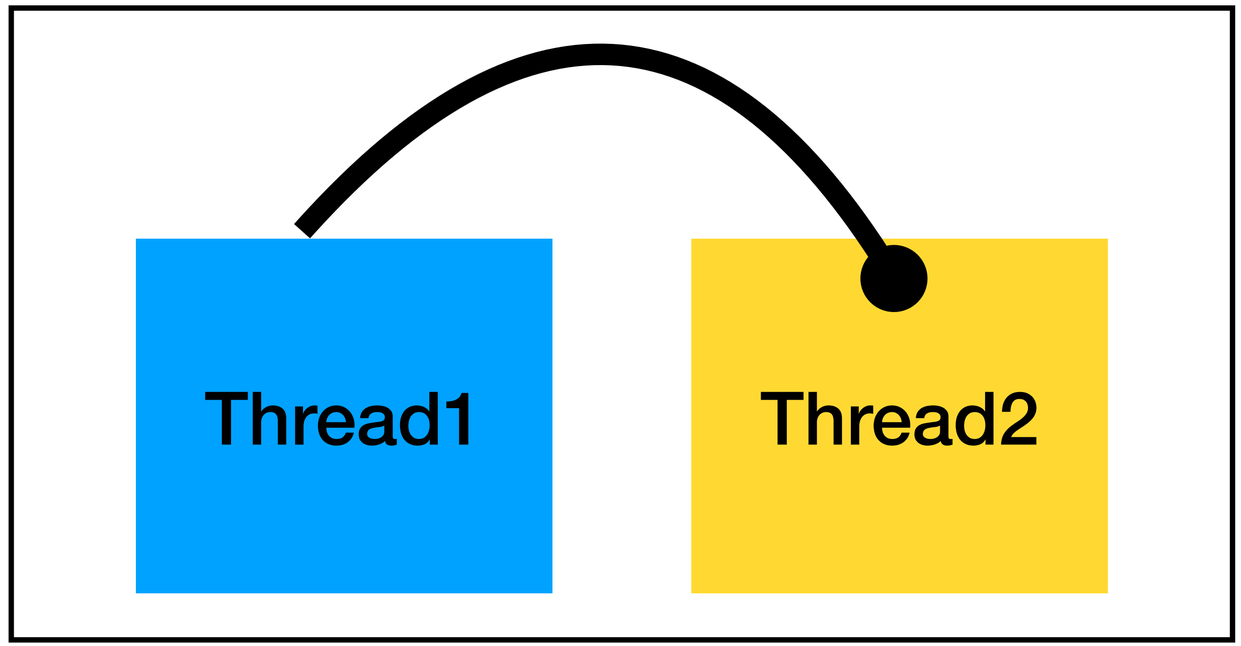
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

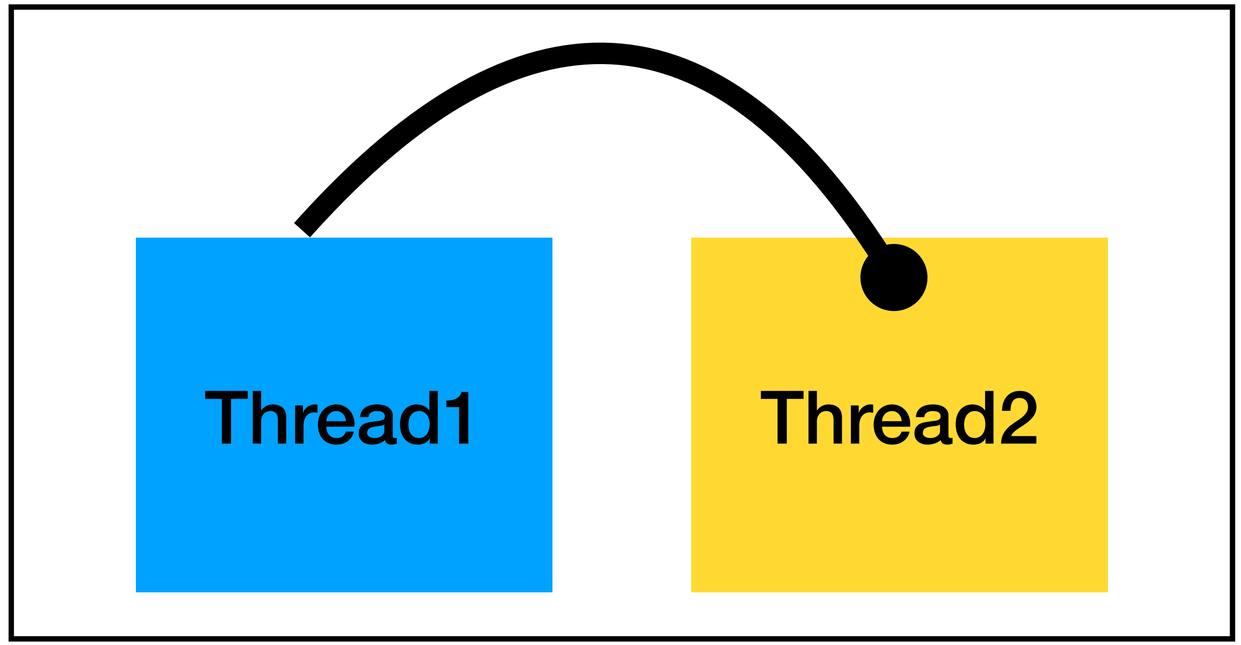
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

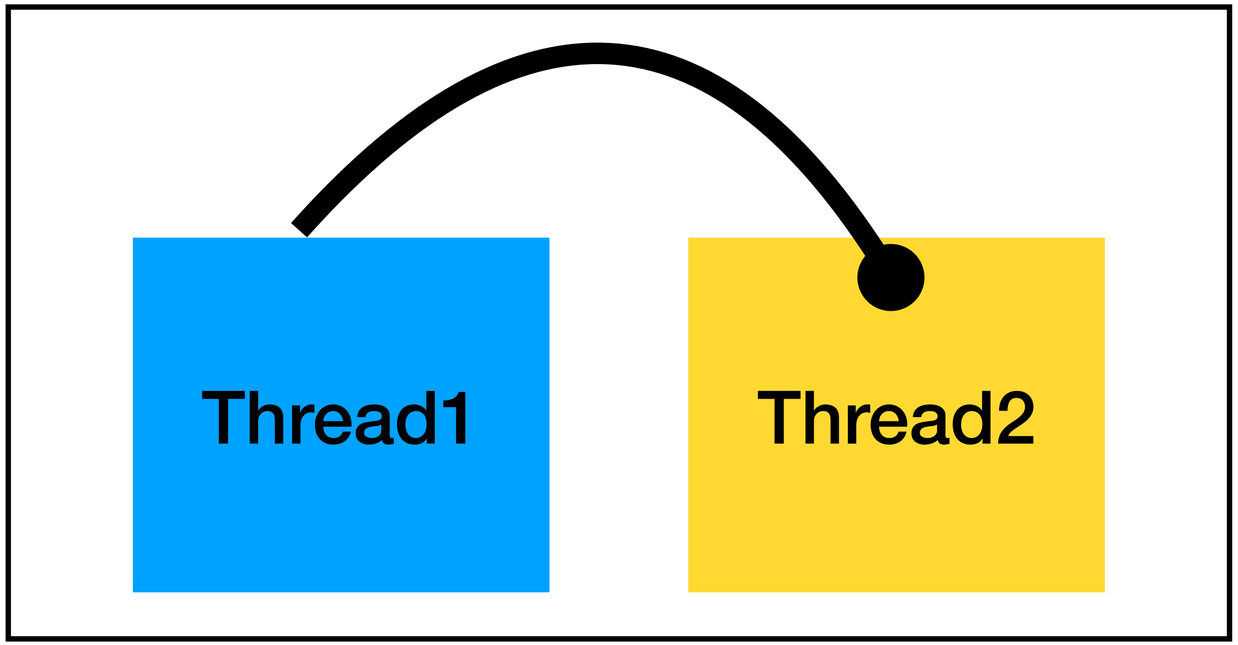
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Background

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

Processor:

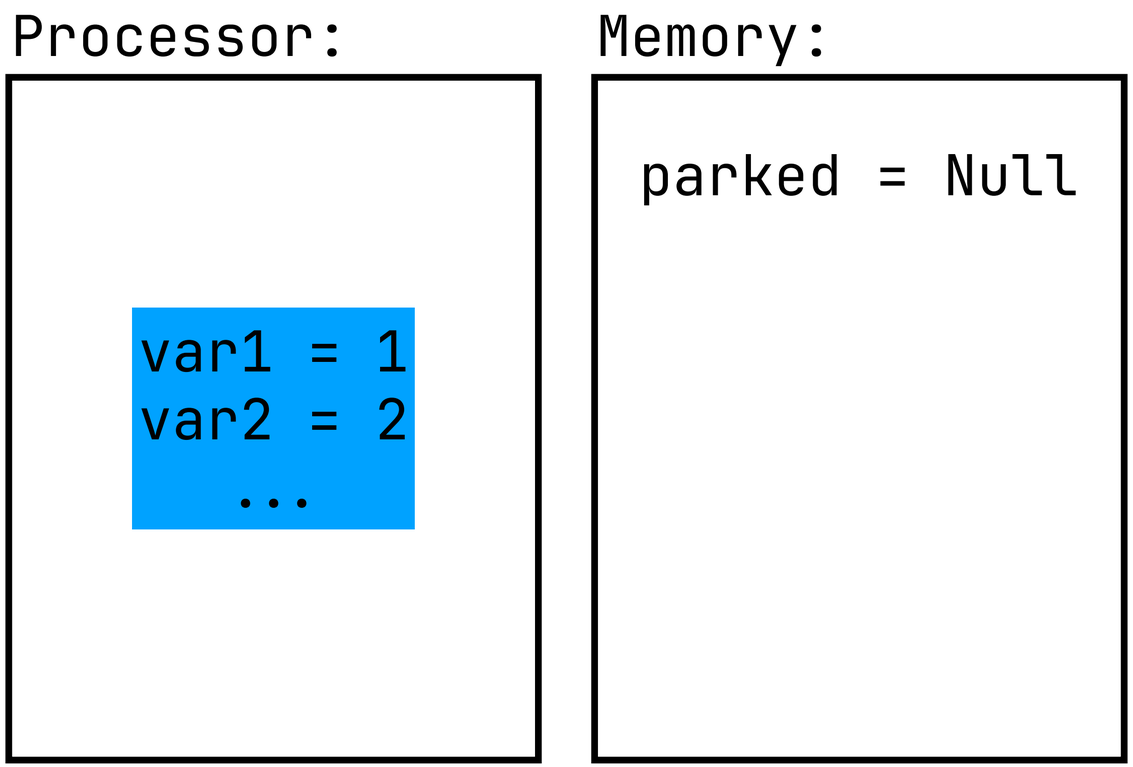
Memory:

parked = Null

Effect handler

A powerful language feature

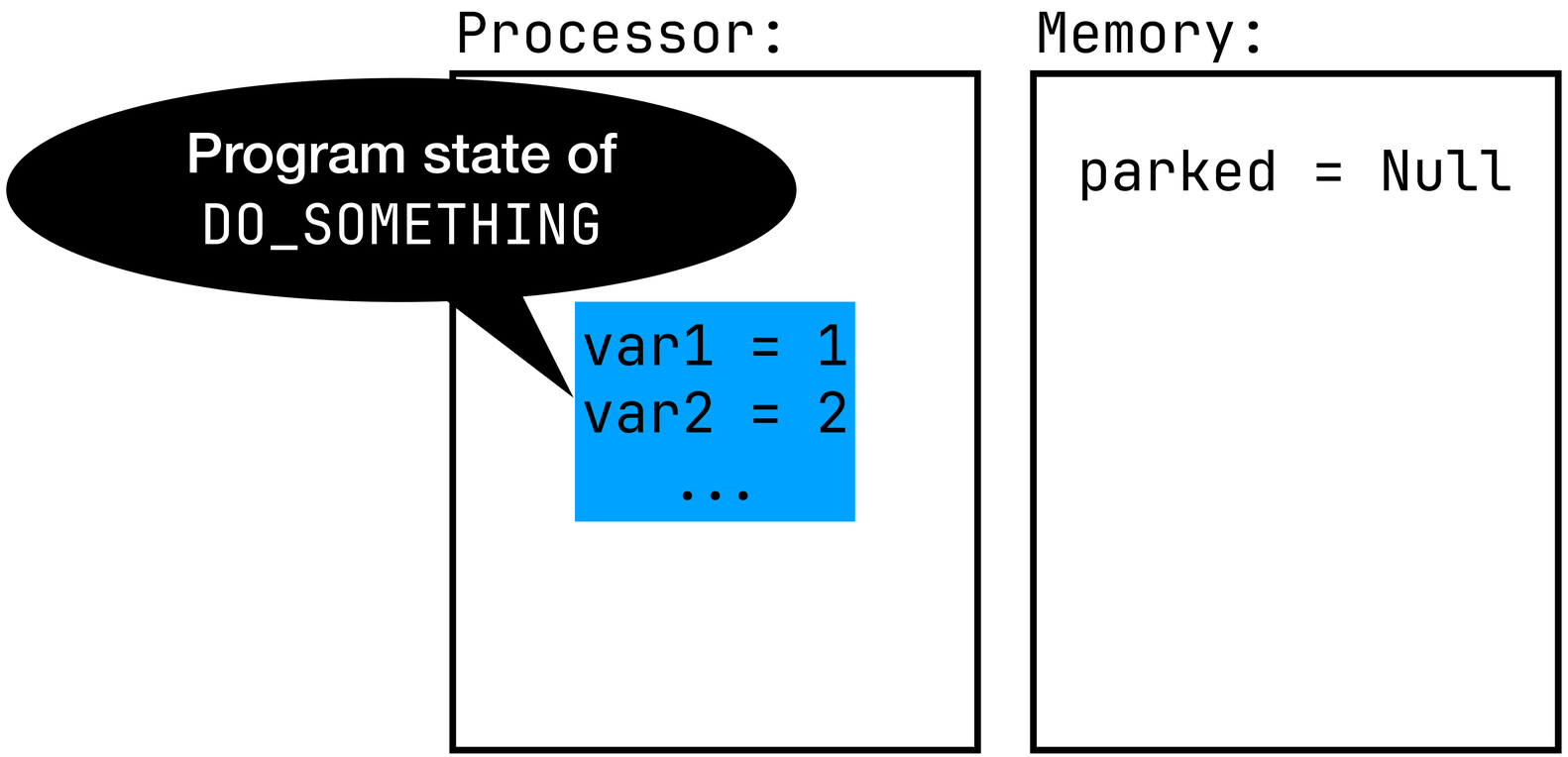
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

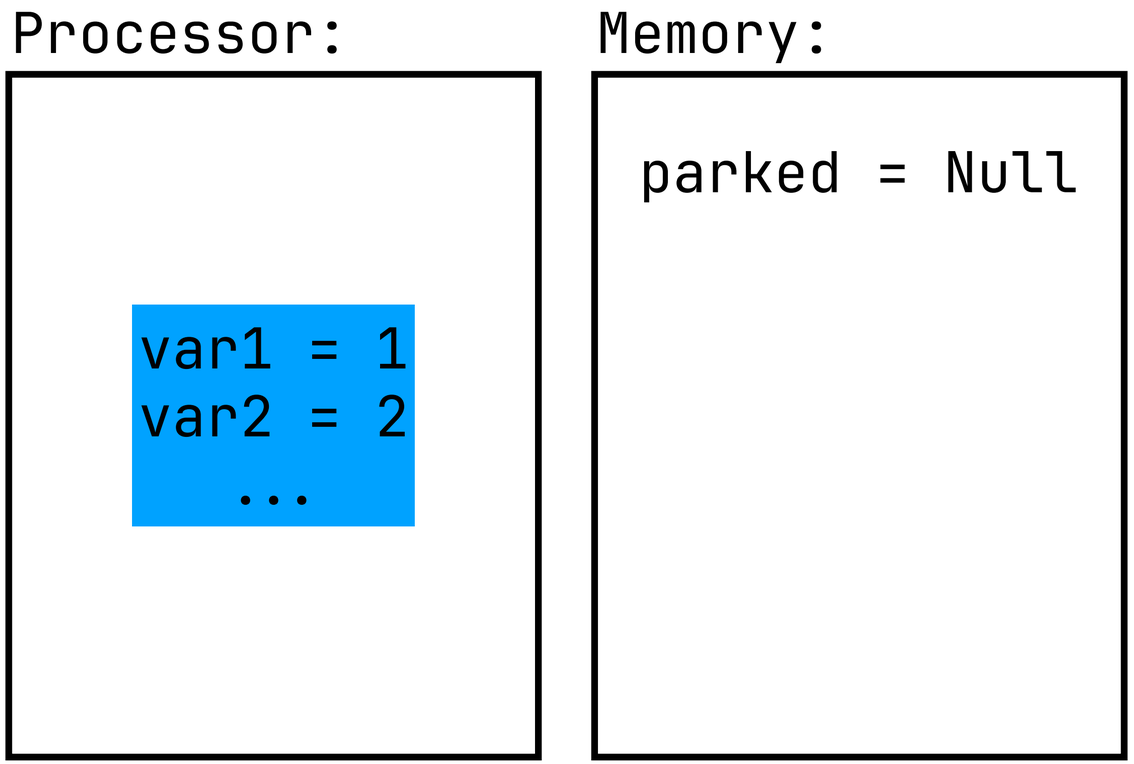
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

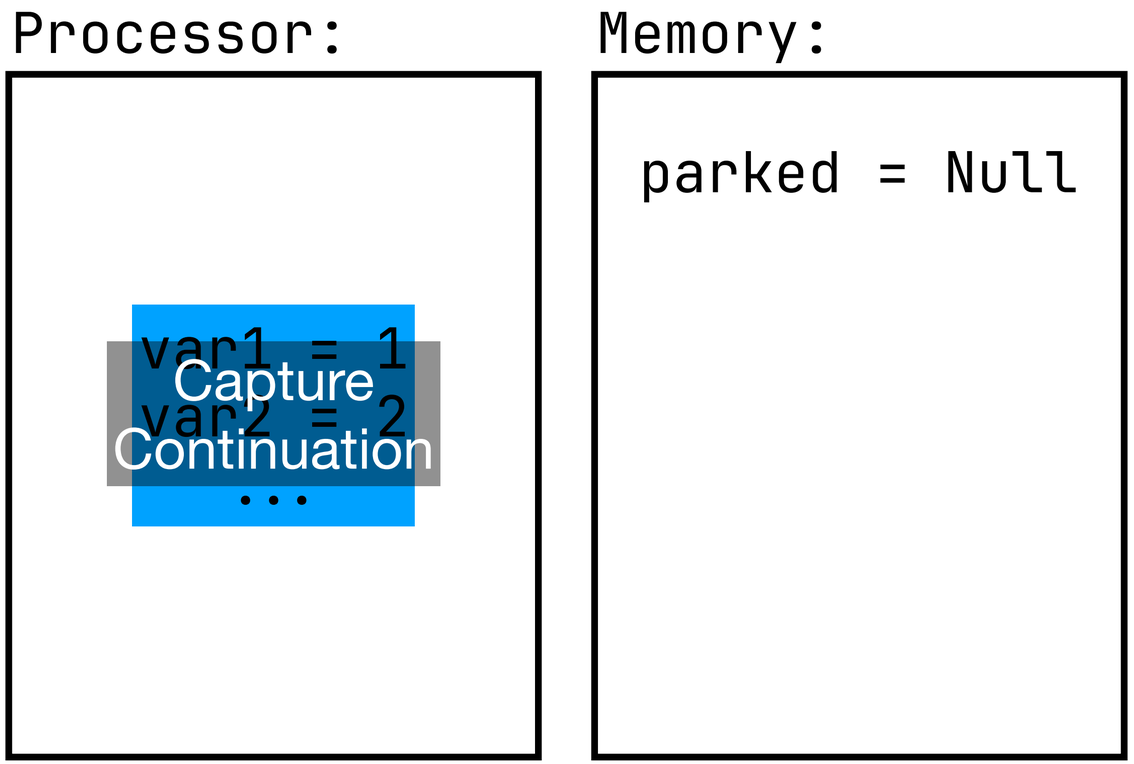
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

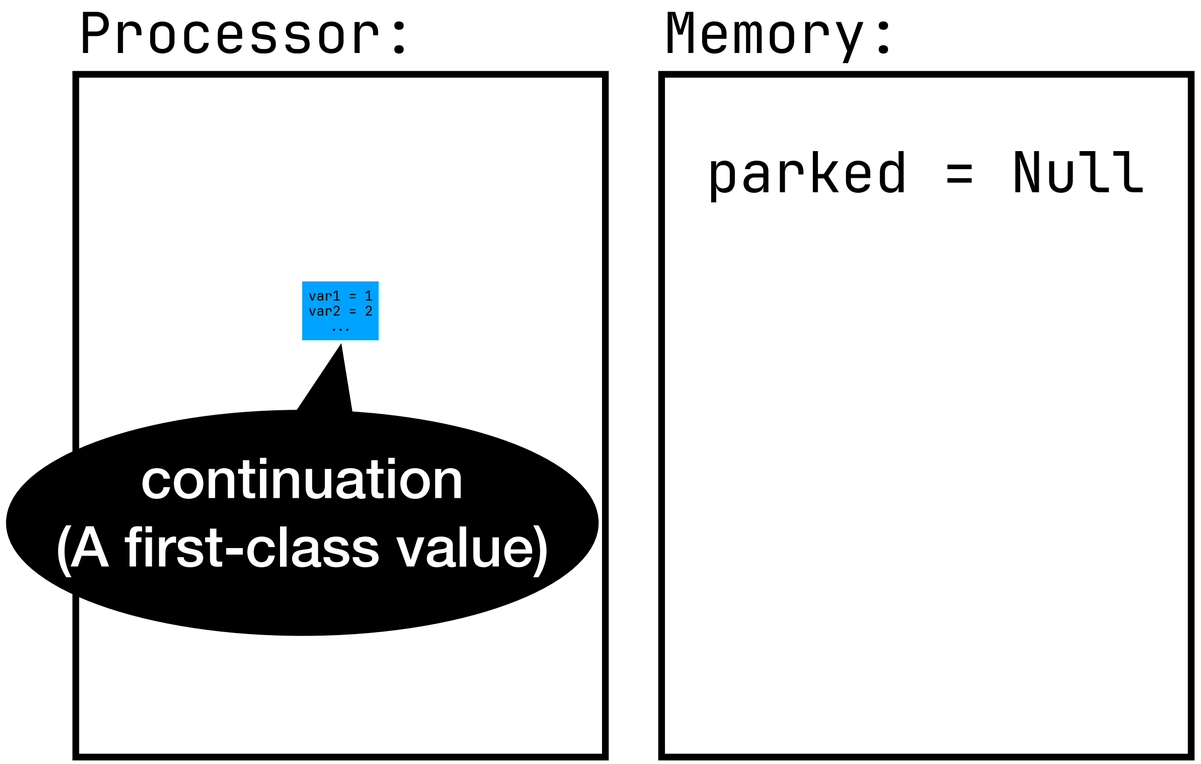
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

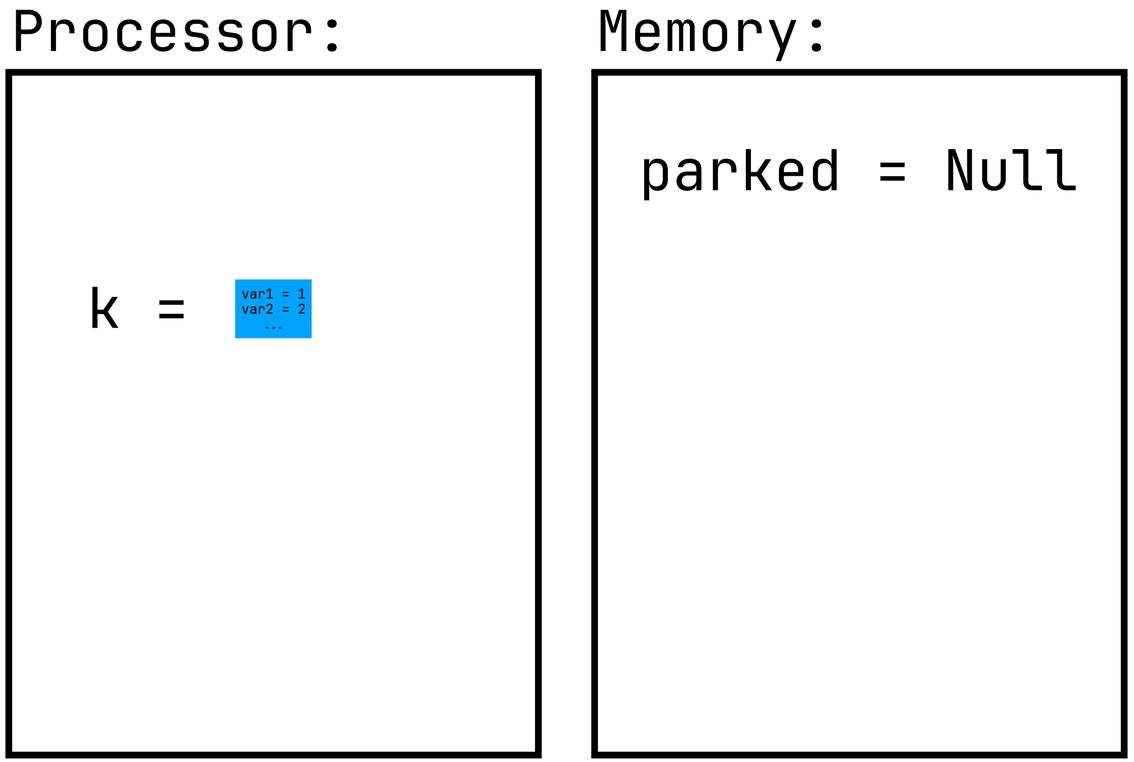
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) =>
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

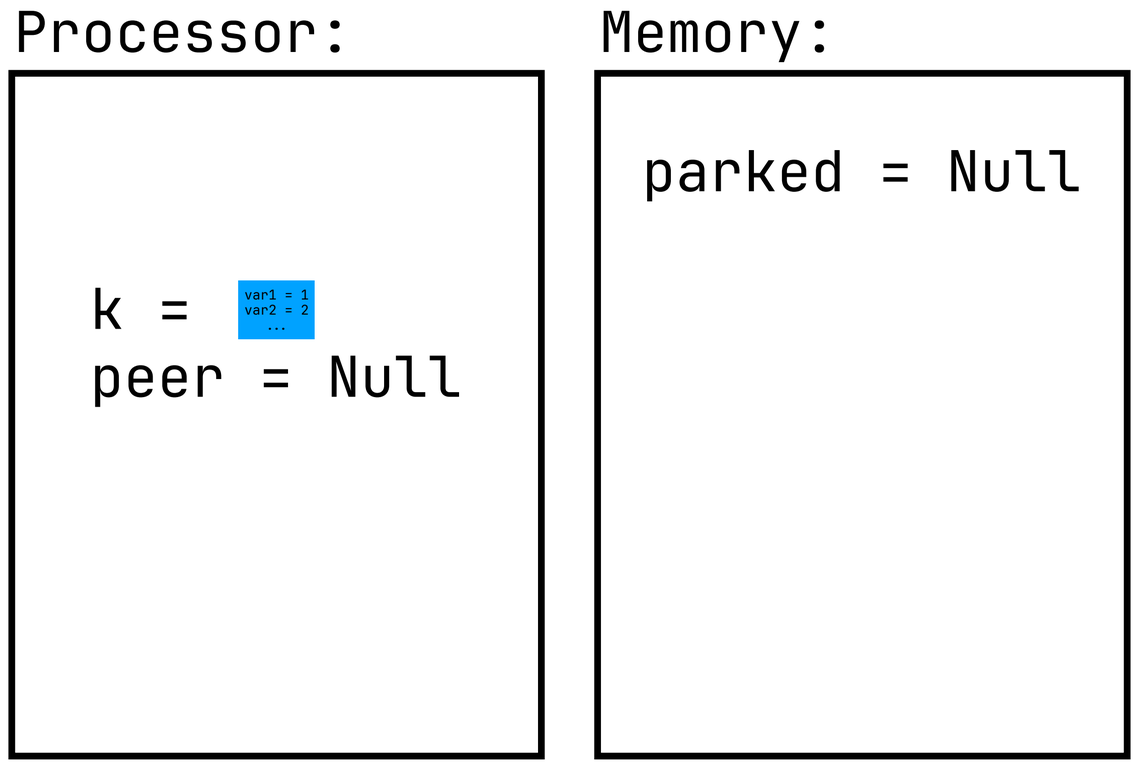


Background

Effect handler

A powerful language feature

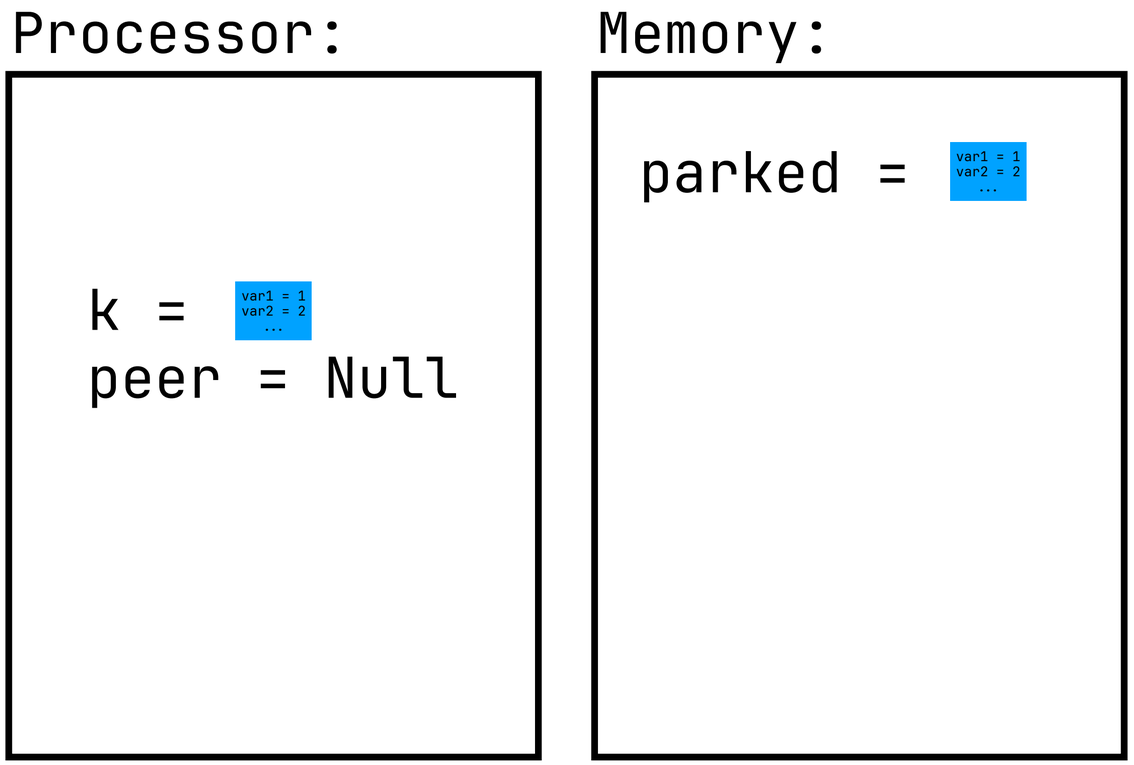
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

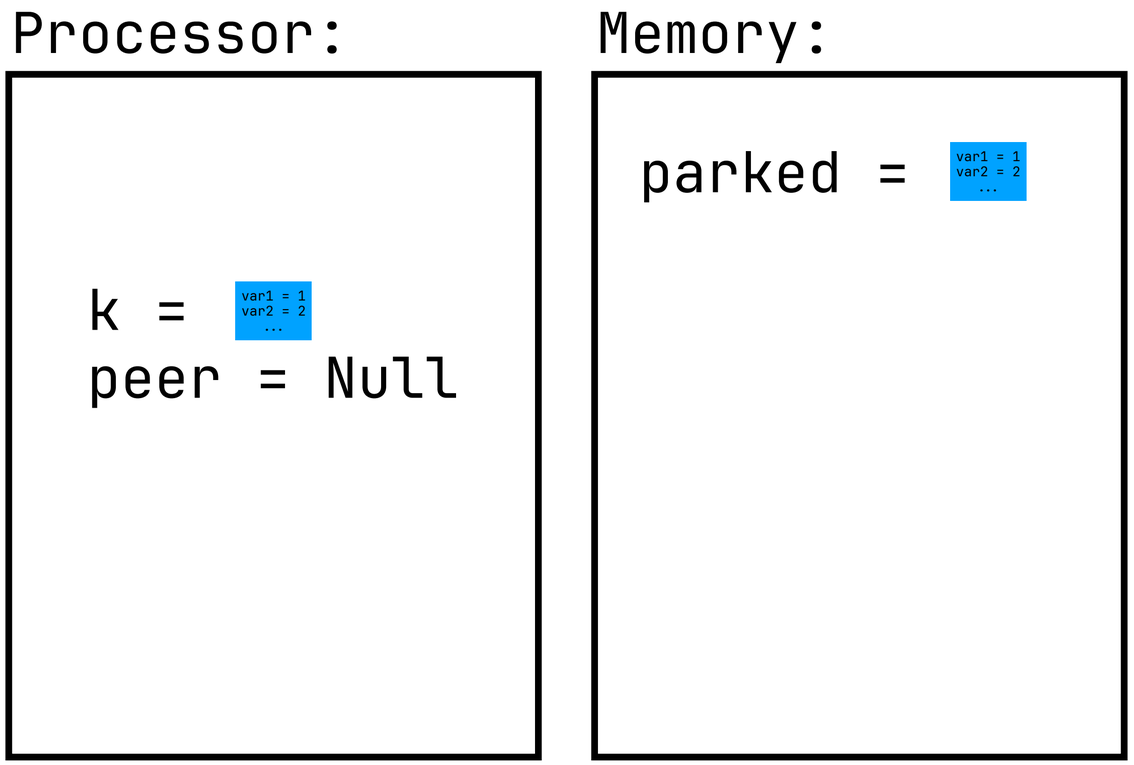
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

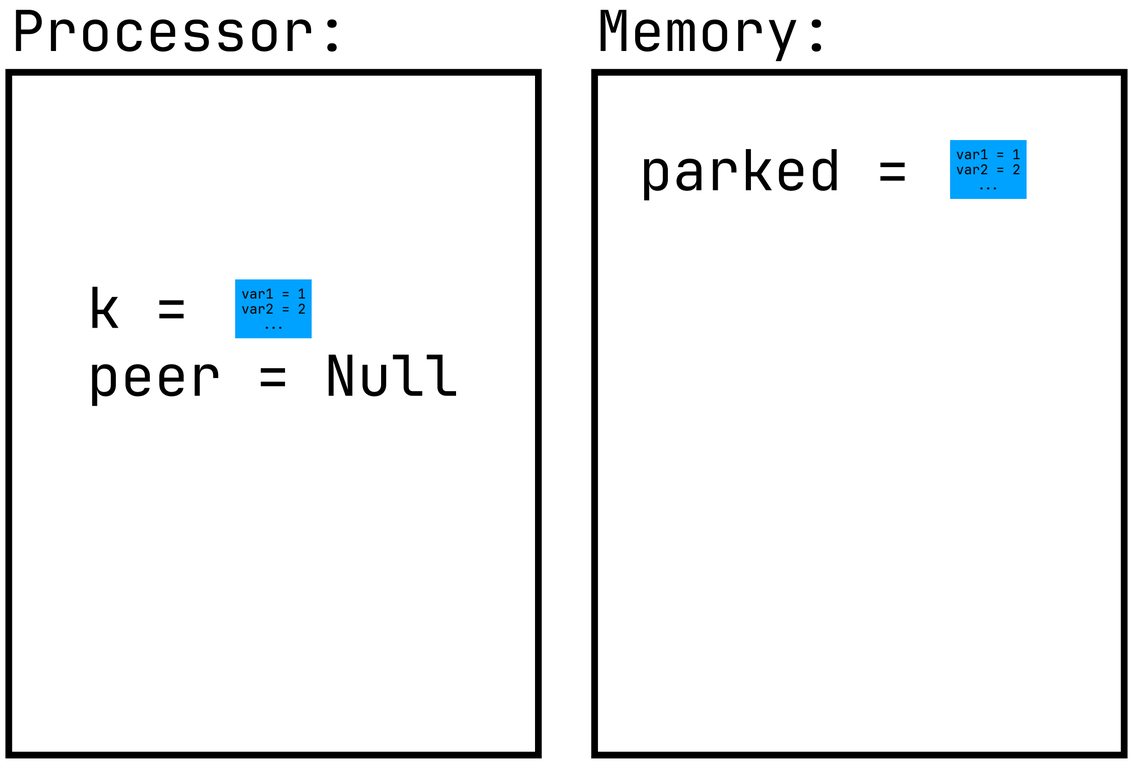
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

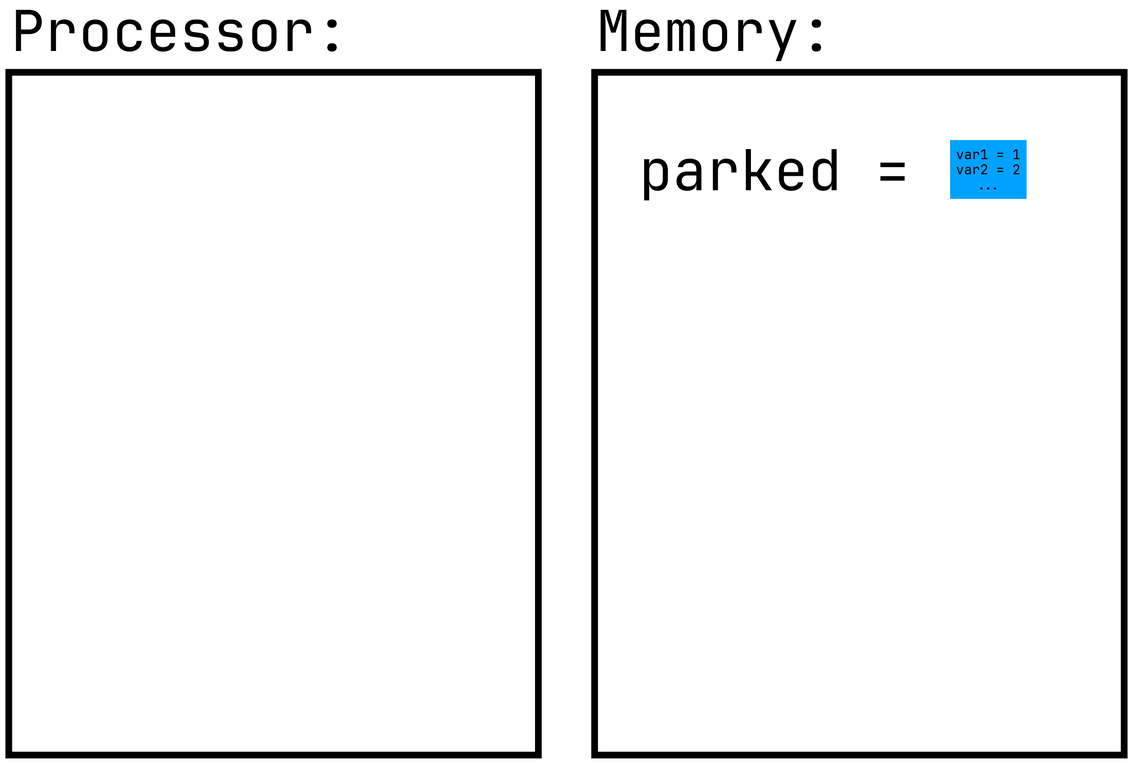


Background

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

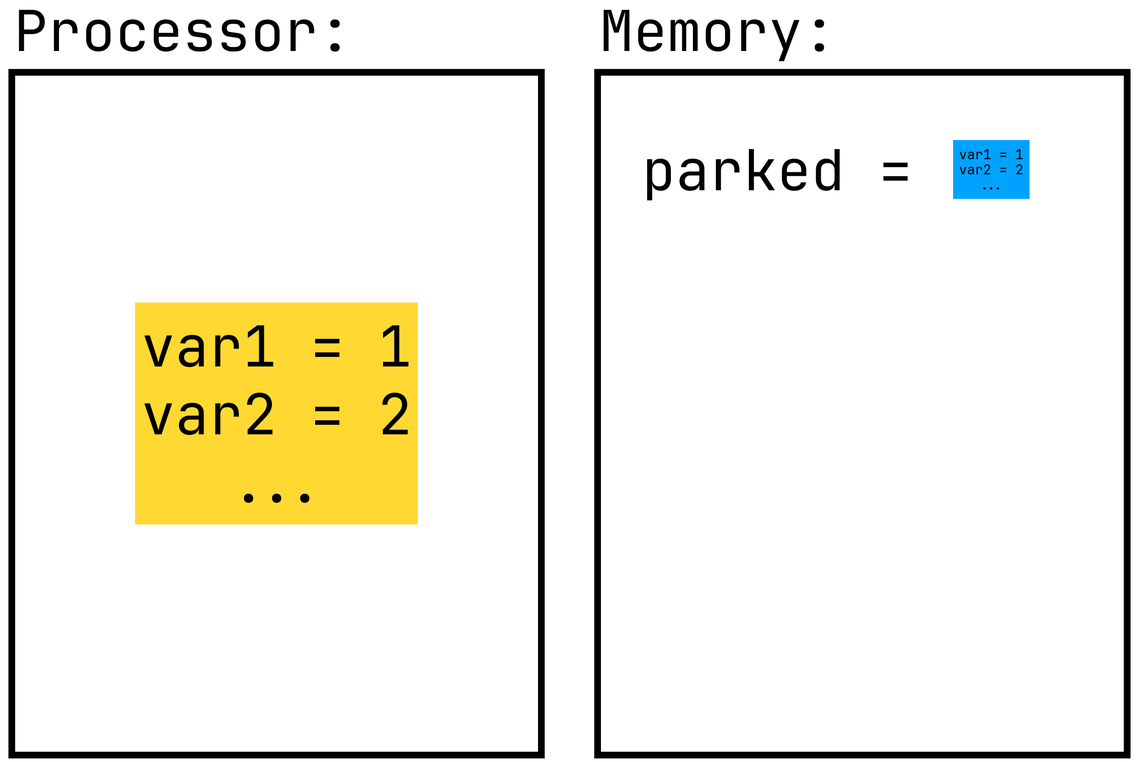


Background

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

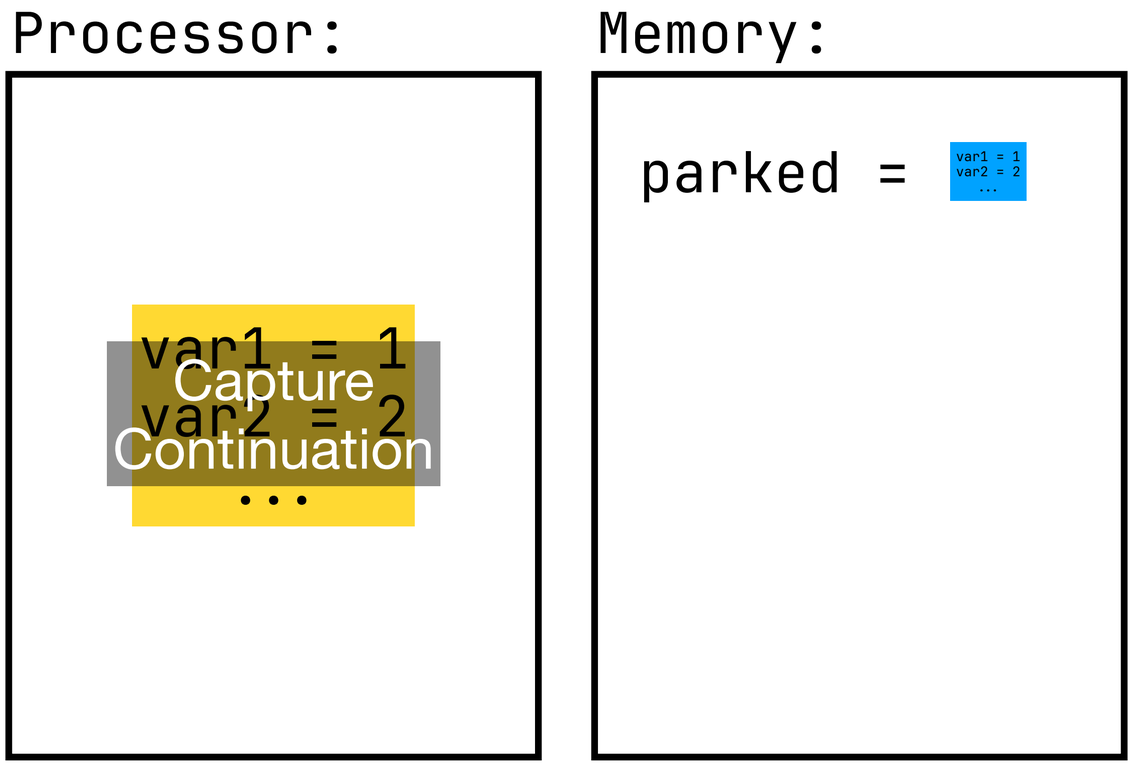


Background

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

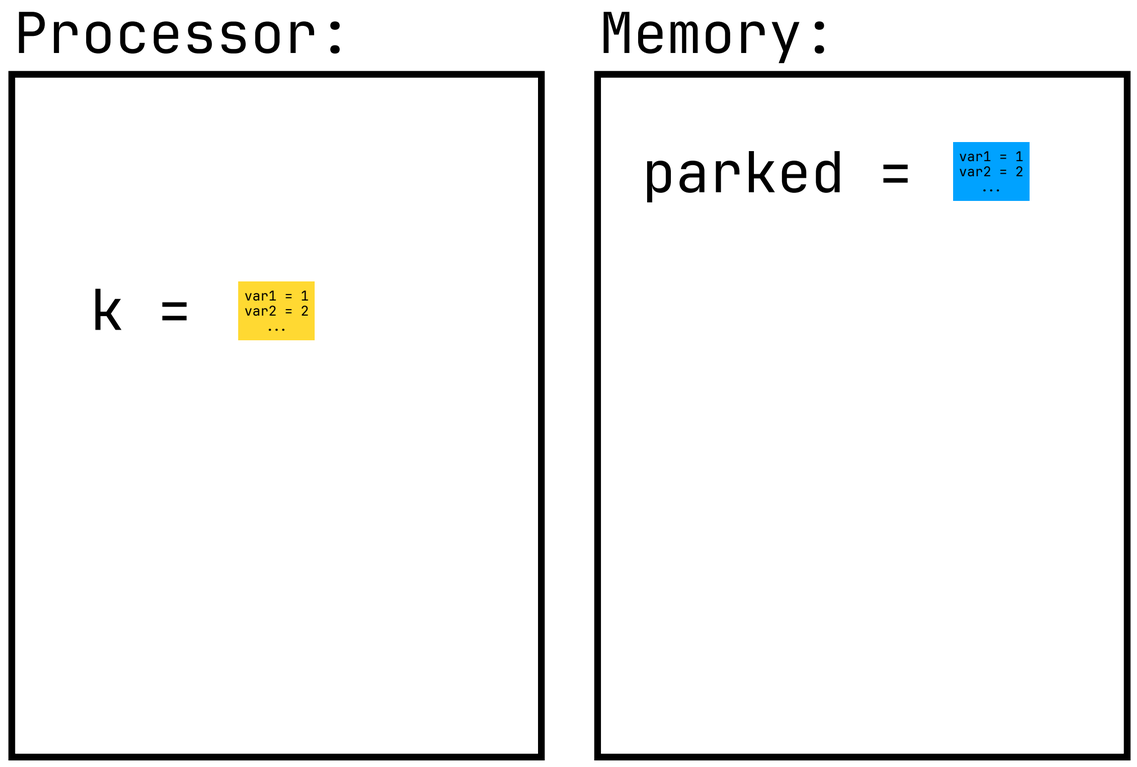


Background

Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) =>
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

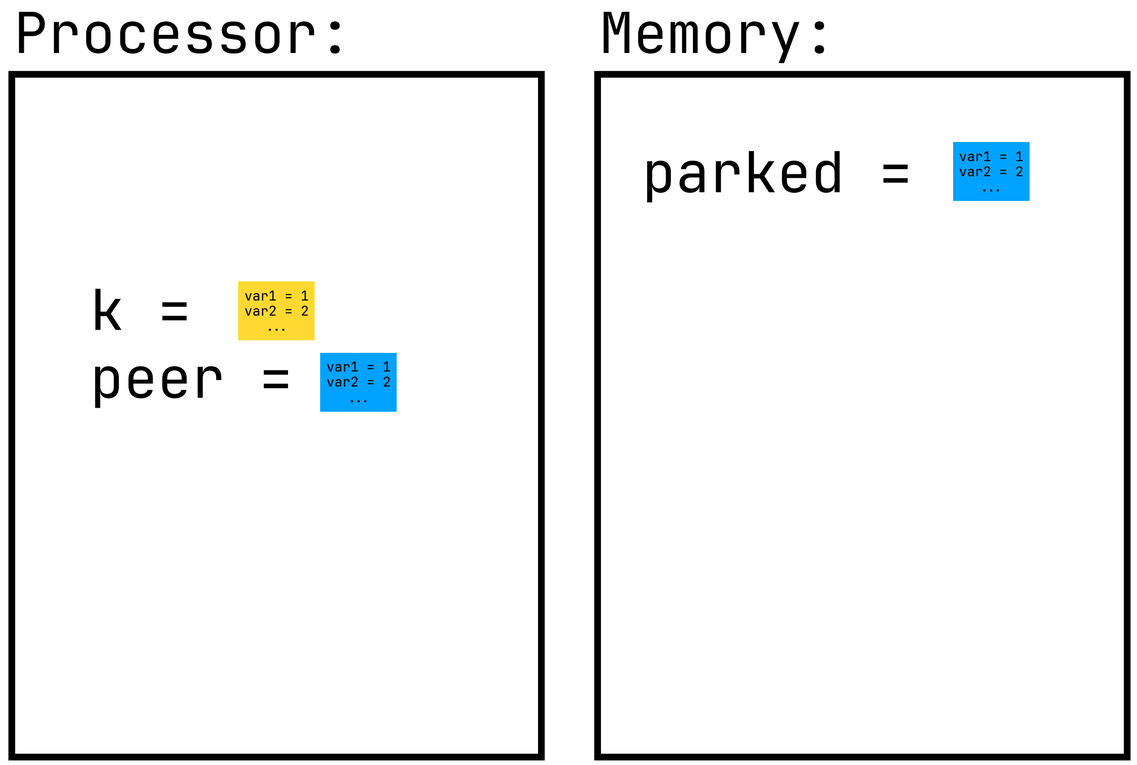


Background

Effect handler

A powerful language feature

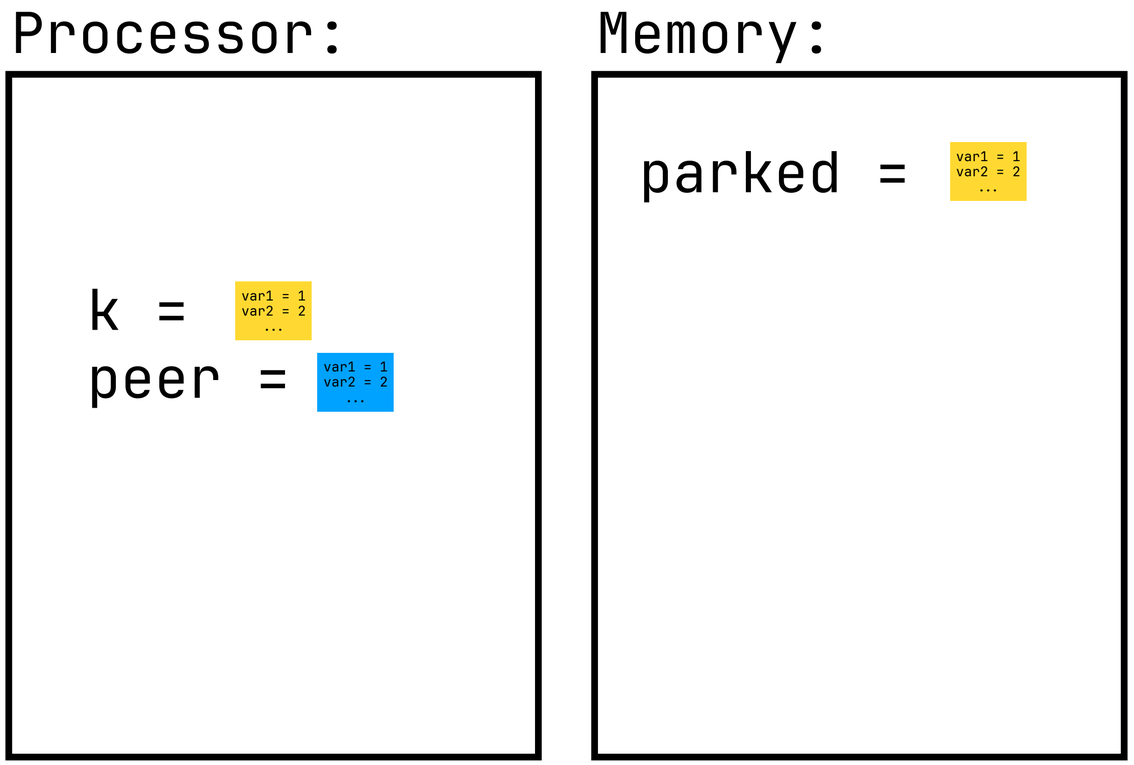
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

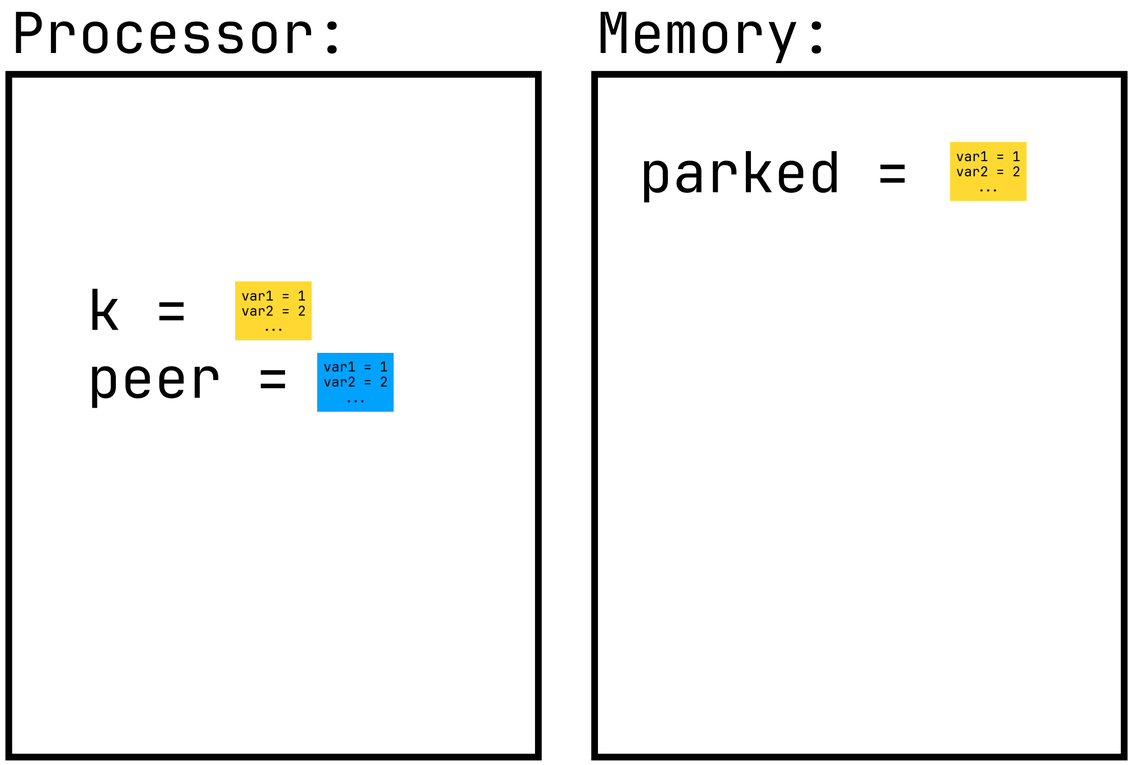
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

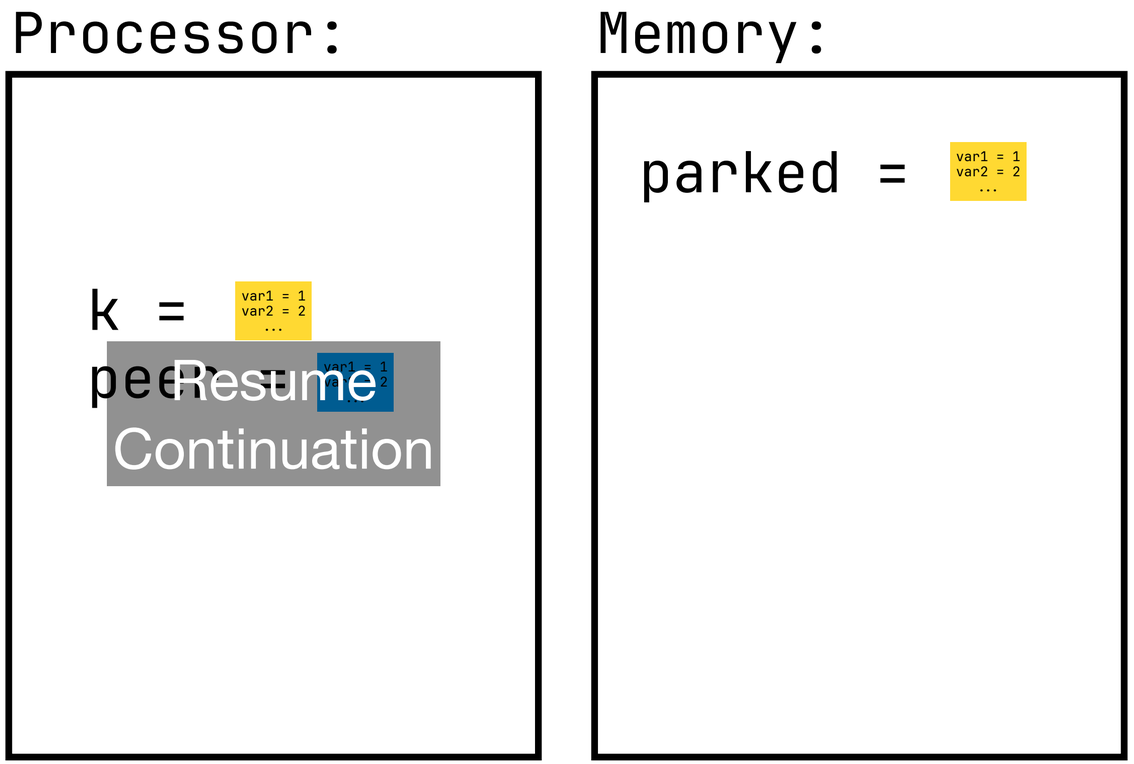


Background

Effect handler

A powerful language feature

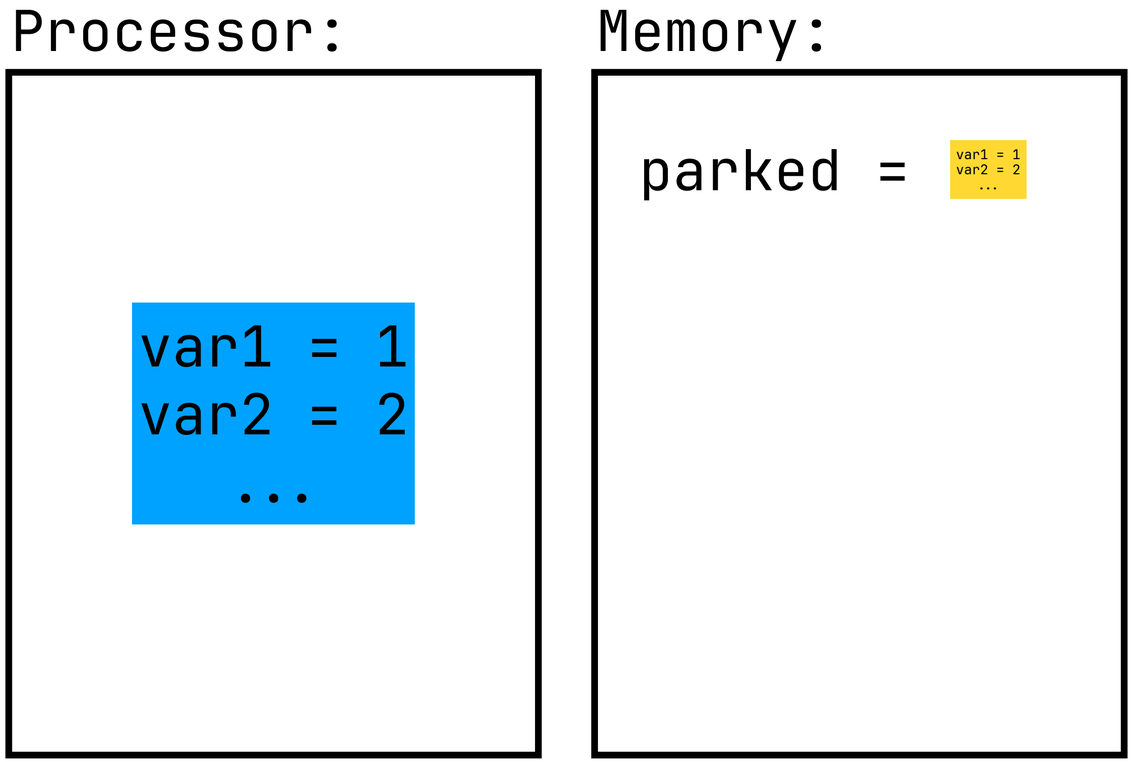
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

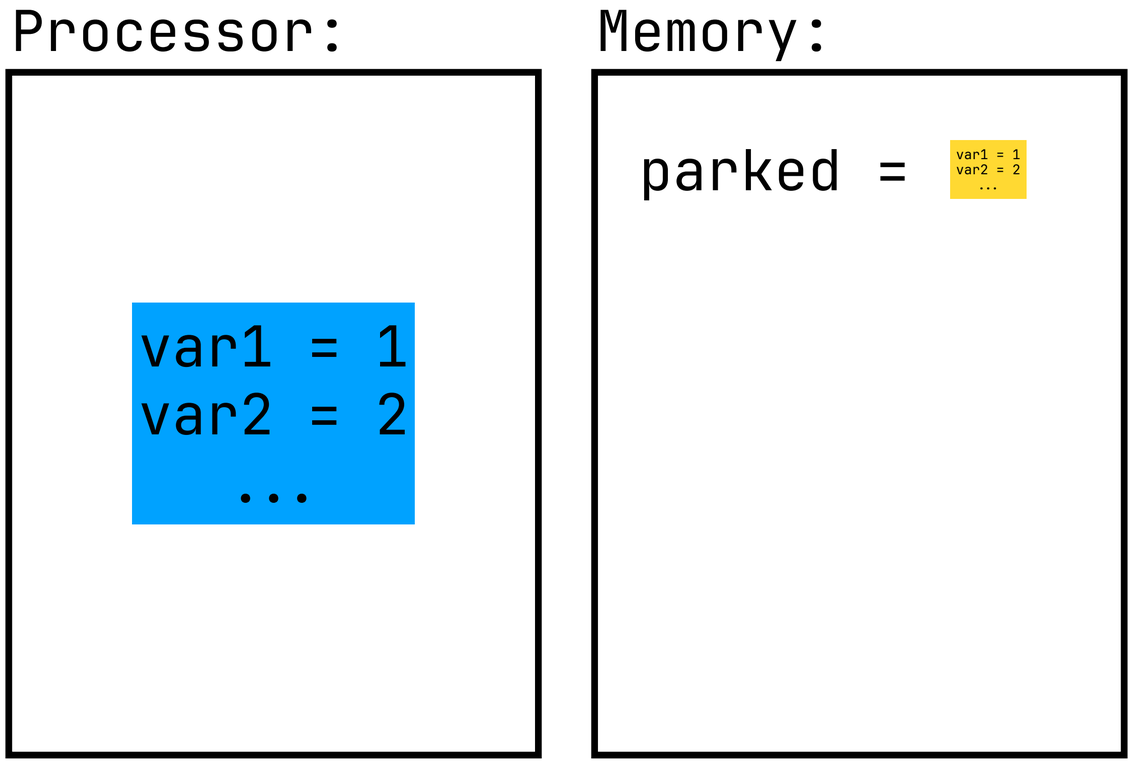
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

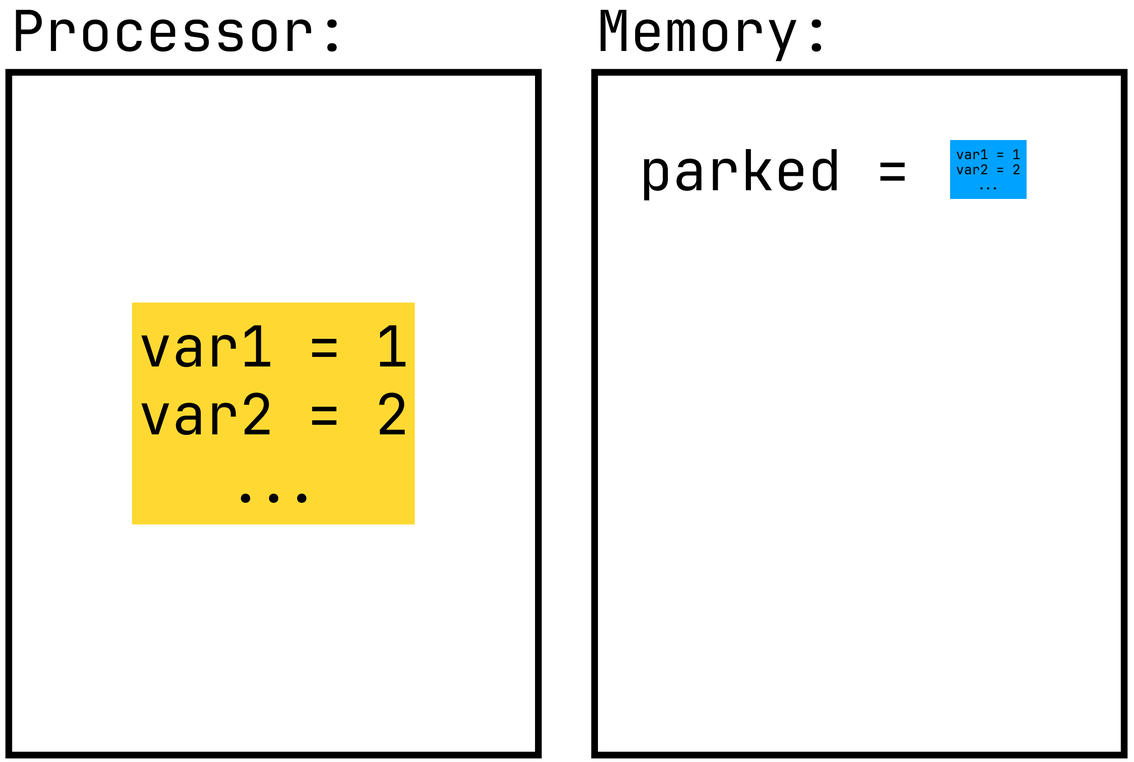
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

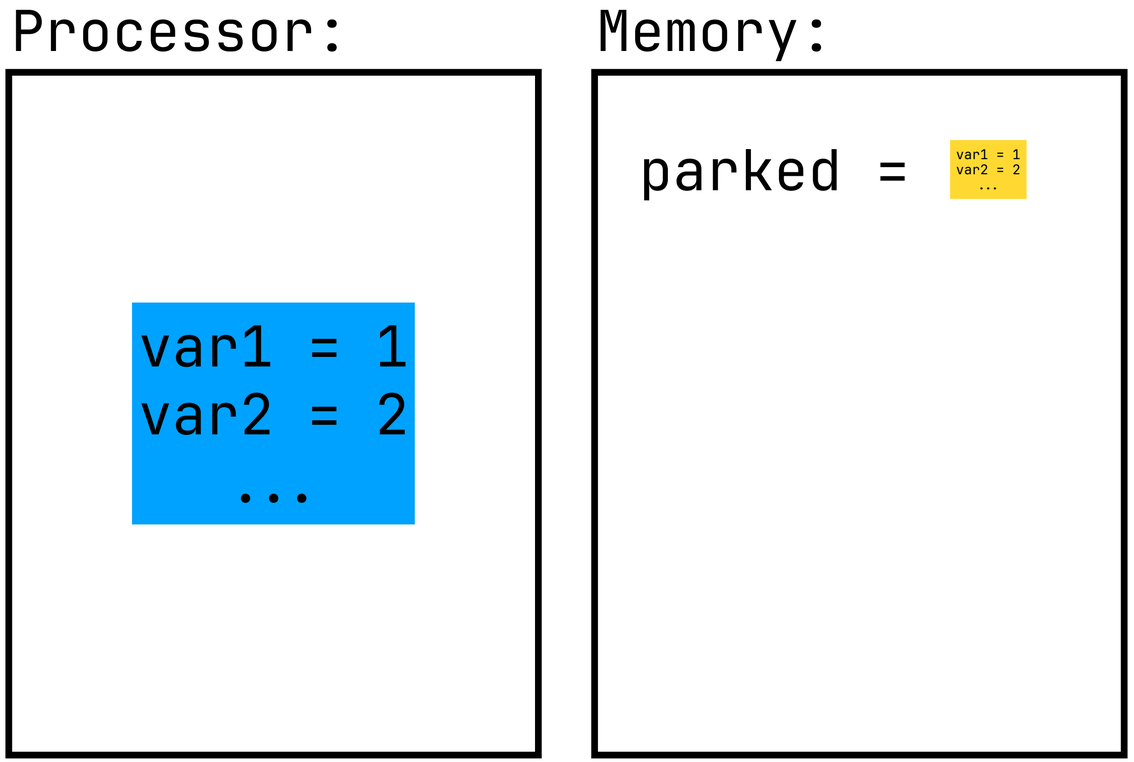
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

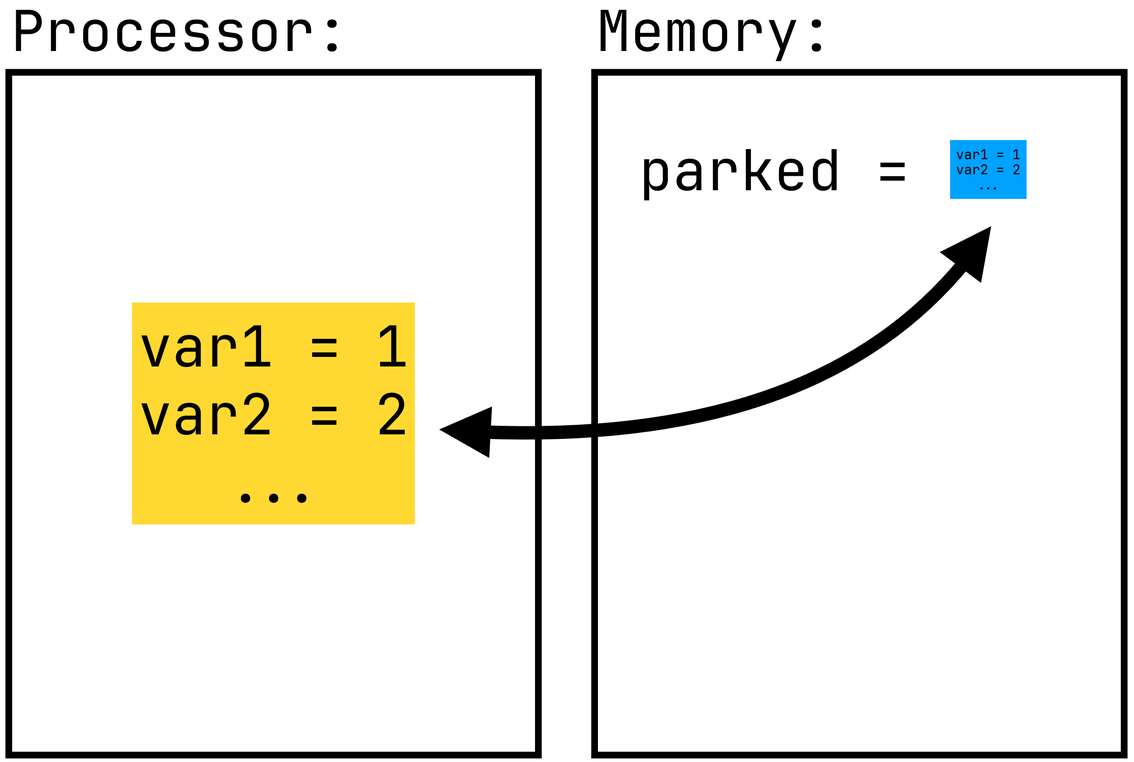
```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise Yield()
  with Yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```



Effect handler

A powerful language feature

Language Definition

Syntax + **Semantics**

Effect handler

A powerful language feature

Language Definition

Syntax + **Semantics**

```
handle E with H
```

```
raise ...
```

Capture continuation

```
resume ...
```

Resume continuation

Effect handler

A powerful language feature

Language Definition

Syntax + **Semantics**

```
handle E with H    Install a handler
  raise ...        Find handler; Capture continuation
  resume ...       Resume continuation
```

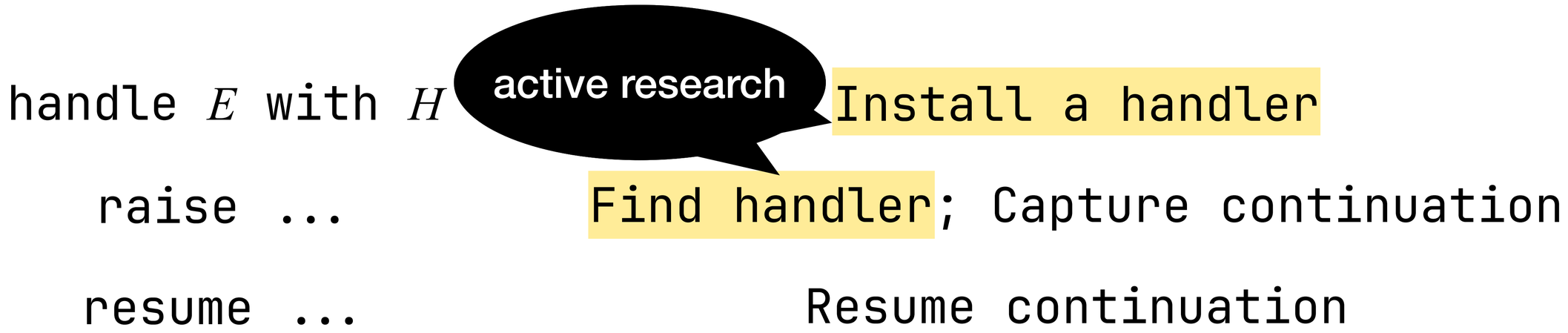
active research

Effect handler

A powerful language feature

Language Definition

Syntax + **Semantics**



Introducing lexical effect handler

Lexical effect handler

Last bit of semantics: install a handler

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise yield()
  with yield(k) ⇒
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

Lexical effect handler

Last bit of semantics: install a handler

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

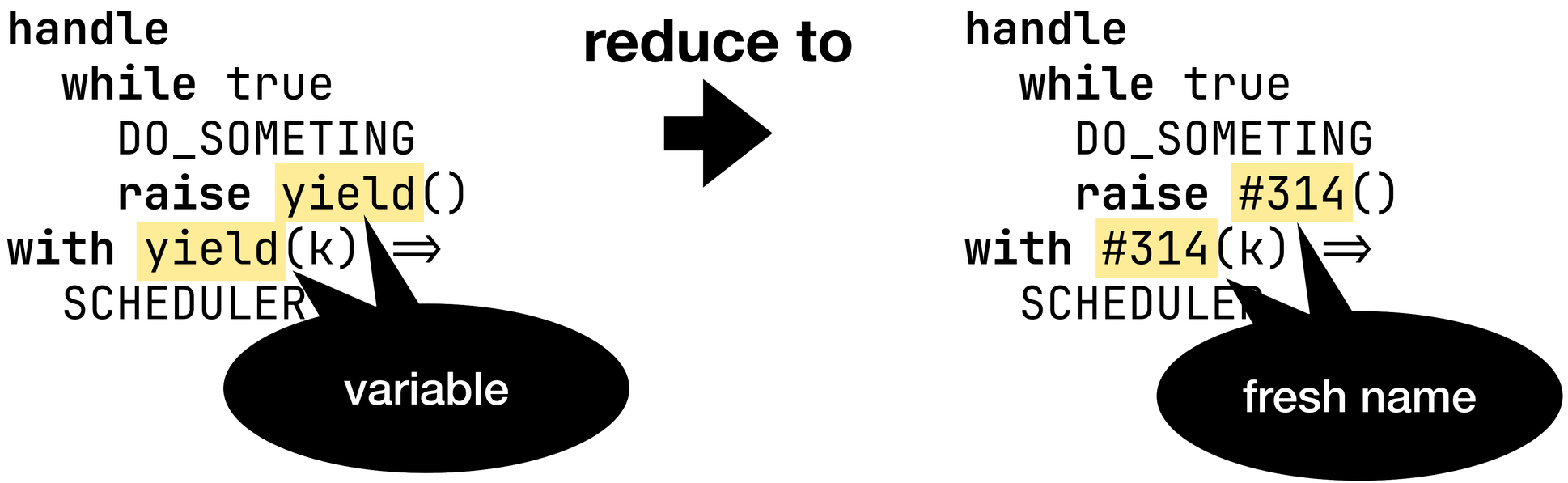
Lexical effect handler

Last bit of semantics: install a handler

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

Lexical effect handler

Last bit of semantics: install a handler



Lexical effect handler

Last bit of semantics: install a handler

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to

```
handle
  while true
    DO_SOMETING
    raise #314()
with #314(k) ⇒
  SCHEDULER
```

handler frame

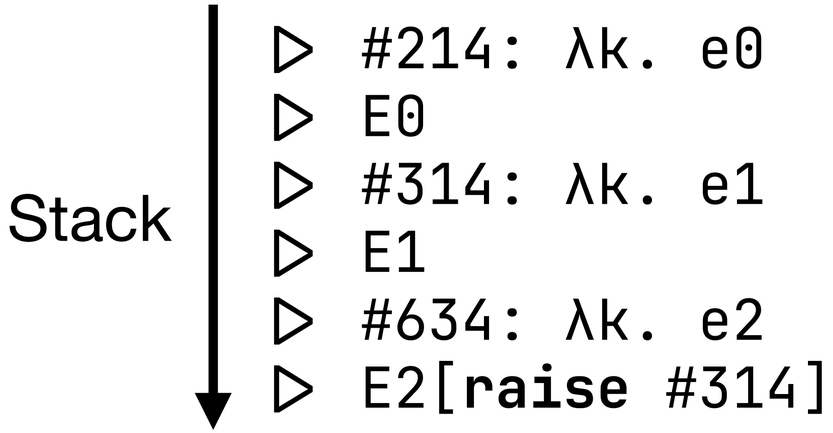
function frame

Another presentation:

```
▷ #314: SCHEDULER
▷ while true {...raise #314()}
```

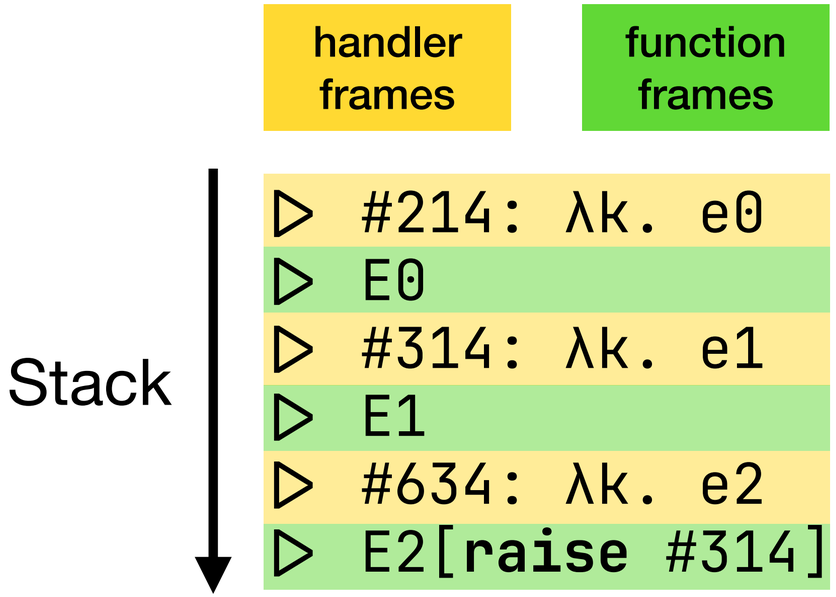
Lexical effect handler

Last bit of semantics: install a handler



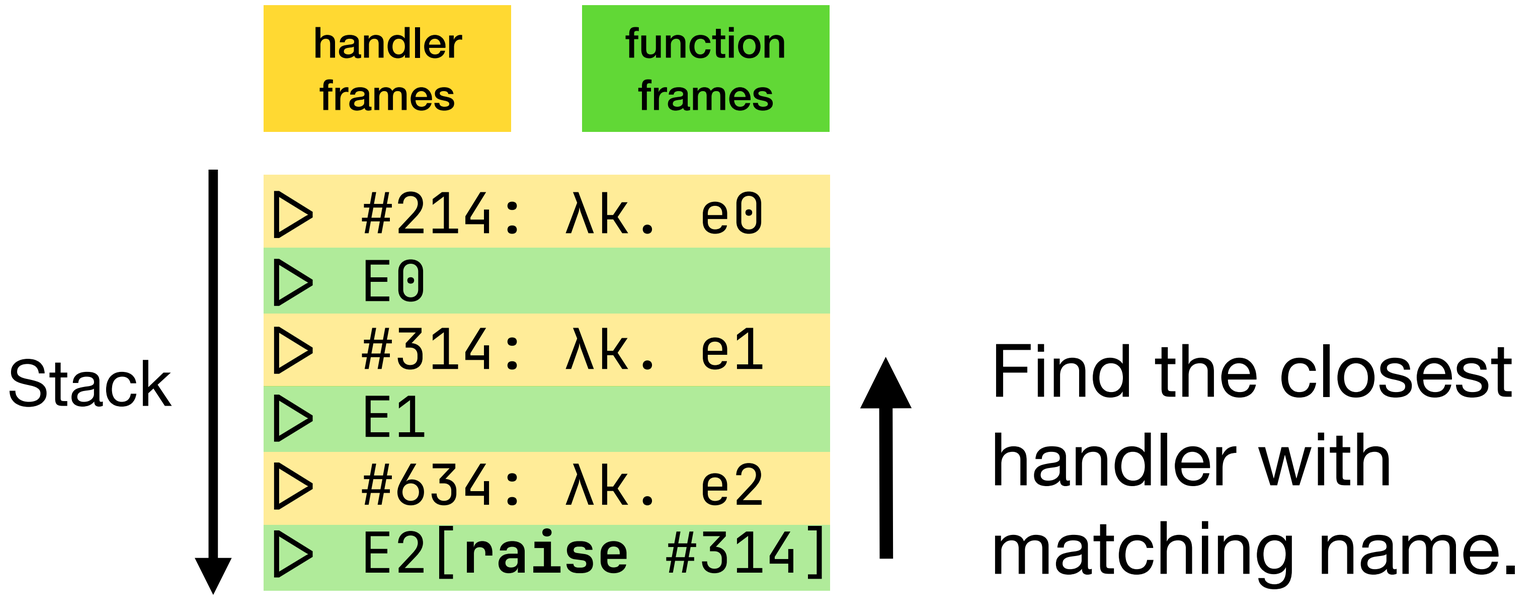
Lexical effect handler

Last bit of semantics: install a handler



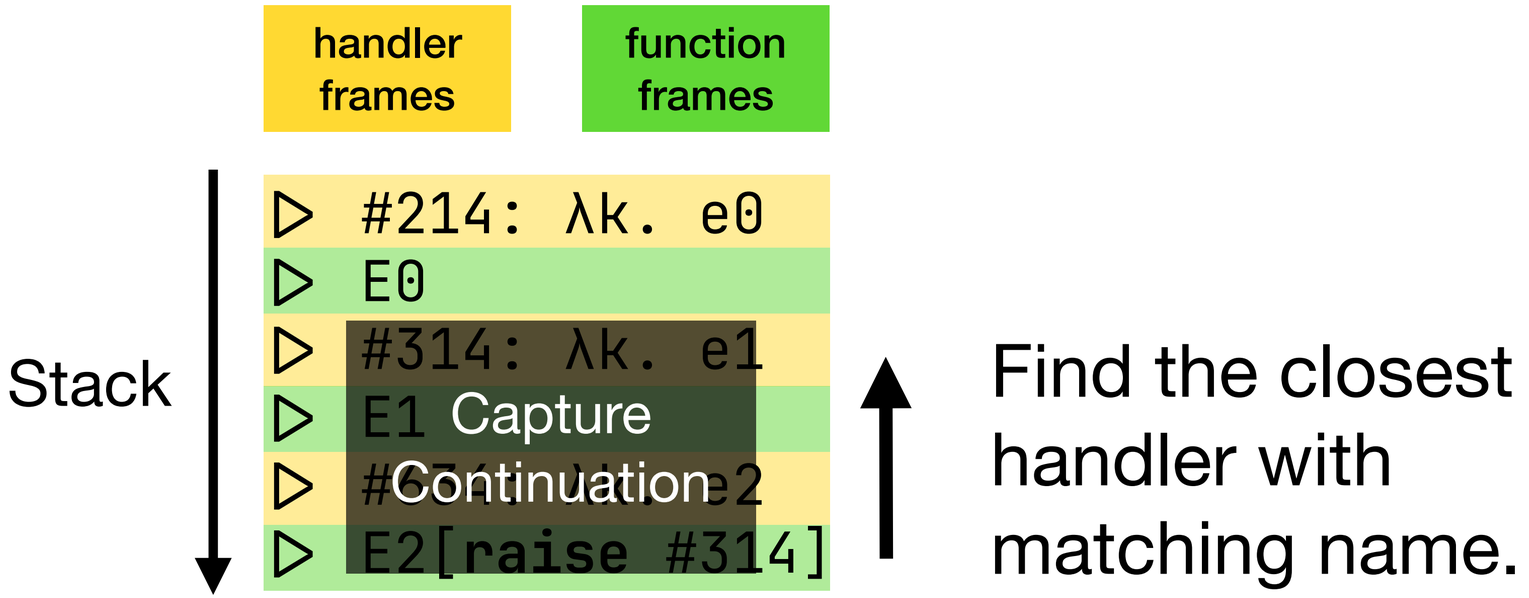
Lexical effect handler

Last bit of semantics: install a handler



Lexical effect handler

Last bit of semantics: install a handler



Lexical effect handler

A powerful language feature

Syntax	+	Semantics
`handle` E `with` H		Install a handler
`raise ...`		Find handler; Capture continuation
`resume ...`		Resume continuation

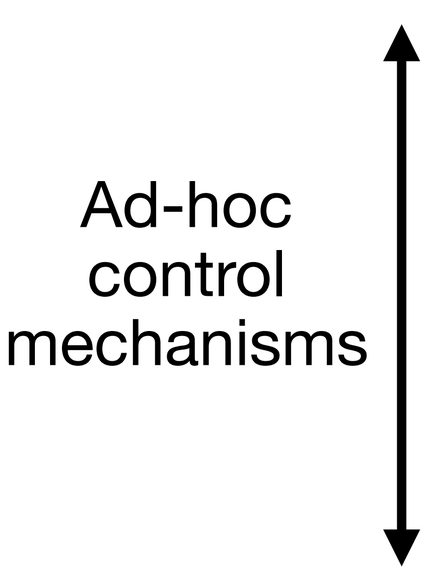
Lexical effect handler

A powerful language feature

Syntax	+	Semantics
`handle` E `with` H		Install a handler
`raise ...`		Find handler; Capture continuation
`resume ...`		Resume continuation

Effect handler

A powerful language feature



async/await

Dependency injection

Coroutine

Generator

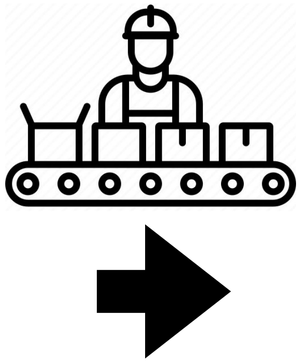
Multitasking

Exception Handler

Effect Handler

high-level effect handlers in Lexa

```
handle E with H

raise ...

resume ...
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

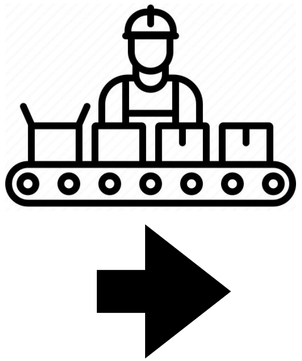
Direct Lexa
OOPSLA24

Zero Lexa
OOPSLA25

Multi Lexa
work-in-progress

high-level effect handlers in Lexa

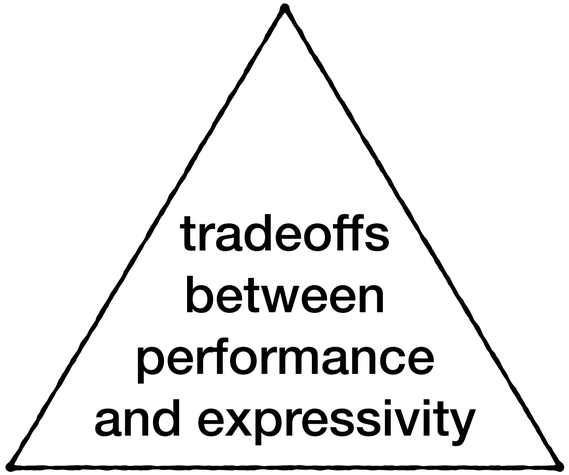
```
handle E with H

raise ...

resume ...
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

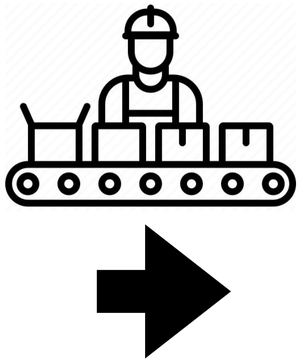


Multi Lexa

Zero Lexa

high-level effect handlers in Lexa

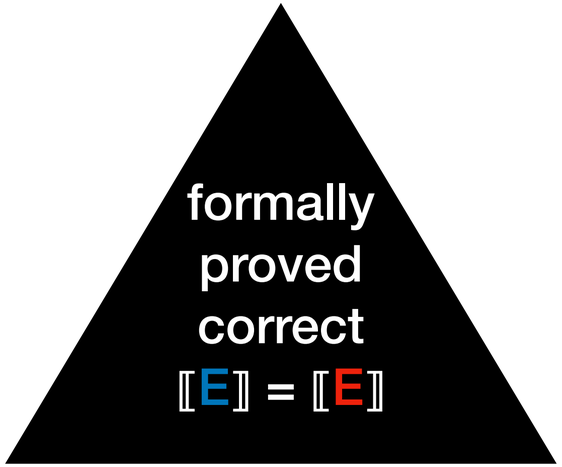
```
handle E with H

raise ...

resume ...
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

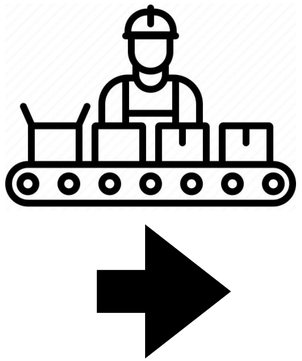


Multi Lexa

Zero Lexa

high-level effect handlers in Lexa

```
handle E with H

raise ...

resume ...
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

formally proved correct
⟦E⟧ = ⟦E⟧

Multi Lexa

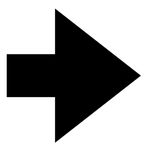
Zero Lexa

Direct Lexa

Idea 1: use stack addresses as handler names

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



globally unique names

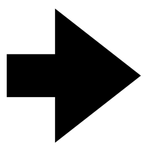
```
▷ #314: SCHEDULER
▷ E[raise #314()]
```

Direct Lexa

Idea 1: use stack addresses as handler names

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



stack addresses

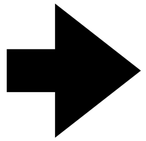
```
0x123 ▷ #314: SCHEDULER
0x124 ▷ E[raise #314()]
```

Direct Lexa

Idea 1: use stack addresses as handler names

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



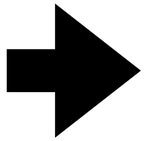
```
0x123 ▷ SCHEDULER
0x124 ▷ E[raise ?()]
```

Direct Lexa

Idea 1: use stack addresses as handler names

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



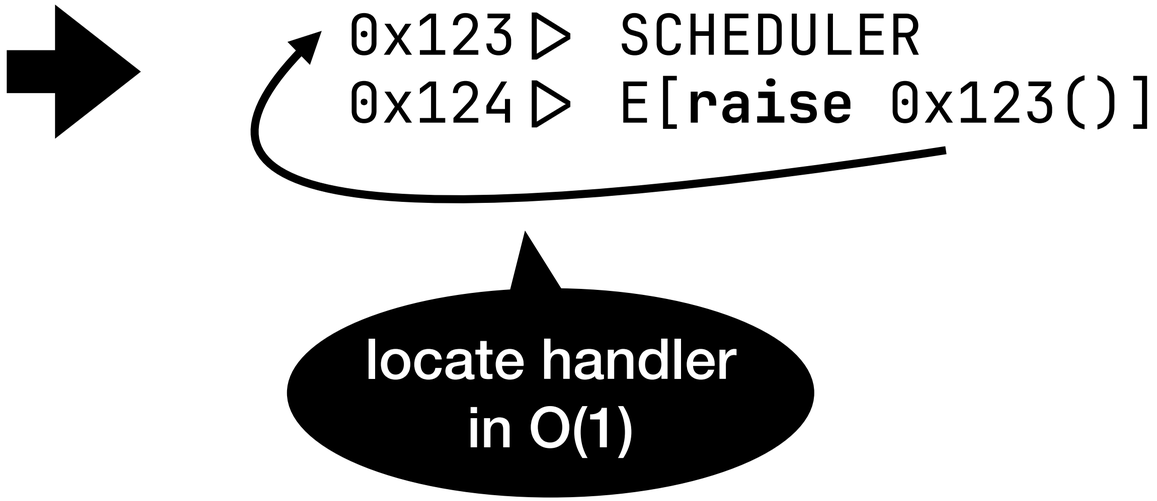
```
0x123 ▷ SCHEDULER
0x124 ▷ E[raise 0x123()]
```

Direct Lexa

Idea 1: use stack addresses as handler names

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



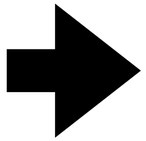
Direct Lexa

Idea 1: use stack addresses as handler names

This design will come back and bite us—more on that later.

```
handle
  while true
    DO_SOMETING
    raise yield()
with yield(k) ⇒
  SCHEDULER
```

reduce to



```
0x123 ▷ SCHEDULER
0x124 ▷ E[raise 0x123()]
```

Direct Lexa

Idea 2: Stack switching

Direct Lexa

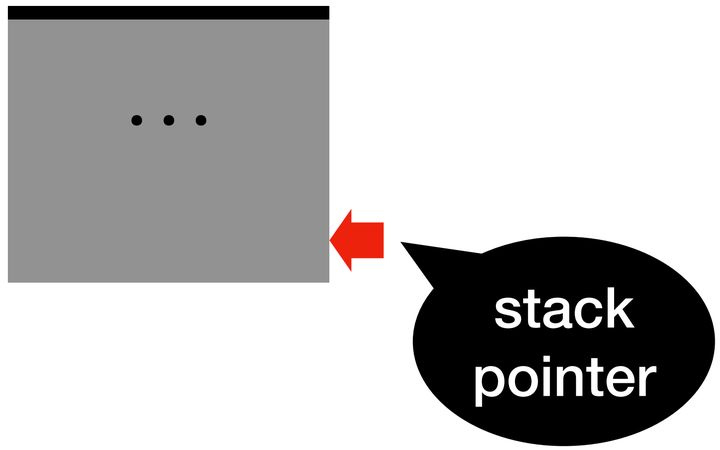
Idea 2: Stack switching

```
INSTALL
RAISE
RESUME
```

Direct Lexa

Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```



INSTALL

RAISE

RESUME

Direct Lexa

Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

INSTALL

RAISE

RESUME

Direct Lexa

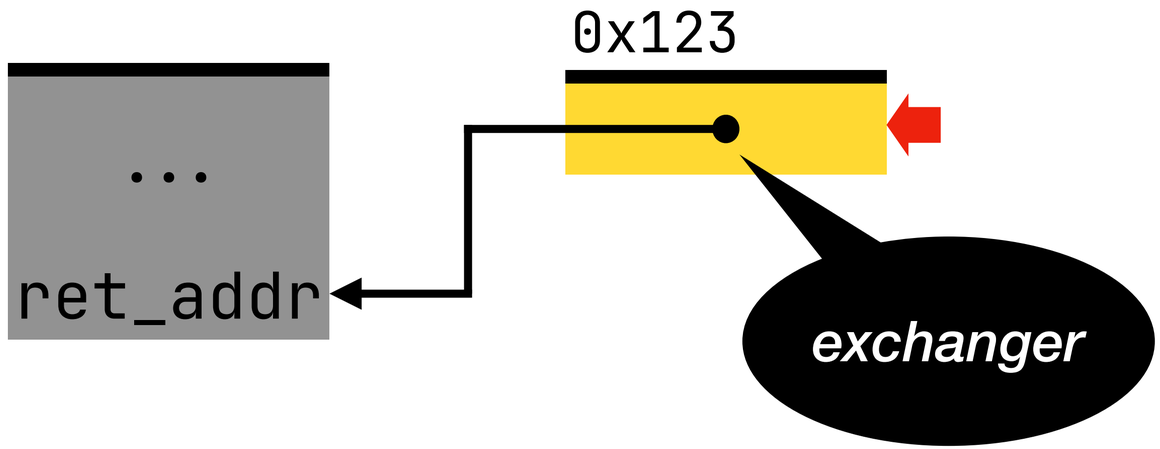
Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

INSTALL

RAISE

RESUME



Direct Lexa

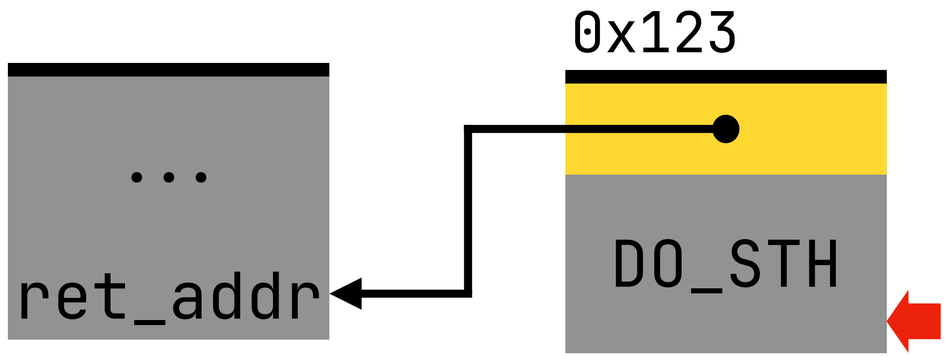
Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

INSTALL

RAISE

RESUME



Direct Lexa

Stack switching

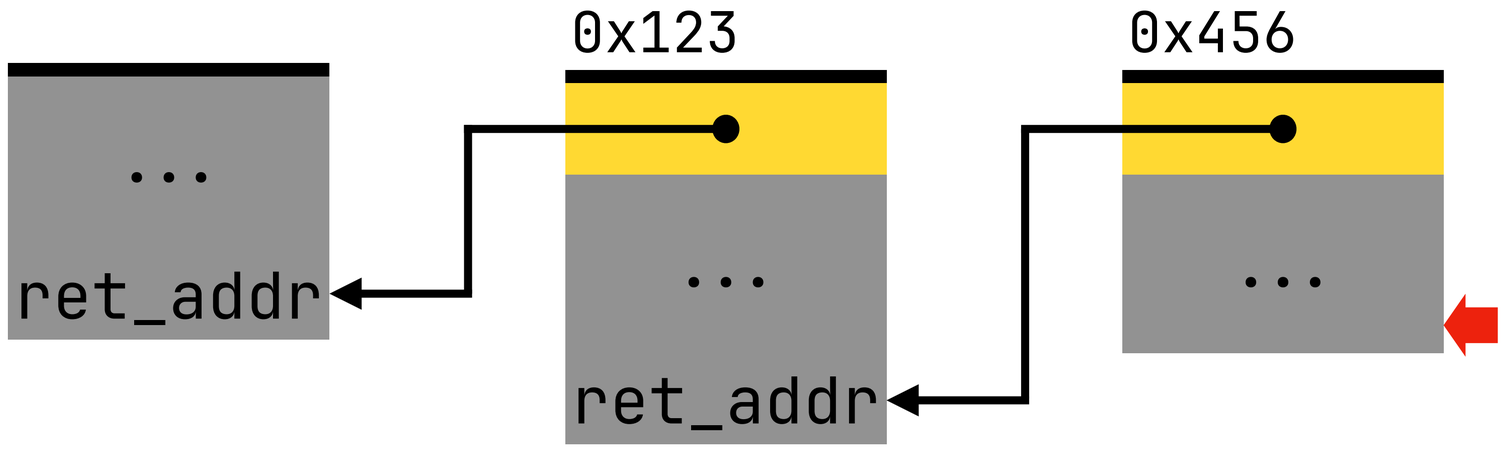
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



Direct Lexa

Stack switching

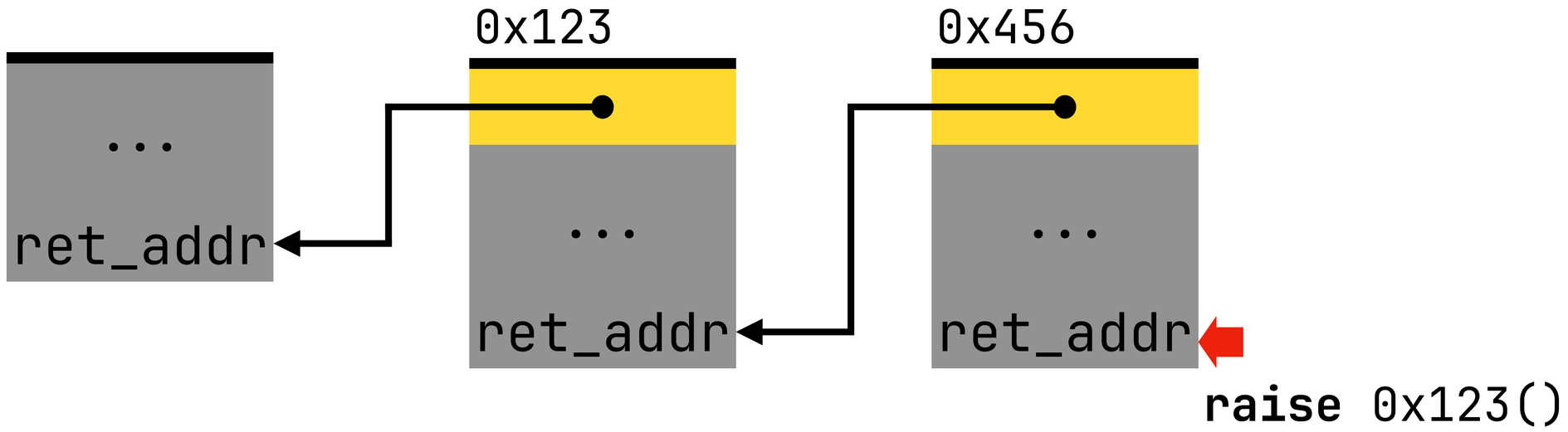
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



Direct Lexa

Stack switching

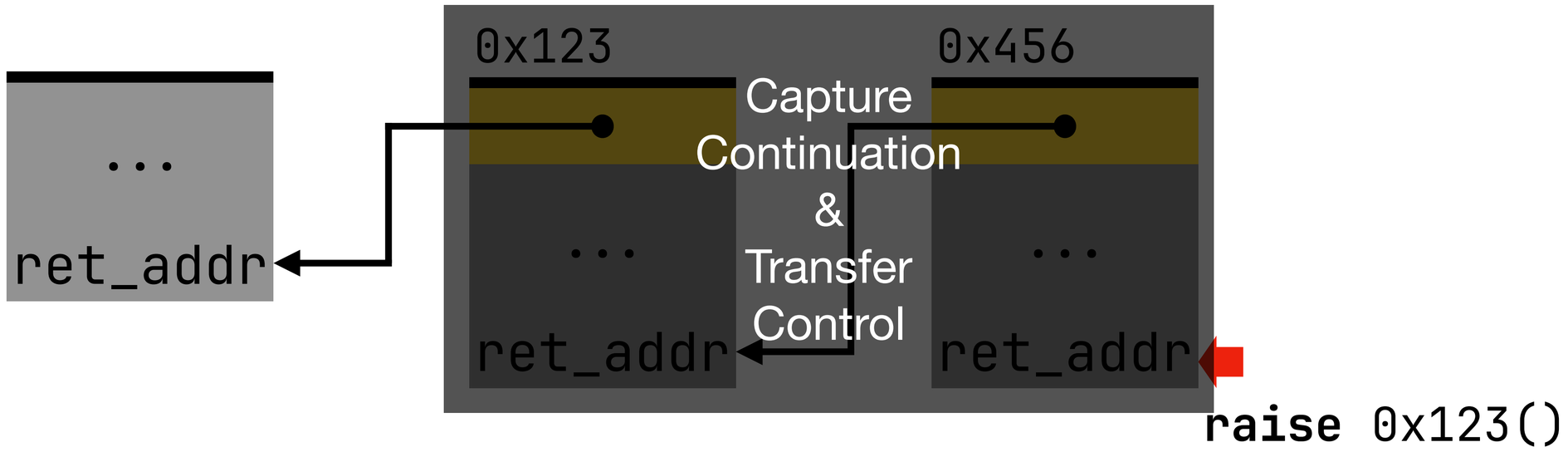
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



Direct Lexa

Stack switching

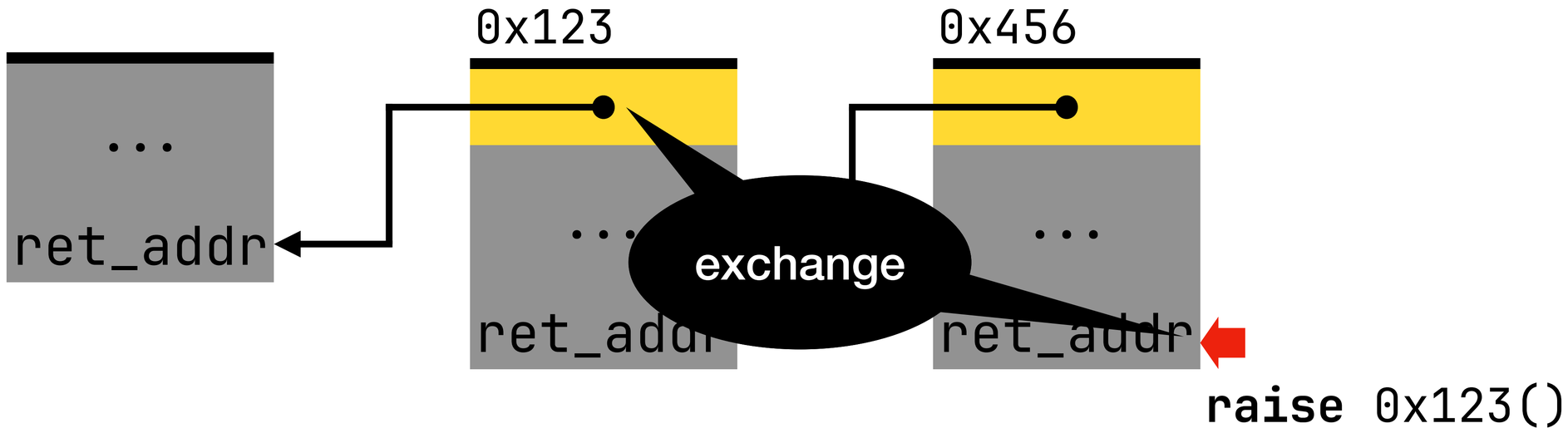
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



raise 0x123()

Direct Lexa

Stack switching

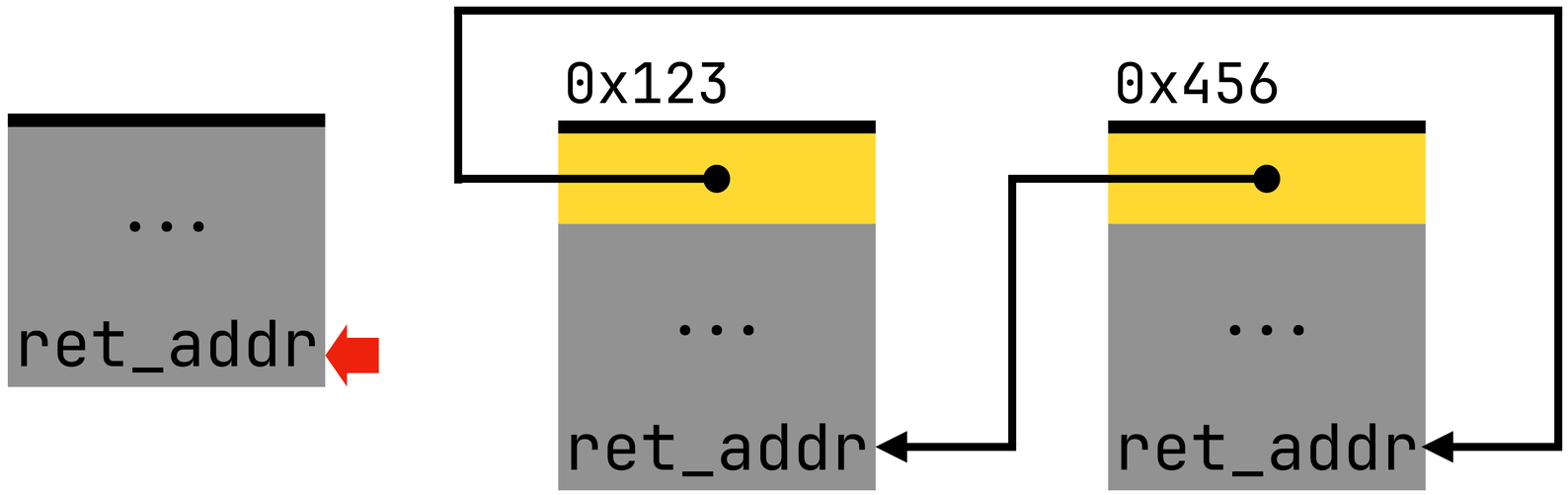
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



Direct Lexa

Stack switching

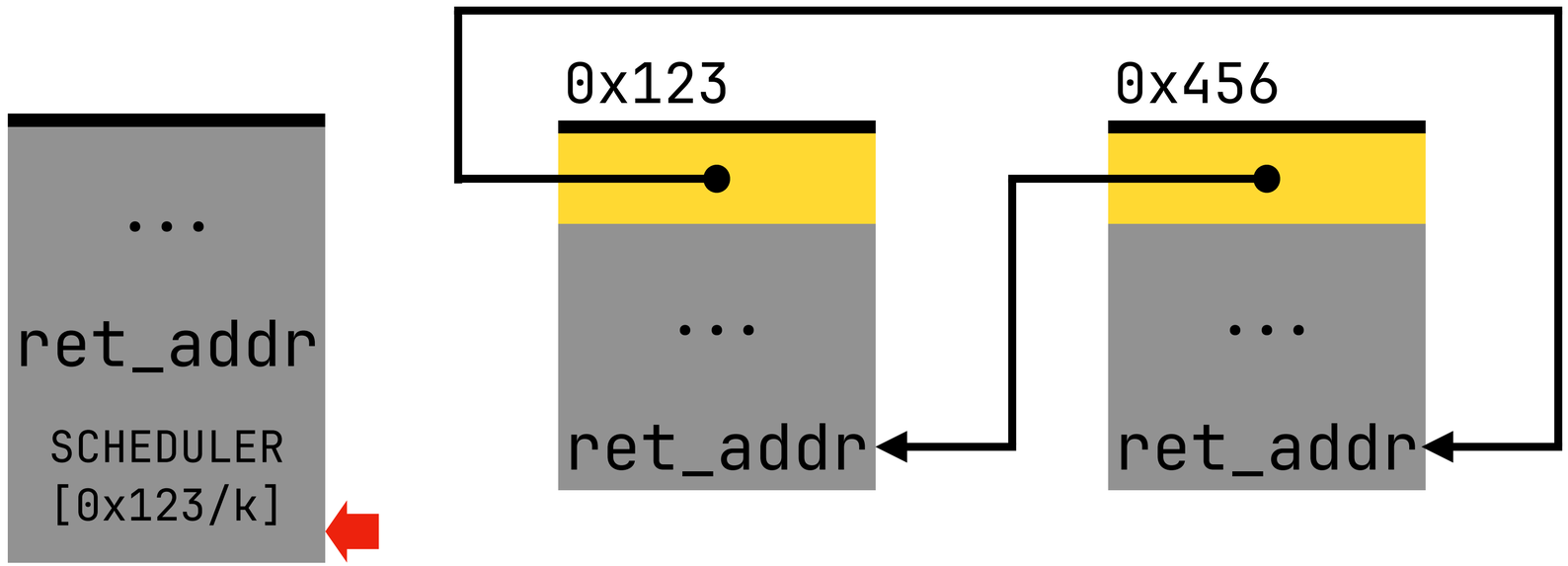
```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  SCHEDULER
```

```
handle
  ...
  raise yield()
with ...
```

INSTALL

RAISE

RESUME



Direct Lexa

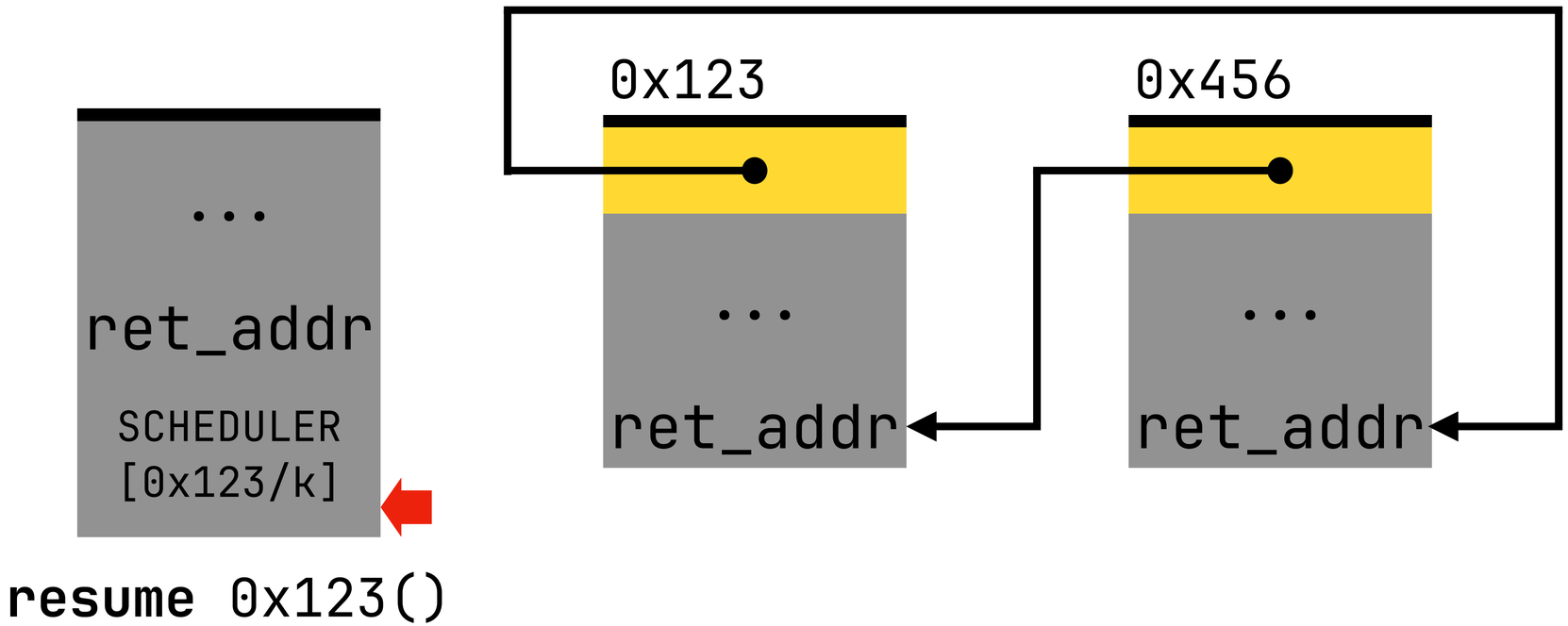
Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  ...
  resume peer ()
```

INSTALL

RAISE

RESUME




```
resume 0x123()
```

Direct Lexa

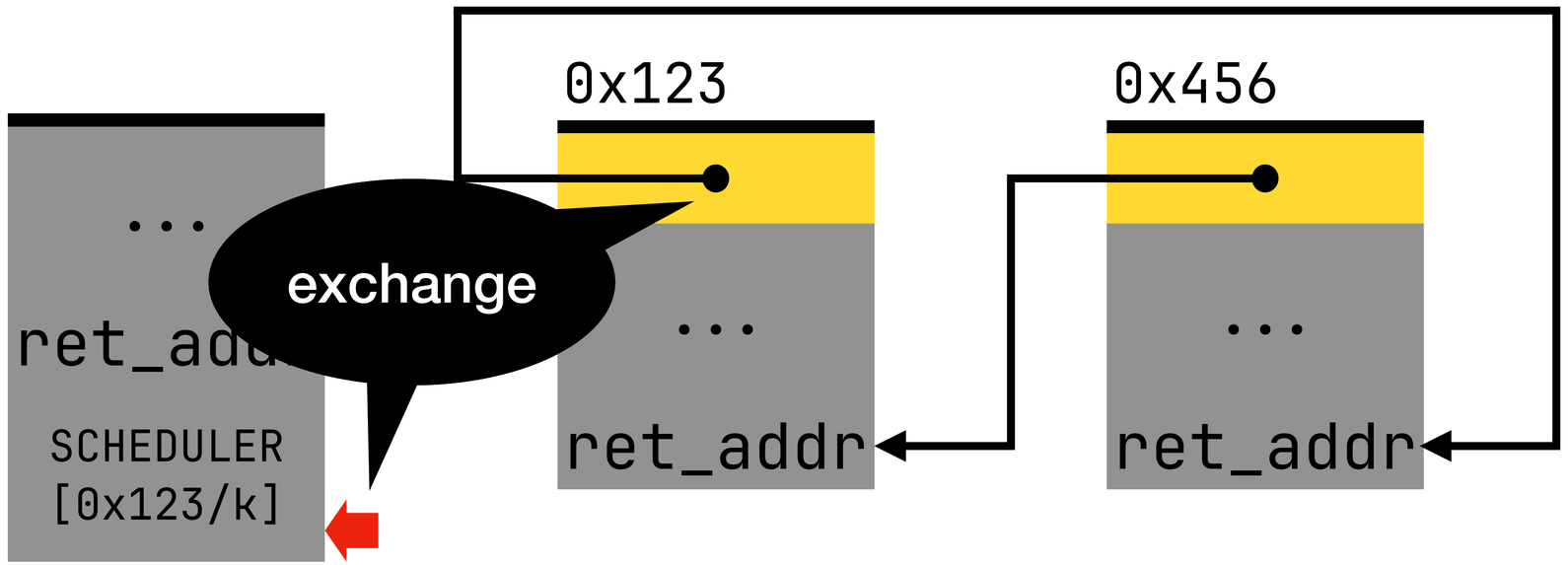
Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  ...
  resume peer ()
```

INSTALL

RAISE

RESUME




```
resume 0x123()
```

Direct Lexa

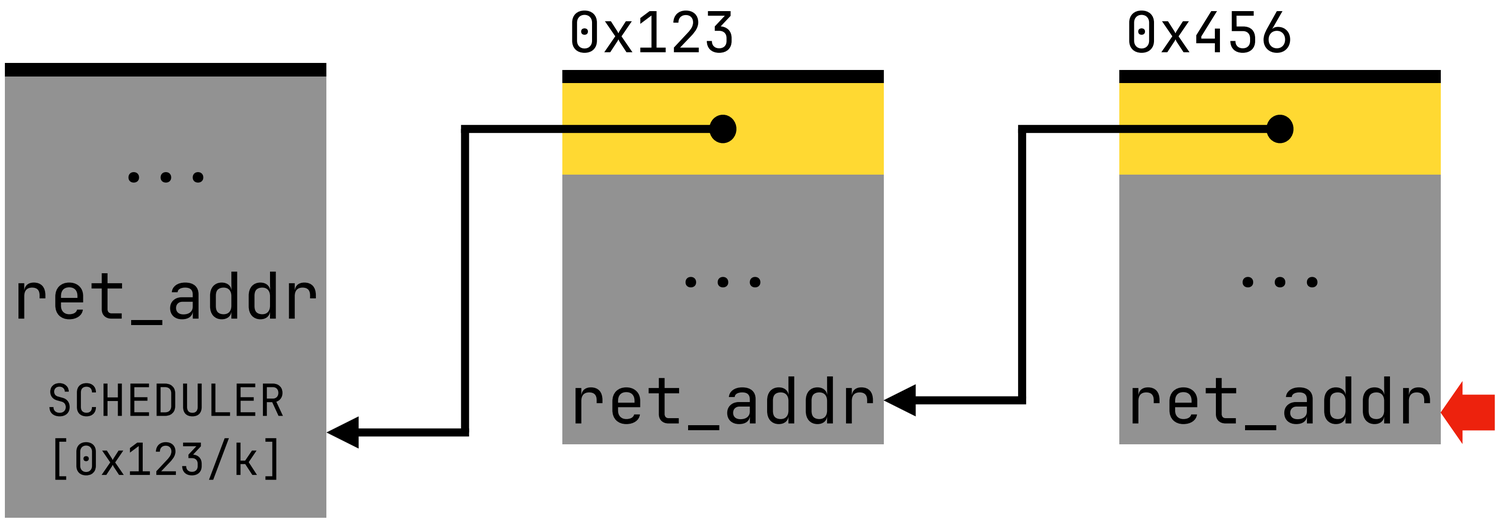
Stack switching

```
handle
  while true
    DO_SOMETING
with yield(k) ⇒
  ...
  resume peer ()
```

INSTALL

RAISE

RESUME

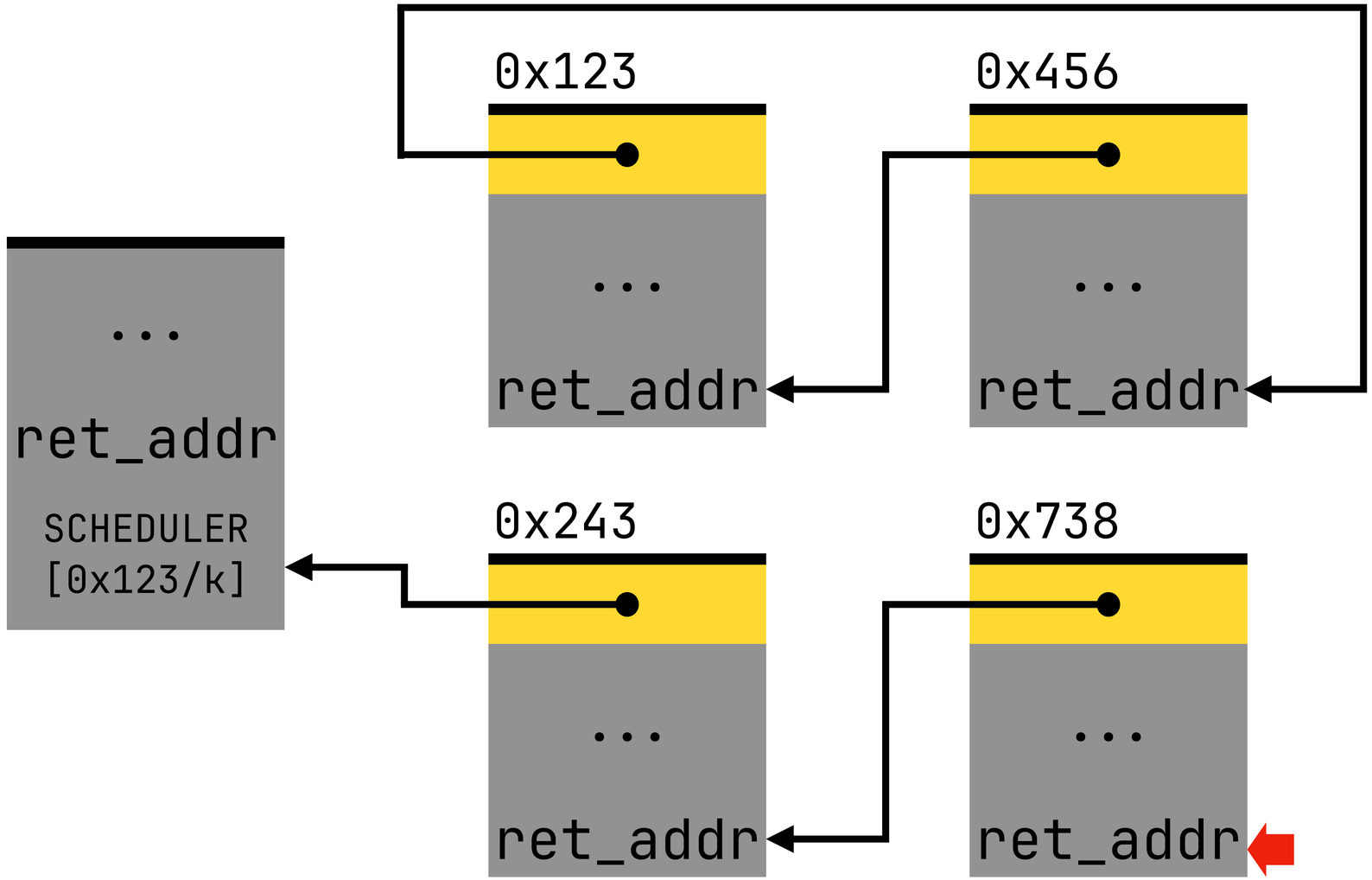



```
resume 0x123()
```

Direct Lexa

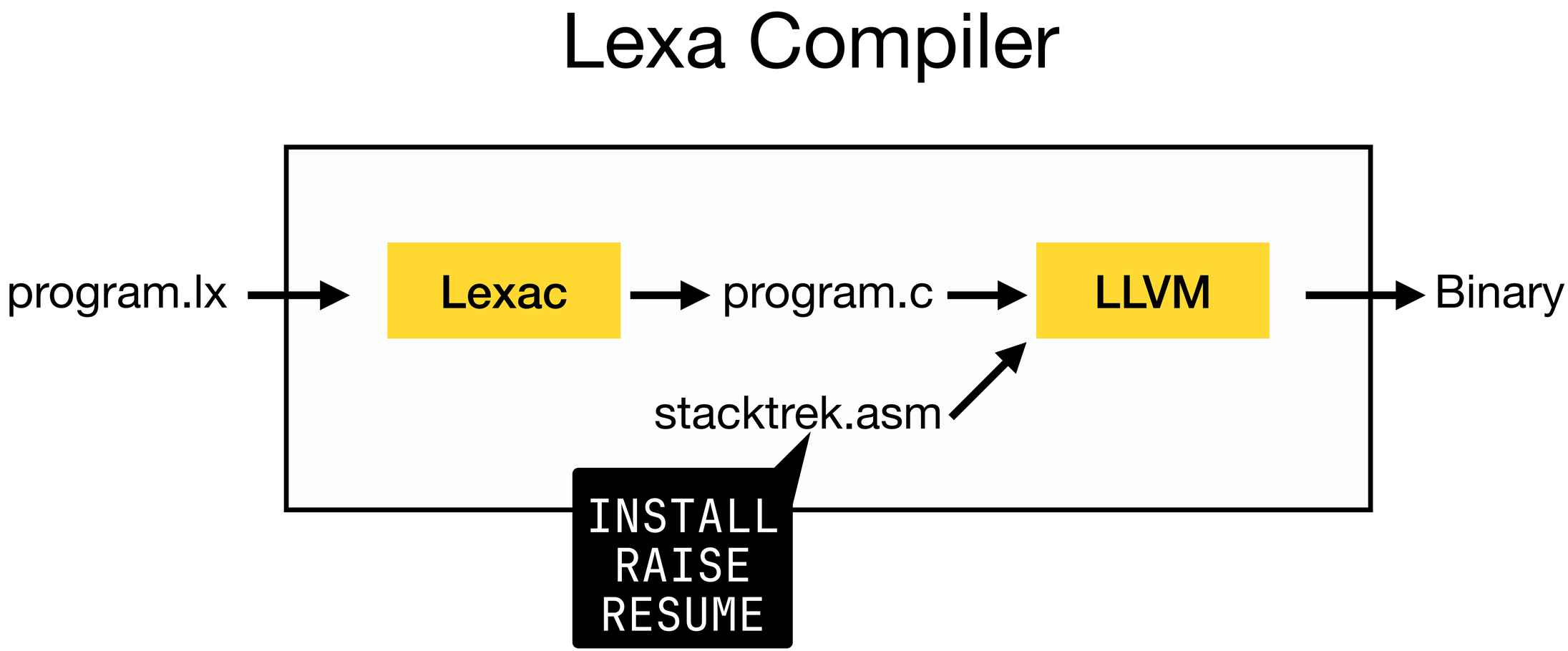
Stack switching

Running Example
The Full Picture



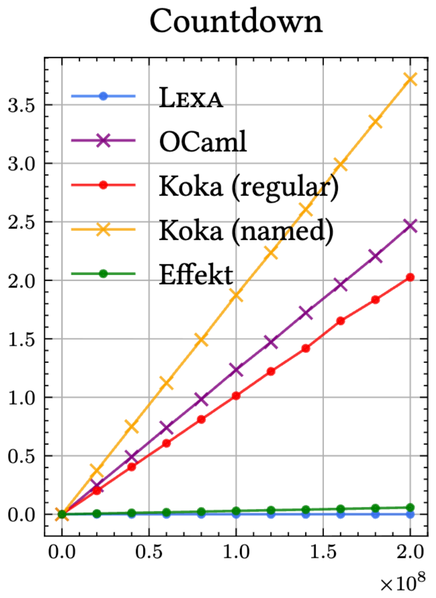
Lexa compiler implementation

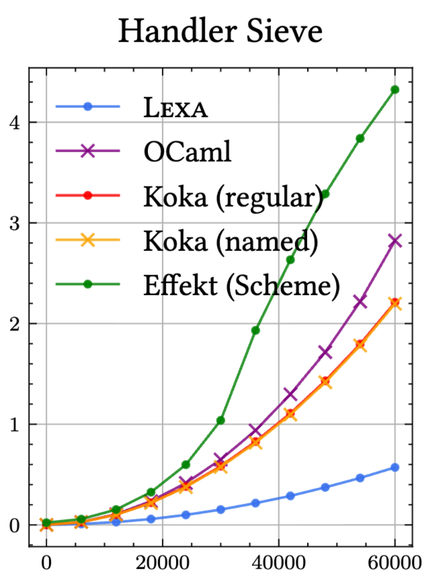
stacktrek Library

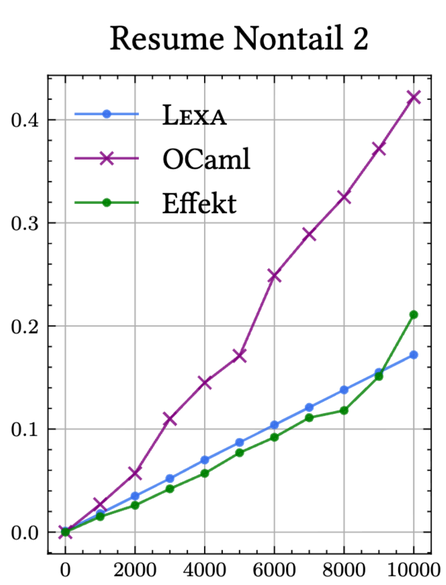


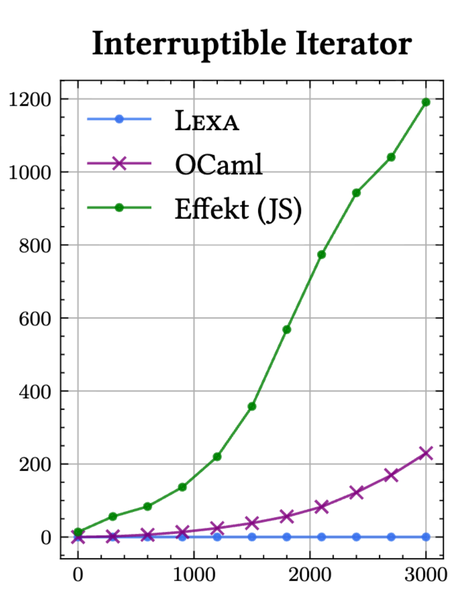
Lexa is fast

Selected benchmark results



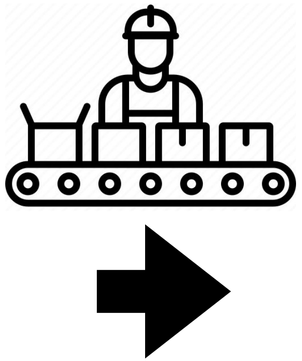






high-level effect handlers in Lexa

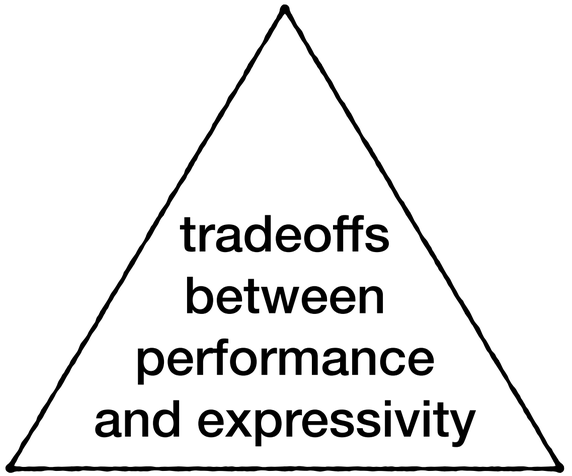
```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

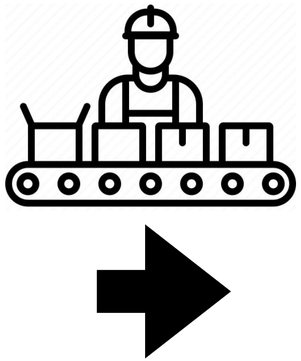


Multi Lexa

Zero Lexa

high-level effect handlers in Lexa

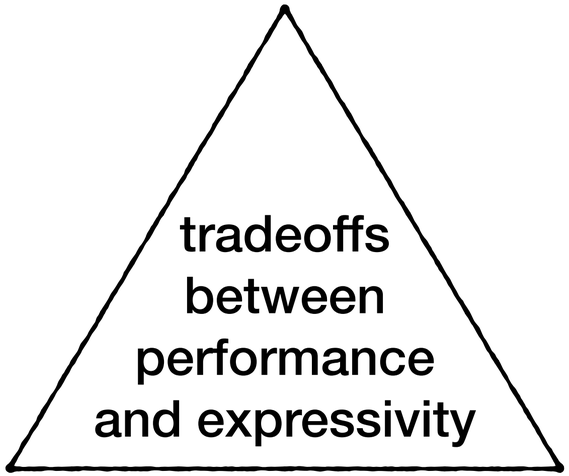
```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa



Multi Lexa

Zero Lexa

Multi Lexa

Multishot resumptions are expressive.

Multi Lexa

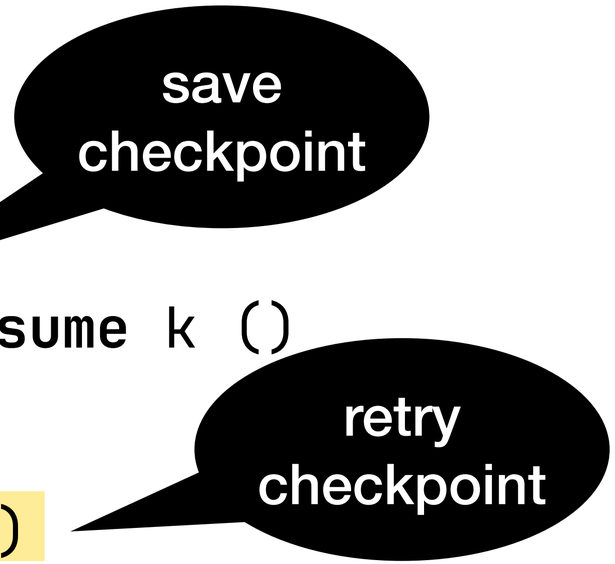
Multishot resumptions are expressive.

```
def checkpointing(program):
  var cp = None;
  handle
      program(save, retry)
  with save =
      λx,k. cp = Some(k); resume k ()
  with retry =
      match cp
      | Some k ⇒ resume k ()
      | None ⇒ checkpointing(program)
```

Multi Lexa

Multishot resumptions are expressive.

```
def checkpointing(program):
  var cp = None;
  handle
      program(save, retry)
  with save =
      λx,k. cp = Some(k); resume k ()
  with retry =
      match cp
      | Some k ⇒ resume k ()
      | None ⇒ checkpointing(program)
```



Multi Lexa

Direct Lexa does not support multishot resumptions.

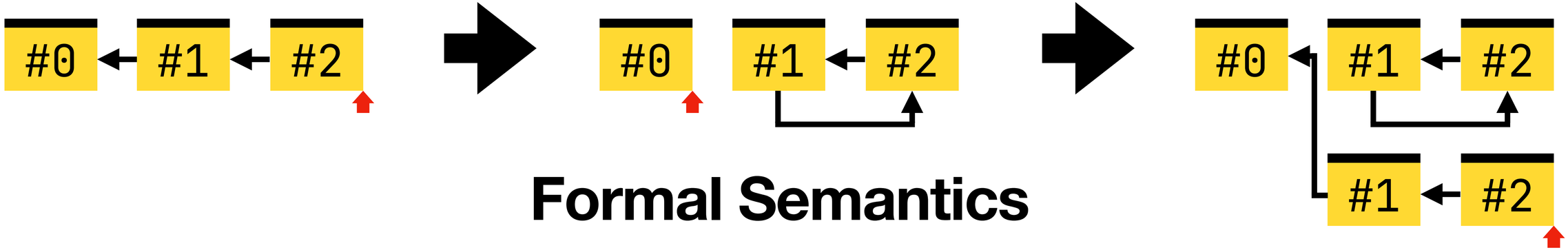
Direct Lexa's use of stack address comes back and bites us.

Formal Semantics

Direct Lexa Implementation

Multi Lexa

Direct Lexa does not support multishot resumptions.

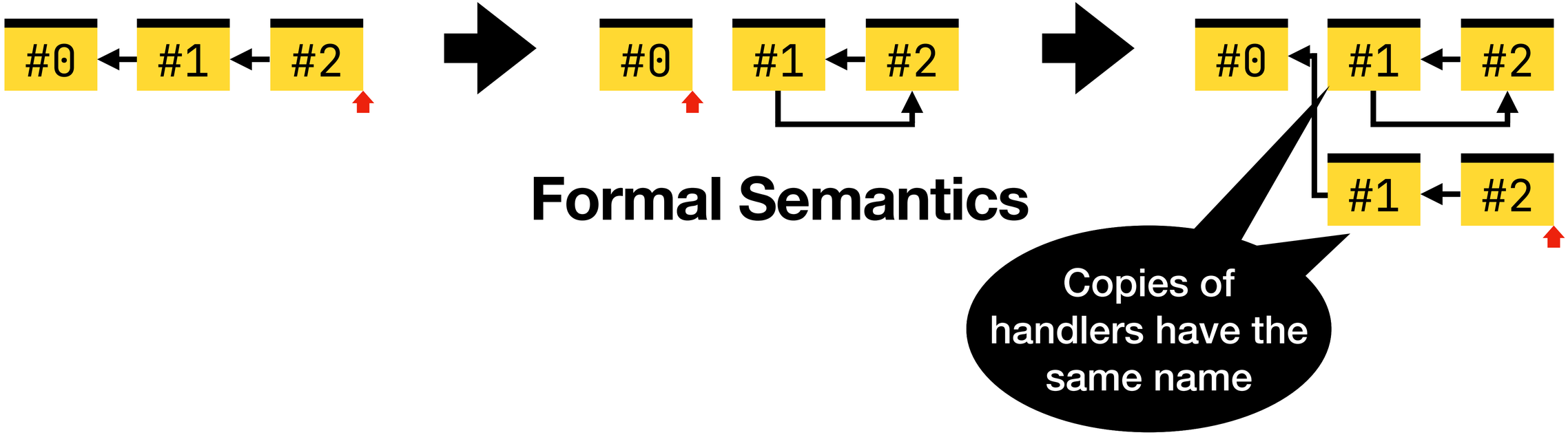


Formal Semantics

Direct Lexa Implementation

Multi Lexa

Direct Lexa does not support multishot resumptions.

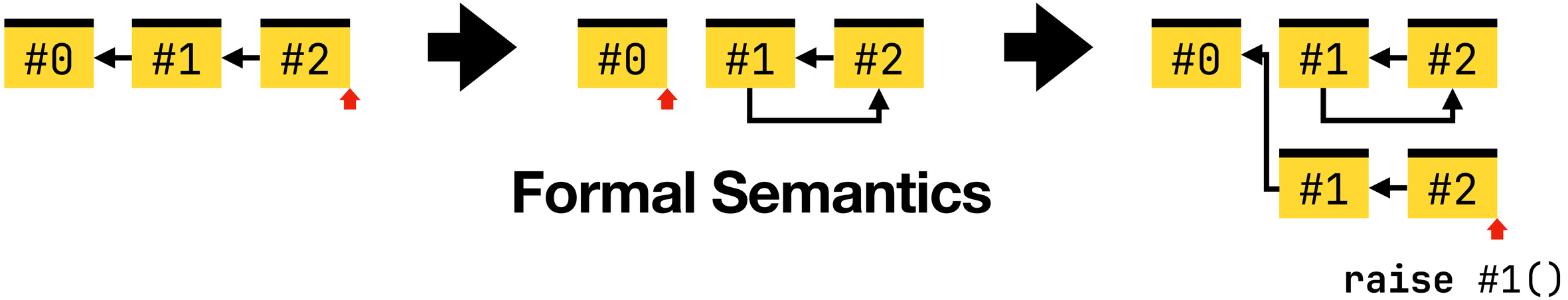


Formal Semantics

Direct Lexa Implementation

Multi Lexa

Direct Lexa does not support multishot resumptions.

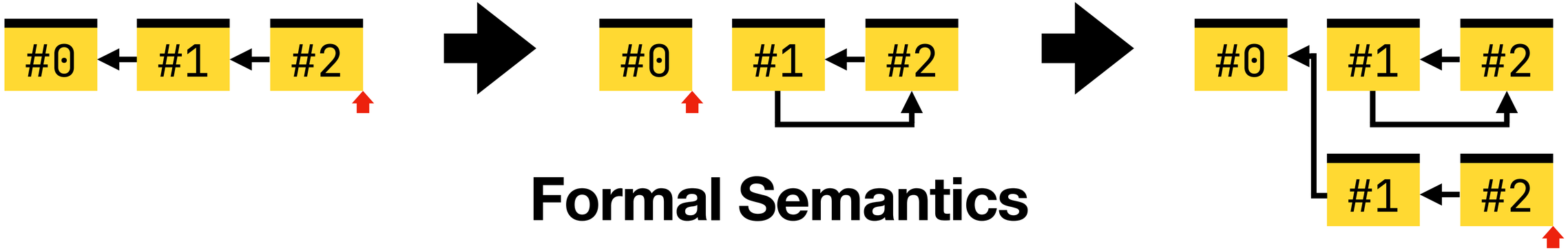


Formal Semantics

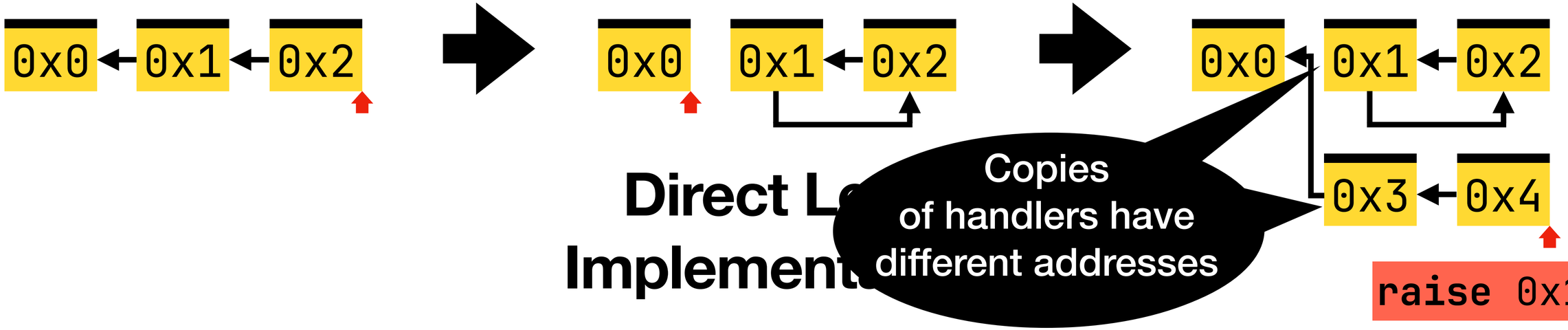
Direct Lexa Implementation

Multi Lexa

Direct Lexa does not support multishot resumptions.



Formal Semantics



Direct L
Implement

Multi Lexa

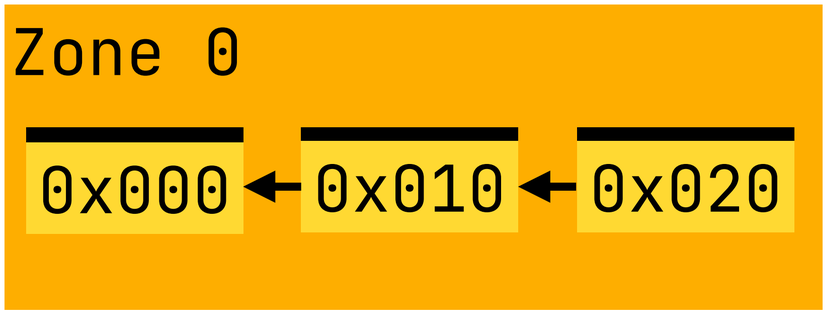
Main idea: virtual resumption

Multi Lexa

Main idea: virtual resumption

- Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

- Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

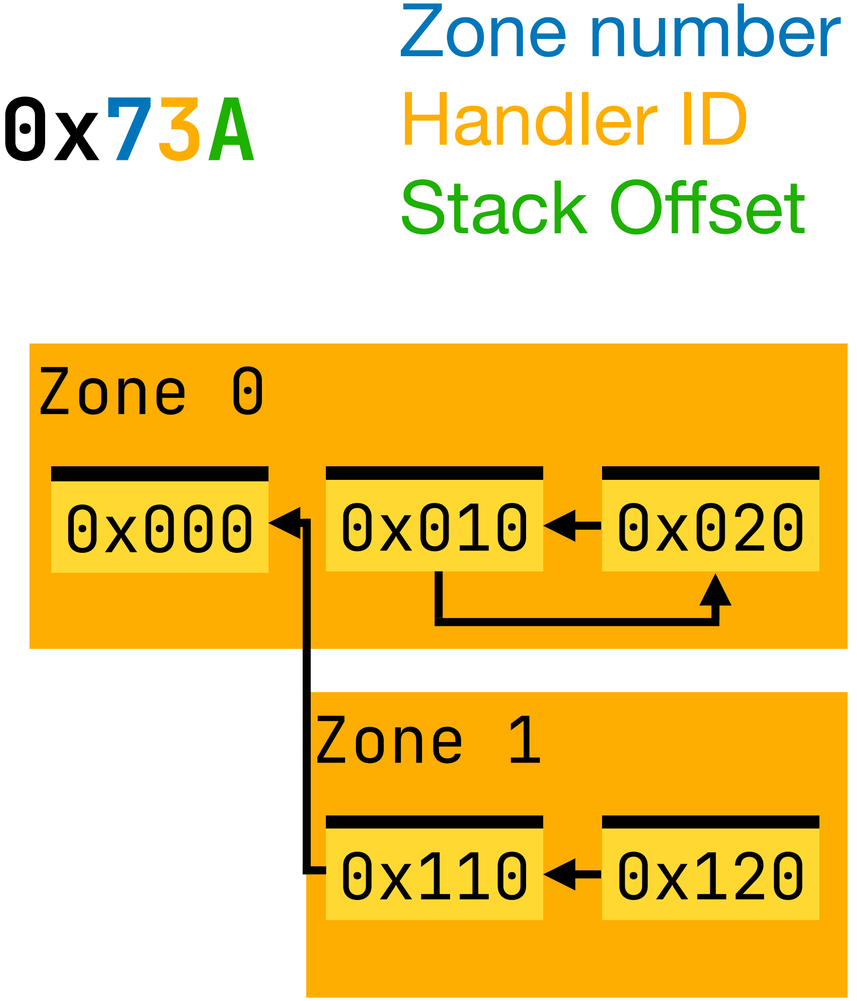


Multi Lexa

Main idea: virtual resumption

- Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

- Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones

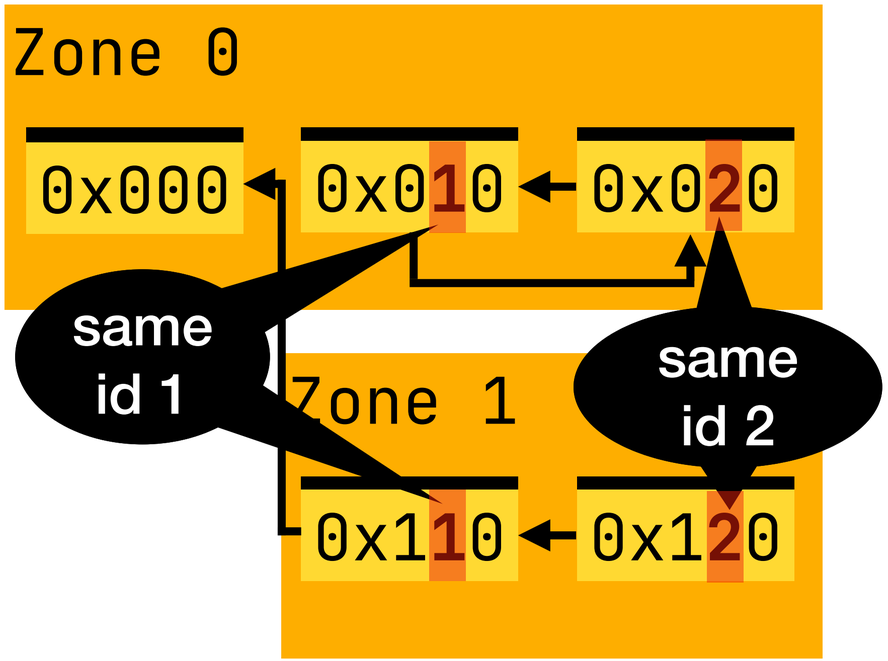


Multi Lexa

Main idea: virtual resumption

- Stacks and handlers live in virtual memory space, and all addresses are virtual addresses

- Each continuation lives in a different zone; copies of the same handlers in different continuation live at the same relative address within zones



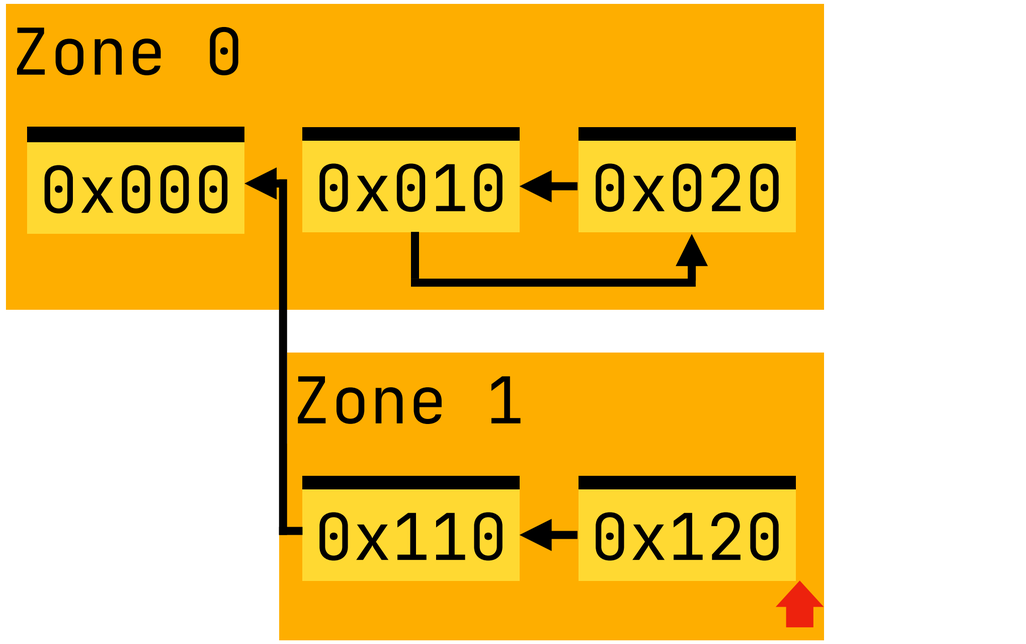
Multi Lexa

Main idea: virtual resumption

0x73A

Zone number
Handler ID
Stack Offset

- A software-level memory management unit(**sMMU**) is in charge of memory allocation and address translation



raise 0x020()

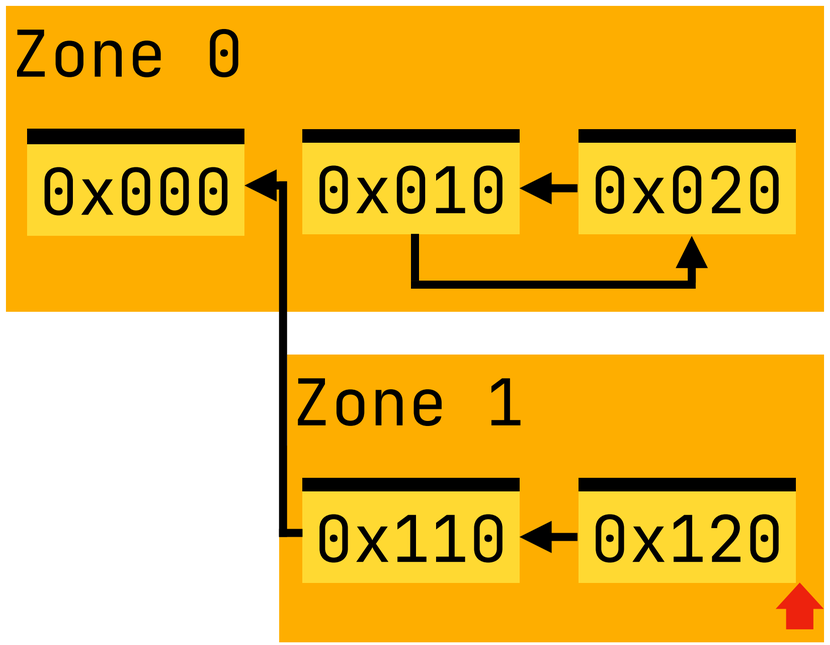
Multi Lexa

Main idea: virtual resumption

`0x73A`

Zone number
Handler ID
Stack Offset

- A software-level memory management unit(**sMMU**) is in charge of memory allocation and address translation



Computed from sp

sMMU:
1. Start from Zone 1, find handler with ID 2

Computed from handler address

`raise 0x020()`

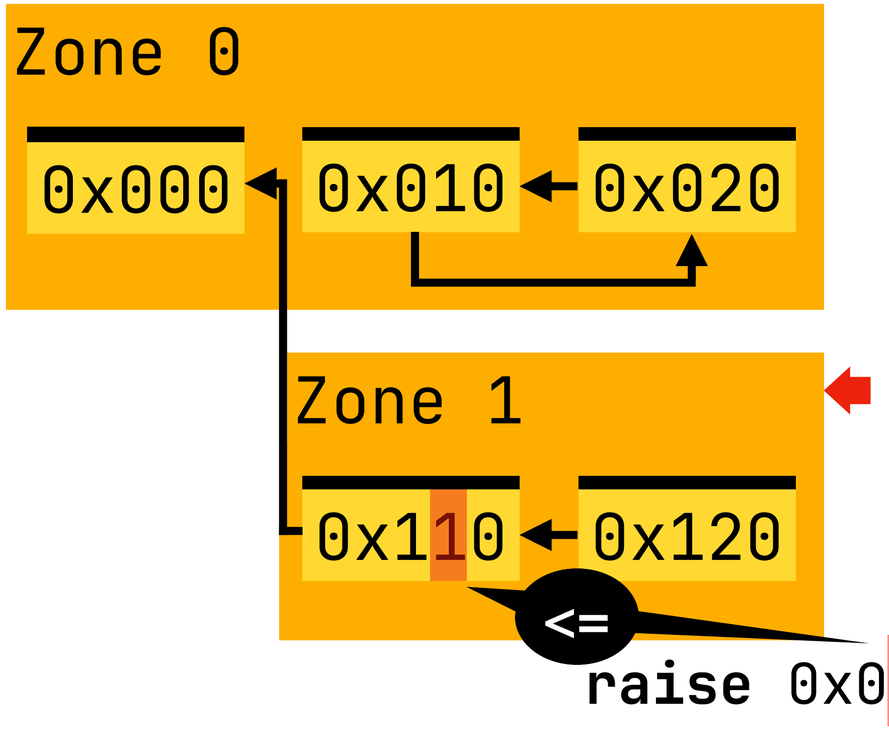
Multi Lexa

Main idea: virtual resumption

0x73A

Zone number
Handler ID
Stack Offset

- A software-level memory management unit(**sMMU**) is in charge of memory allocation and address translation



sMMU:
1. Start from Zone 1, find handler with ID 2
2. Zone walk: check if Zone 1 has Handler 2

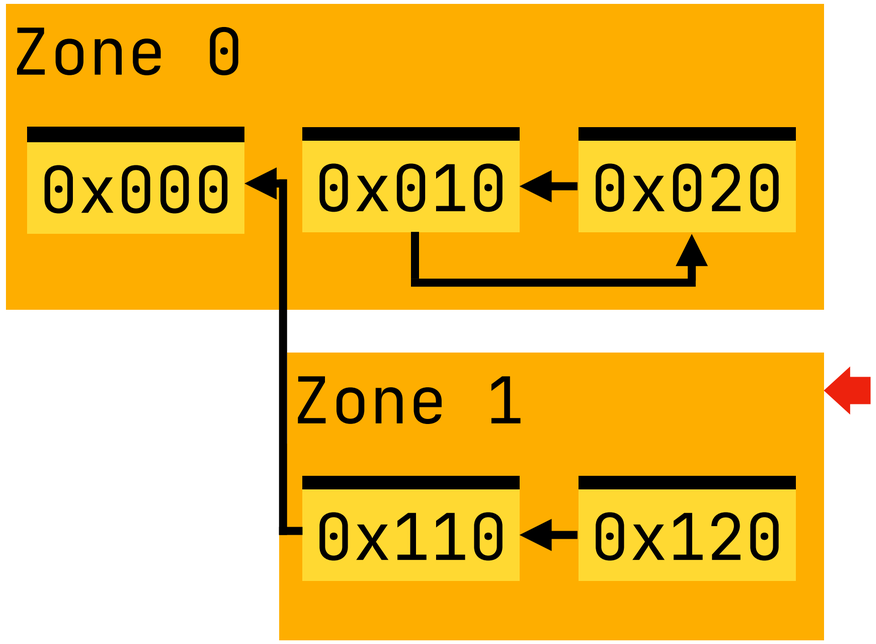
Multi Lexa

Main idea: virtual resumption

`0x73A`

Zone number
Handler ID
Stack Offset

- A software-level memory management unit(**sMMU**) is in charge of memory allocation and address translation



sMMU:
1. Start from Zone 1, find handler with ID 2
2. Zone walk: check if Zone 1 has Handler 2
3. Yes. Modify the zone number in virtual address to the target zone.

`0x020 → 0x120`

raise `0x020()`

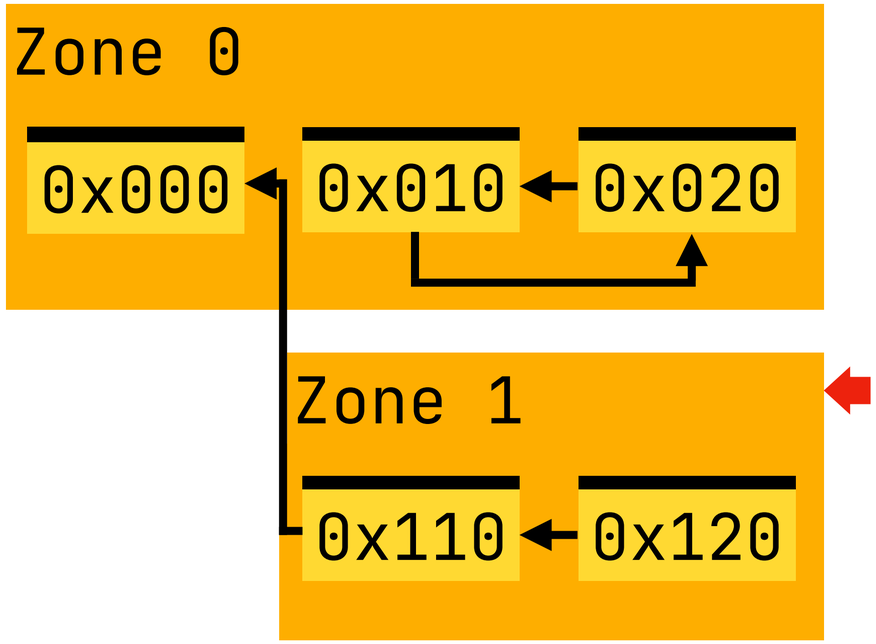
Multi Lexa

Main idea: virtual resumption

`0x73A`

Zone number
Handler ID
Stack Offset

- A software-level memory management unit(**sMMU**) is in charge of memory allocation and address translation

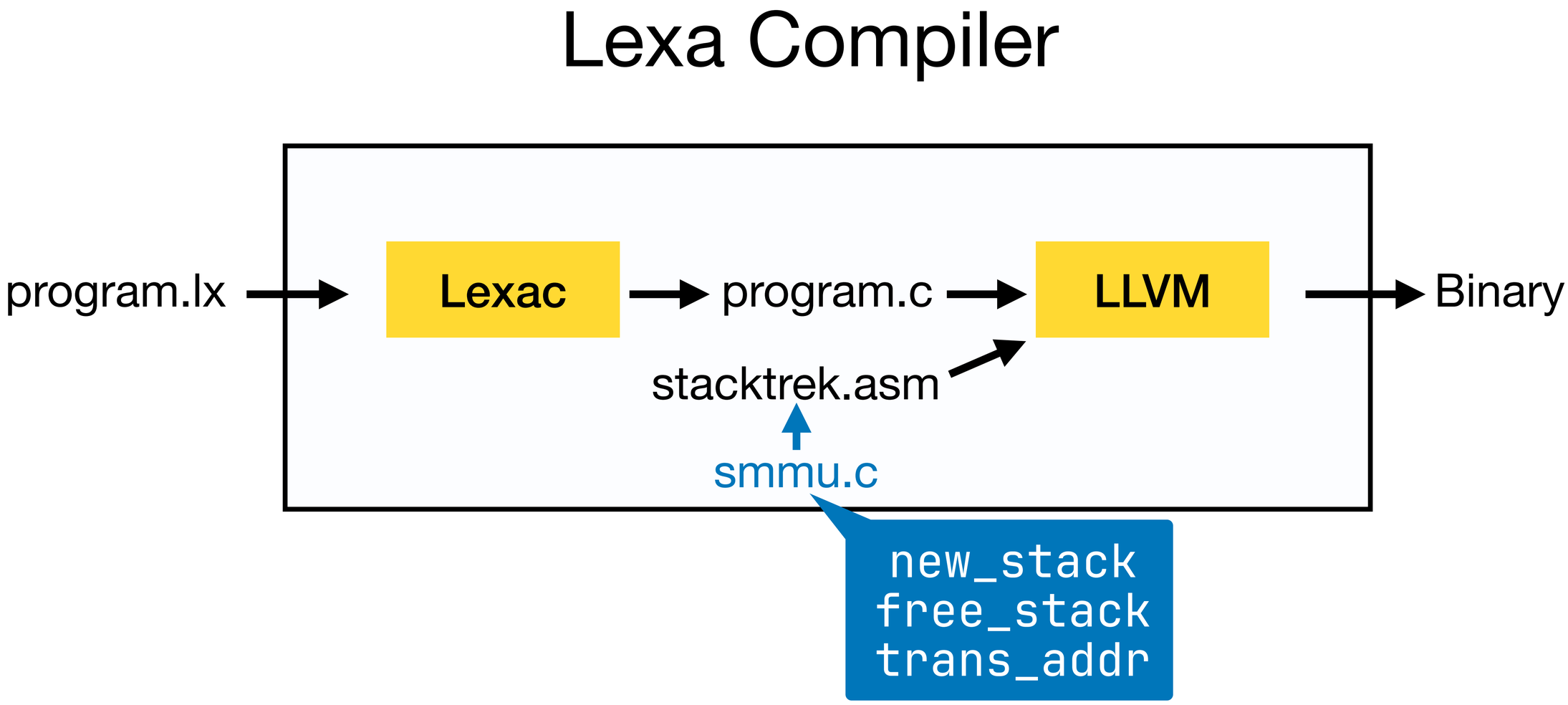


sMMU:
1. Start from Zone 1, find handler with ID 2
2. Zone walk: check if Zone 1 has Handler 2
3. Yes. Modify the zone number in virtual address to the target zone.
4. Return 0x120

```
raise 0x020()
```

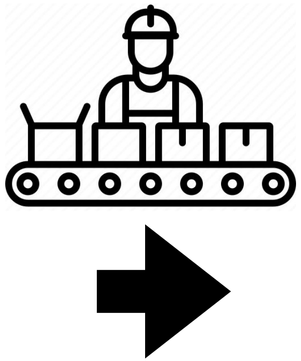
Multi Lexa

Library sMMU



high-level effect handlers in Lexa

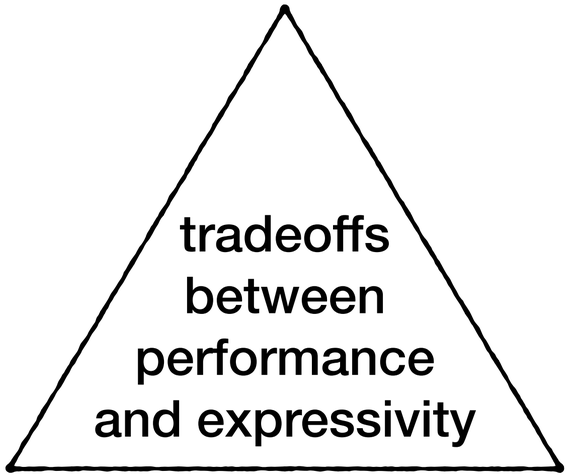
```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa



Multi Lexa

Zero Lexa

Effect handler

A powerful language feature

Language Definition

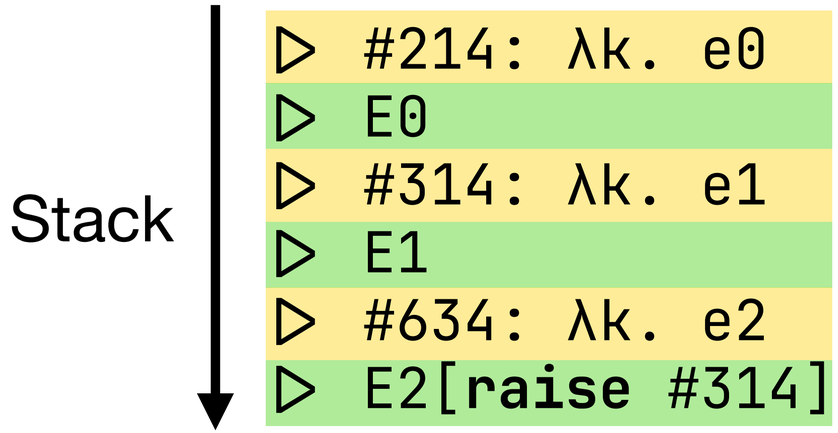
Syntax + **Semantics**

active research

```
handle E with H      Install a handler
raise ...            Find handler; Capture continuation
resume ...           Resume continuation
```

A glimpse of two semantics

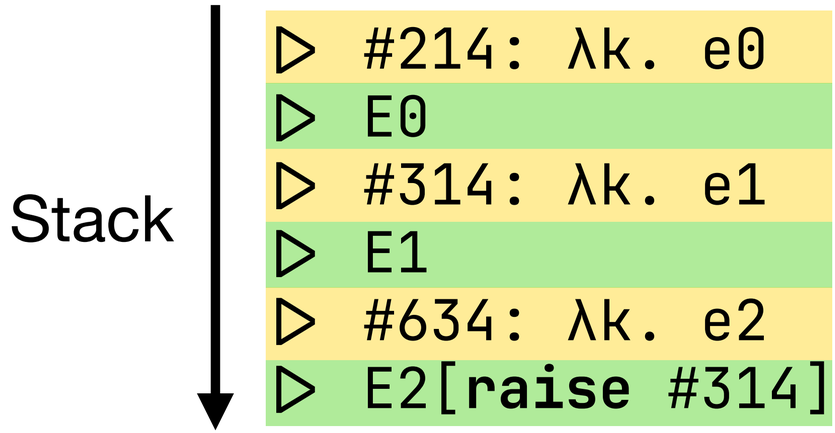
Lexical Semantics



A glimpse of two semantics

```
def f(n, exception_handler) =
  ...
  if (condition)
    raise exception_handler(...);
  ...
```

Lexical Semantics



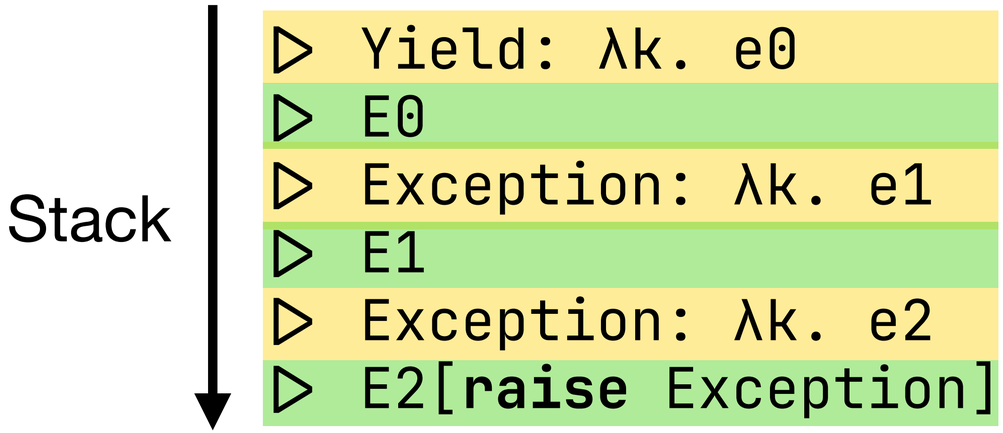
A glimpse of two semantics

```
def f(n) =
  ...
  if (condition)
    raise Exception(...);
  ...
```

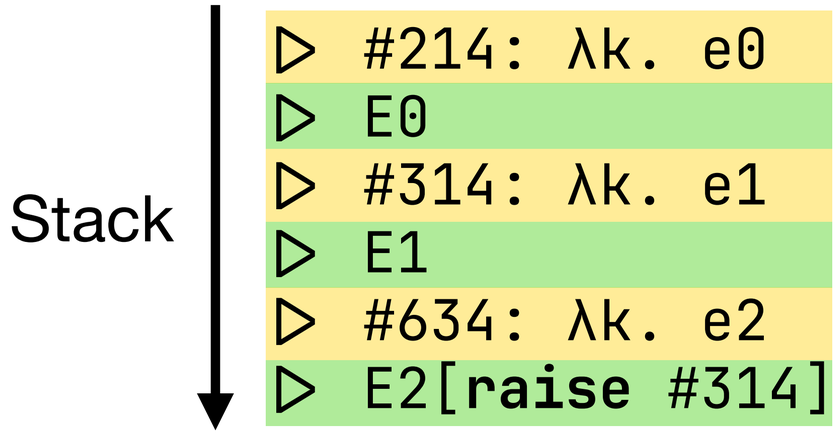
V.S.

```
def f(n, exception_handler) =
  ...
  if (condition)
    raise exception_handler(...);
  ...
```

Traditional Semantics



Lexical Semantics



Problem

All existing implementations impose a runtime cost


```
def f(n) =
  ...
  if (condition)
    raise Exception(...);
  ...
```

Traditional Semantics

V.S.

```
def f(n, exception_handler) =
  ...
  if (condition)
    raise exception_handler(...);
  ...
```

Lexical Semantics

Problem

All existing implementations impose a runtime cost

extra parameters

```
def f(n) =
  ...
  if (condition)
    raise Exception(...);
  ...
```

Traditional Semantics

V.S.

```
def f(n, exception_handler) =
  ...
  if (condition)
    raise exception_handler(...);
  ...
```

Lexical Semantics

Programmers are used to the zero-overhead handlers.

Problem

All existing implementations impose a runtime cost

extra parameters

```
def f(n) =
  ...
  if (condition)
    raise Exception(...);
  ...
```

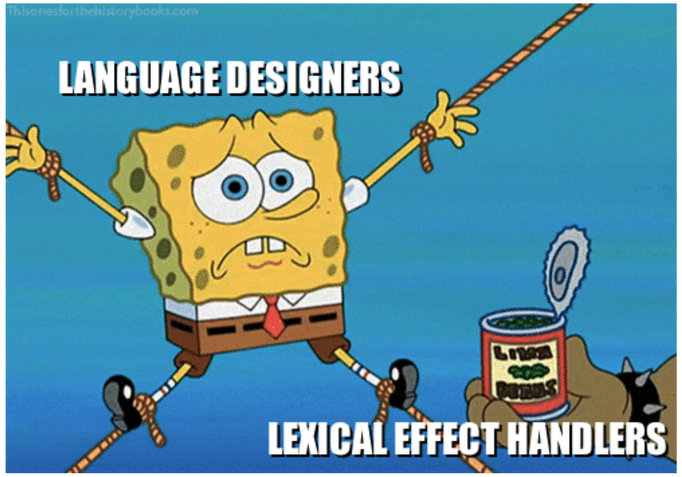
Traditional Semantics

V.S.

```
def f(n, exception_handler) =
  ...
  if (condition)
    raise exception_handler(...);
  ...
```

Lexical Semantics

Programmers are used to the zero-overhead handlers.



It’s a hard sell.

Zero Lexa

Zero-overhead Principle

“What you don’t use, you don’t pay for.”

- Bjarne Stroustrup [1995, 2012]

“Normal case execution efficiency should not be impaired at all.”

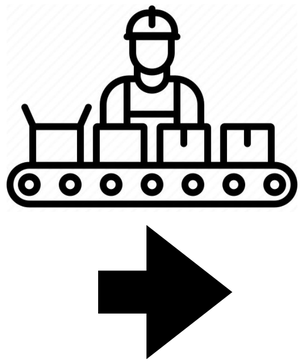
- Atkinson, Liskov, and Scheifler [1978]

- Liskov and Snyder [1979]

Zero Lexa

Zero-overhead Lexical effect handler

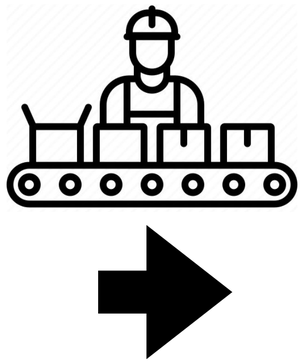
```
def f(n, handler) =
  ...
  if (condition)
    raise handler
  ...
```



Zero Lexa

Zero-overhead Lexical effect handler

```
def f(n, handler) =
  ...
  if (condition)
    raise handler
  ...
```



```
def f(n) =
  ...
  if (condition)
    raise _
  ...
```

Compile away handler passing, yet

retain the same behavior

Zero Lexa

Zero-overhead Lexical effect handler

We will first see a non-zero-overhead execution.

We will then figure out how to make it zero-overhead.

Zero Lexa, Example 1

```
val parked = Null
def work() {
  handle
    while true
      DO_SOMETING
      raise yield()
  with yield(k) =
    val peer = parked
    parked = k
    if peer then
      resume peer ()
}
work()
work()
```

Zero Lexa, Example 1

```
def work() {
  handle
    DO_SOMETING
  with yield = ...
}
```

Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}
```

Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...
```

Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```

Zero Lexa, Example 1

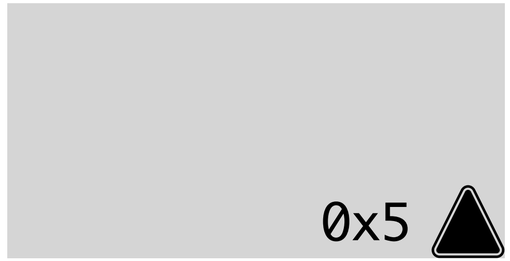
```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```

work:

Zero Lexa, Example 1

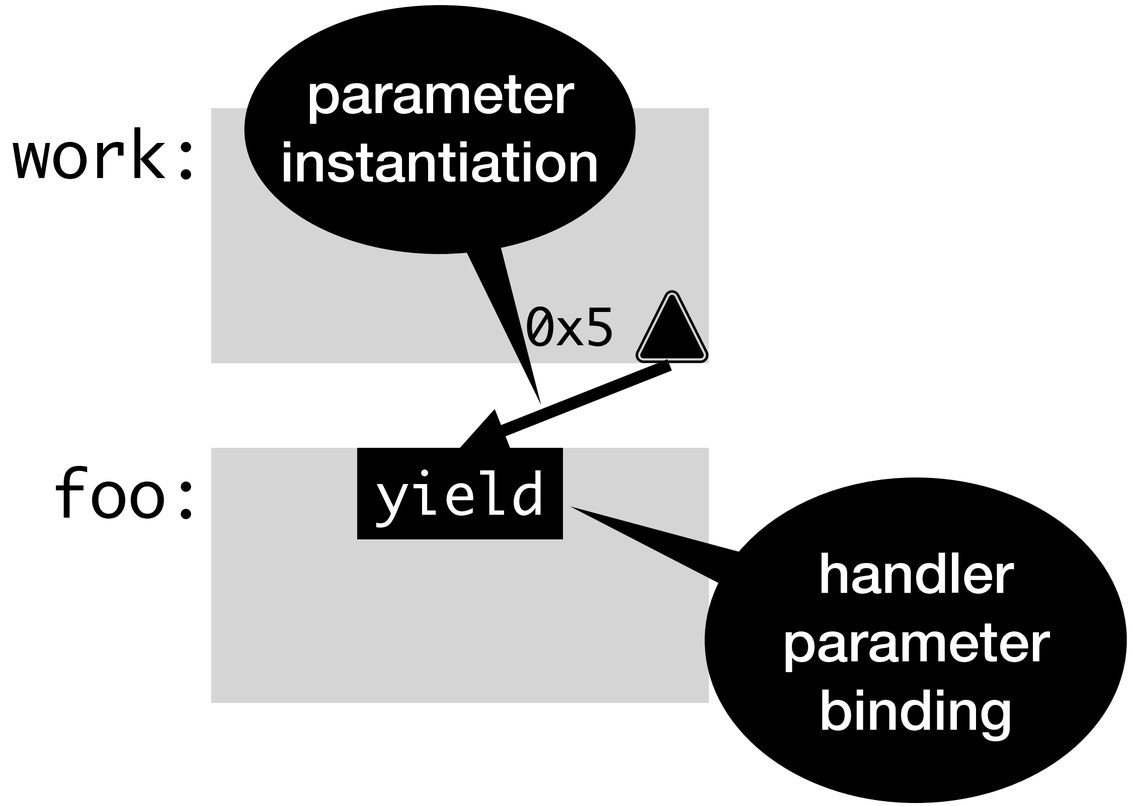
```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```

work:



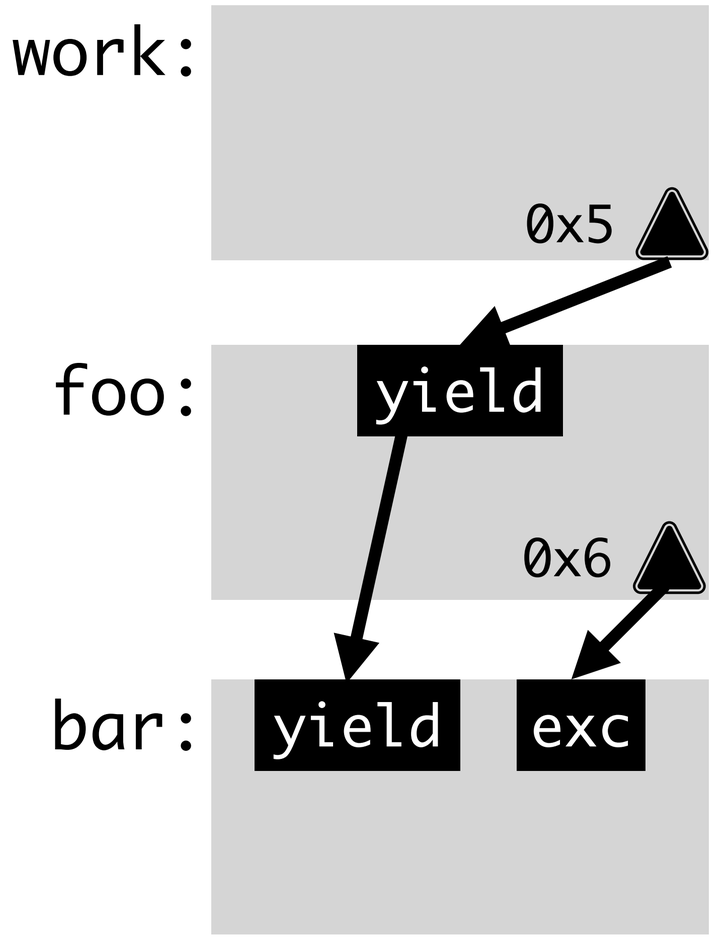
Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



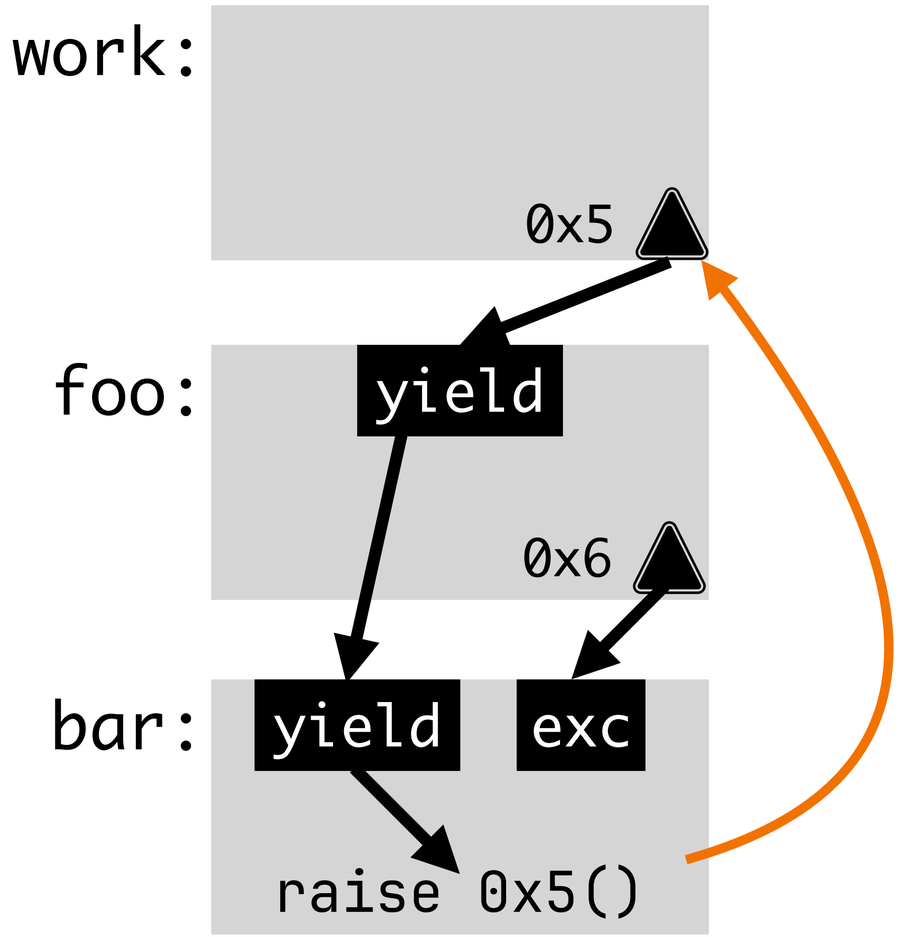
Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



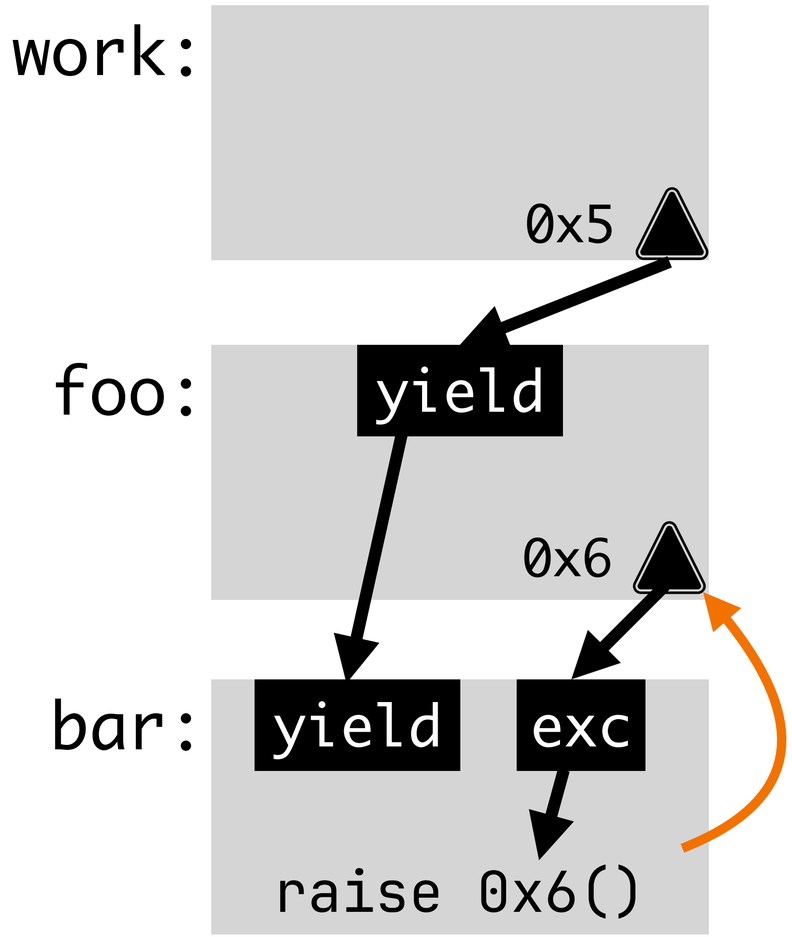
Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



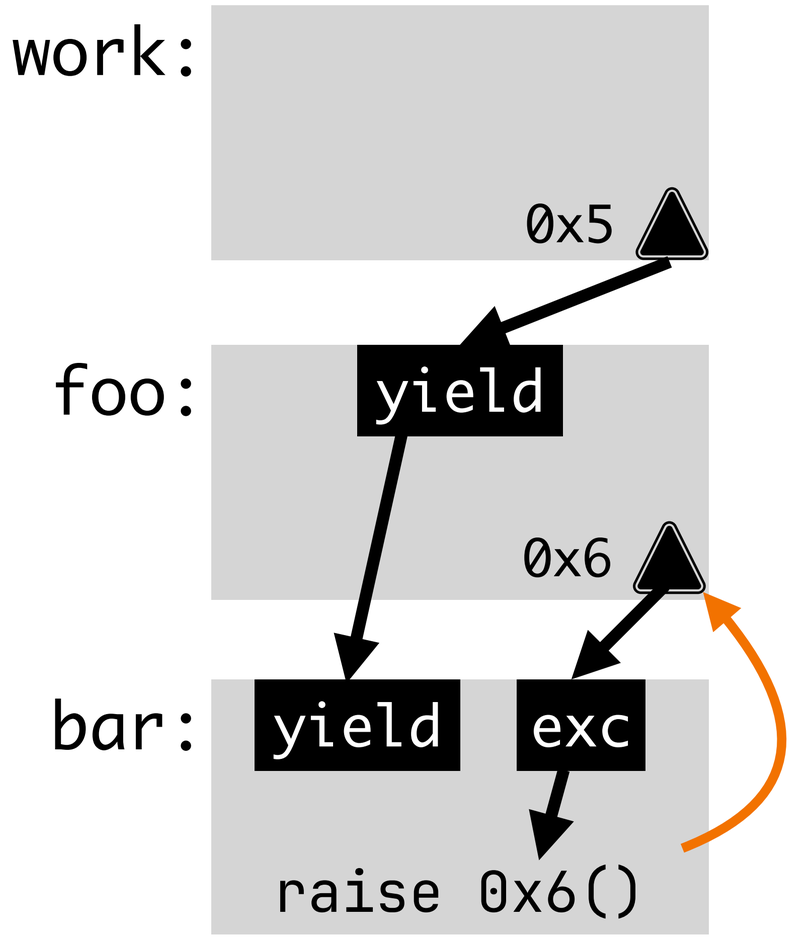
Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



Zero Lexa, Example 1

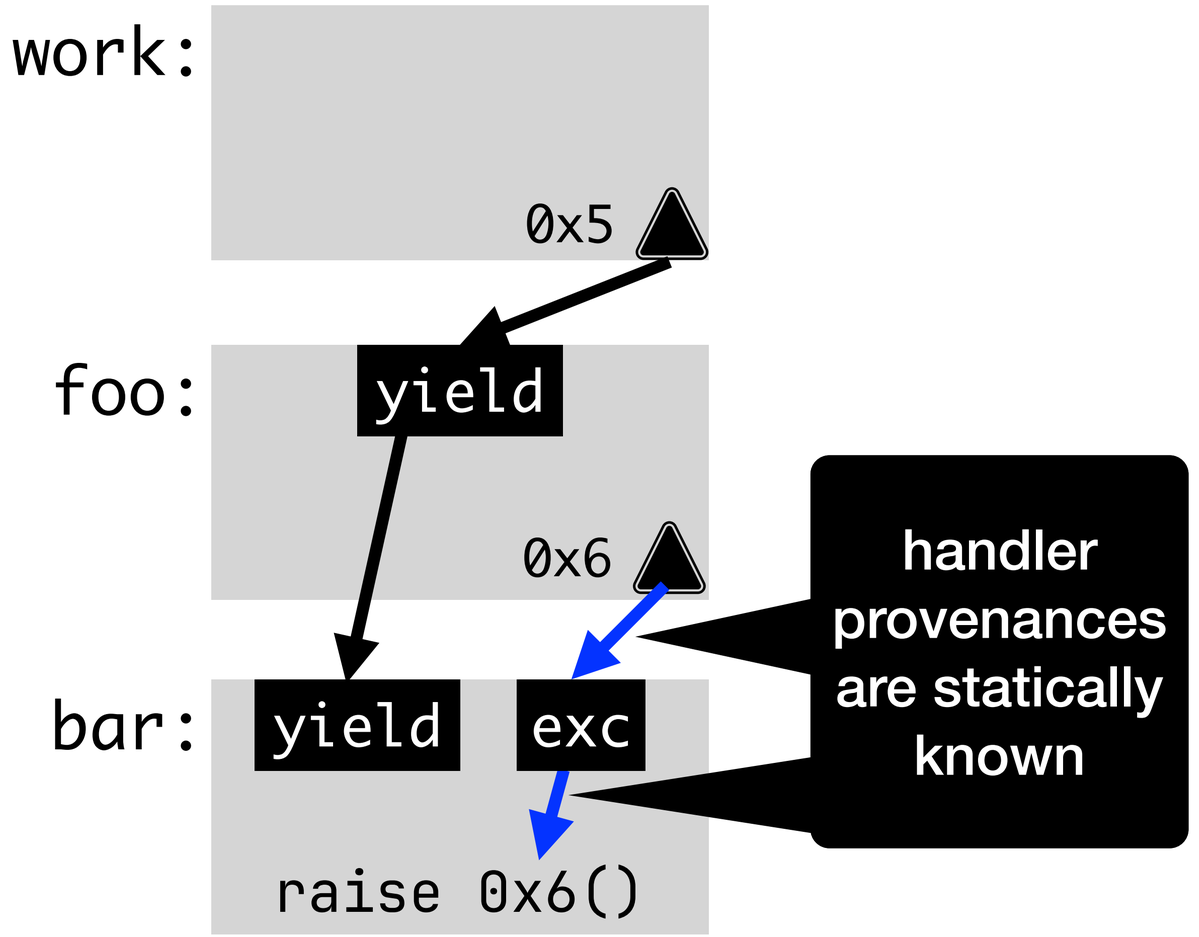
```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



Can we locate the intended handler without passing down addresses?

Zero Lexa, Example 1

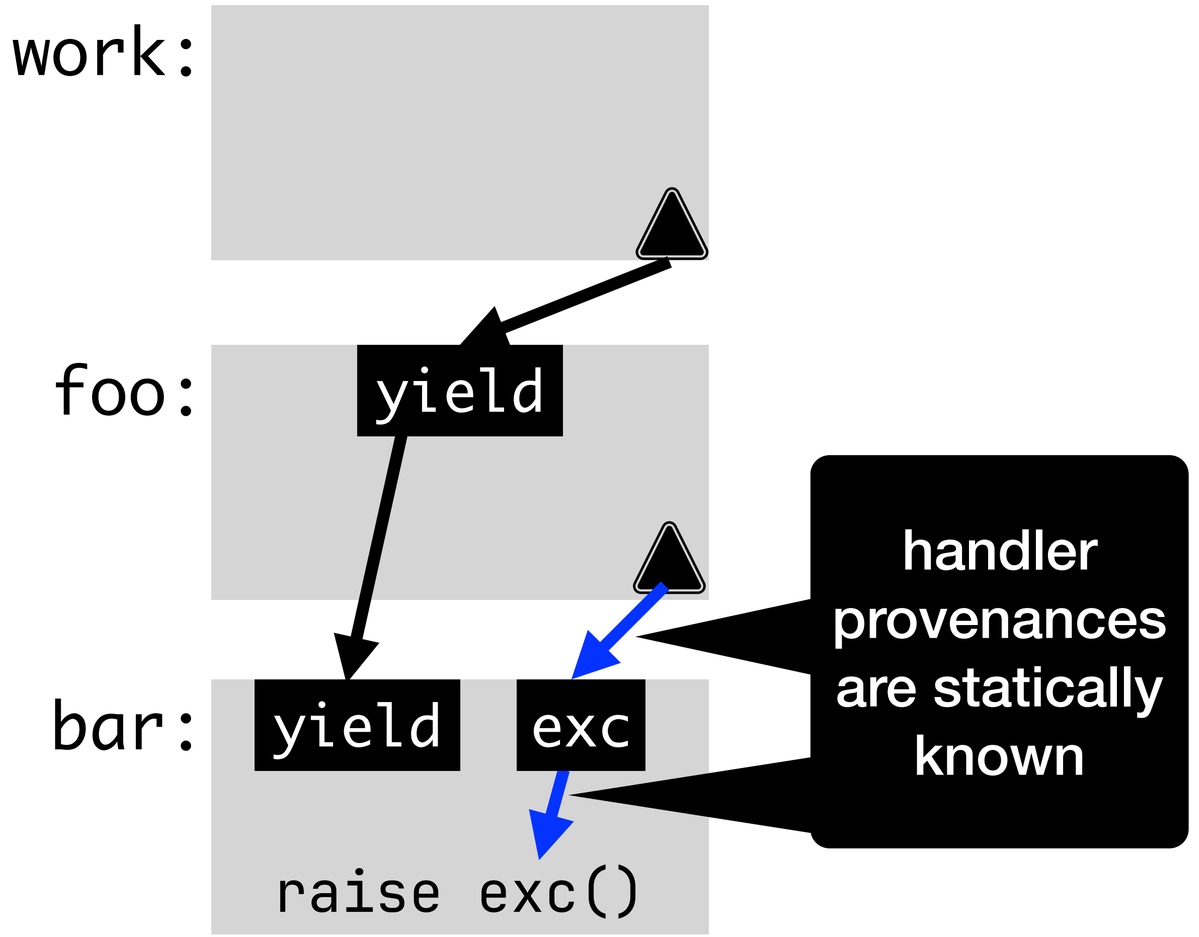
```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



Can we locate the intended handler without passing down addresses?

Zero Lexa, Example 1

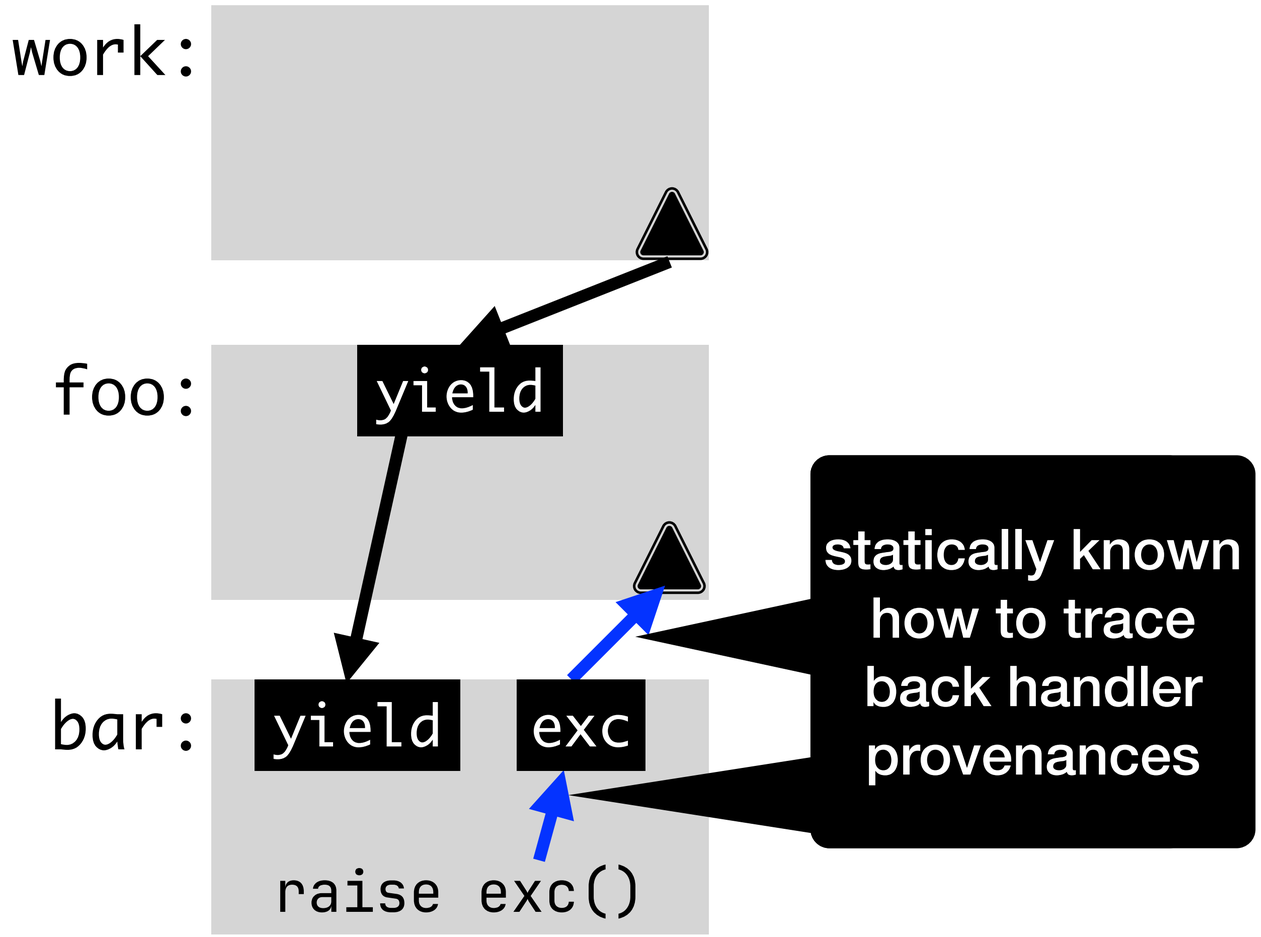
```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



Can we locate the intended handler without passing down addresses?

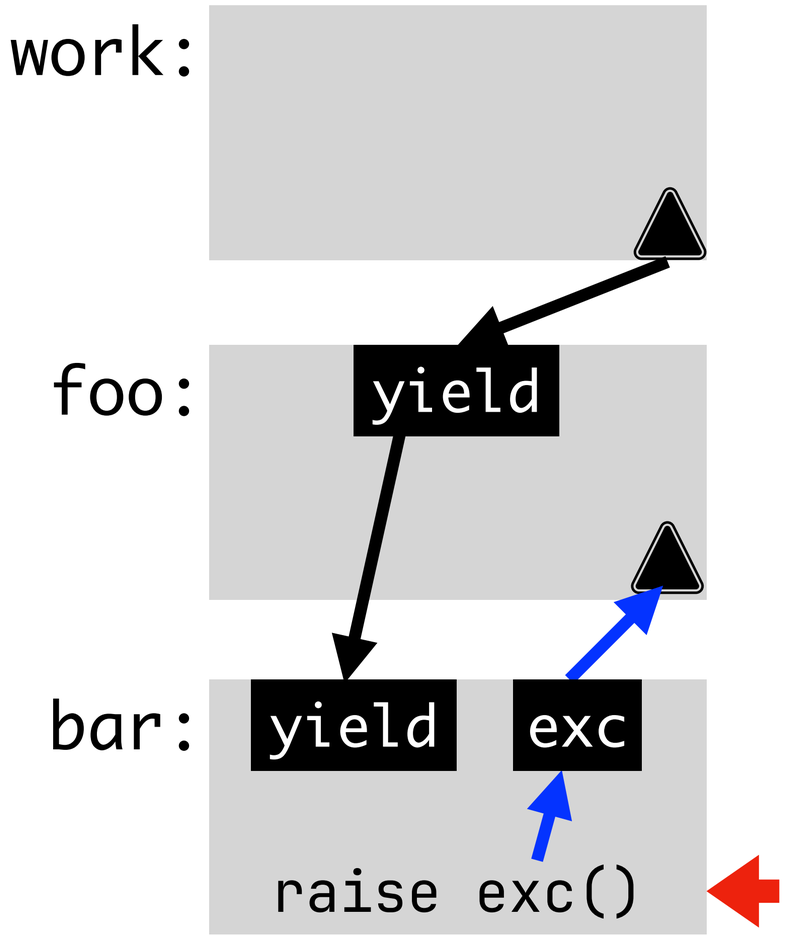
Zero Lexa, Example 1

```
def work() {
  handle
    foo(yield)
  with yield = ...
}

def foo(yield):
  handle
    bar(yield, exc)
  with exc = ...

def bar(yield, exc):
  raise yield()
  raise exc()
```



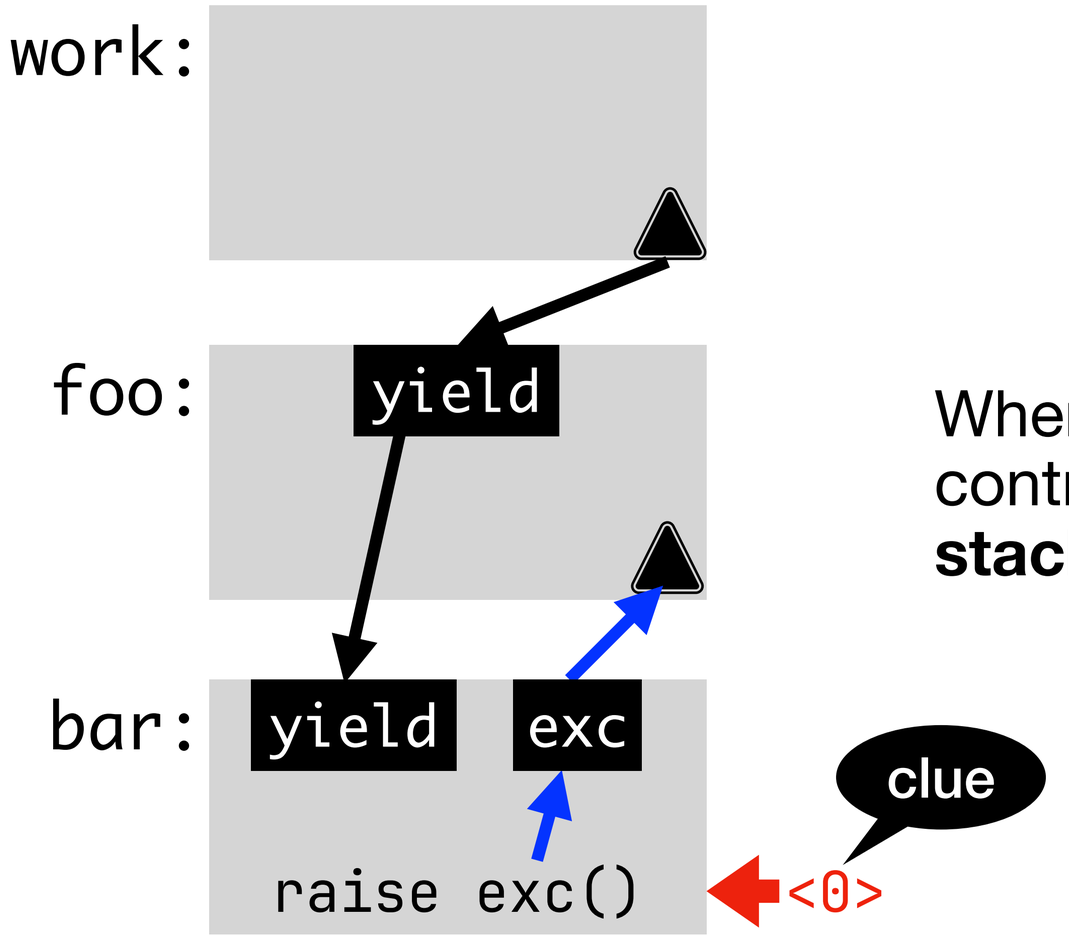
Can we locate the intended handler without passing down addresses?

Zero Lexa, Example 1



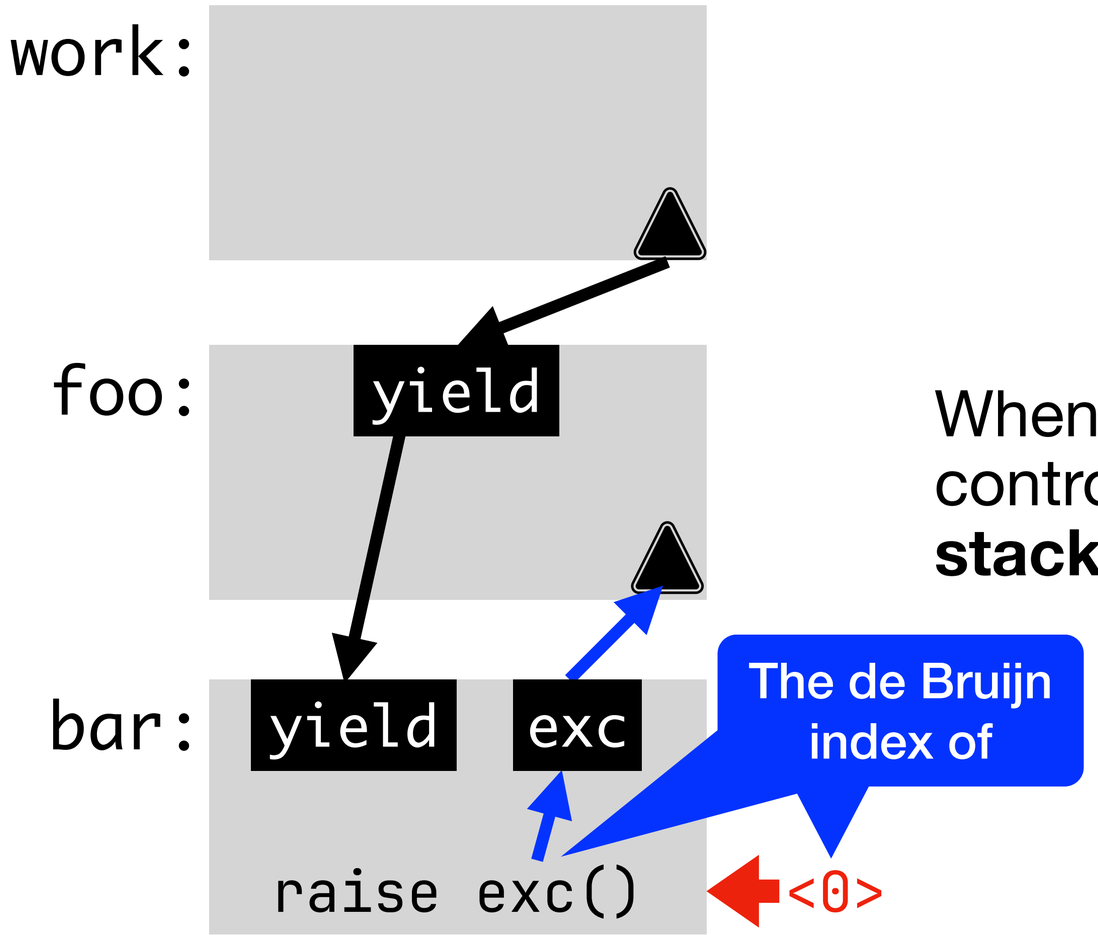
When an effect is raised, control goes to a runtime **stack-walker**.

Zero Lexa, Example 1



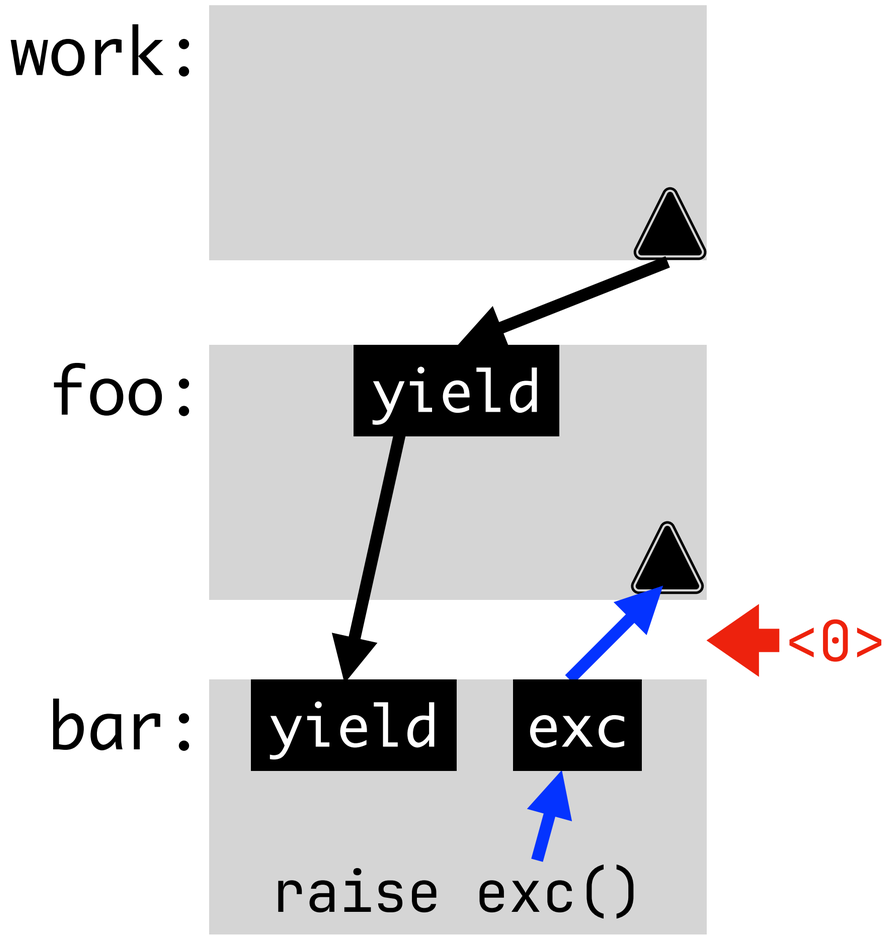
When an effect is raised, control goes to a runtime **stack-walker**.

Zero Lexa, Example 1

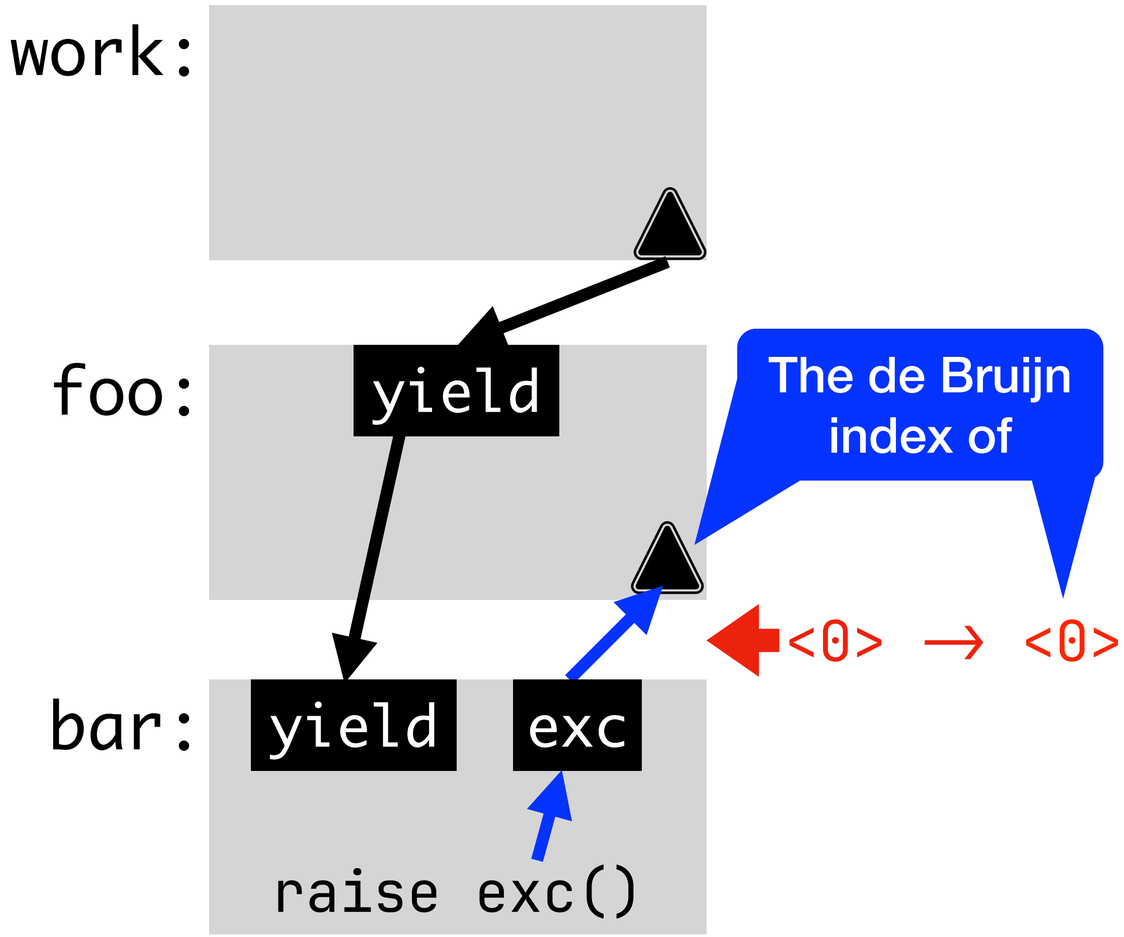


When an effect is raised, control goes to a runtime **stack-walker**.

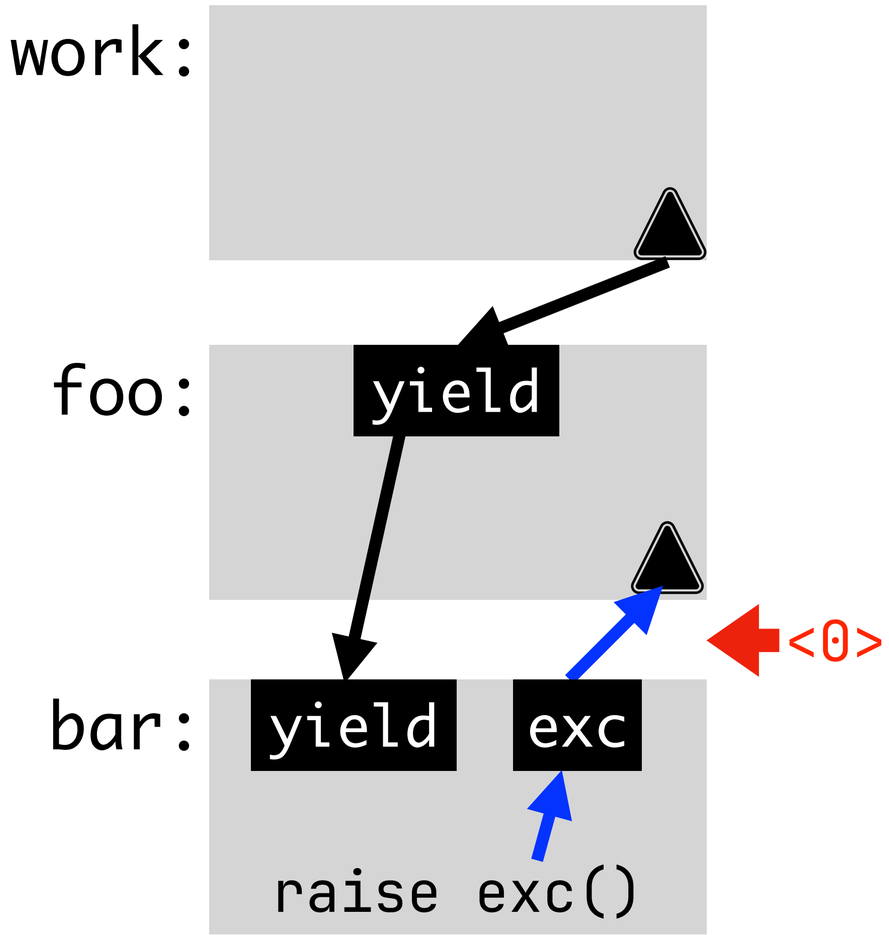
Zero Lexa, Example 1



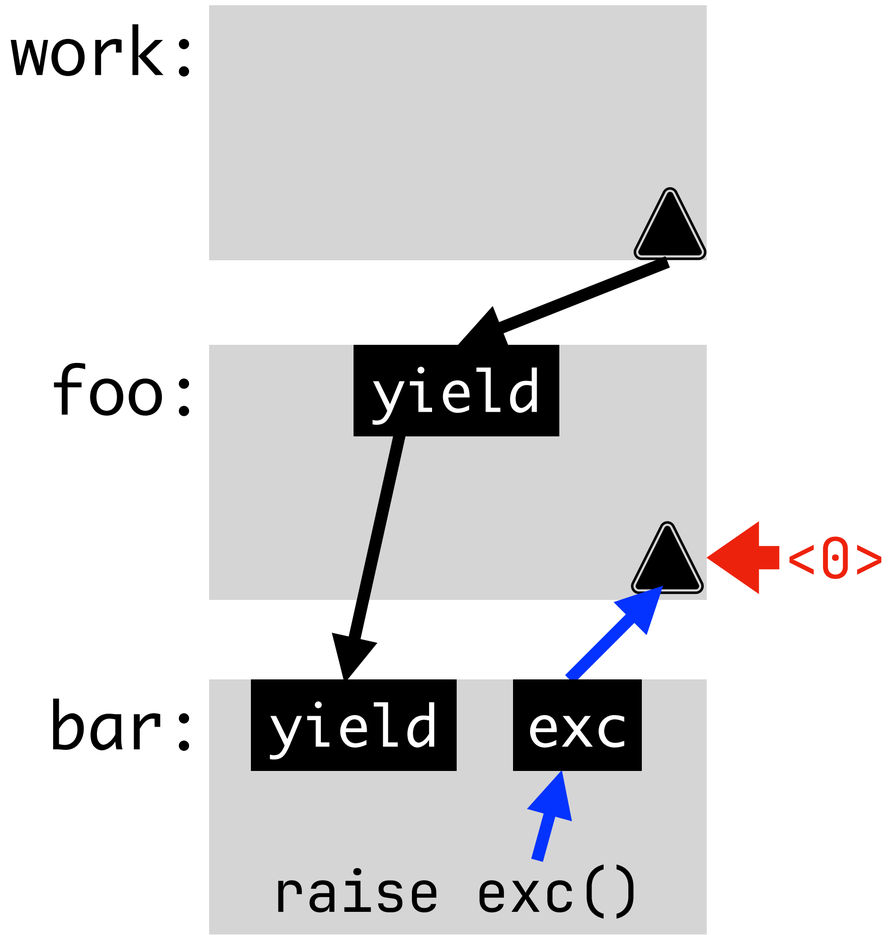
Zero Lexa, Example 1



Zero Lexa, Example 1



Zero Lexa, Example 1



Zero Lexa, Example 2

Our compilation can also deal with higher-order functions.

Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```

Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```

Zero Lexa, Example 2

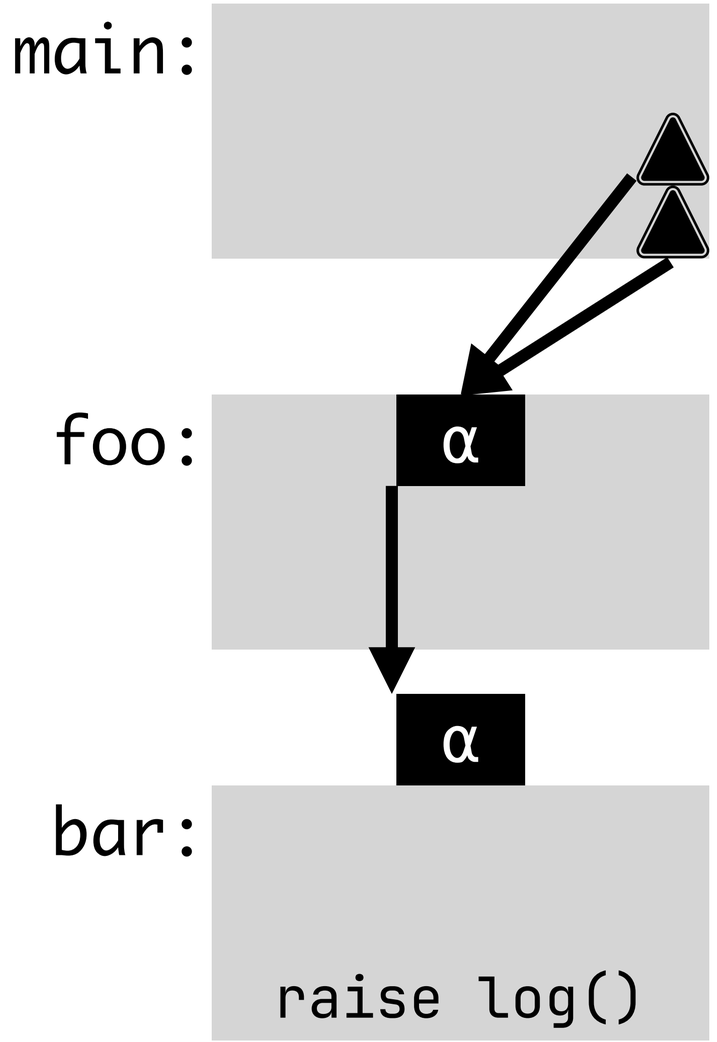
```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```

Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```

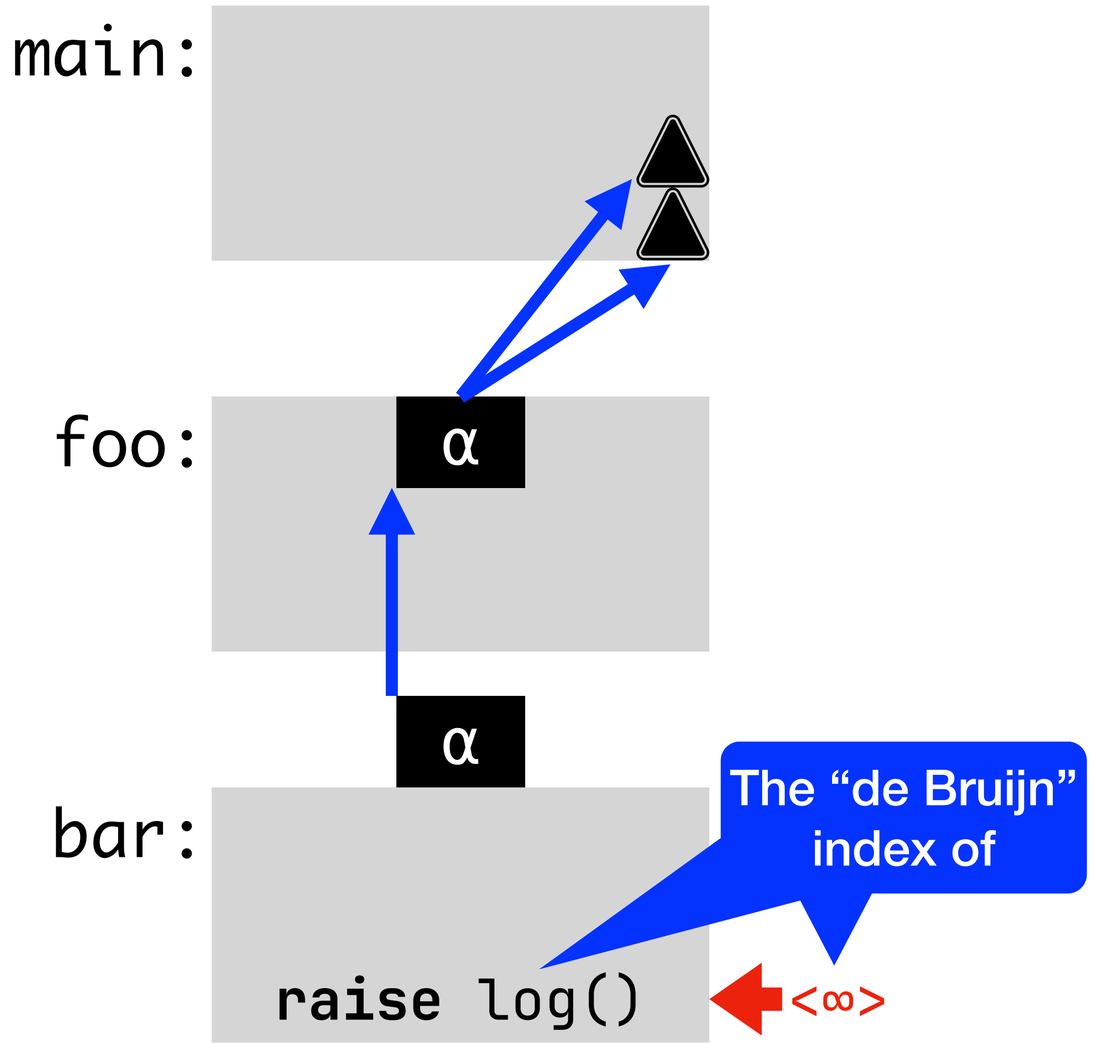
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



Zero Lexa, Example 2

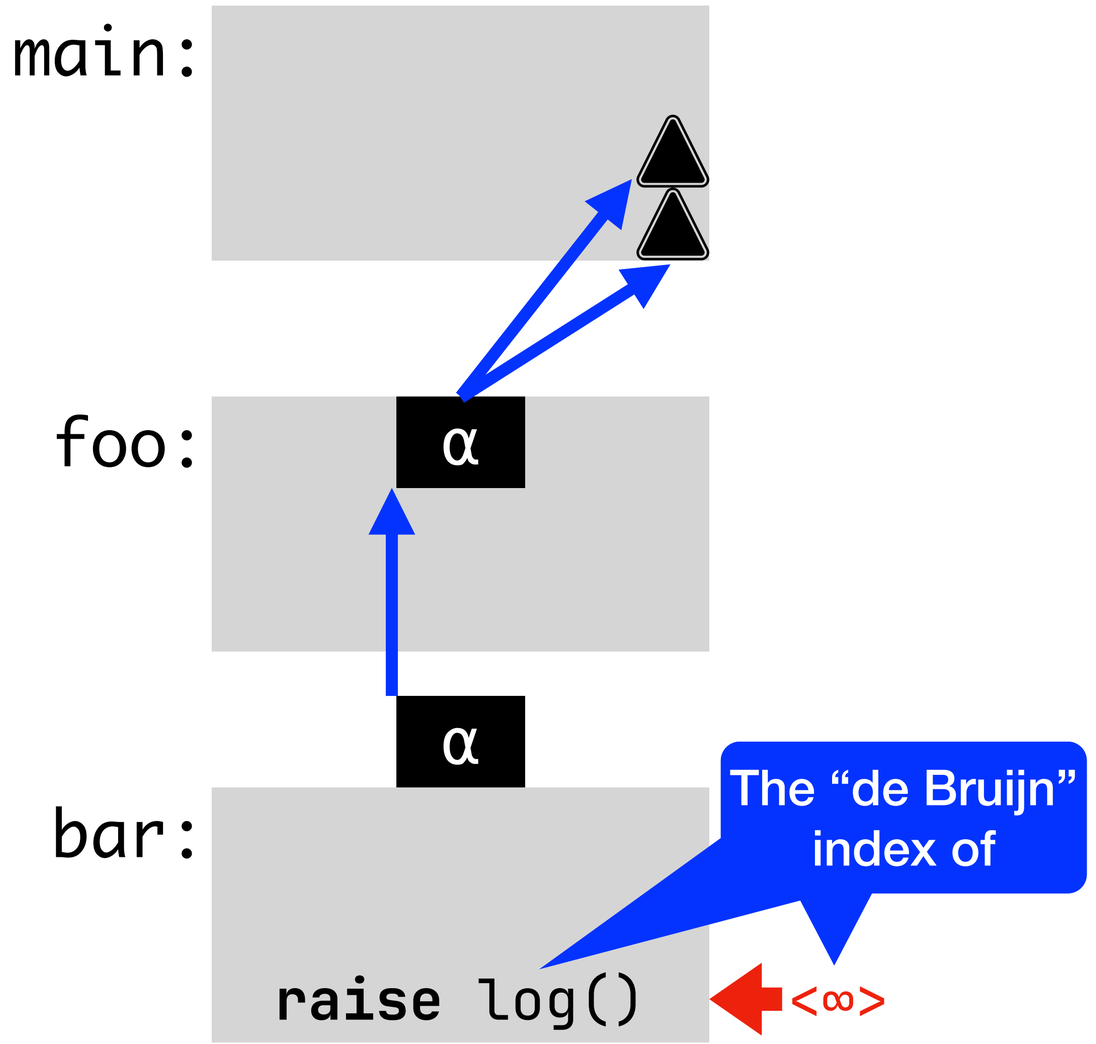
```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



Zero Lexa, Example 2

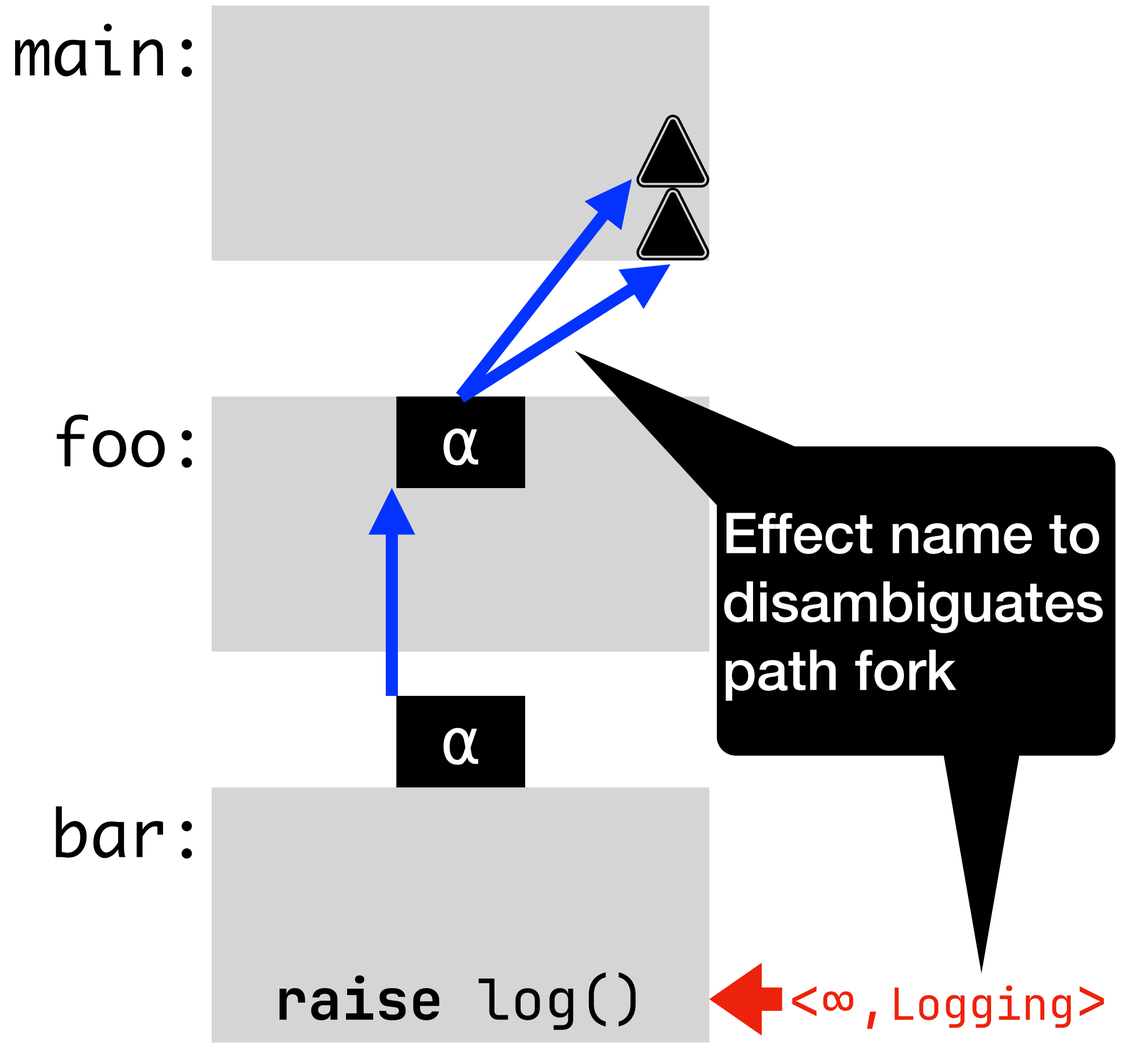
```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```

bound outside



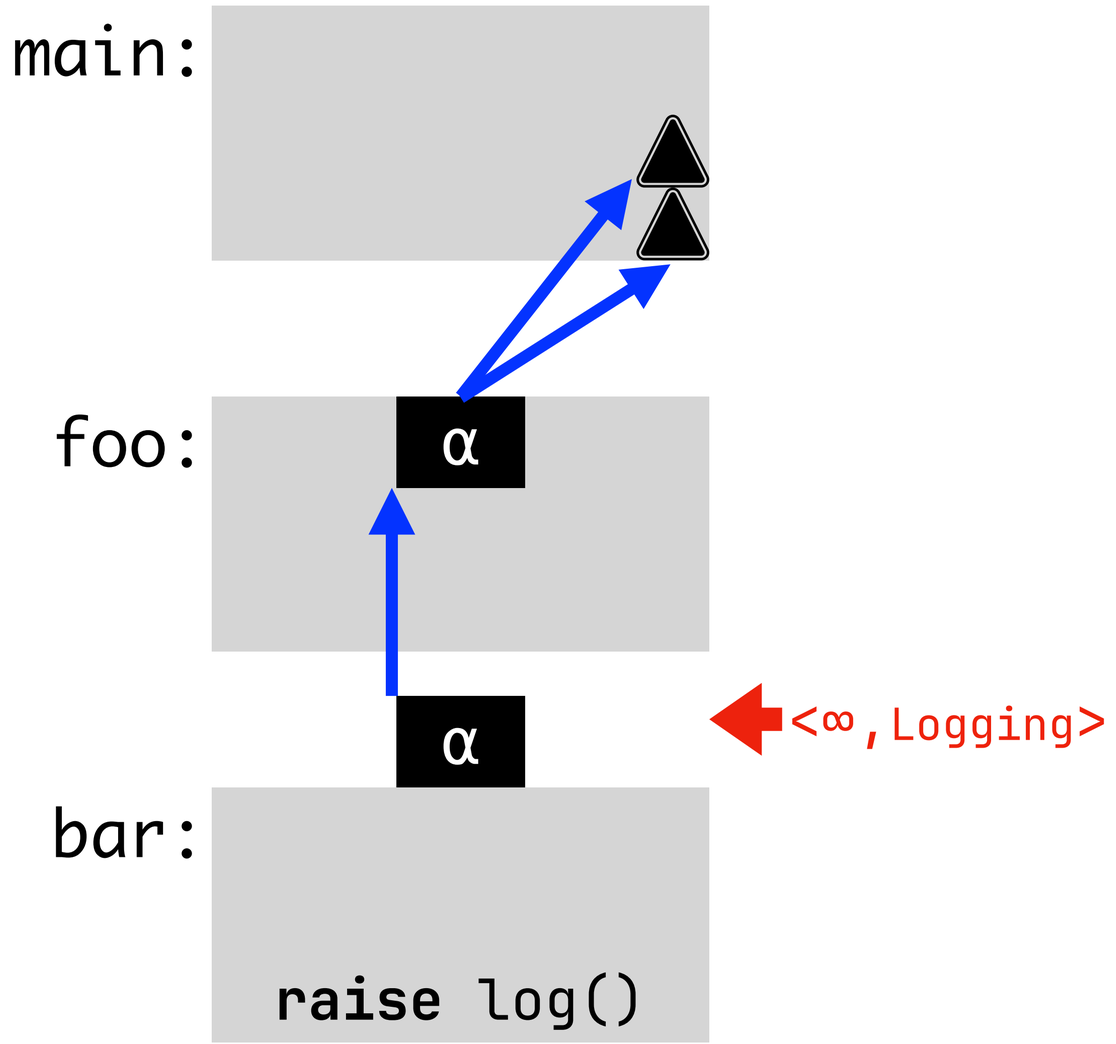
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



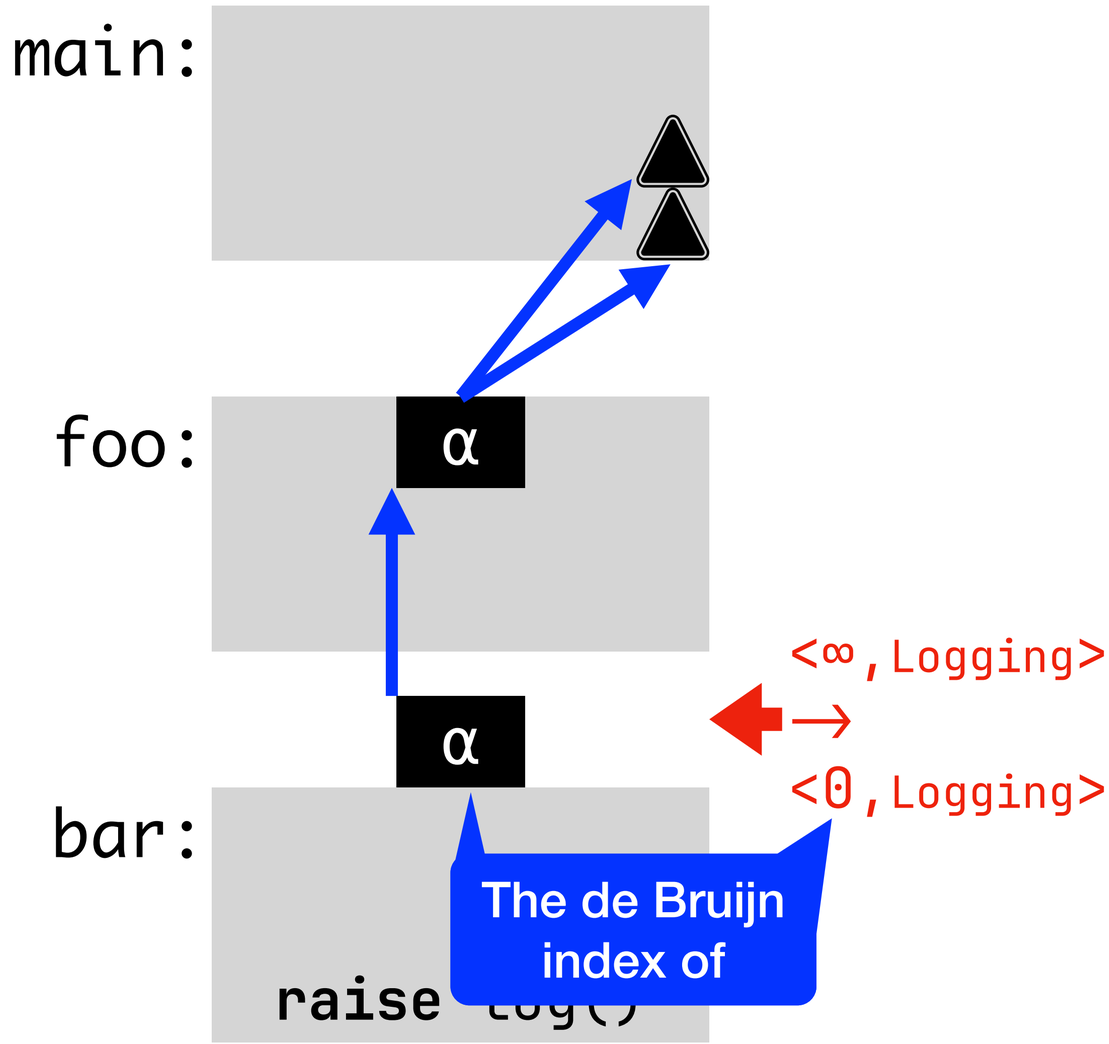
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



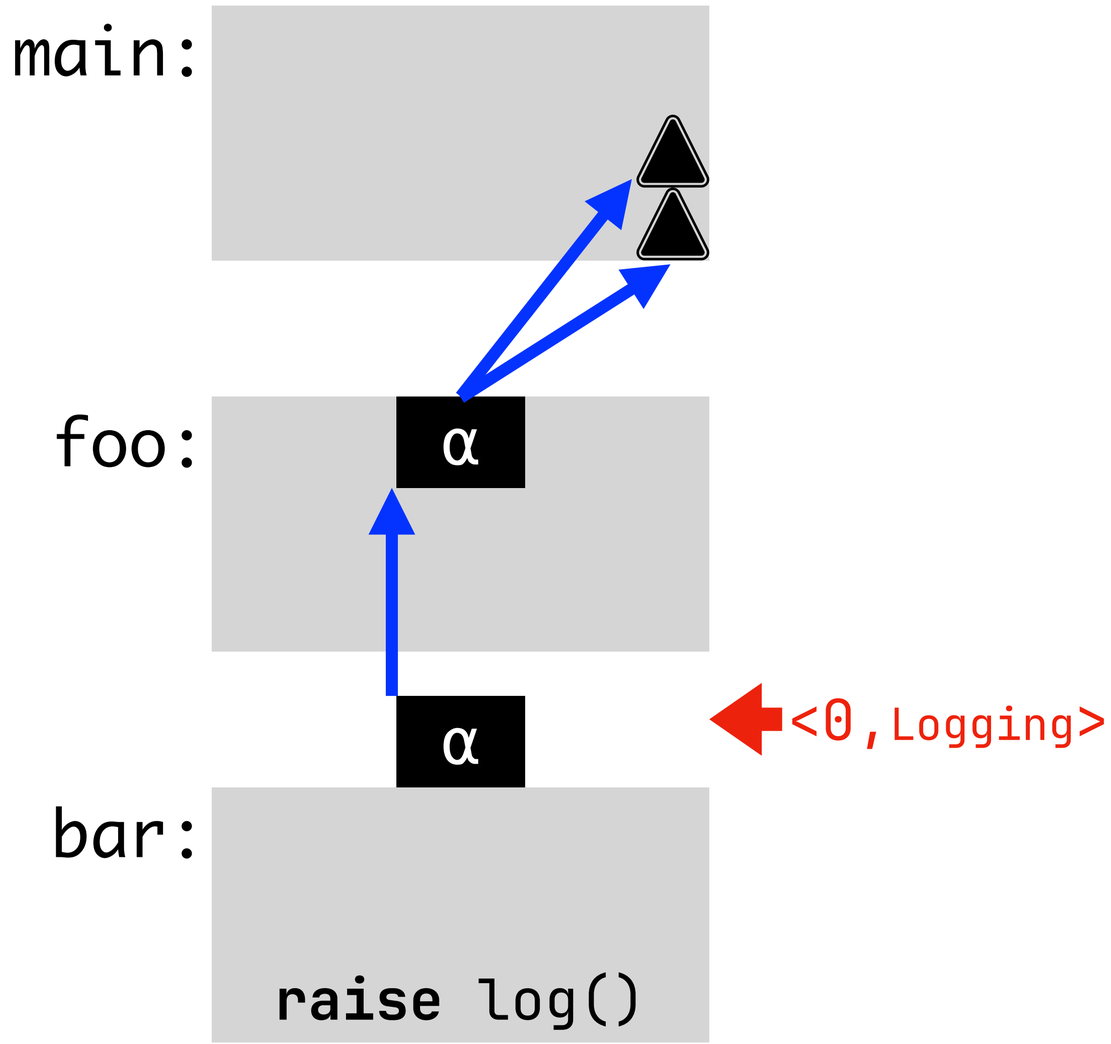
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



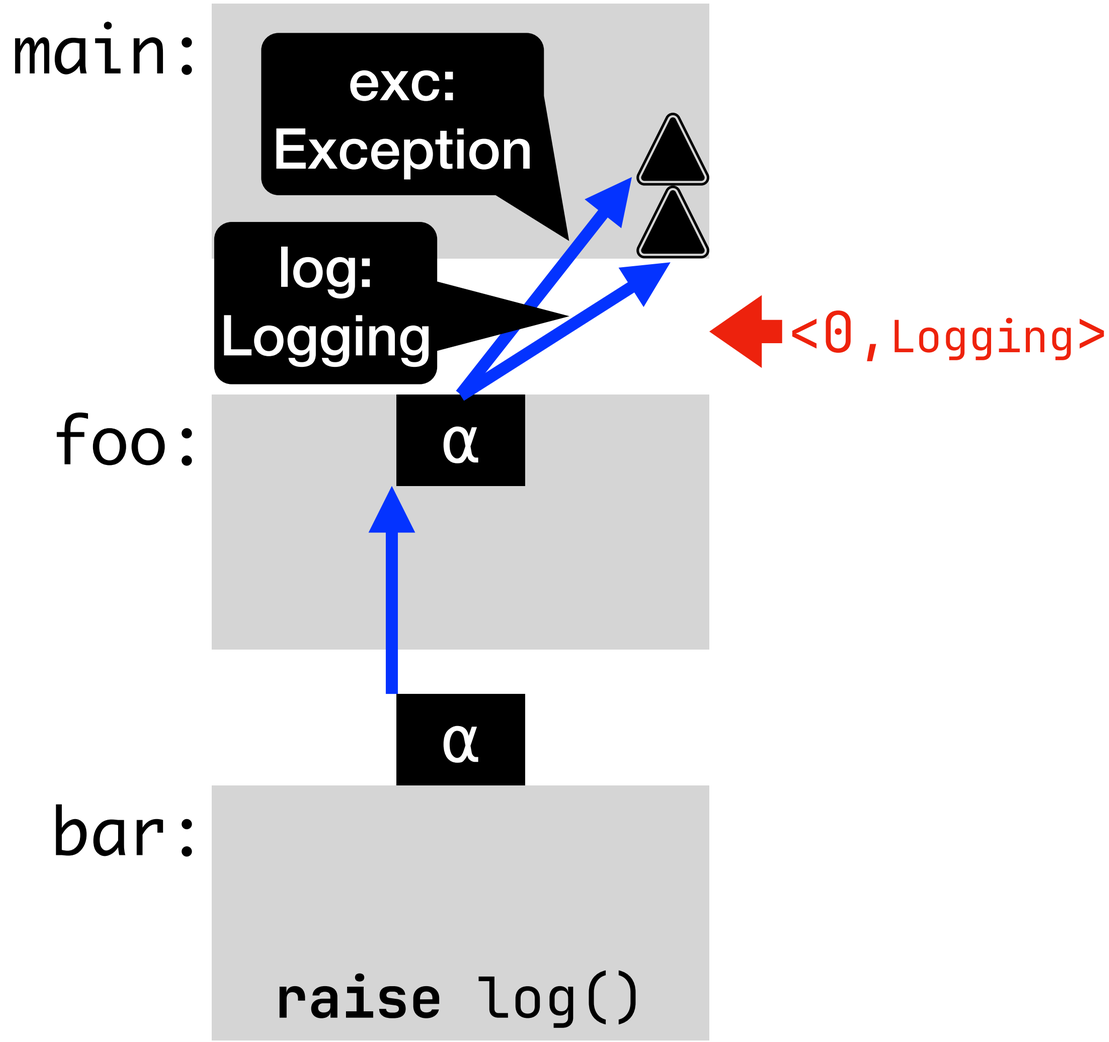
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



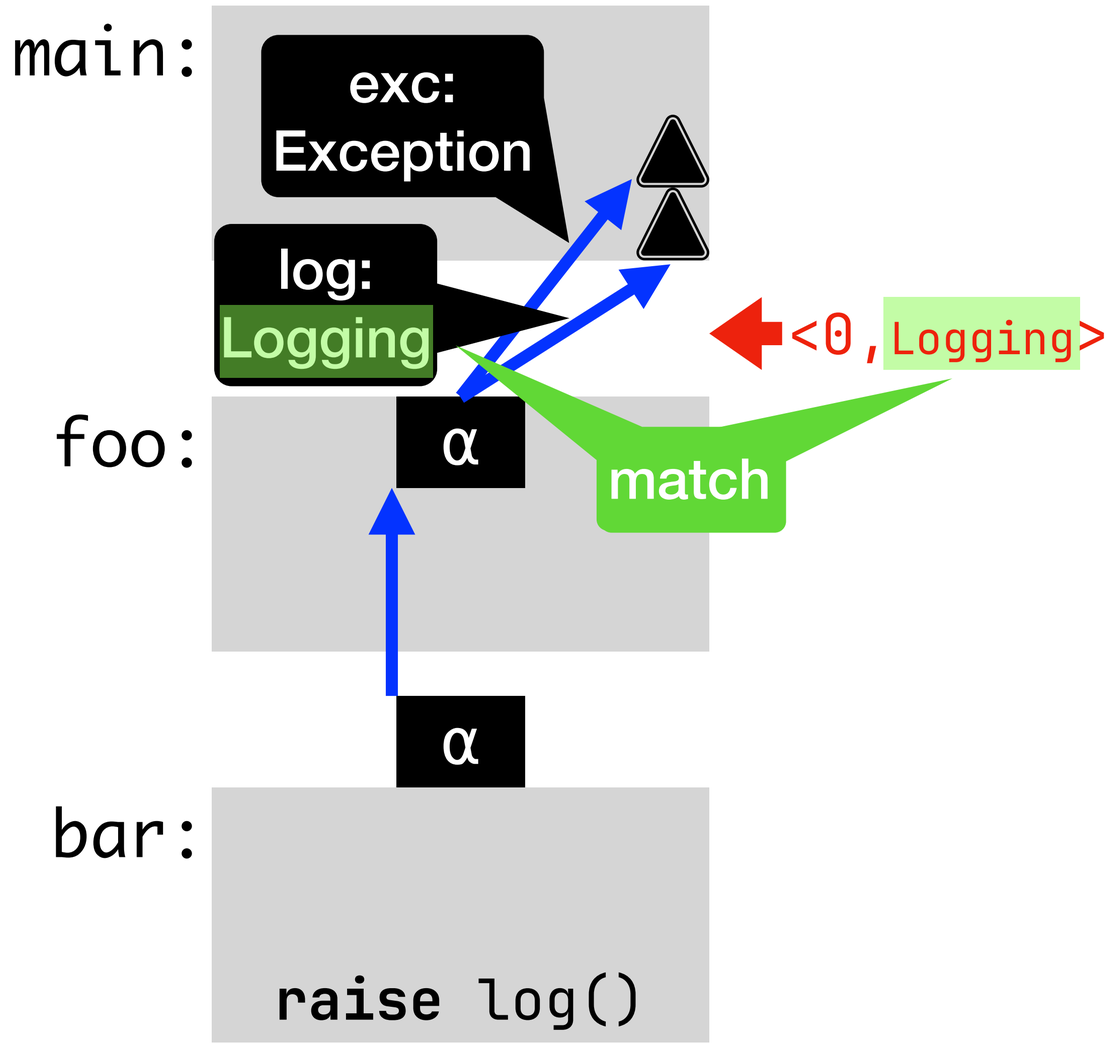
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



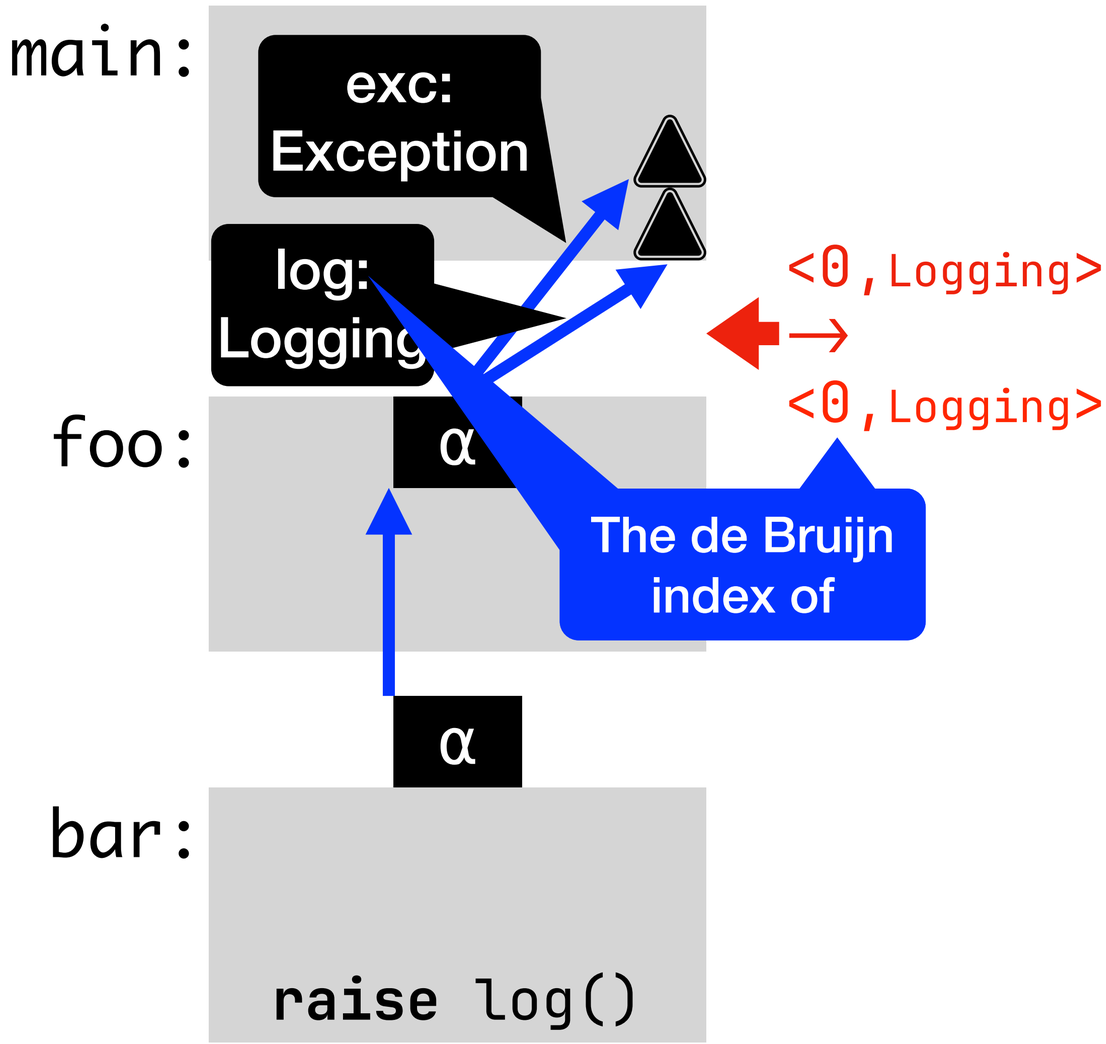
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



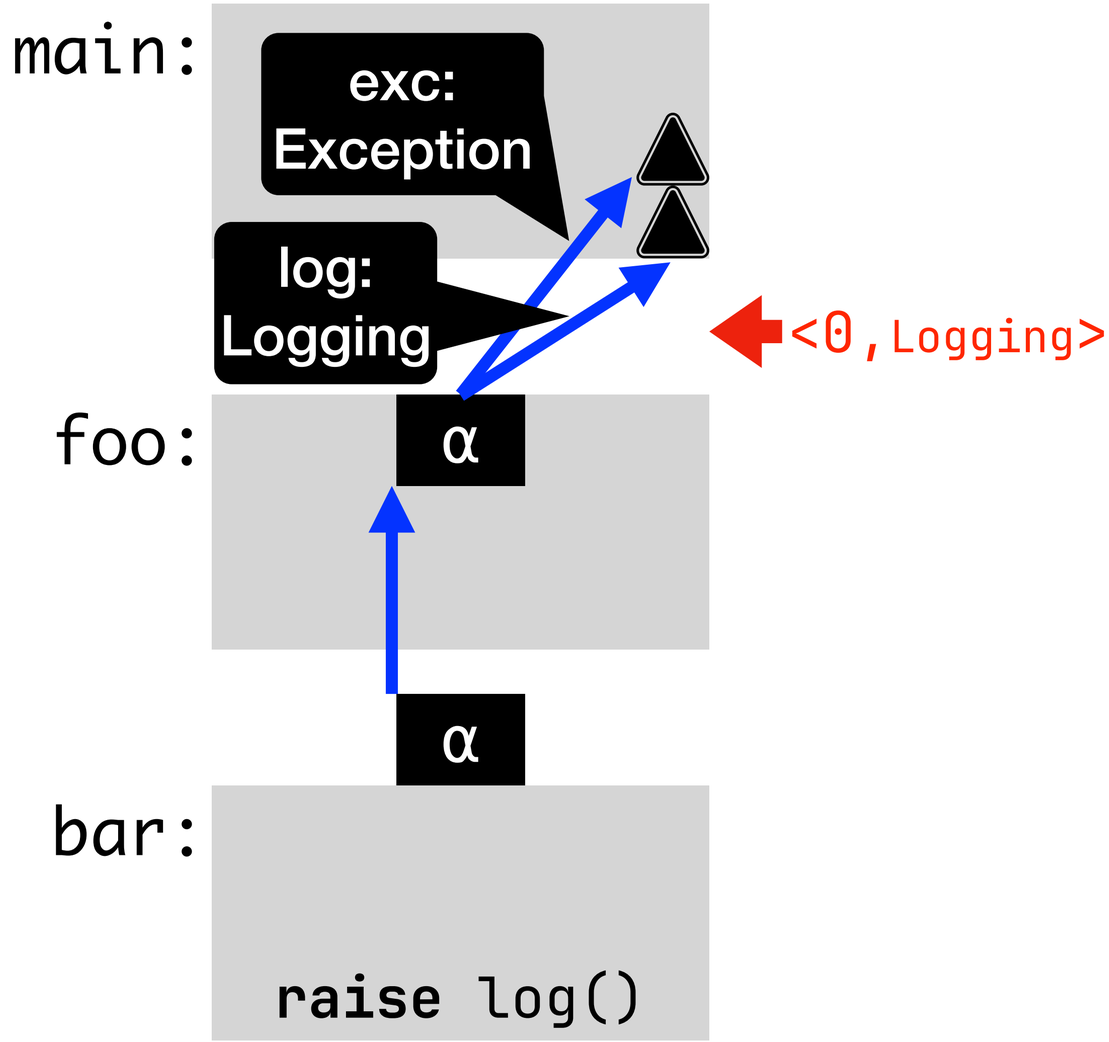
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



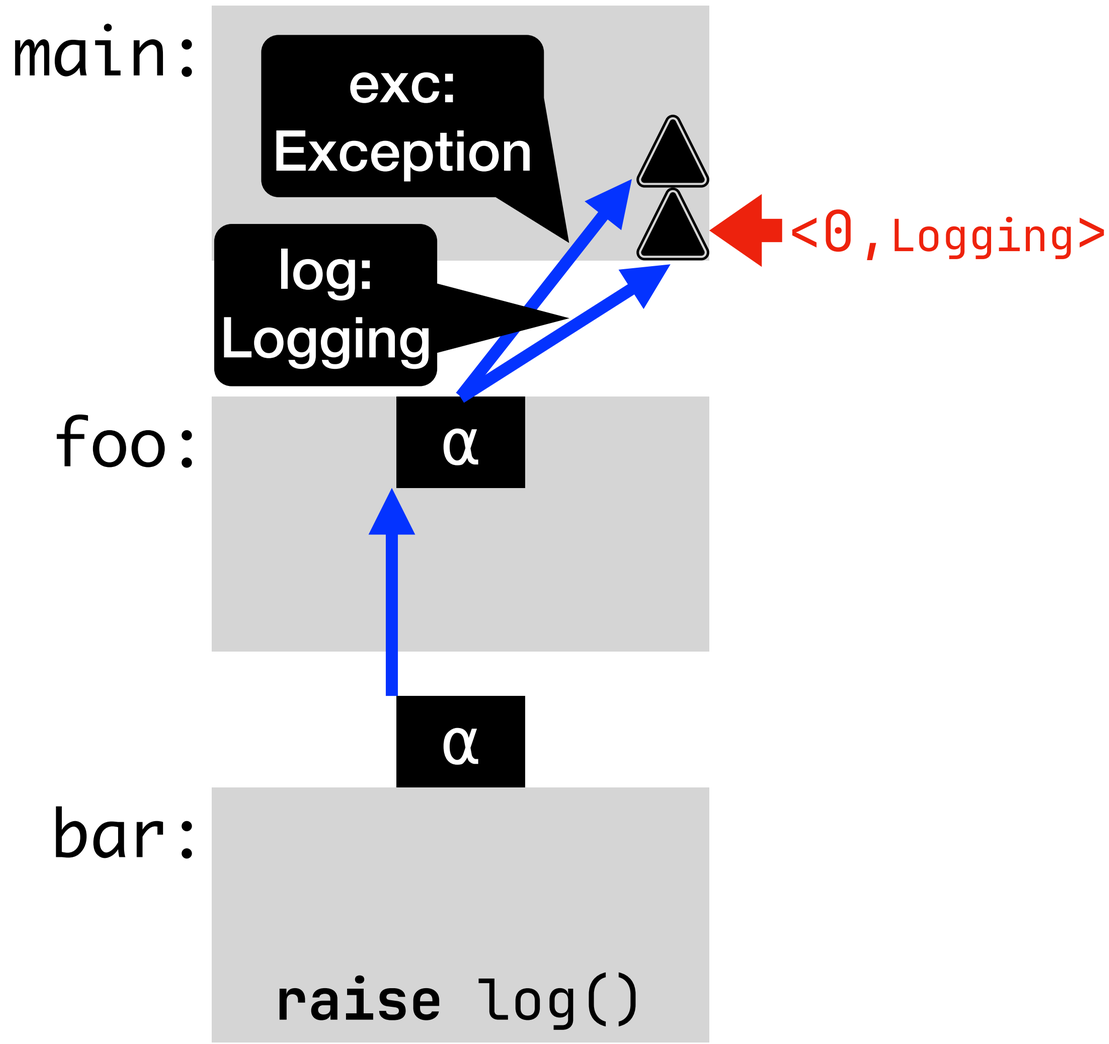
Zero Lexa, Example 2

```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



Zero Lexa, Example 2

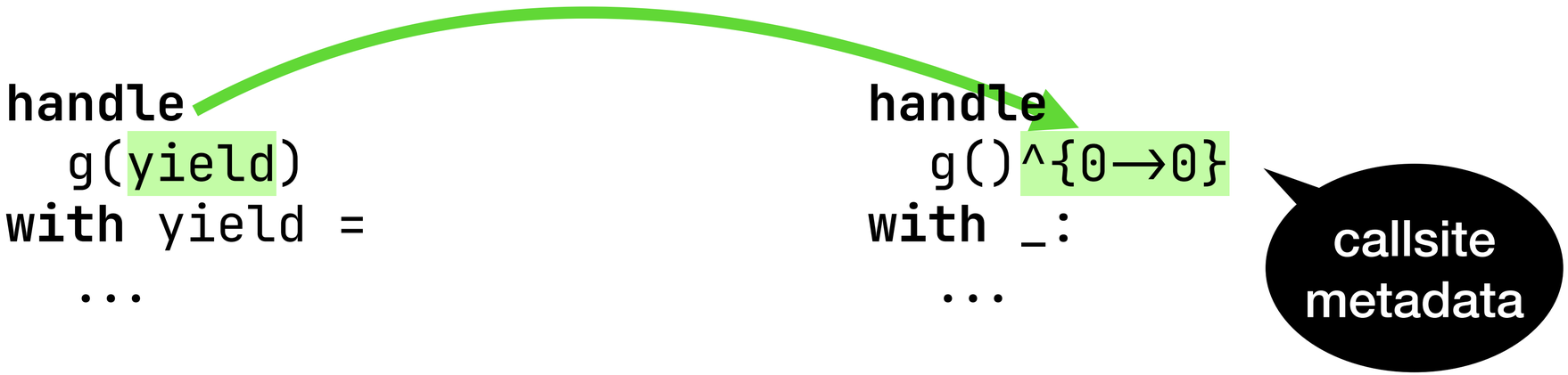
```
def main():
  handle
    handle
      let
        bar = λ().raise log();
                  raise exc()
      in
        foo[log, exc](bar)
    with log: Logging = ...
  with exc: Exception = ...

def foo[α](g):
  ...
  g()
  ...
```



Zero Lexa

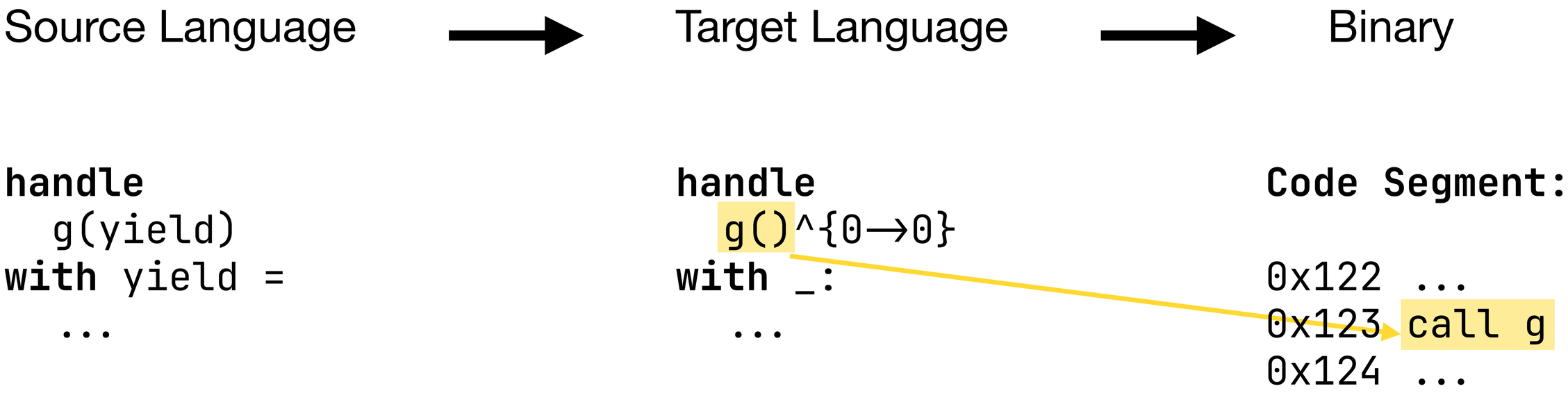
Implementation

Source Language → Target Language



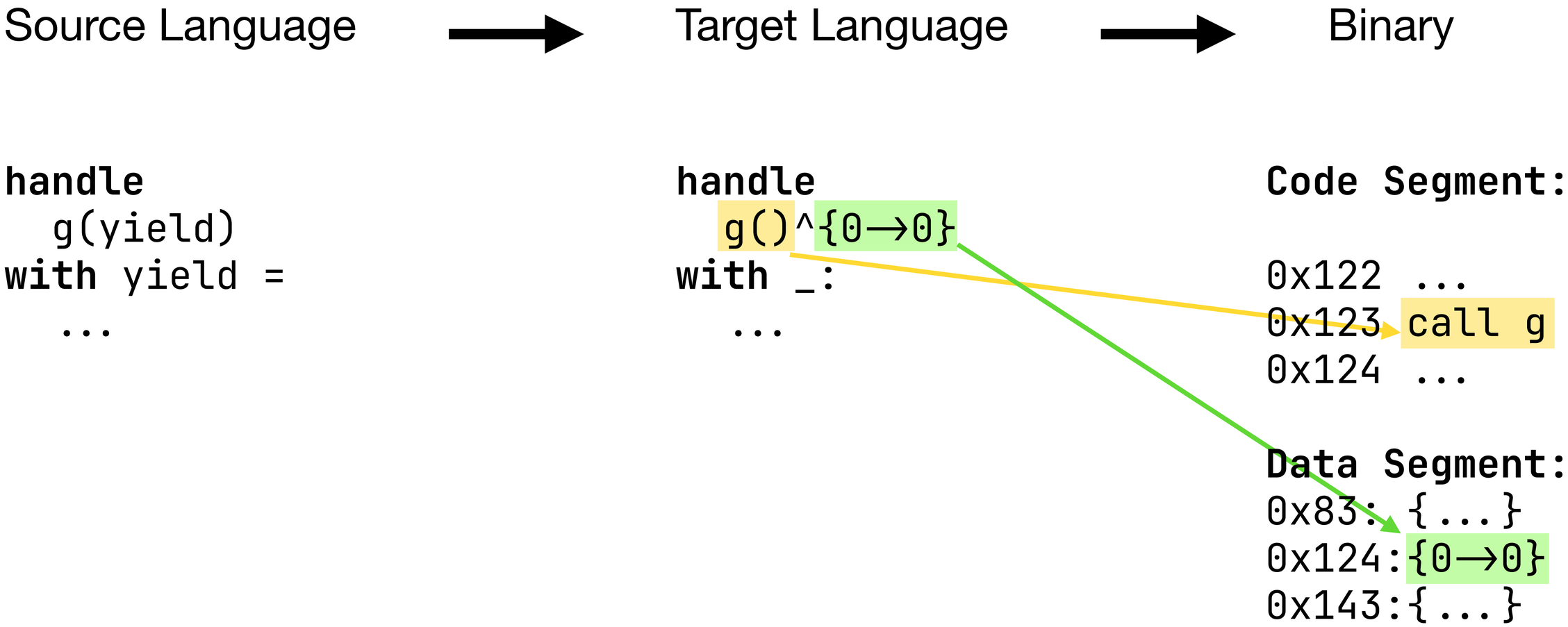
Zero Lexa

Implementation



Zero Lexa

Implementation



Zero Lexa

Implementation

Source Language → Target Language → Binary

```
handle
  g(yield)
with yield =
  ...
```

```
handle
  g()^{0→0}
with _:
  ...
```

```
Code Segment:

0x122  ...
0x123  call g
0x124  ...

Data Segment:
0x83:  {...}
0x124: {0→0}
0x143: {...}
```

return address of call g

Zero Lexa

Lexa Compiler

```
effect State {
    get: () → int
    set: (int) → unit
}

exceptional effect Exception {
  exc: (int) → int
}

...
```

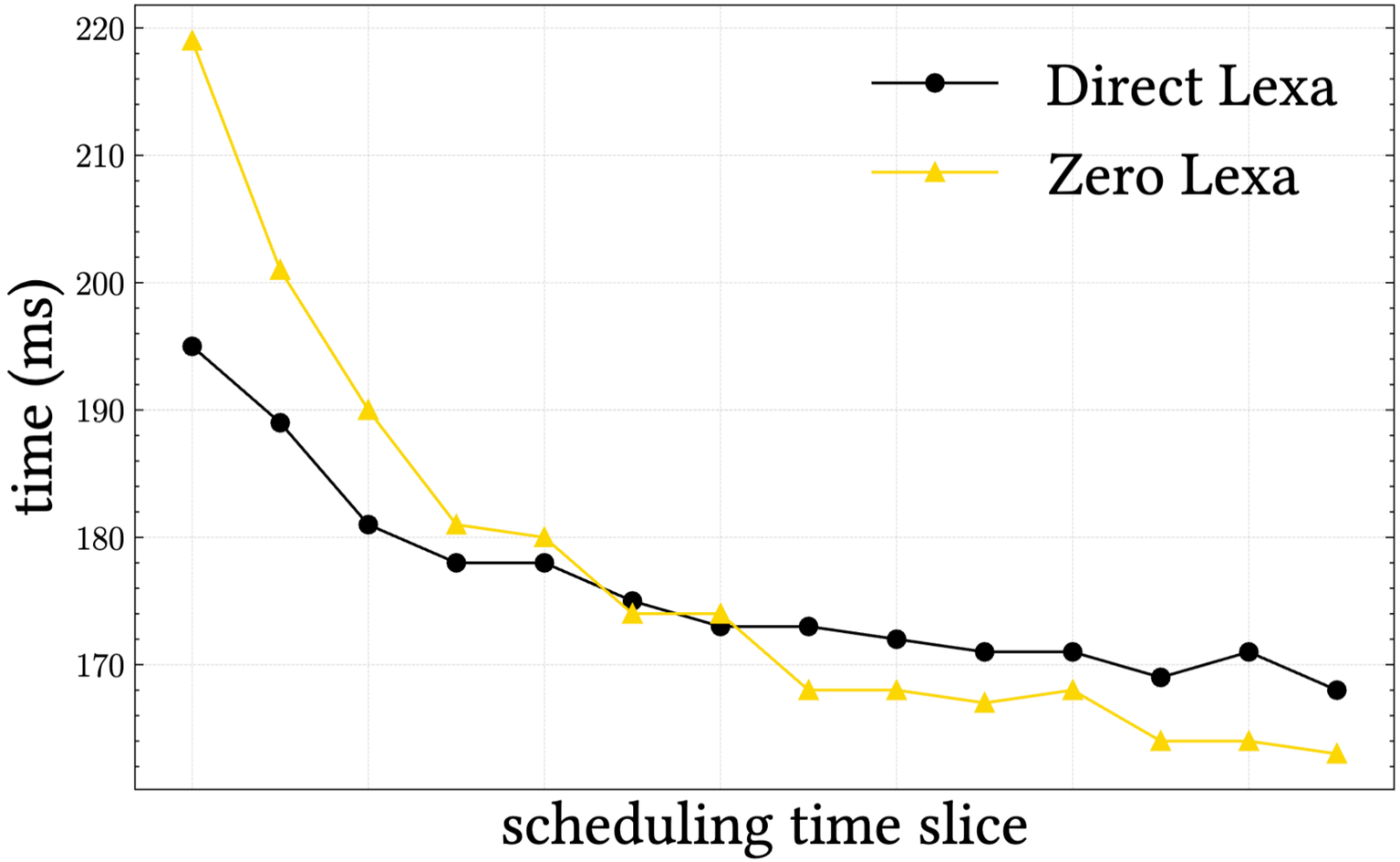
Evaluation

Hypothesis: zero-overhead implementation benefits exceptional handlers that are rarely used.

Evaluation

Case Study

A program with two coorperativley scheduled co-routine.



Better

Infrequent Yield

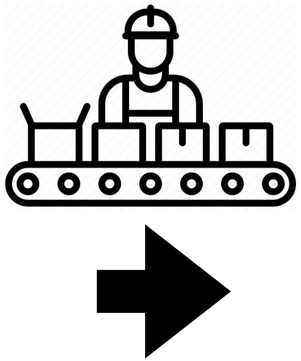
Evaluation

On new benchmarks

Benchmarks	Direct Lexa (ms)	Zero Lexa (ms)	Loss / Gain Zero vs. Direct
Catalan	344	301	12.5%
Bézout	680	671	1.3%
Golomb	651	563	13.5%
Hofstadter Q	576	559	3.0%
Karatsuba	804	702	12.7%
Ackermann	3975	3954	0.5%
Palindrome Partition	112	113	−0.9%
Lattice Path	1002	971	3.1%
Two Threads	153	140	8.5%

high-level effect handlers in Lexa

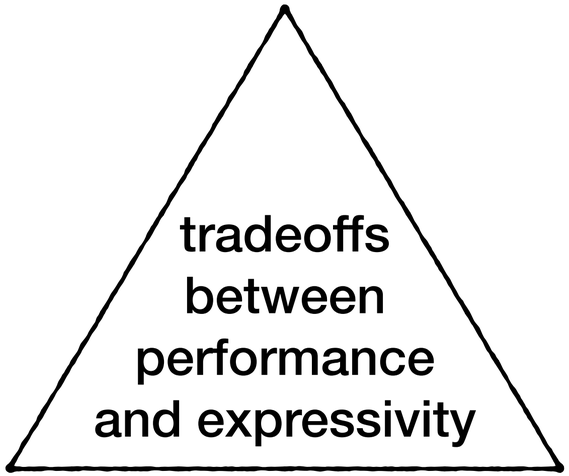
```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

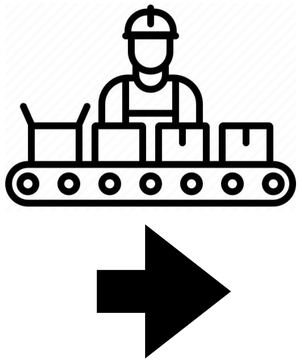


Multi Lexa

Zero Lexa

high-level effect handlers in Lexa

```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa

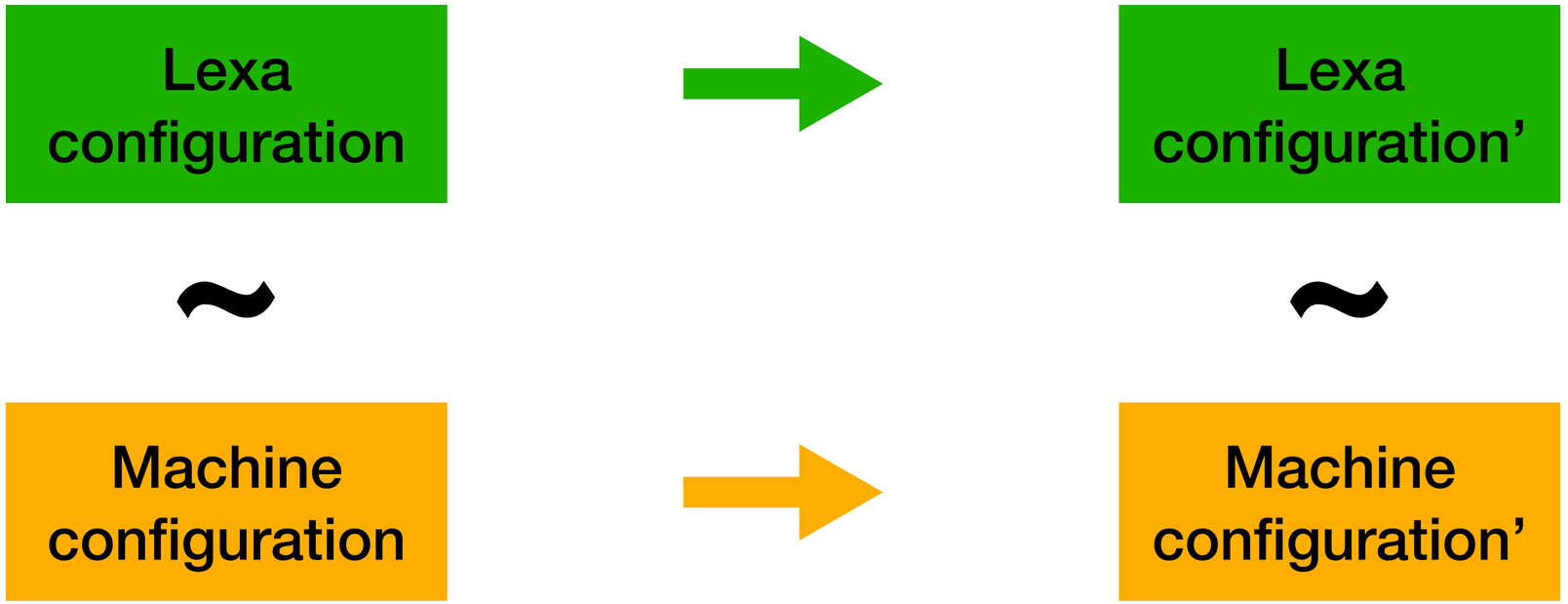
formally proved correct
⟦E⟧ = ⟦E⟧

Multi Lexa

Zero Lexa

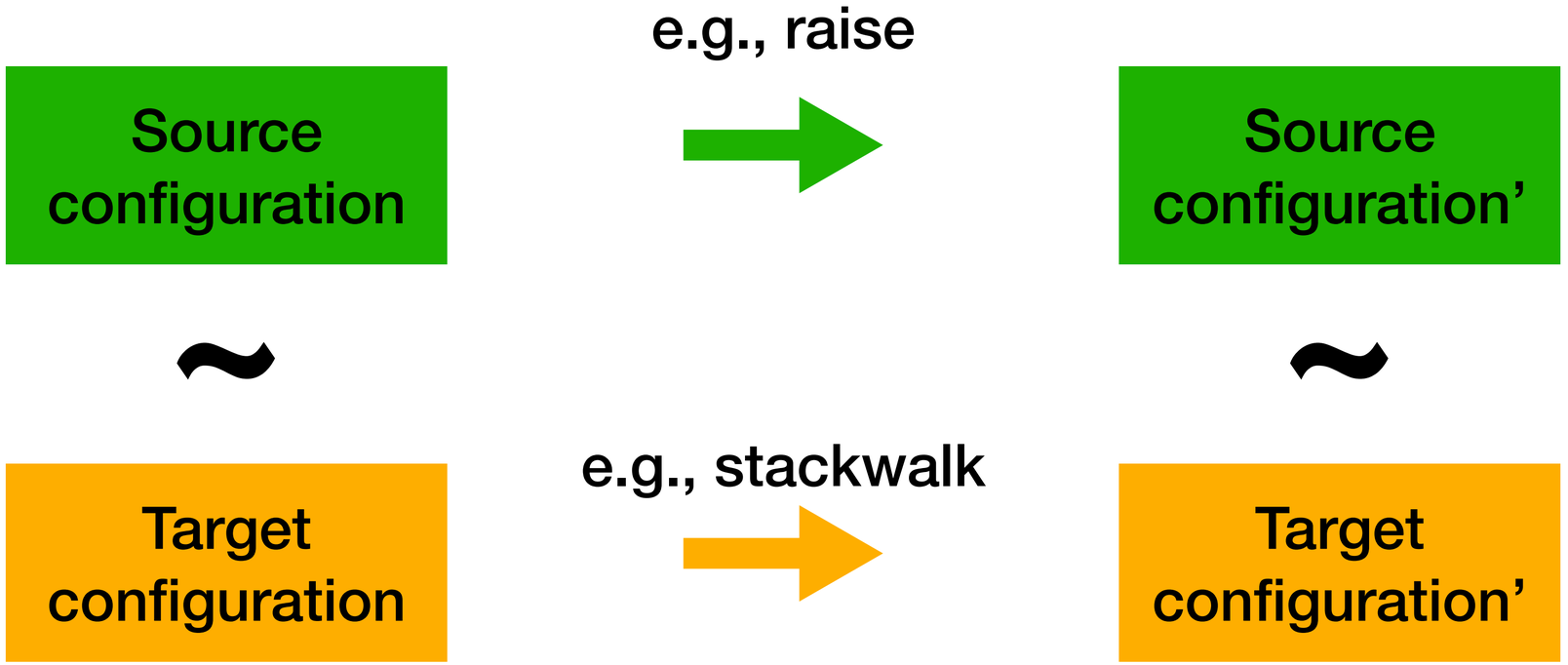
Compiler Correctness

CompCert-style Simulation Proofs for Direct, Multi & Zero Lexa



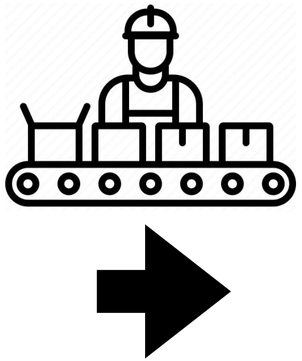
Compiler Correctness

e.g., Simulation Proofs for Zero Lexa



high-level effect handlers in Lexa

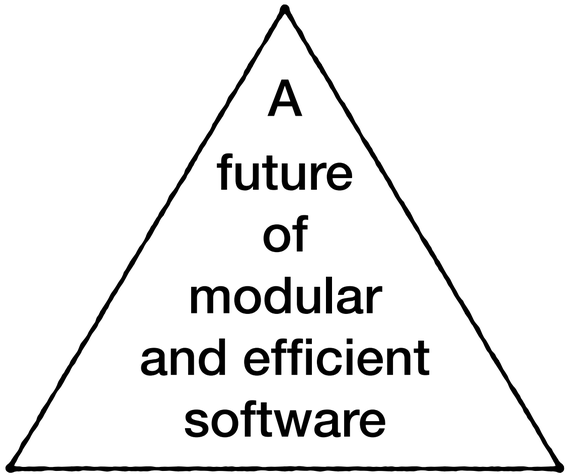
```
handle E with H

raise …

resume …
```



low-level stack switching in assembly

```
ENTER

RAISE

RESUME
```

Direct Lexa



Multi Lexa

Zero Lexa