Lexical Effect Handlers, Directly

Compiling high-level, modular algebraic effects to low-level, swift stack-switching

Cong Ma, Zhaoyi Ge, Edward Lee, Yizhou Zhang
University of Waterloo

Effect handler

- Effect handlers subsume an array of control flow features: async/await, coroutine, generator…

Lexical effect handler

- Dynamically scoped effect handler vs. lexical effect handler:

    Lexical handler fix a modularity problem of dynamic scoping

Accepting Blame for Safe Tunneled Exceptions

Yizhou Zhang* Guido Salvaneschi† Quinn Beightol*
Barbara Liskov‡ Andrew C. Myers*

*Cornell University, USA †TU Darmstadt, Germany ‡MIT, USA
yizhou@cs.cornell.edu salvaneschi@cs.tu-darmstadt.de qeb2@cornell.edu
liskov@csail.mit.edu andru@cs.cornell.edu

PLDI'16

Abstraction-Safe Effect Handlers via Tunneling

YIZHOU ZHANG, Cornell University, USA
ANDREW C. MYERS, Cornell University, USA

POPL'19

POPL'20

Binders by Day, Labels by Night

Effect Instances via Lexically Scoped Handlers

DARIUSZ BIERNACKI, University of Wrocław, Poland
MACIEJ PIRÓG, University of Wrocław, Poland
PIOTR POLESIUK, University of Wrocław, Poland
FILIP SIECZKOWSKI, University of Wrocław, Poland

Lexical effect handler

- Dynamically scoped effect handler vs. lexical effect handler:

    Lexical handler fix a modularity problem of dynamic scoping

- Lexical effect handlers are expressive.

Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism

JONATHAN IMMANUEL BRACHTHÄUSER, EPFL, Switze
PHILIPP SCHUSTER and KLAUS OSTERMANN, Univers

OOPSLA'20

First-Class Names for Effect Handlers

NINGNING XIE, University of Cambridge, UK
YOUYOU CONG, Tokyo Institute of Technology
KAZUKI IKEMORI, Tokyo Institute of Technolog
DAAN LEIJEN, Microsoft Research, USA

OOPSLA'22

Handling Bidirectional Control Flow

YIZHOU ZHANG, University of Waterloo, Canada
GUIDO SALVANESCHI, University of St. Gallen, Switzerland
ANDREW C. MYERS, Cornell University, USA

OOPSLA'20

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

Source Code

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

Runtime

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

Source Code

```
handle
  (raise #314()) + 1
with #314
  λk. resume k 42
```

freshly generated label

Runtime

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

Source Code

```
handle
  (□) + 1
with #314
  λk. resume k 42
```

Runtime

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

```
(λk. resume k 42)((□) + 1)
```

Source Code

Runtime

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

```
resume ((□) + 1) 42
```

Source Code

Runtime

Lexical effect handler 101

```
handle
  (raise ask()) + 1
with ask =
  λk. resume k 42
```

```
(42 + 1)
```

Source Code

Runtime

Lexical effect handler

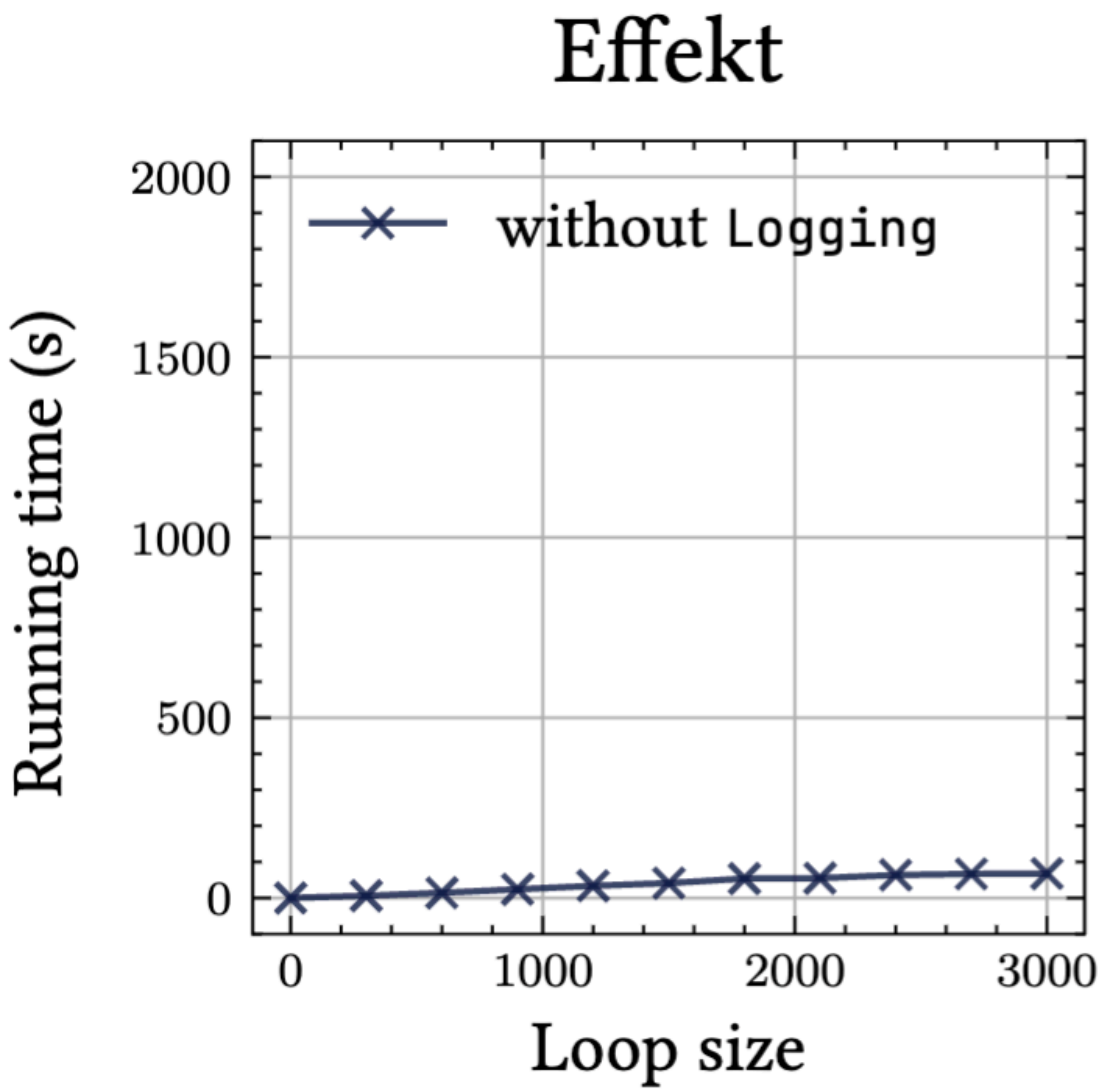
However, ...

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

Lexical effect handler

However, ...

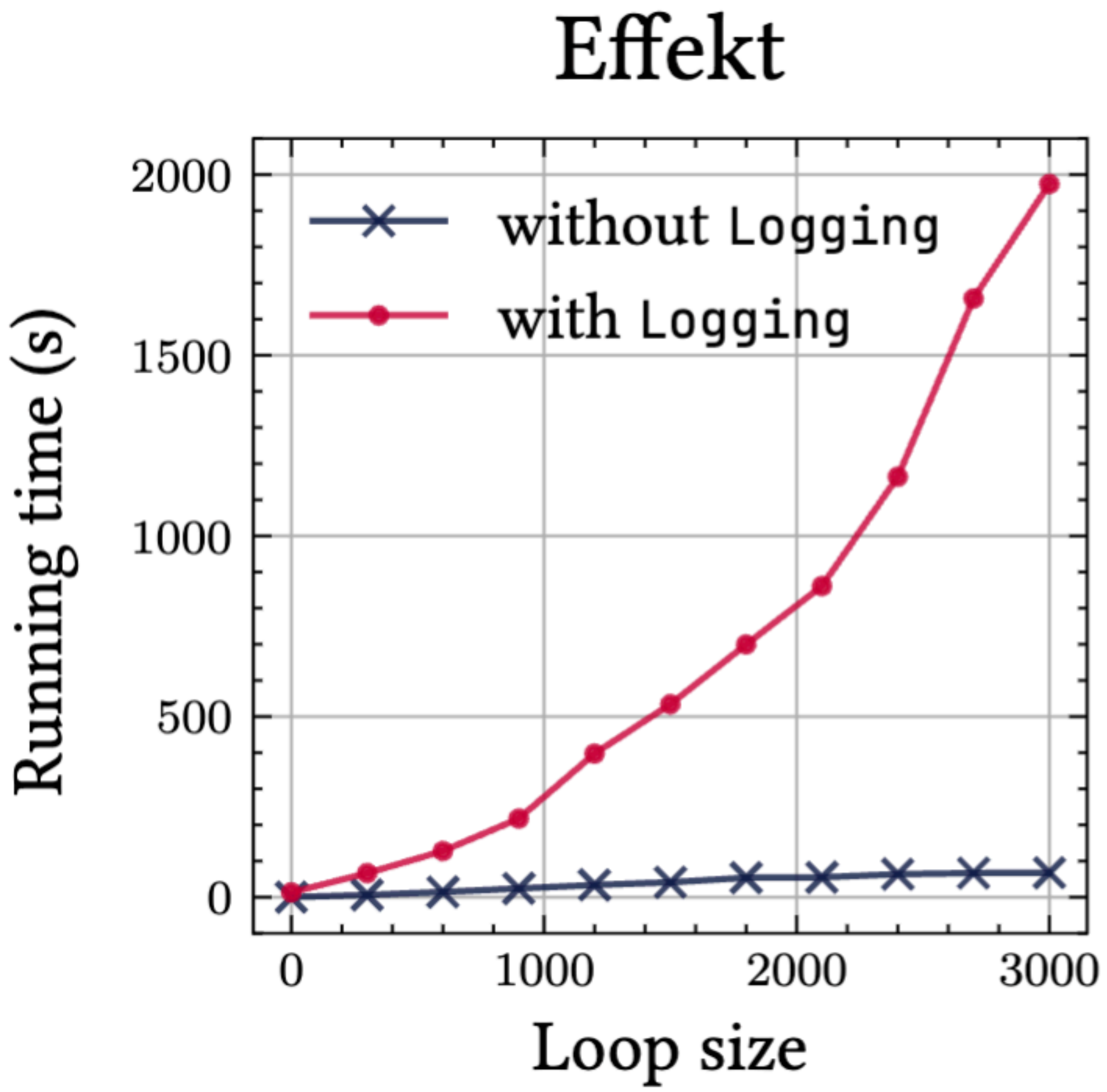
```
def loop(n) raises Logging =
  handle
    // raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```



Lexical effect handler

However, ...

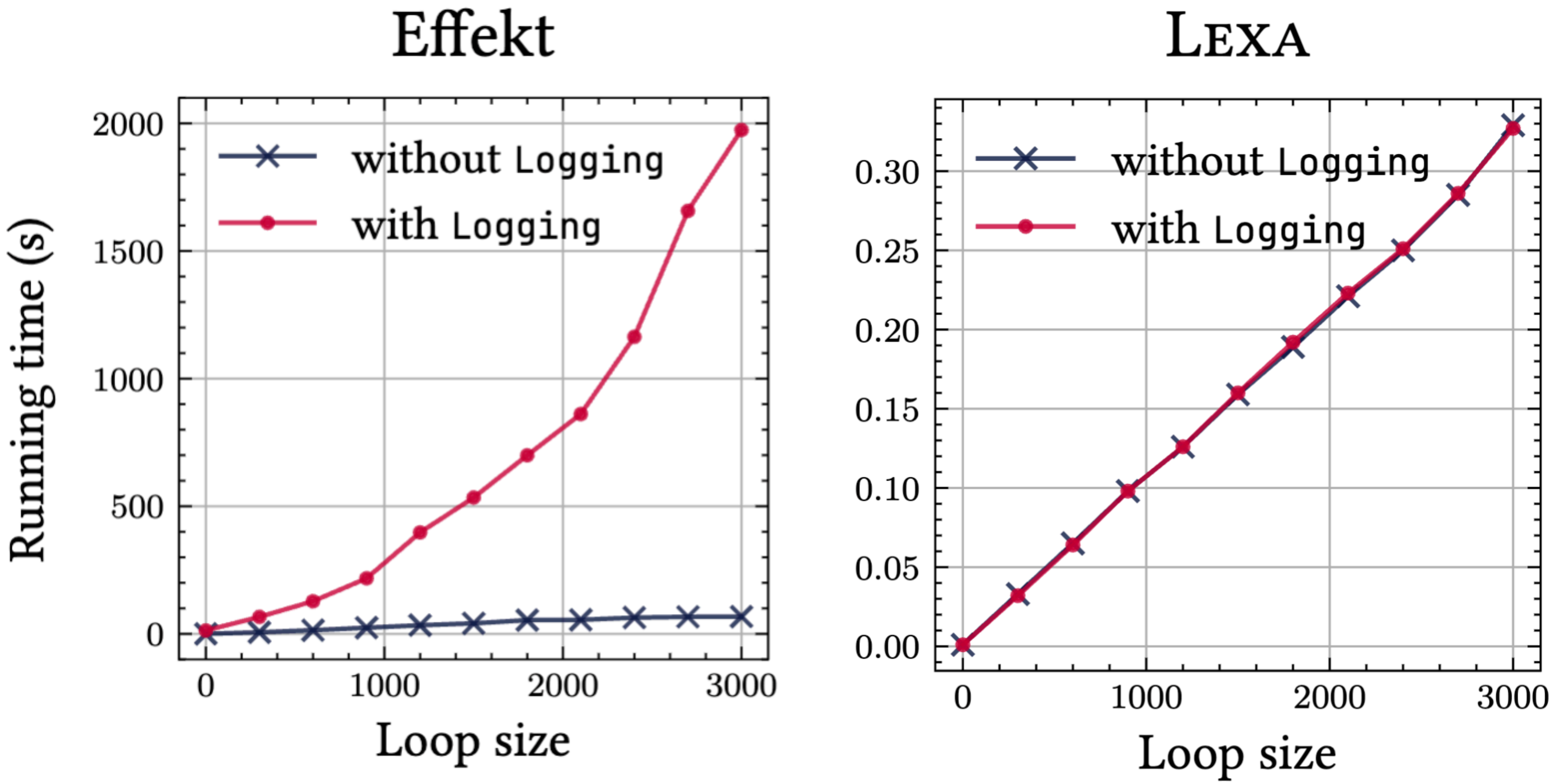
```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```



Lexical effect handler

However, ...

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```



For this program, our language Lexa has linear scaling.

high-level, modular algebraic effects in Lexa

```
handle E with H
raise …
resume …
```

this paper

low-level, swift stack switching in assembly

```
ENTER
RAISE
RESUME
```

Time to find the handler: O(1)

Time to capture the continuation: O(1)

$[\![\text{source program}]\!] = [\![\text{compiled program}]\!]$

Compiling lexical effect handlers…

requires searching for handlers, if the target language is too high-level

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

Compiling lexical effect handlers…

requires searching for handlers, if the target language is too high-level

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

```
#123, Logging handler
```

Compiling lexical effect handlers…

requires searching for handlers, if the target language is too high-level

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

```
#123, Logging handler

#456, Exception handler

...

#789, Exception handler

raise #123()
```

Compiling lexical effect handlers...

requires searching for handlers, if the target language is too high-level

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

searching for handler with the matching label

#123, Logging handler

#456, Exception handler

...

Exception handler

Seems to be at odd with the spirit of lexical scoping

#123()

raise #123()

Compiling lexical effect handlers…

does not require searching if the target language is low-level enough

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```

#123()

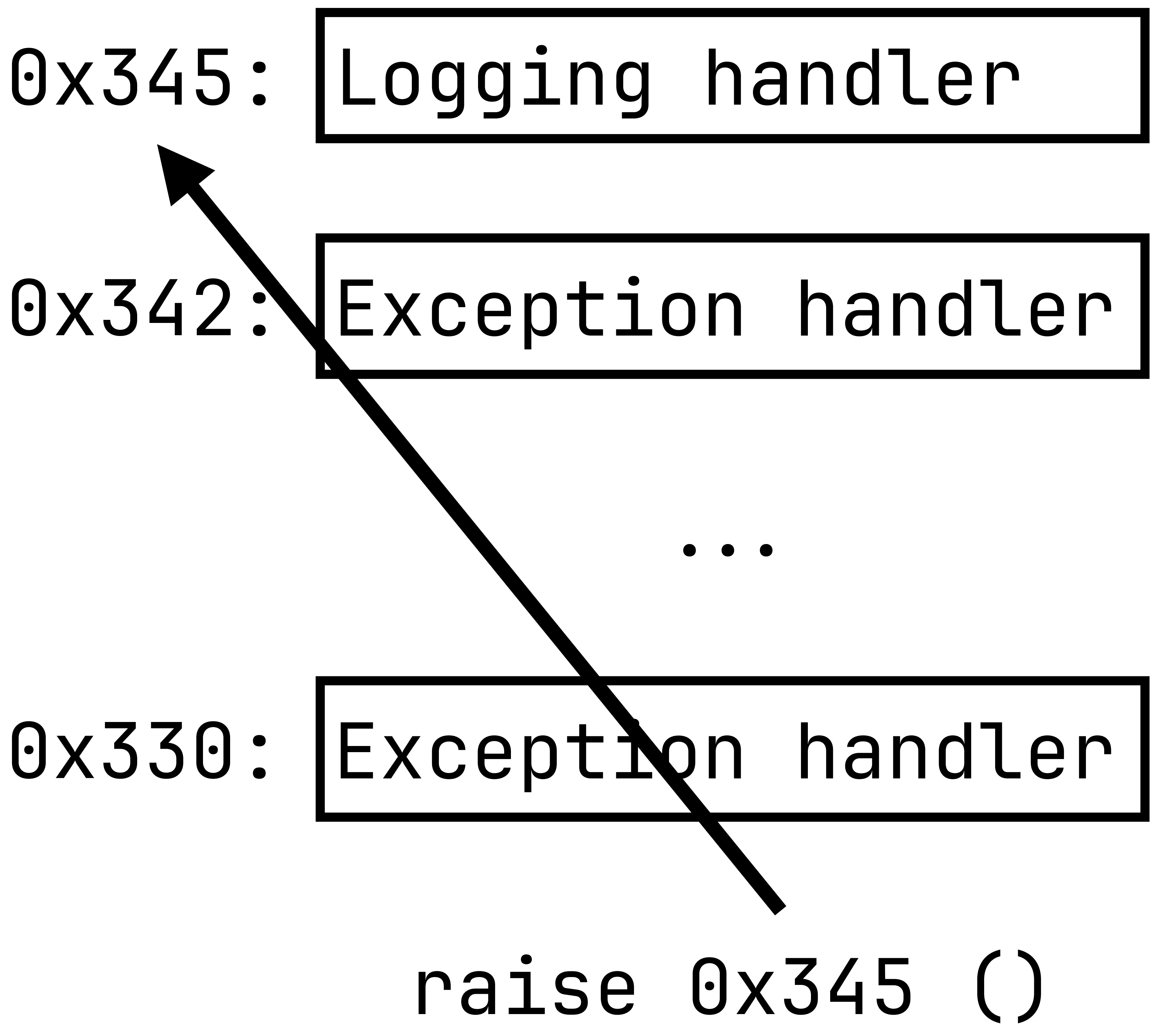
#123, Logging handler
#456, Exception handler
...
#789, Exception handler

`raise #123()`

Compiling lexical effect handlers…

does not require searching if the target language is low-level enough

```
def loop(n) raises Logging =
  handle
    raise Logging(...);
    loop(n-1);
    ...
  with Exception =
    λx,k...
```



high-level, modular algebraic effects in Lexa

```
handle E with H
raise …
resume …
```

this paper

low-level, swift stack switching in assembly

```
ENTER
RAISE
RESUME
```

Time to find the handler: $O(1)$

Time to capture the continuation: $O(1)$

$[\![\text{source program}]\!] = [\![\text{compiled program}]\!]$

```
(λxλy.let z = x + y in z)(42,1)
```

A program without effects.

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

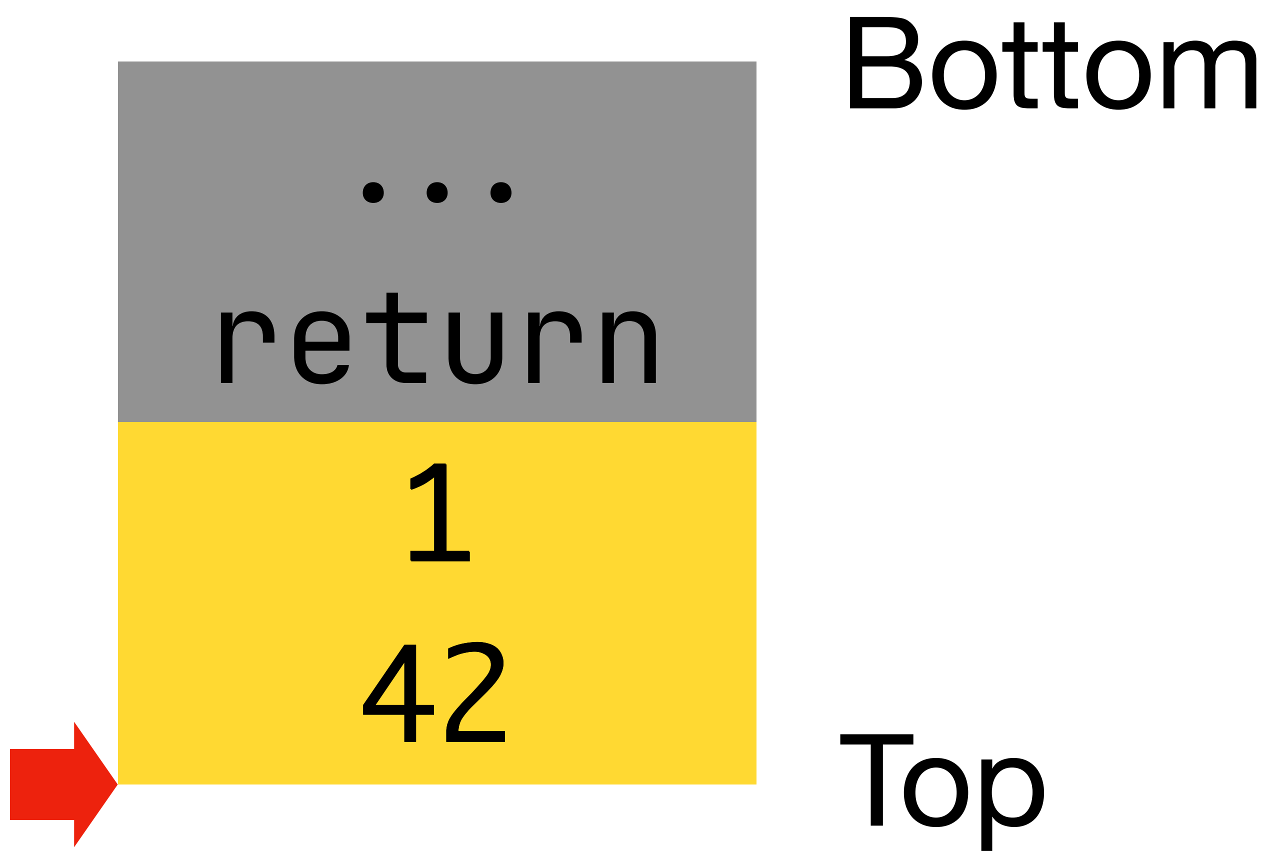
Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```



Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

```
...
return
1
42
```

```
r1 -> 42
r2 → 1
```

Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

...
return
1
42

```
r1 -> 43
r2 → 1
```

Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

...
return
1
42
43

```
r1 -> 43
r2 → 1
```

Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

...
return
1
42
43

```
r1 -> 43
r2 → 1
```

Code

Stack

Register

(λxλy.let z = x + y in z)(42,1)

```
#fun: push r2
      push r1
      load r1, [sp + 0]
      load r2, [sp + 1]
      add r1, r2
      push r1
      load r1, [sp + 0]
      sfree 3
      ret
```

```
...
return
```

```
r1 -> 43
```

Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()


 #ENTER: ...
 #RAISE: ...
#RESUME: ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
         load r1, [sp + 0]
         mov r2, ()
         call #RAISE
         ...
```

```
 handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()


 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
         load r1, [sp + 0]
         mov r2, ()
         call #RAISE
         ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
         load r1, [sp + 0]
         mov r2, ()
         call #RAISE
         ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
         load r1, [sp + 0]
         mov r2, ()
         call #RAISE
         ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()

 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
   #ask: ...
        mov r1, [sp + 1]
        mov r2, ()
        call #RESUME
        ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
   #ask: ...
   #fun: ...
         mov r2, #ask
         mov r1, #body
         call #ENTER
         ...
```

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
 #ENTER: ...
 #RAISE: ...
#RESUME: ...
  #body: ...
   #ask: ...
   #fun: ...
         mov r2, #ask
         mov r1, #body
         call #ENTER
         ...
```

Code

```
...
return
```

Stack

```
r1 → #body
r2 → #ask
```

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#ENTER: mov r3, sp
        mkstk sp
        push r2
        push r3
        mov r3, r1
        mov r1, sp
        call r3
        pop r2
        sfree 2
        mov sp, r2
        return
```

```
...
return
```

```
r1 → #body
r2 → #ask
```

Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#ENTER: mov r3, sp
        mkstk sp
        push r2
        push r3
        mov r3, r1
        mov r1, sp
        call r3
        pop r2
        sfree 2
        mov sp, r2
        return
```

```
...
return
```

```
r1 → #body
r2 → #ask
```

Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#ENTER: mov r3, sp
        mkstk sp
        push r2
        push r3
        mov r3, r1
        mov r1, sp
        call r3
        pop r2
        sfree 2
        mov sp, r2
        return
```

Stack	
... return	#ask

Register
r1 → #body
r2 → #ask

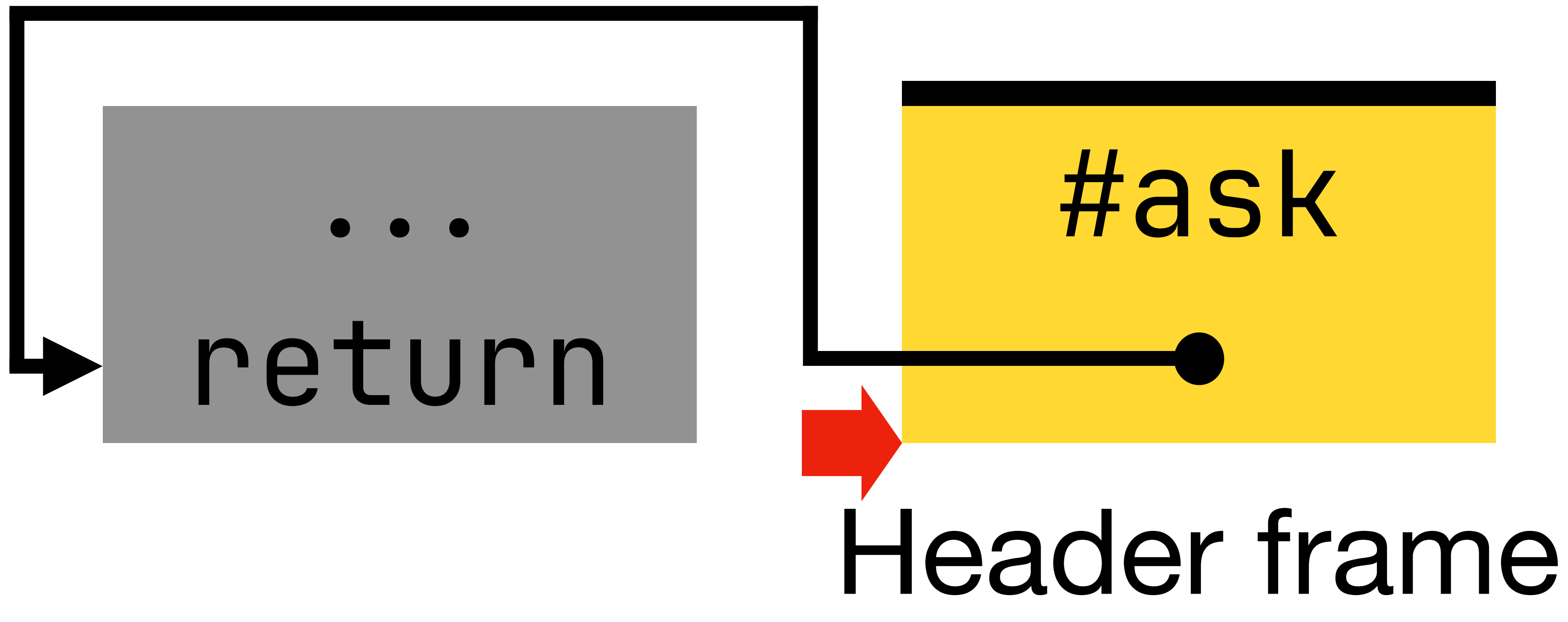
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#ENTER: mov r3, sp
        mkstk sp
        push r2
        push r3
        mov r3, r1
        mov r1, sp
        call r3
        pop r2
        sfree 2
        mov sp, r2
        return
```



r1 → #body
r2 → #ask

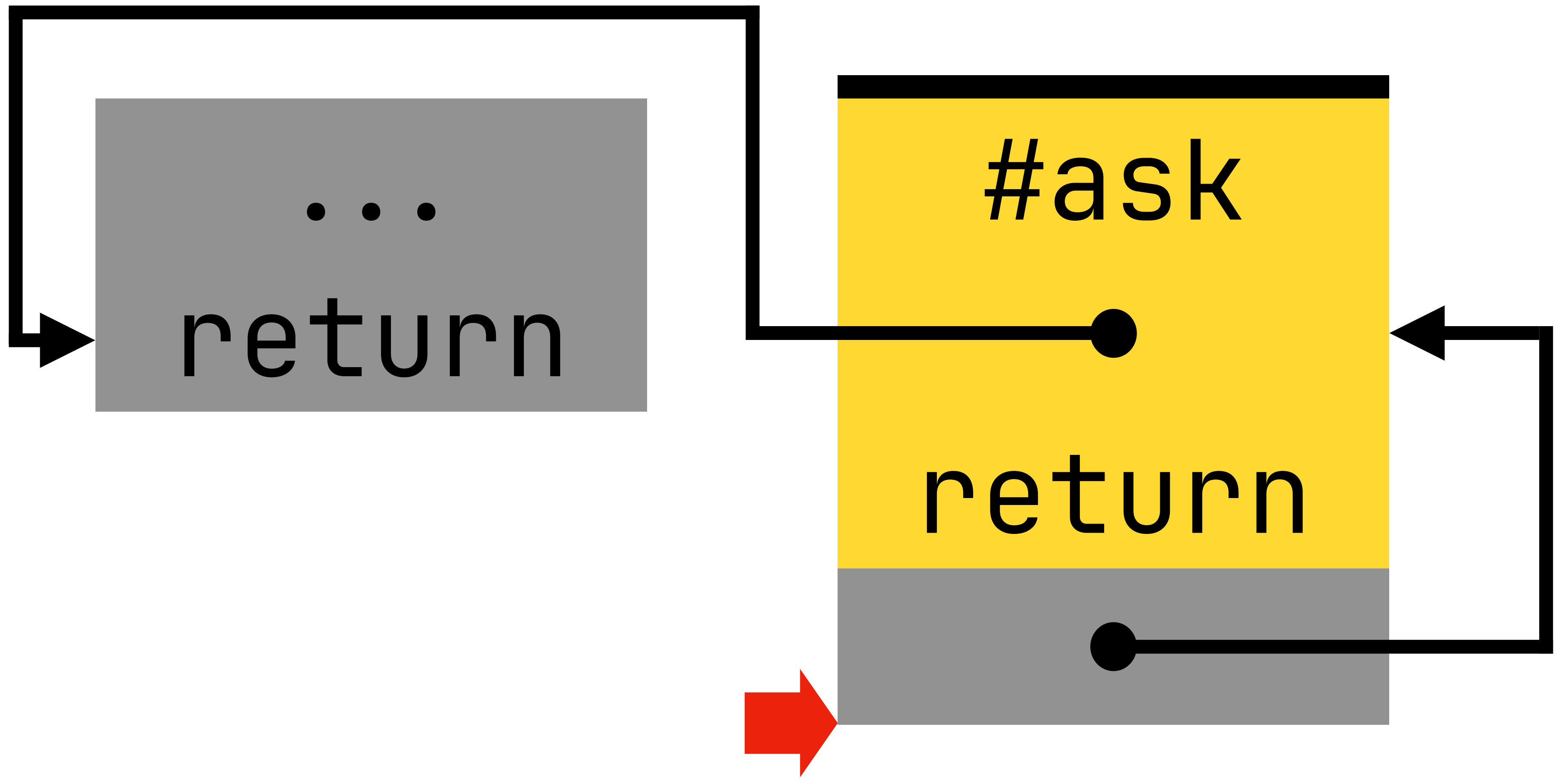
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#ENTER: mov r3, sp
        mkstk sp
        push r2
        push r3
        mov r3, r1
        mov r1, sp
        call r3
        pop r2
        sfree 2
        mov sp, r2
        return
```



r3 → #body

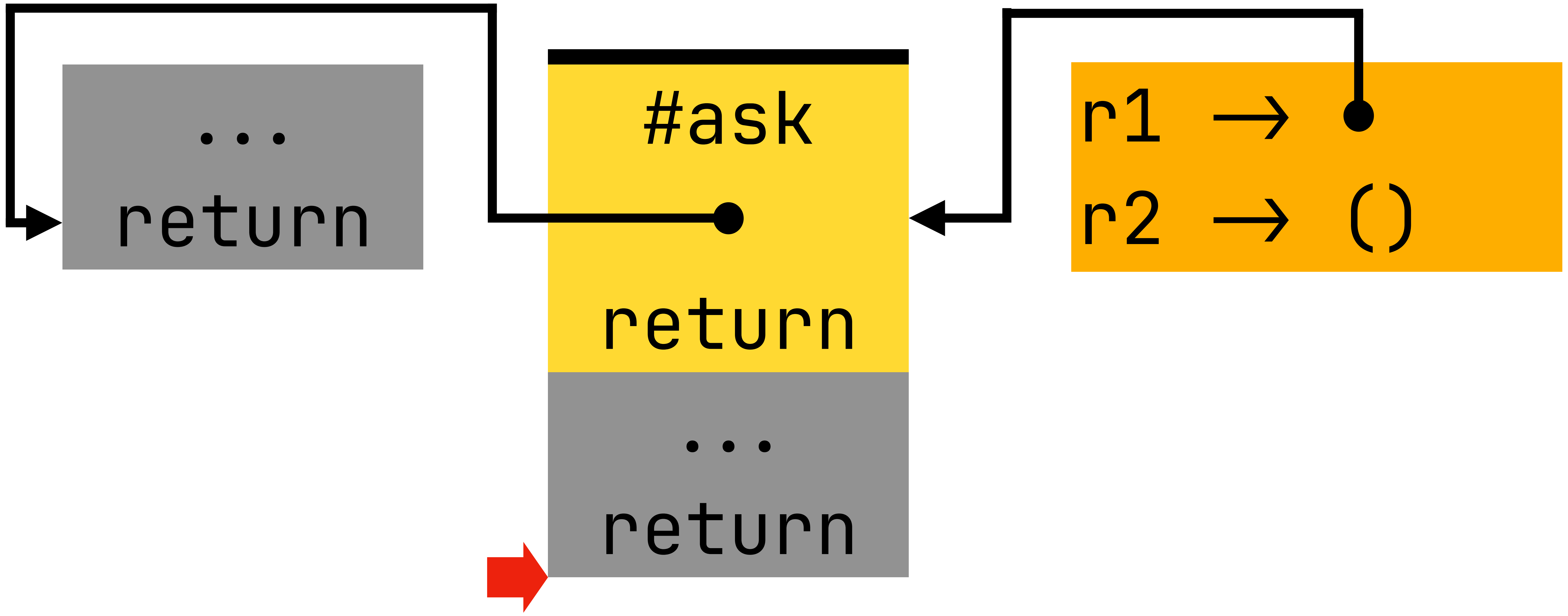
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



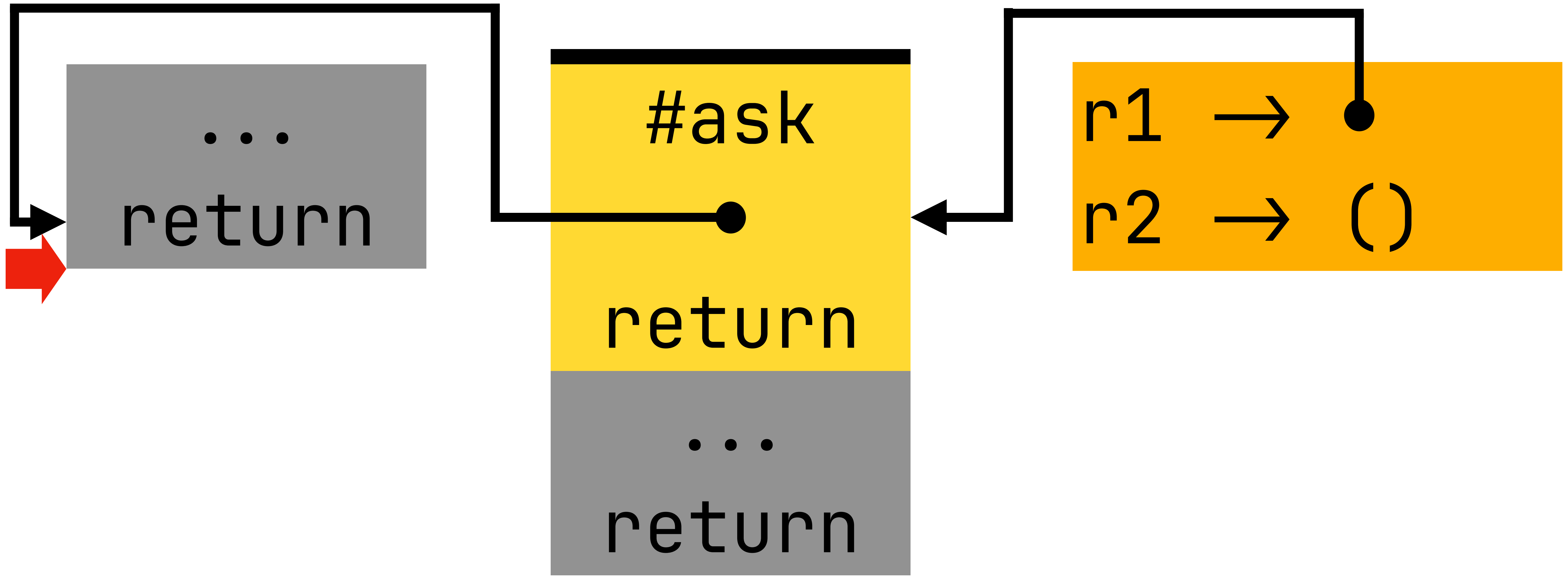
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

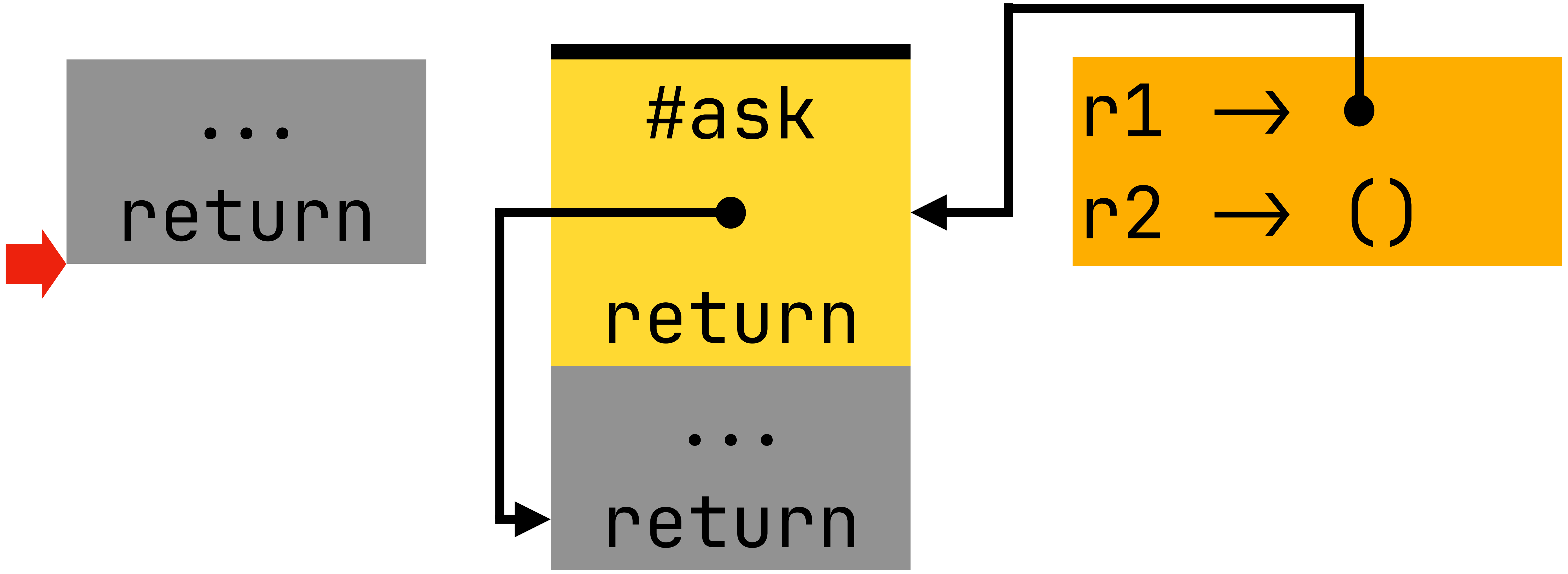
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



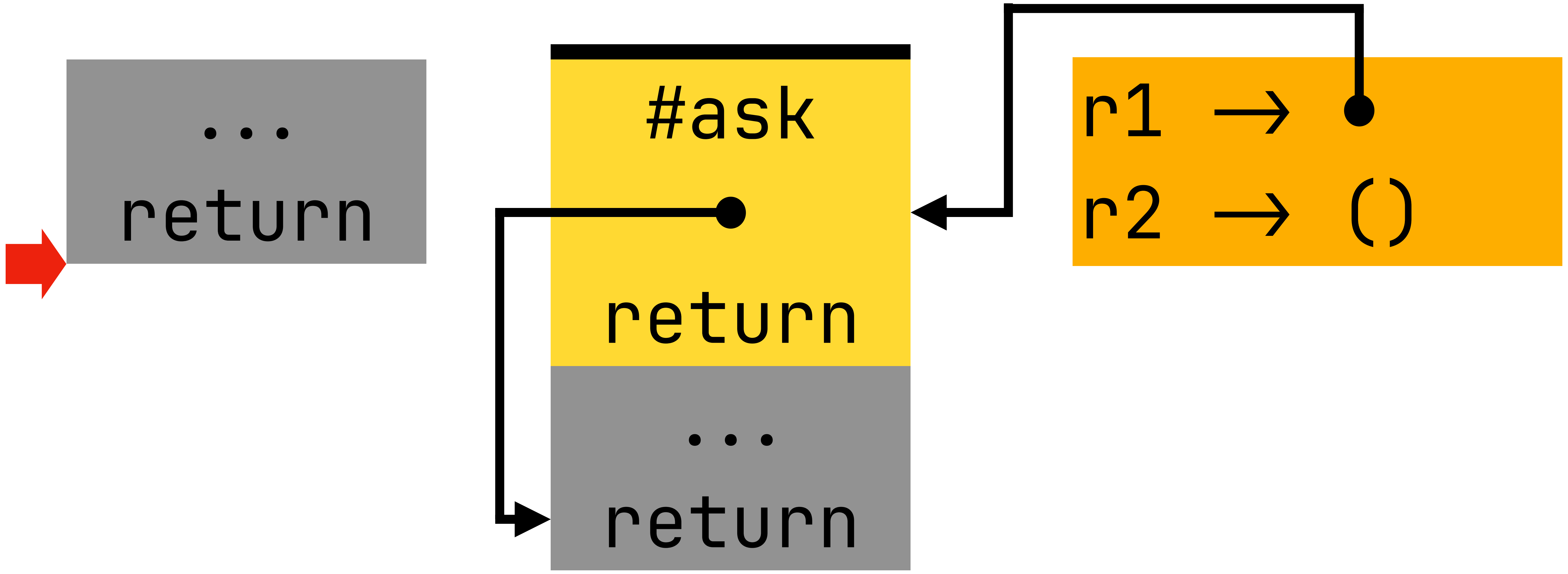
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

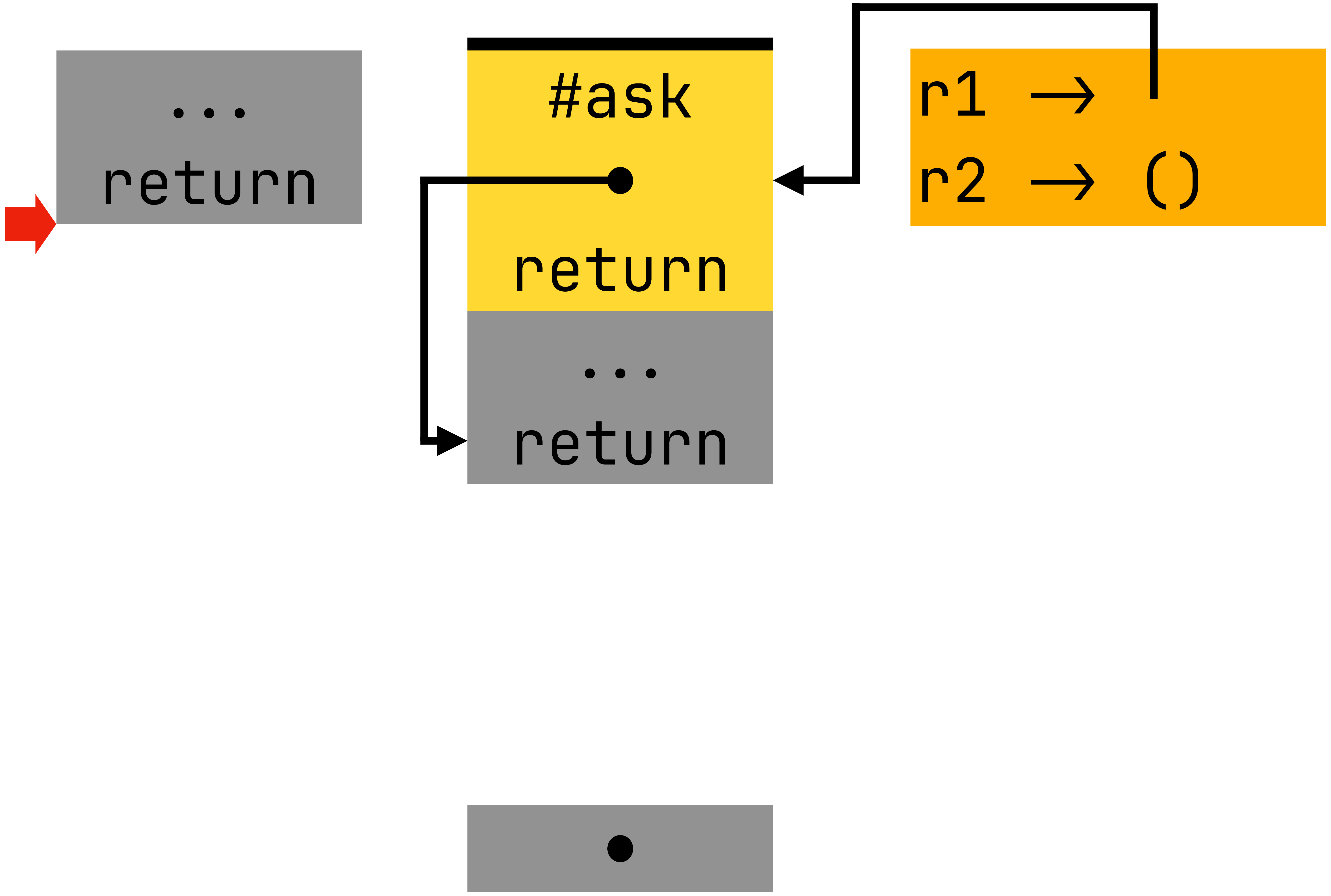
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

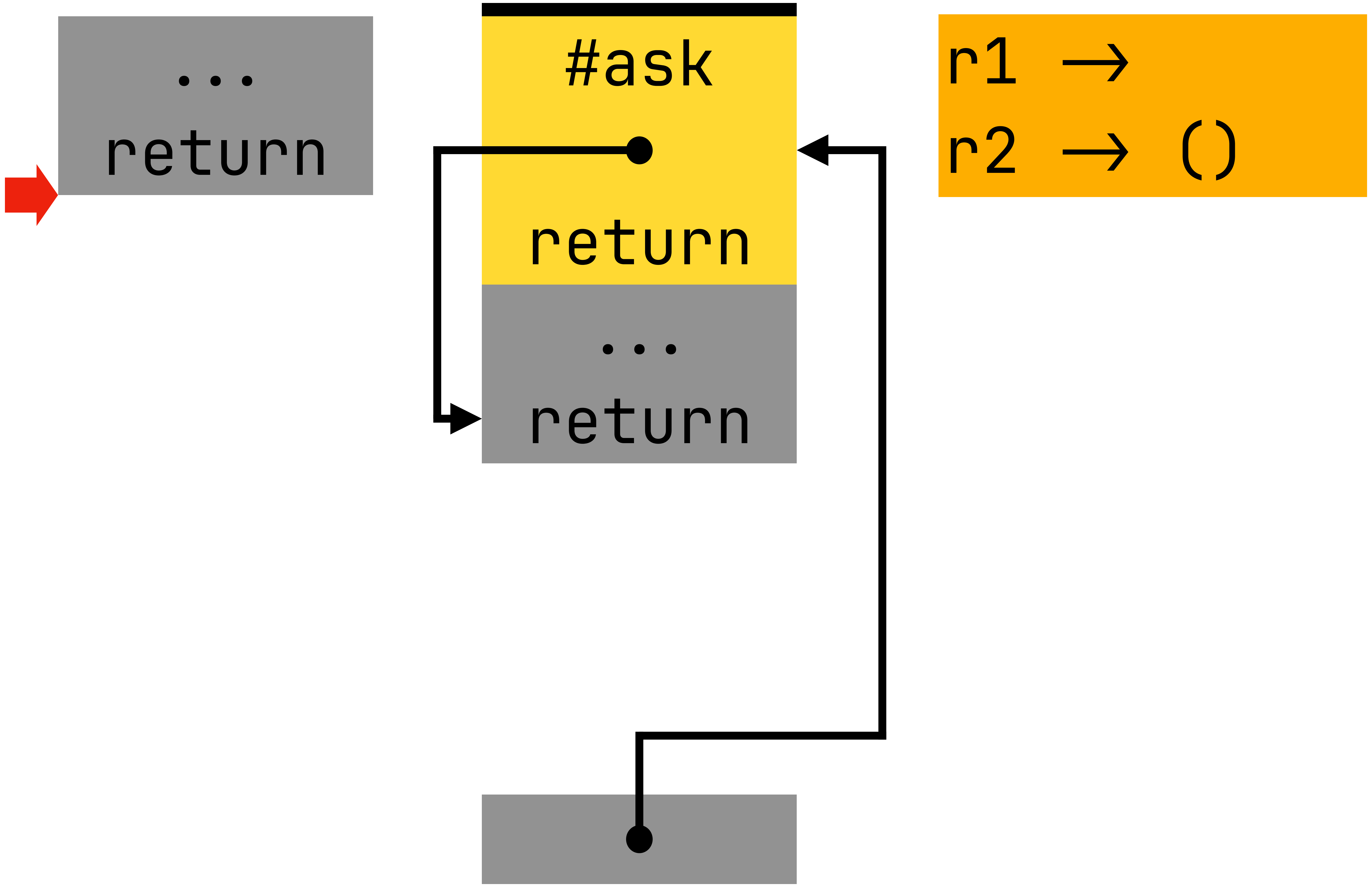
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

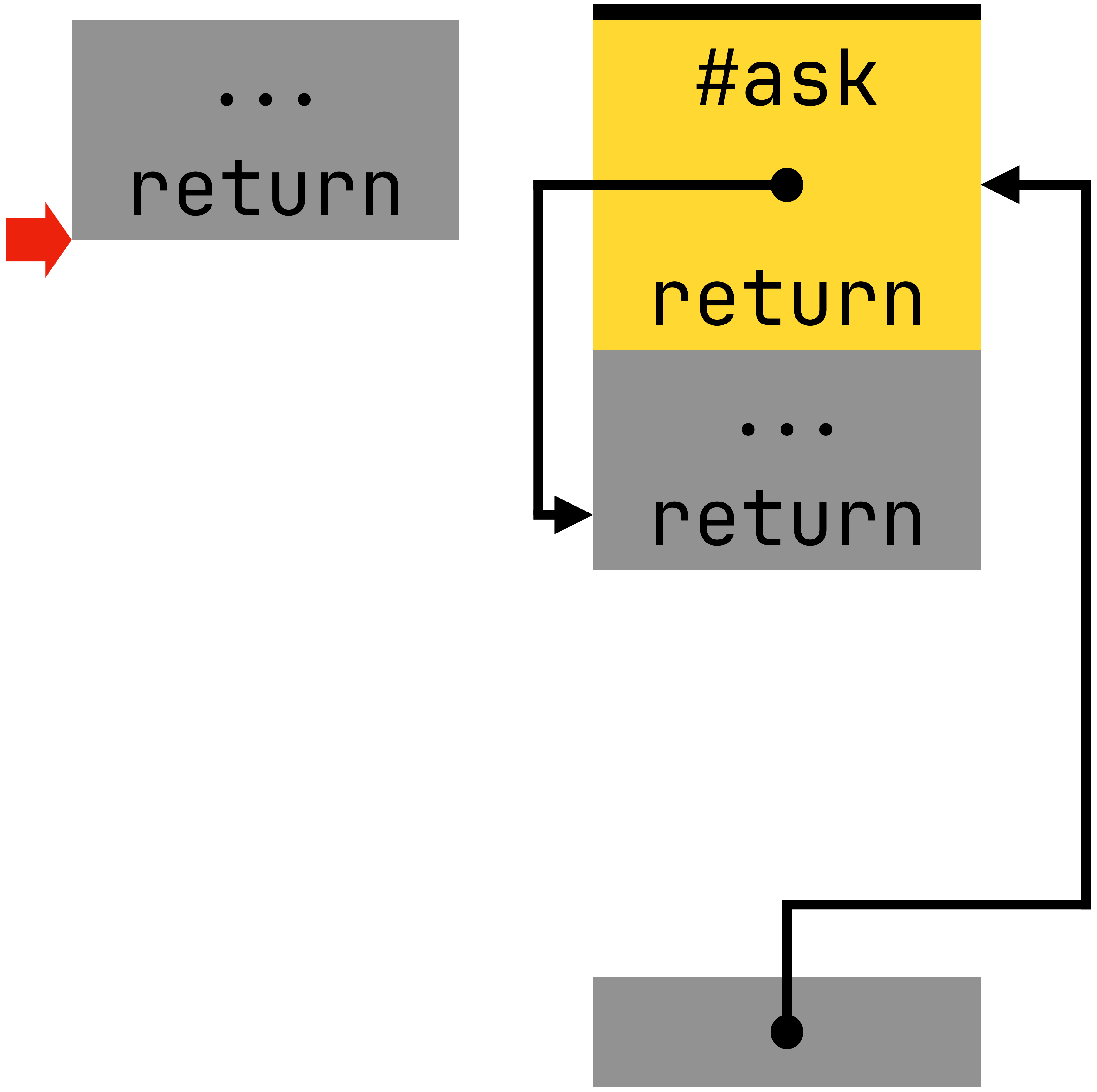
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

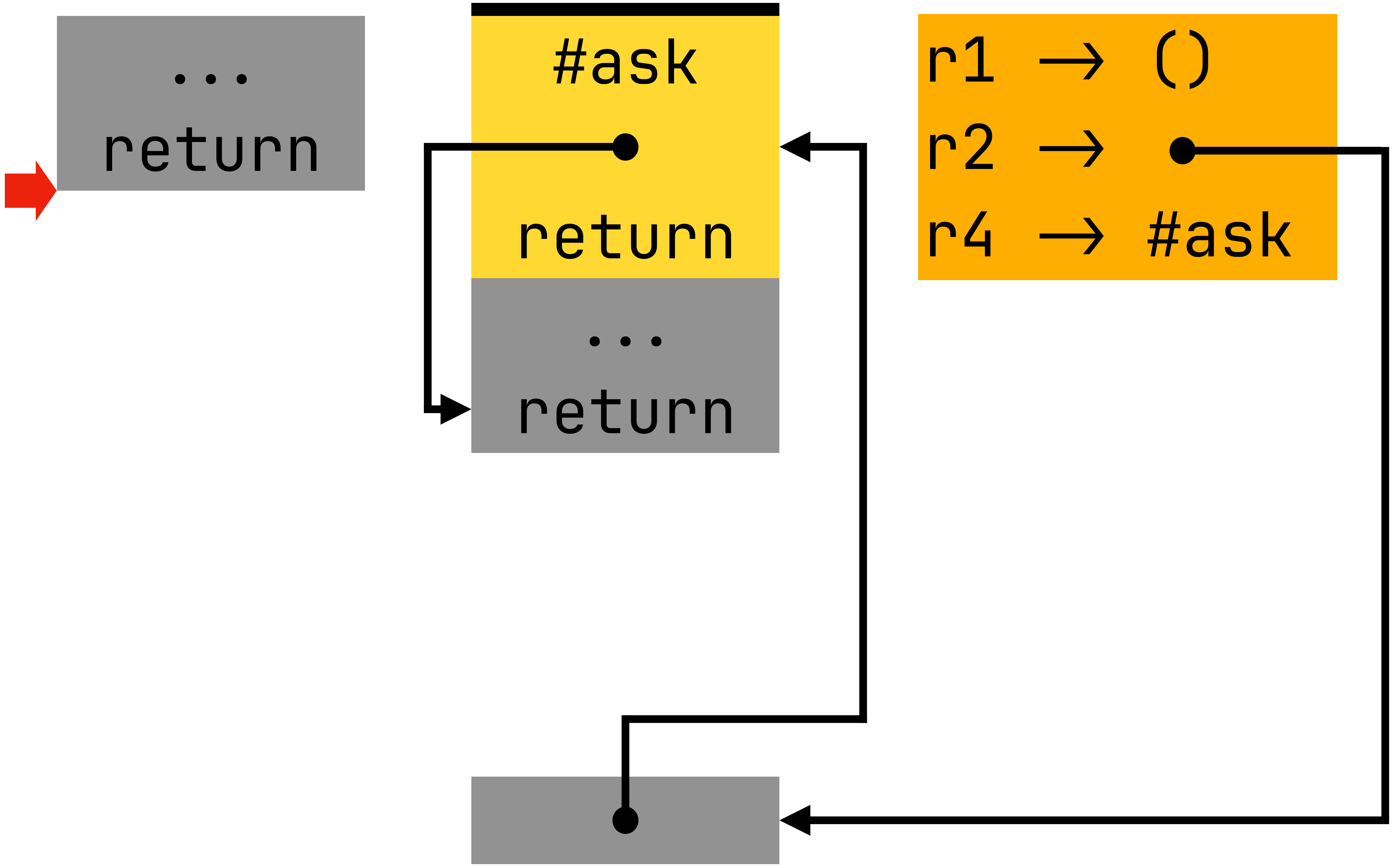
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



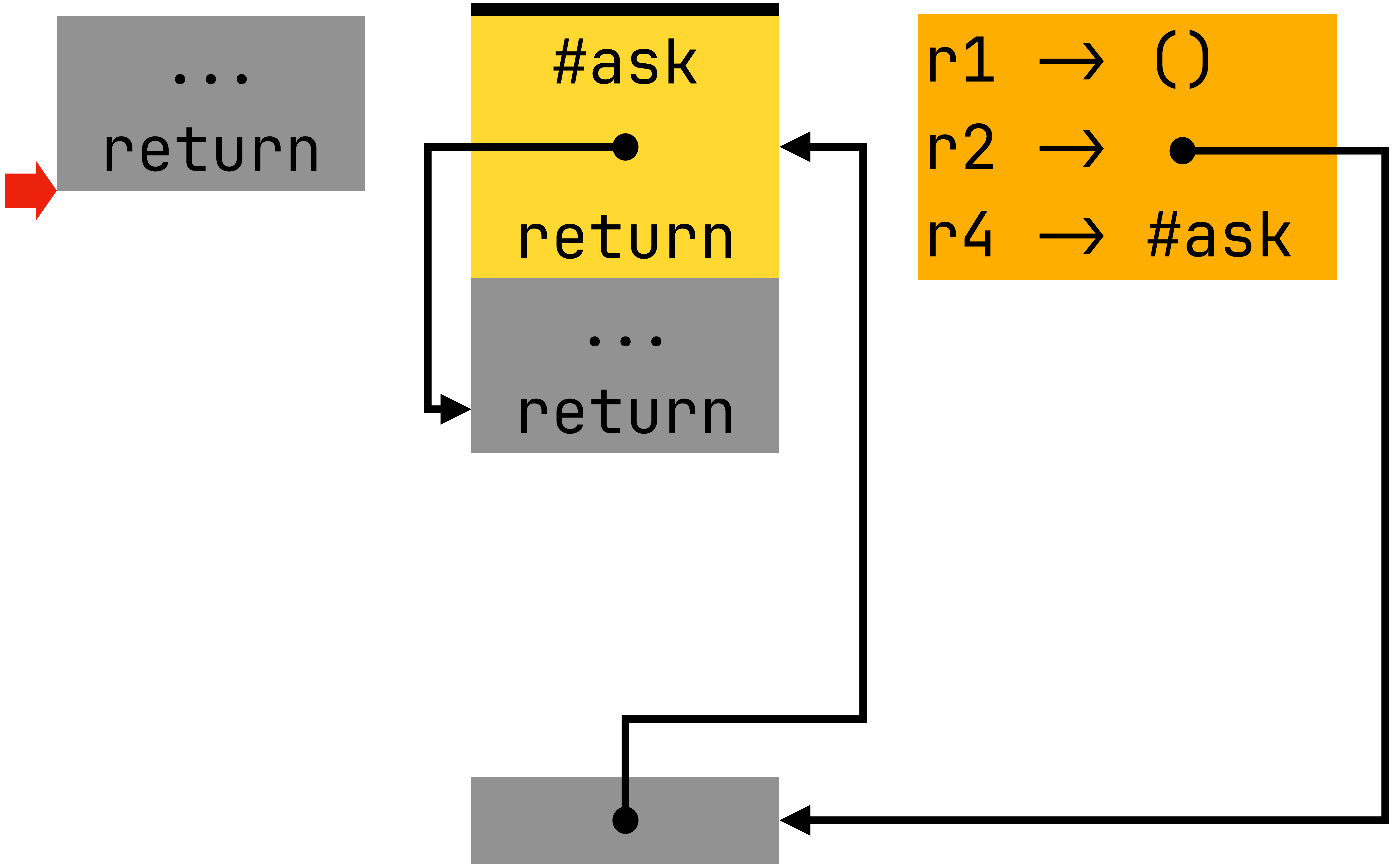
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

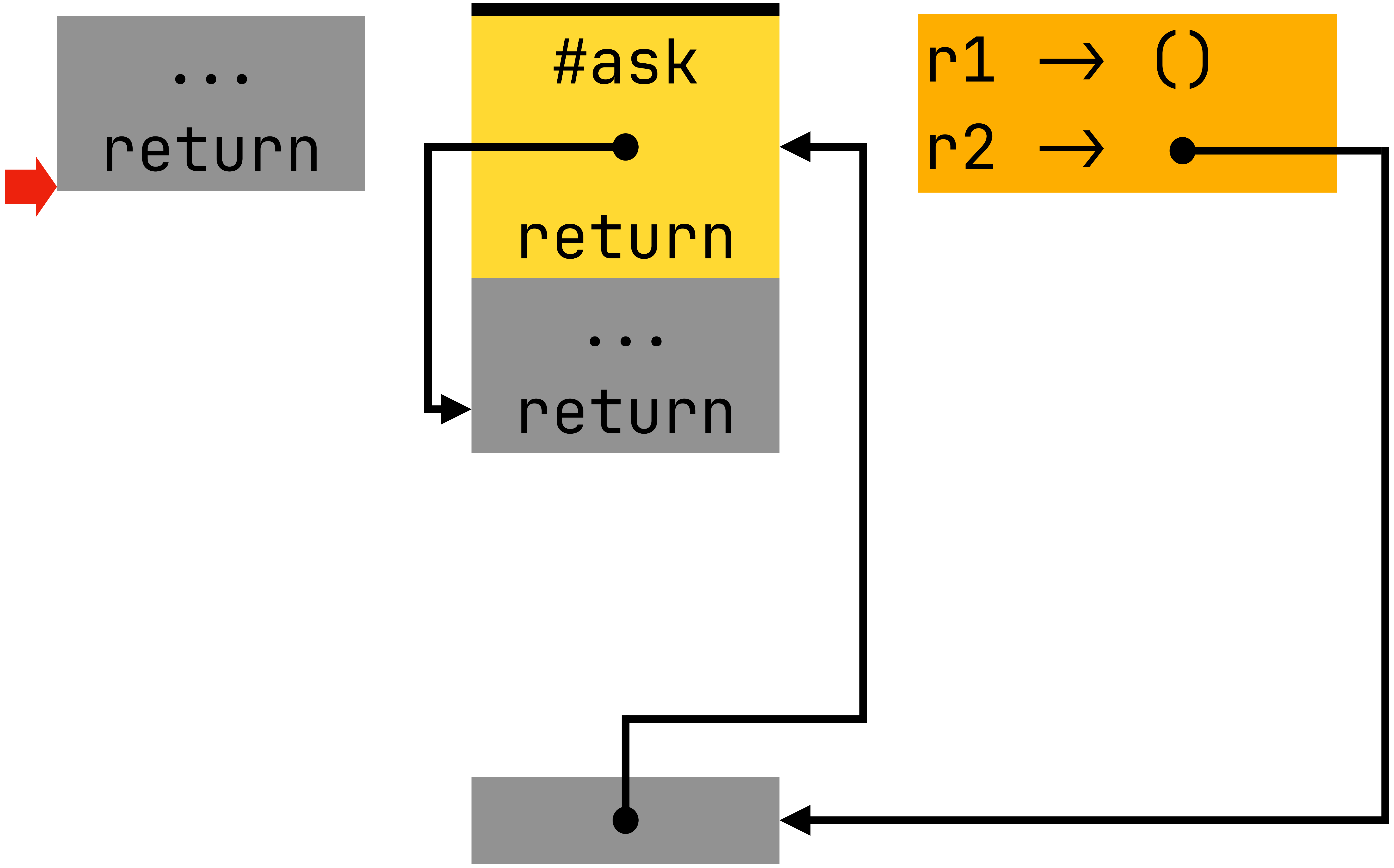
```
#RAISE: load r3, [r1]
        store [r1], sp
        mov sp, r3
        load r4, [r1+1]
        malloc r3, 1
        store [r3], r1
        mov r1, r2
        mov r2, r3
        jmp r4
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

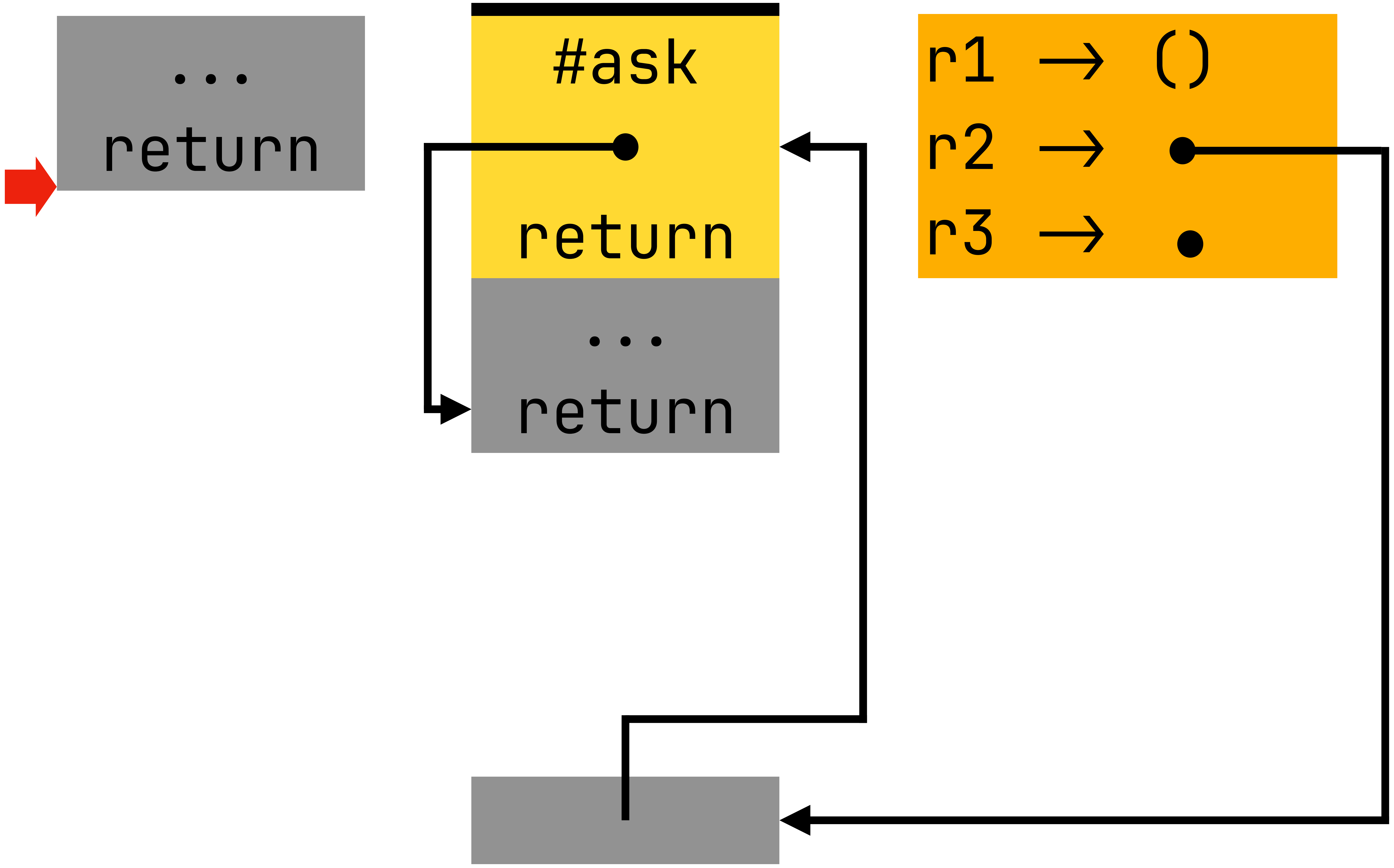
```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



Code Stack Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



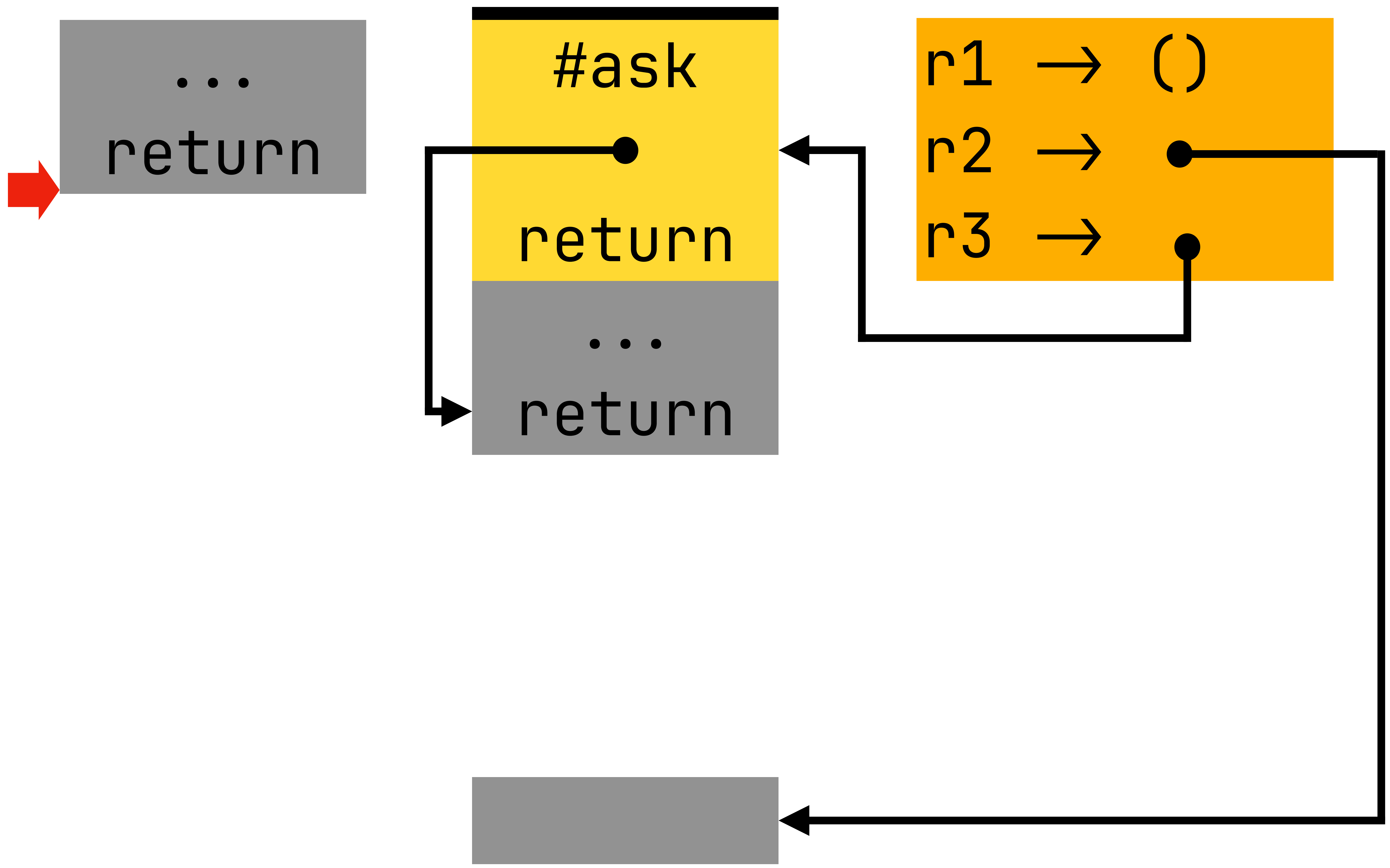
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



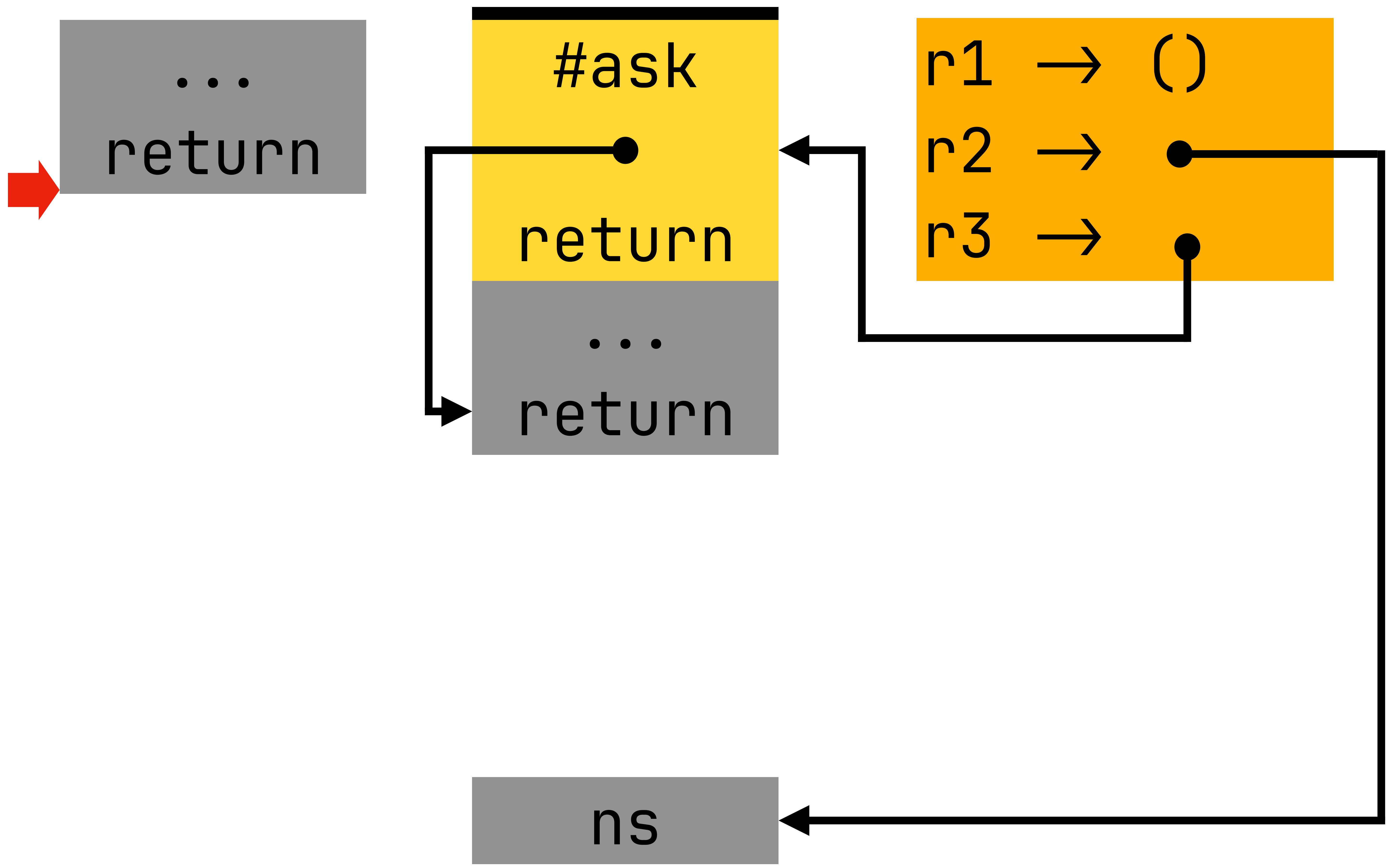
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



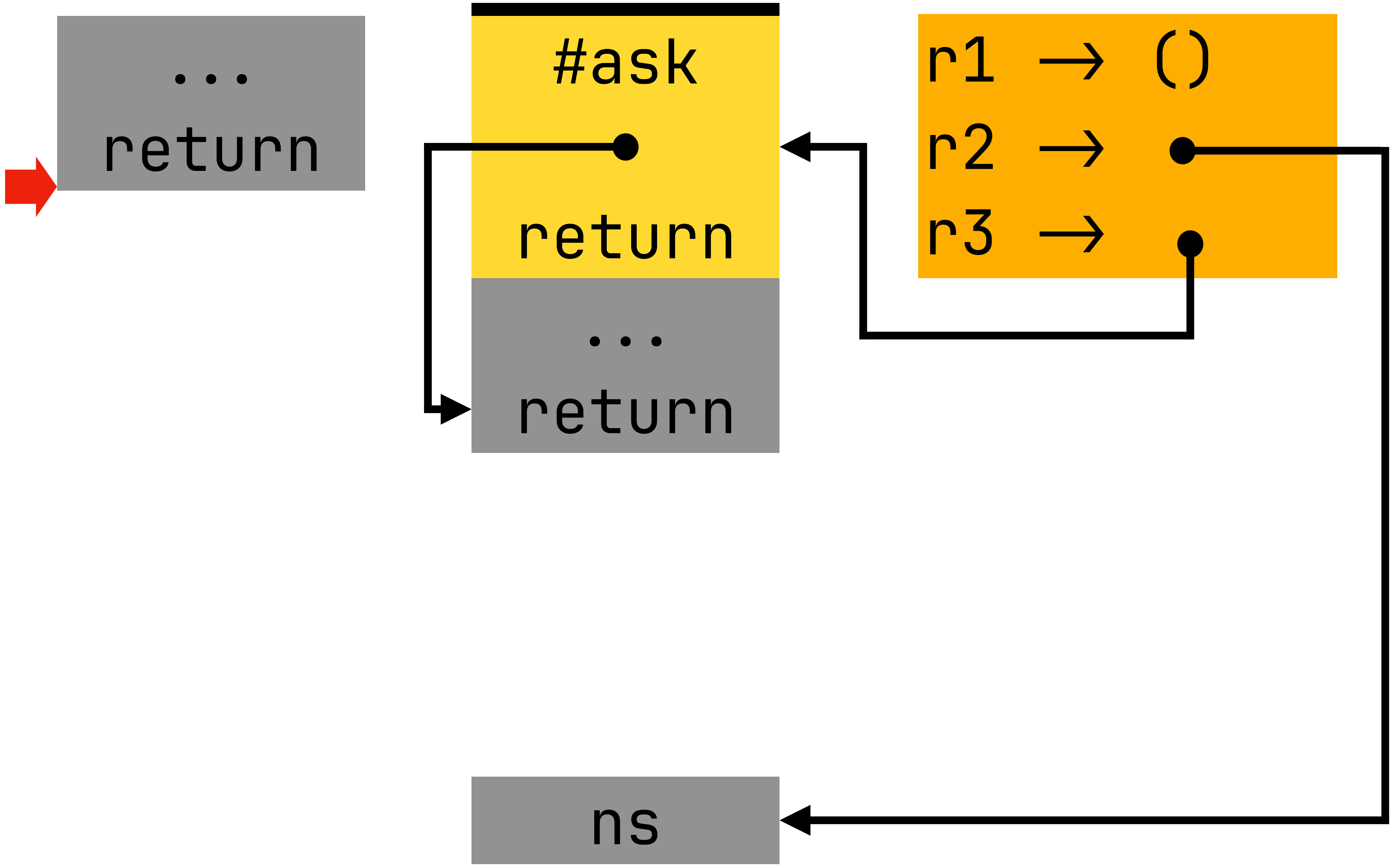
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



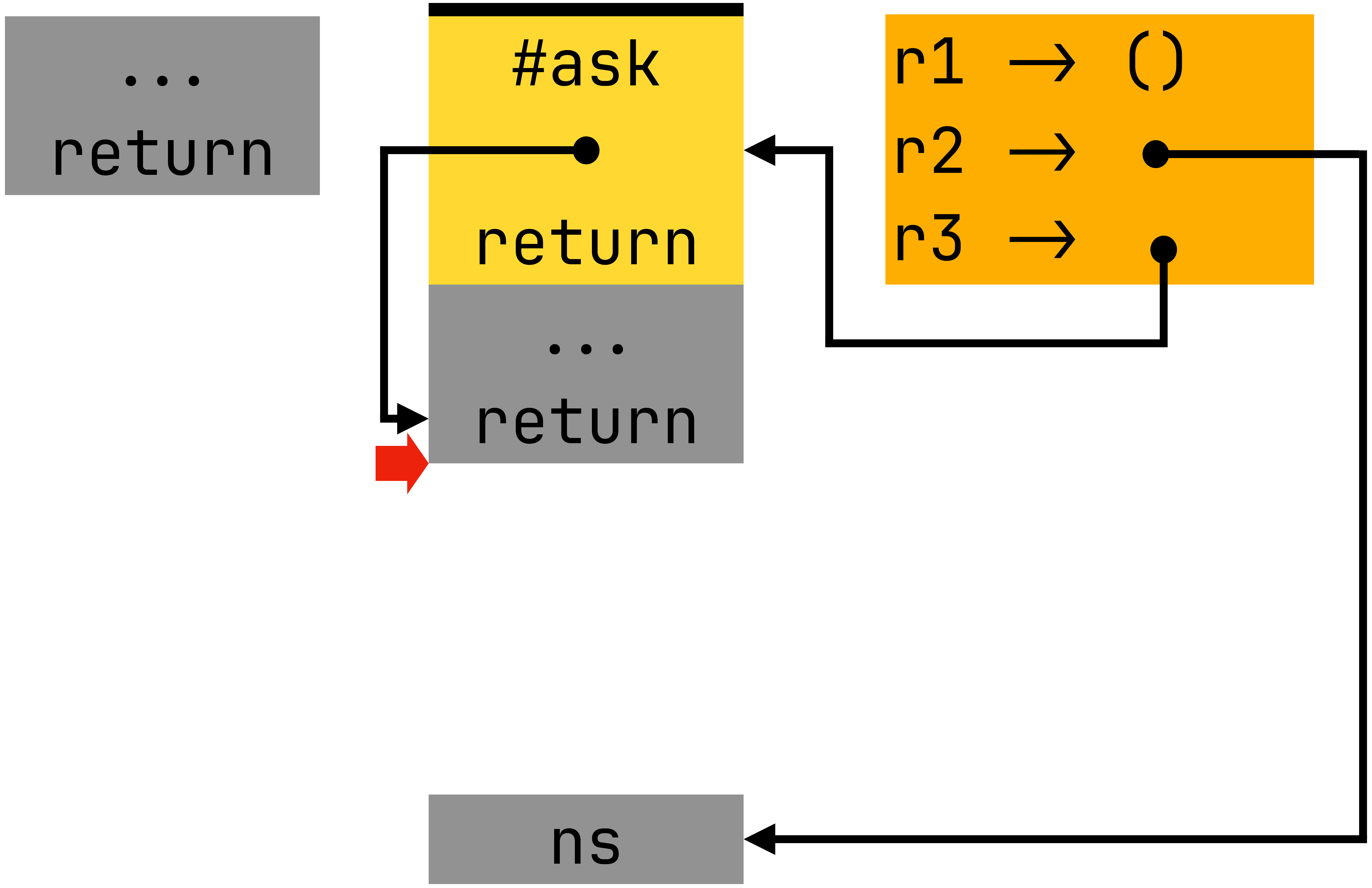
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



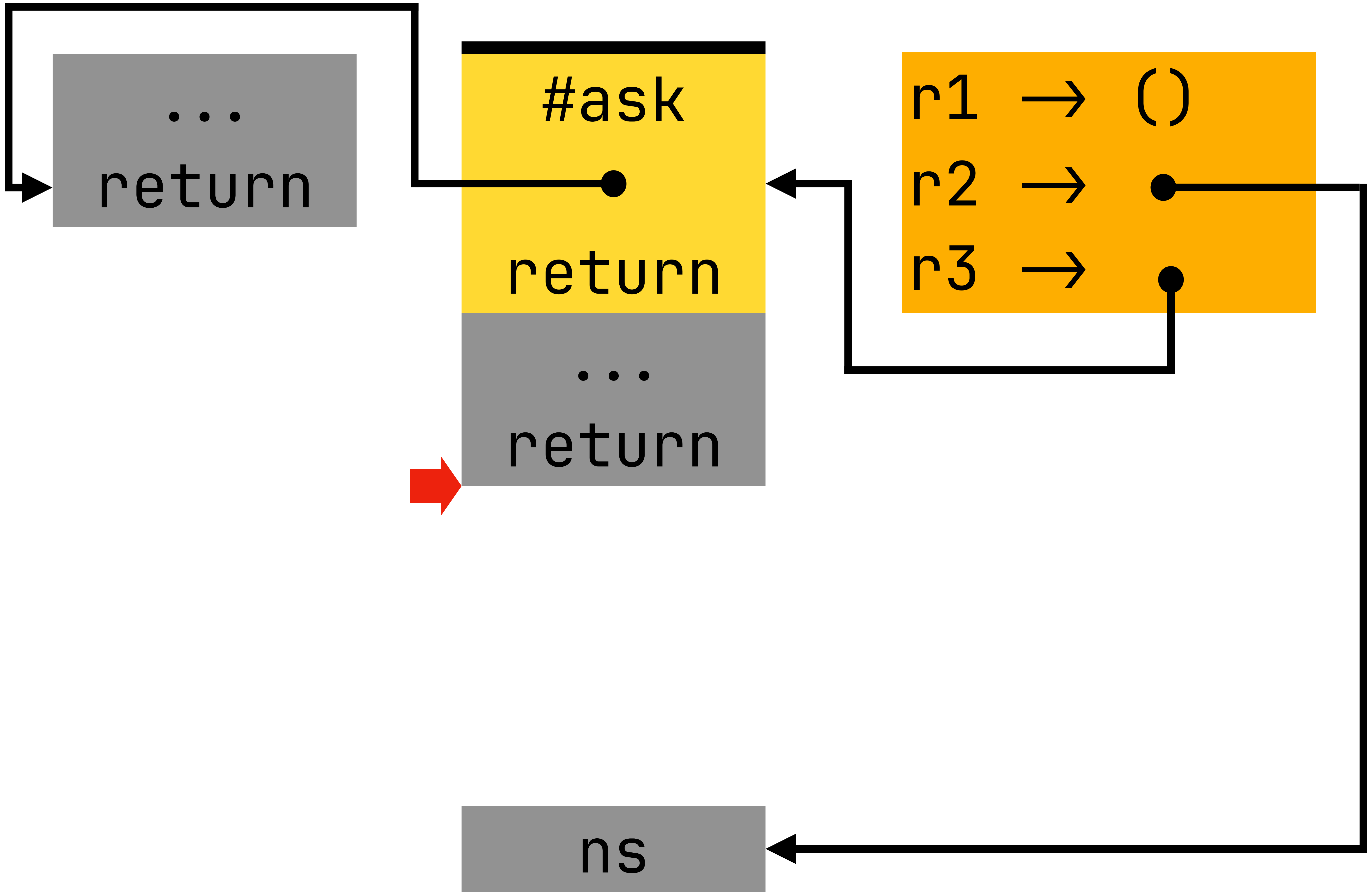
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



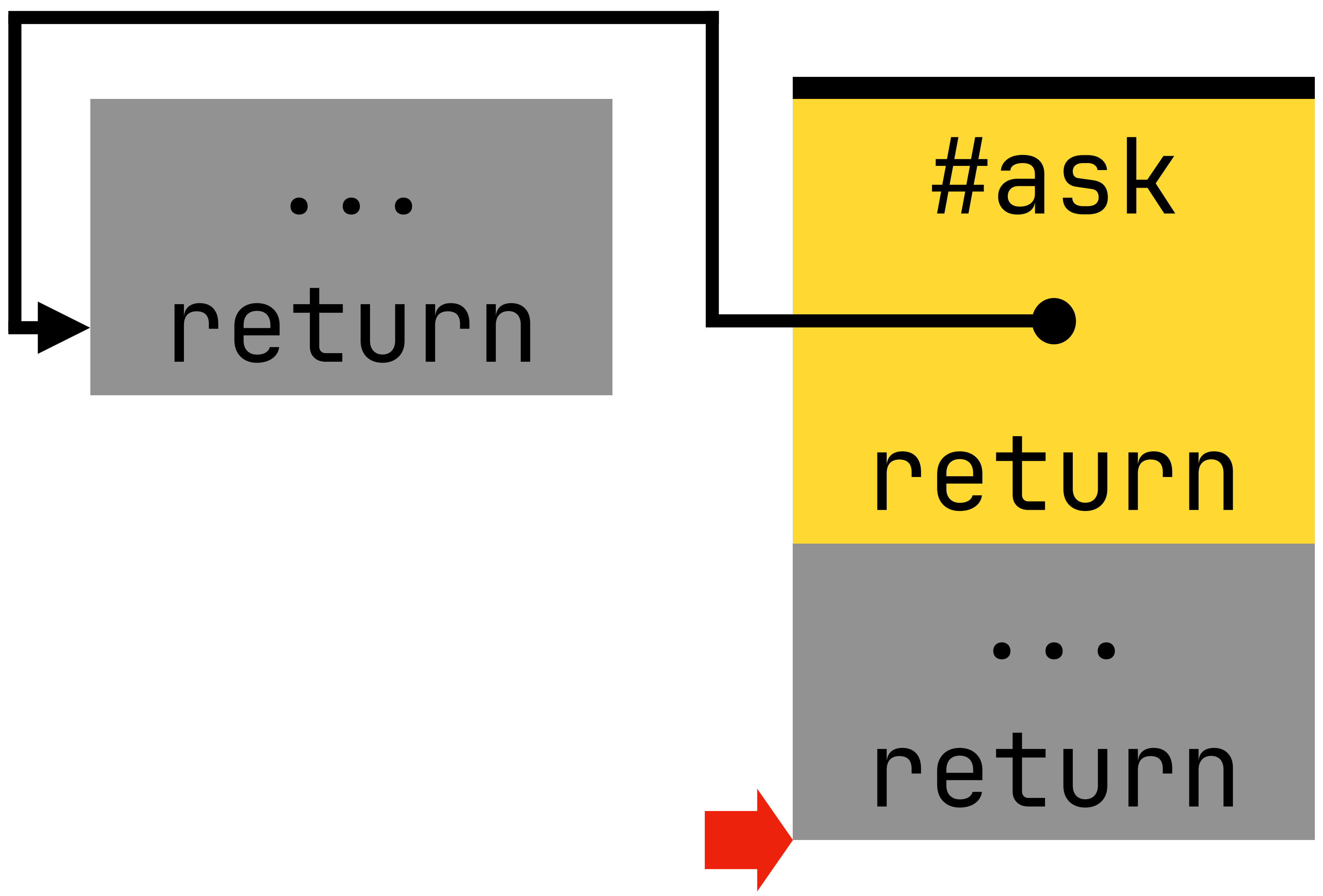
Code

Stack

Register

```
handle { ...; raise ask(); ... } with ask = λx.λk. resume k ()
```

```
#RESUME: load r3, [r2]
         store [r2], ns
         load r4, [r3]
         store [r3], sp
         mov sp, r4
         ret
```



r1 → ()

Code

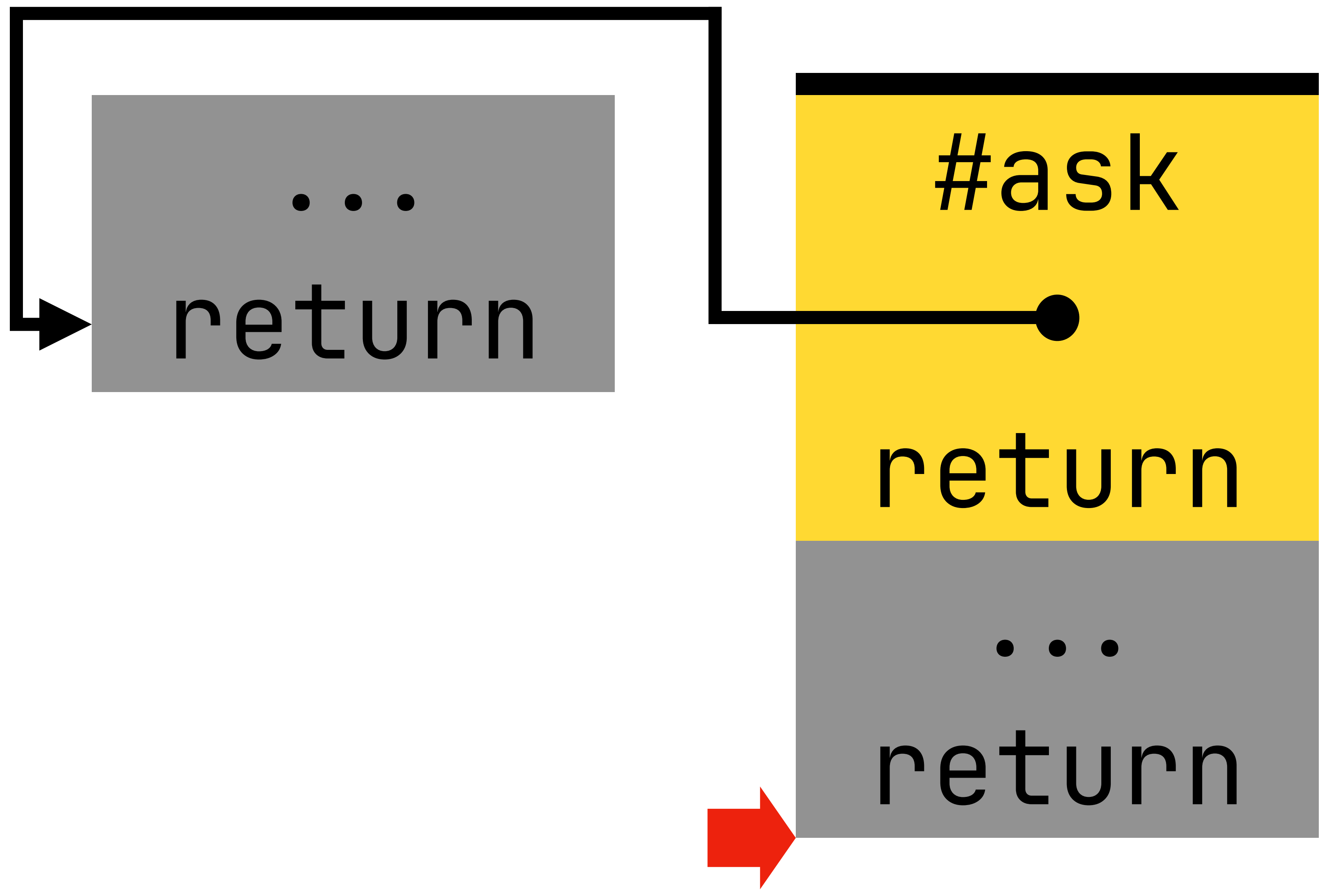
Stack

Register

Compiling lexical effect handlers

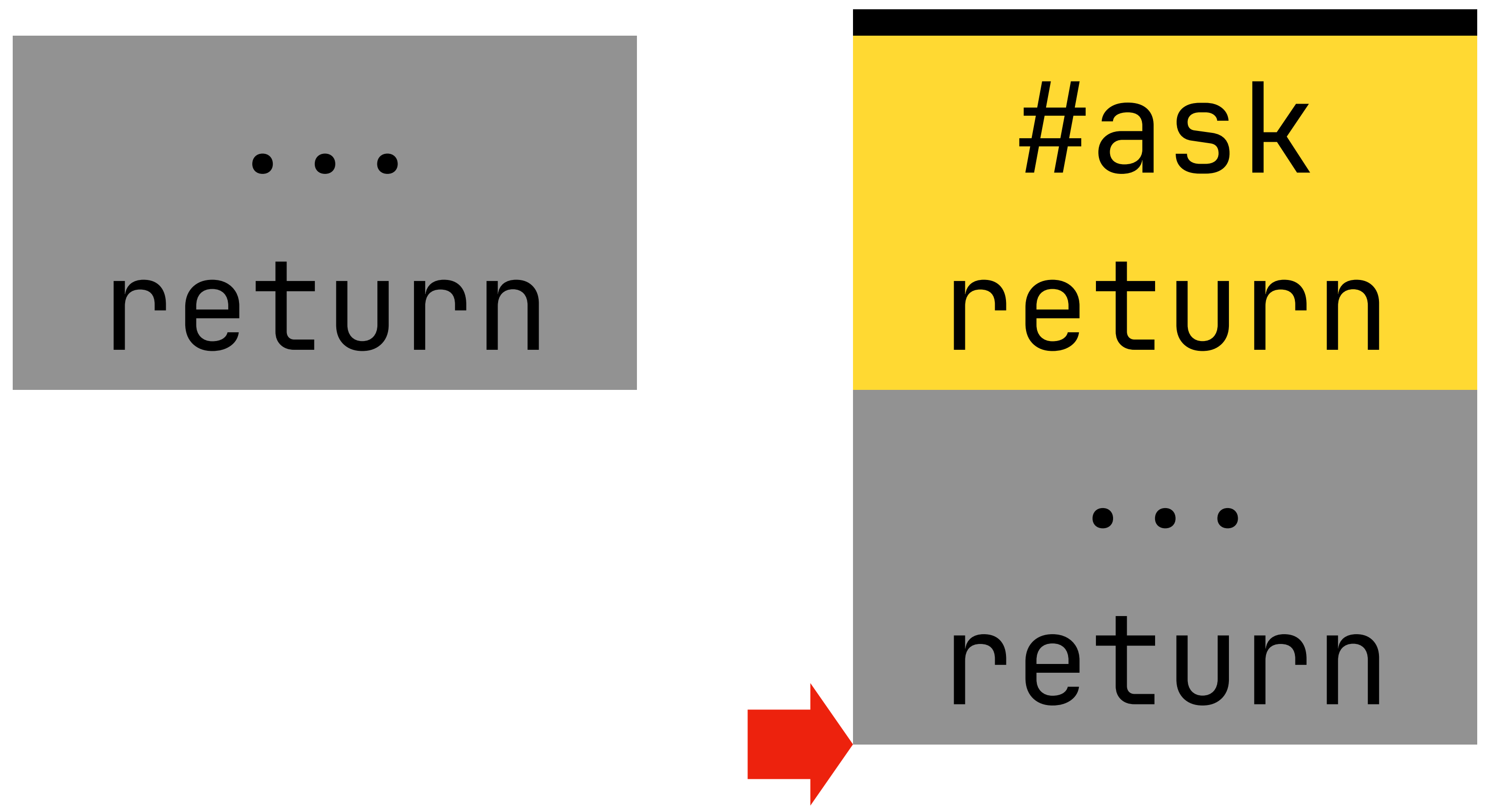
Optimization for tail-resumptive and abortive handler

```
λx.λk. resume k ()
```



Compiling lexical effect handlers

Optimization for tail-resumptive and abortive handler



Compiling lexical effect handlers

Optimization for tail-resumptive and abortive handler

high-level, modular algebraic effects in Lexa

```
handle E with H
raise …
resume …
```

this paper

low-level, swift stack switching in assembly

```
ENTER
RAISE
RESUME
```

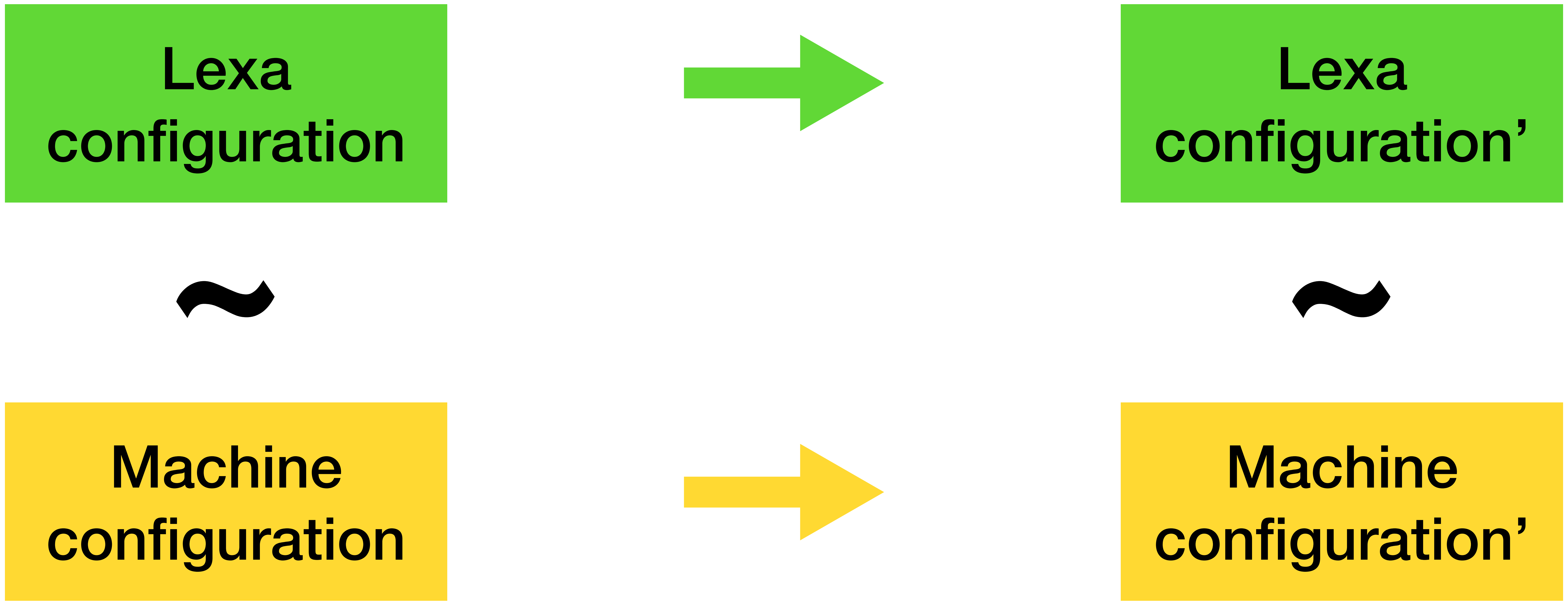
Time to find the handler: O(1)

Time to capture the continuation: O(1)

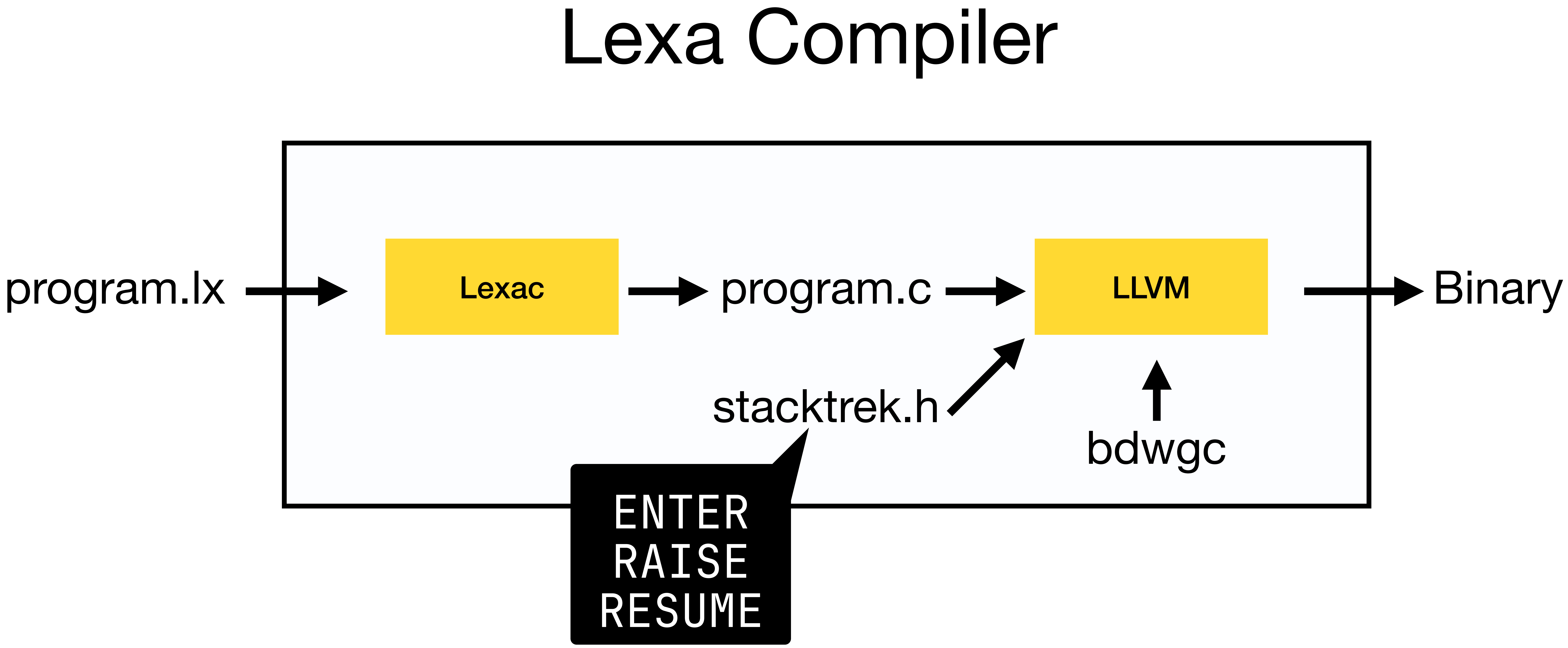
⟦source program⟧ = ⟦compiled program⟧

Lexa formalism

Semantic preservation



Lexa compiler implementation



Lexa compiler implementation

Saving register state: setjmp vs calling conventions

Lexa compiler implementation

```
q = new Queue()
handle {
   ...
  job()
   ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

Lexa compiler implementation

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

Lexa compiler implementation

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

Lexa compiler implementation

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}



...
retval = RESUME(k', ());
return retval;
```

C Code

Lexa compiler implementation

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k’=q.pop();
    resume k’()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}

...
retval = RESUME(k’, ());
return retval;
```

Control goes to a different stack, whose execution can clobber the registers in use

C Code

Lexa compiler implementation

Saving register state: setjmp-style

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}
```

Control goes to a different stack, whose execution can clobber the registers in use

```
...
// Code that saves regs
retval = RESUME(k', ());
// Code that restores regs
return retval;
```

C Code

Lexa compiler implementation

Saving register state: setjmp-style

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}


...
// Code that saves regs
retval = RESUME(k', ());
// Code that restores regs
return retval;
```

RESUME is not a tail-call anymore.

C Code

Lexa compiler implementation

Saving register state: preserve-none calling-convention

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}

...
// Code that saves regs
retval = RESUME(k', ());
// Code that restores regs
return retval;
```

C Code

Lexa compiler implementation

Saving register state: preserve-none calling-convention

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}


...
retval = RESUME(k', ());
return retval;
```

C Code

Lexa compiler implementation

Saving register state: preserve-none calling-convention

```
q = new Queue()
handle {
  ...
  job()
  ...
} with Yield =
  λx,k.
    q.push(k);
    k'=q.pop();
    resume k'()
  with Fork = ...
```

Lexa Code

```
__attribute__((preserve_none))
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
  ret
}


...
retval = RESUME(k', ());
return retval;
```

C Code

Lexa compiler implementation

Saving register state: preserve-none calling-convention

```
q = new Queue()
handle {
  ...
```

```
__attribute__((preserve_none))
i64 RESUME(i64, 64) {
  ...
  mov sp, r4
```

Saving registers through the calling convention
gives the compiler more control over optimizations.

```
    q.push(k);
    k’=q.pop();
    resume k’()
  with Fork = ...
```

```
...
retval = RESUME(k’, ());
return retval;
```

RESUME is tail-call again.

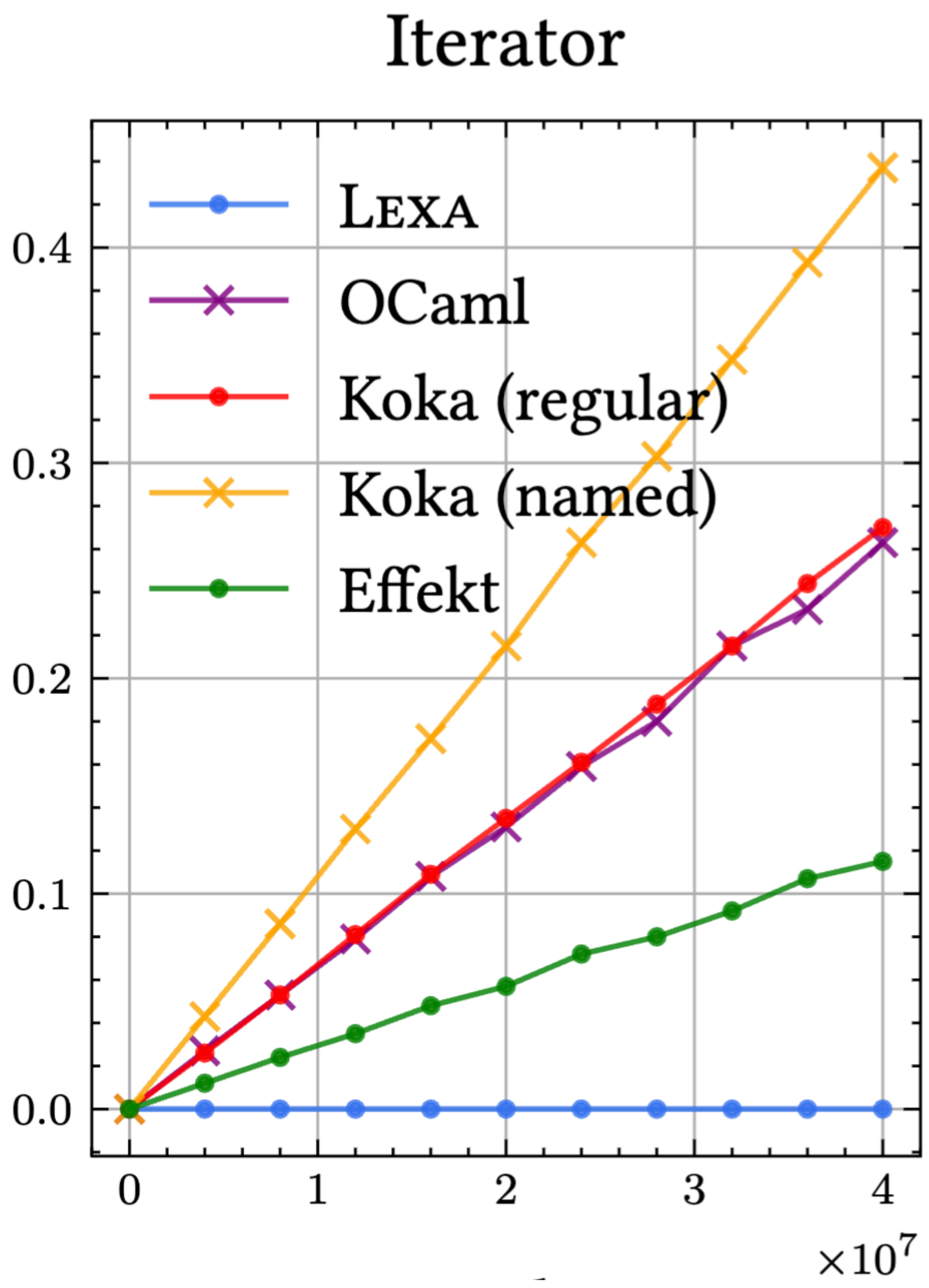
Lexa Code

C Code

Evaluation

We evaluate Lexa on a set of community maintained benchmarks: https://github.com/effect-handlers/effect-handlers-bench

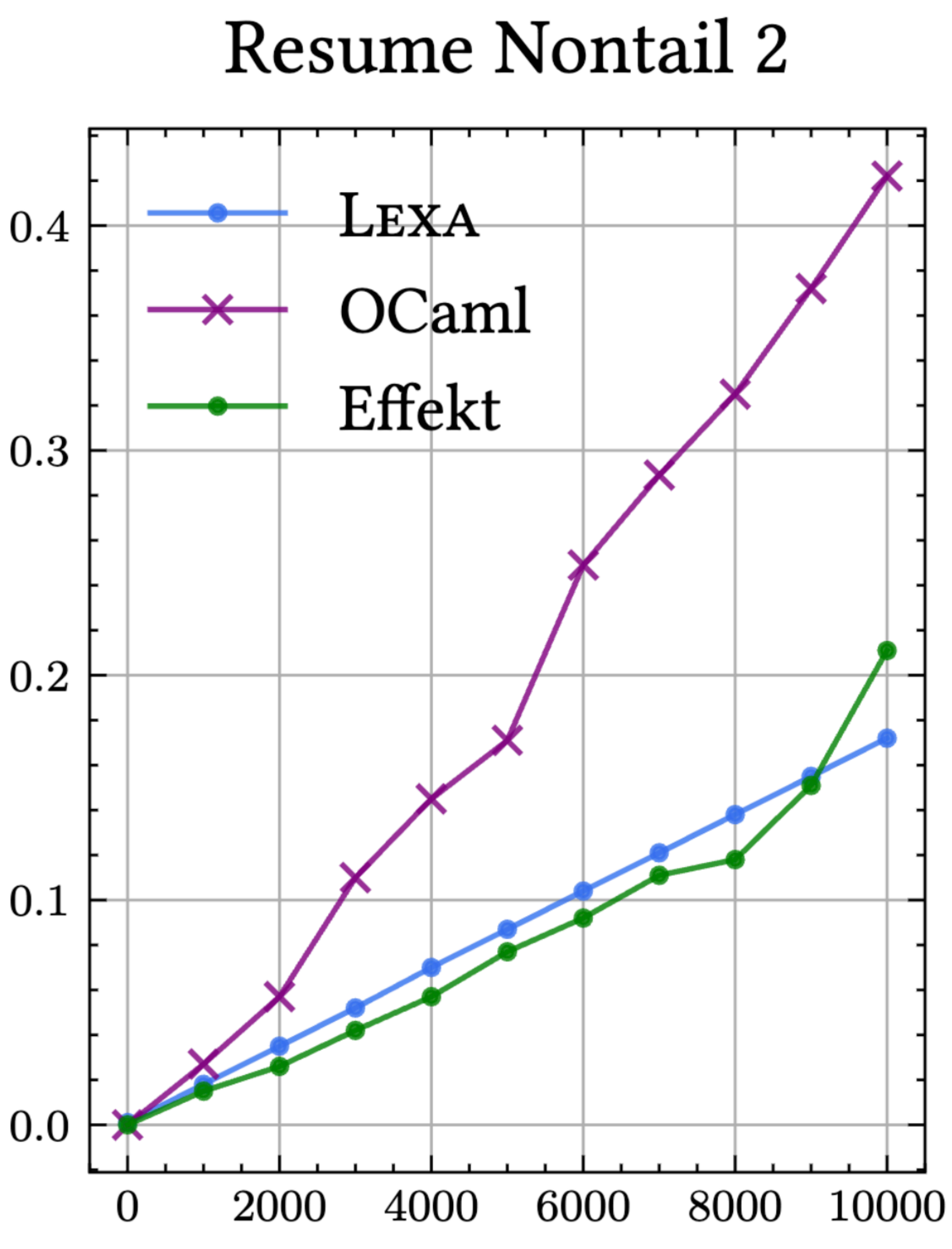
Evaluation: Iterator

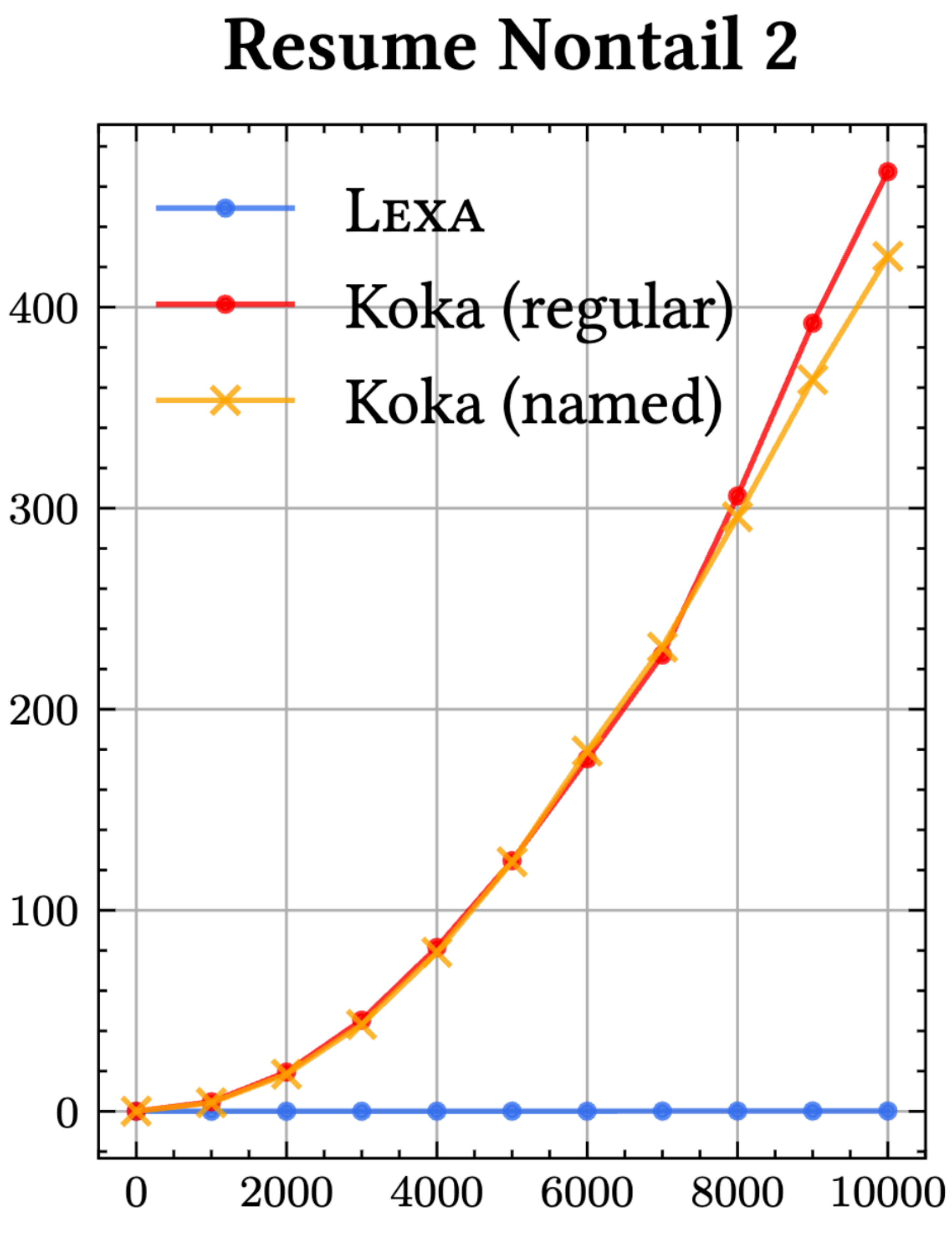


Lexa enjoys constant runtime.

Evaluation: Resume Nontail 2

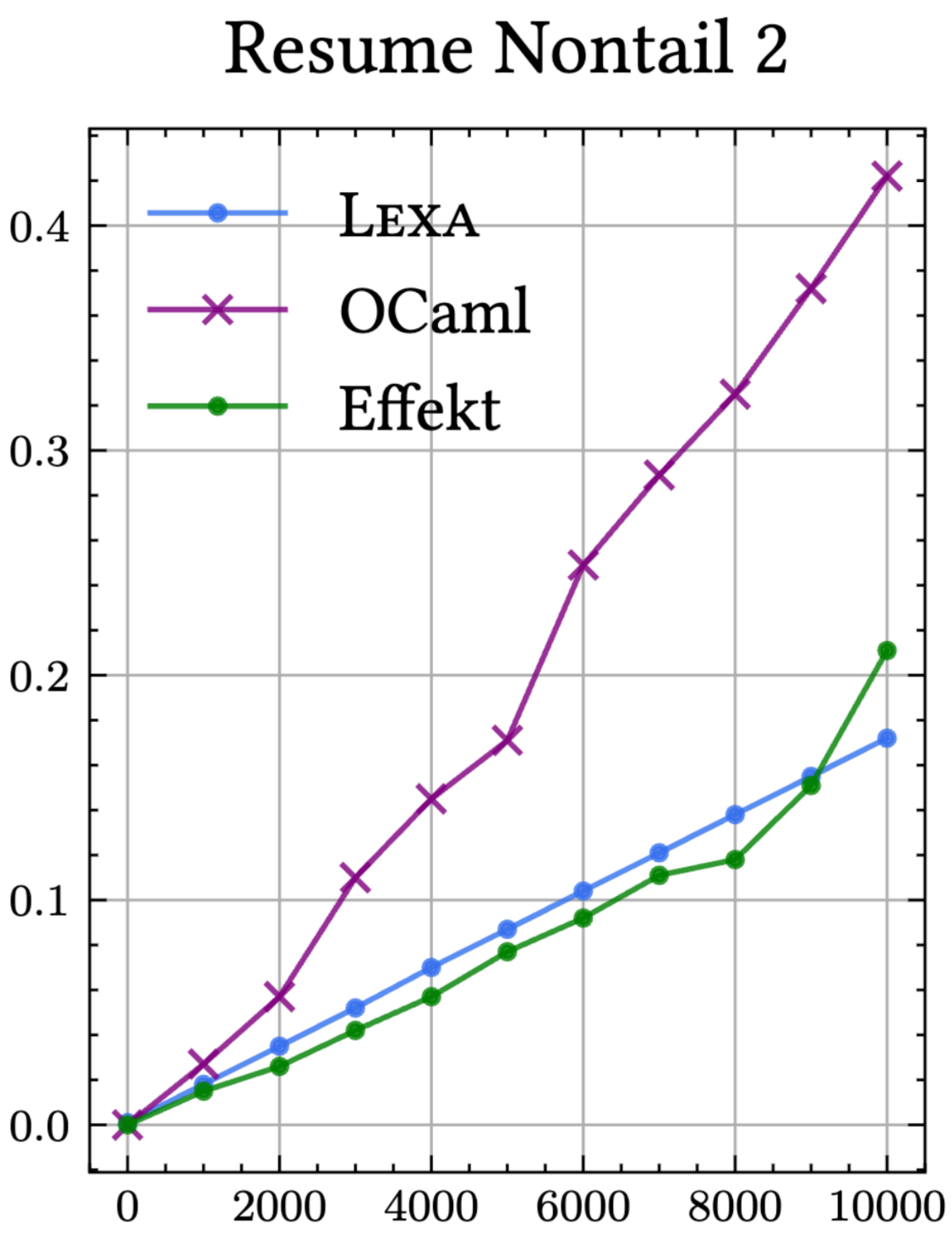
This benchmark checks how efficient a language captures continuations.

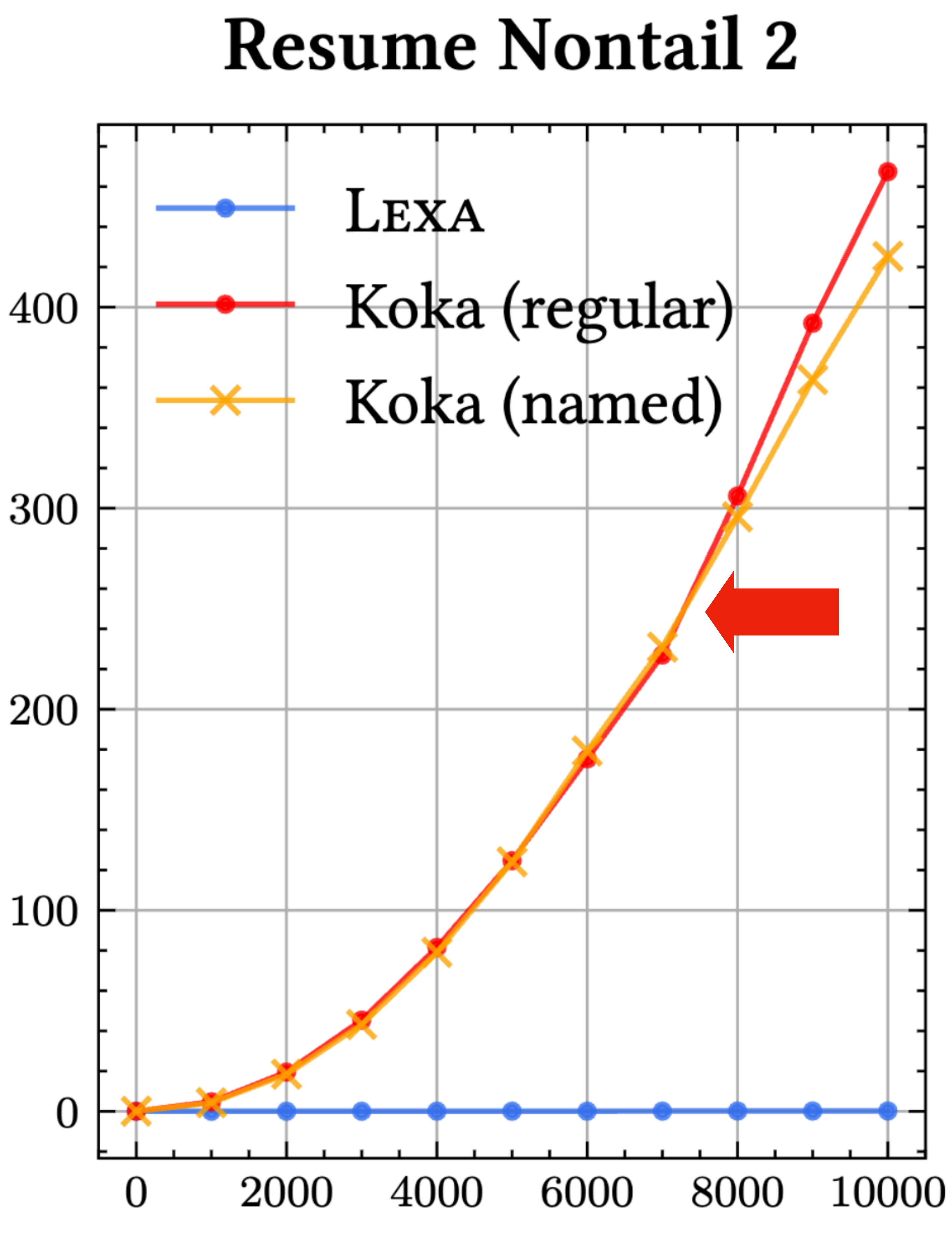




Evaluation: Resume Nontail 2

This benchmark checks how efficient a language captures continuations.



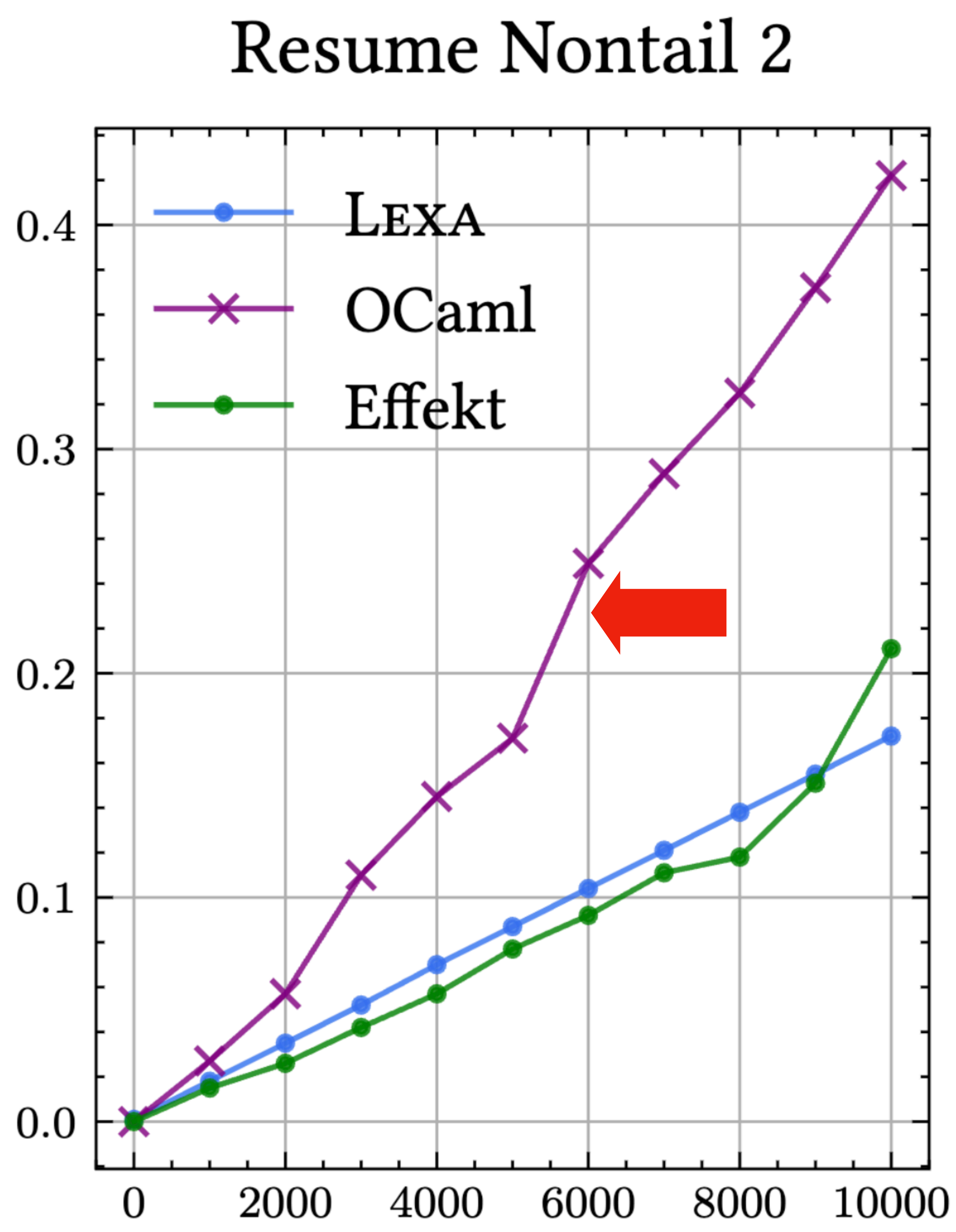


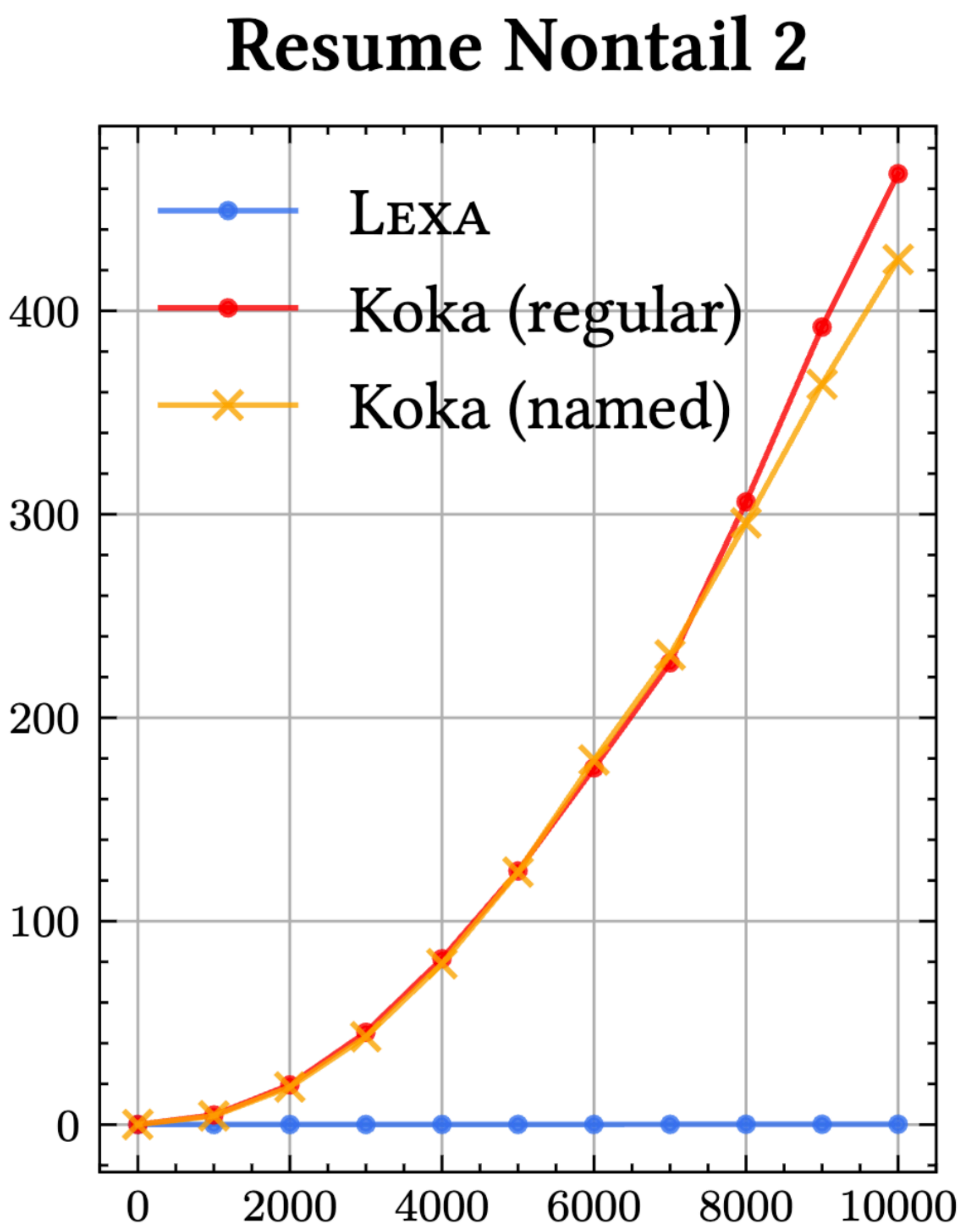
Koka has a polynomial runtime trend because it builds continuation frame by frame.

Evaluation: Resume Nontail 2

This benchmark checks how efficient a language captures continuations.

OCaml is step-wise linear because it copies and doubles the stack upon overflow.

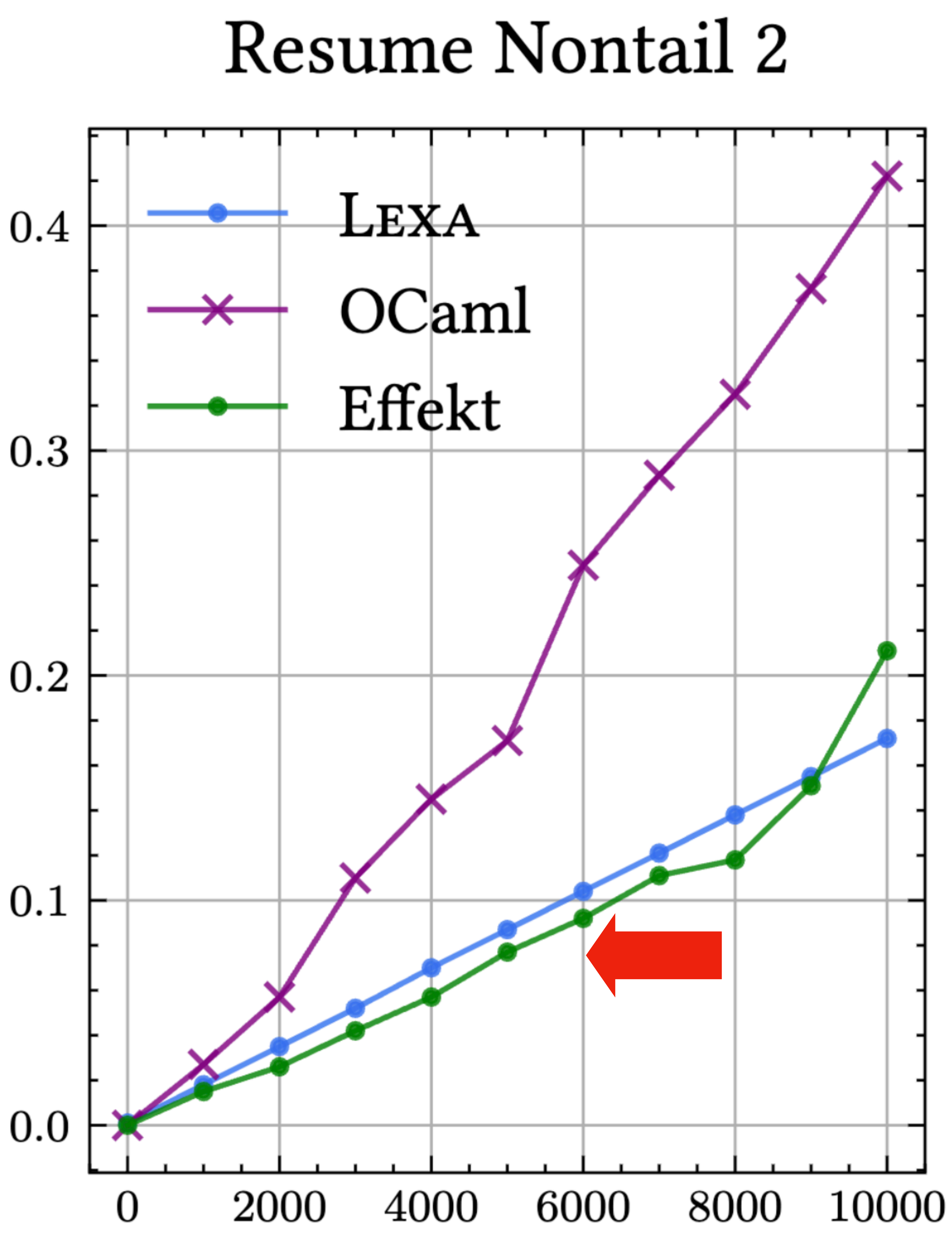


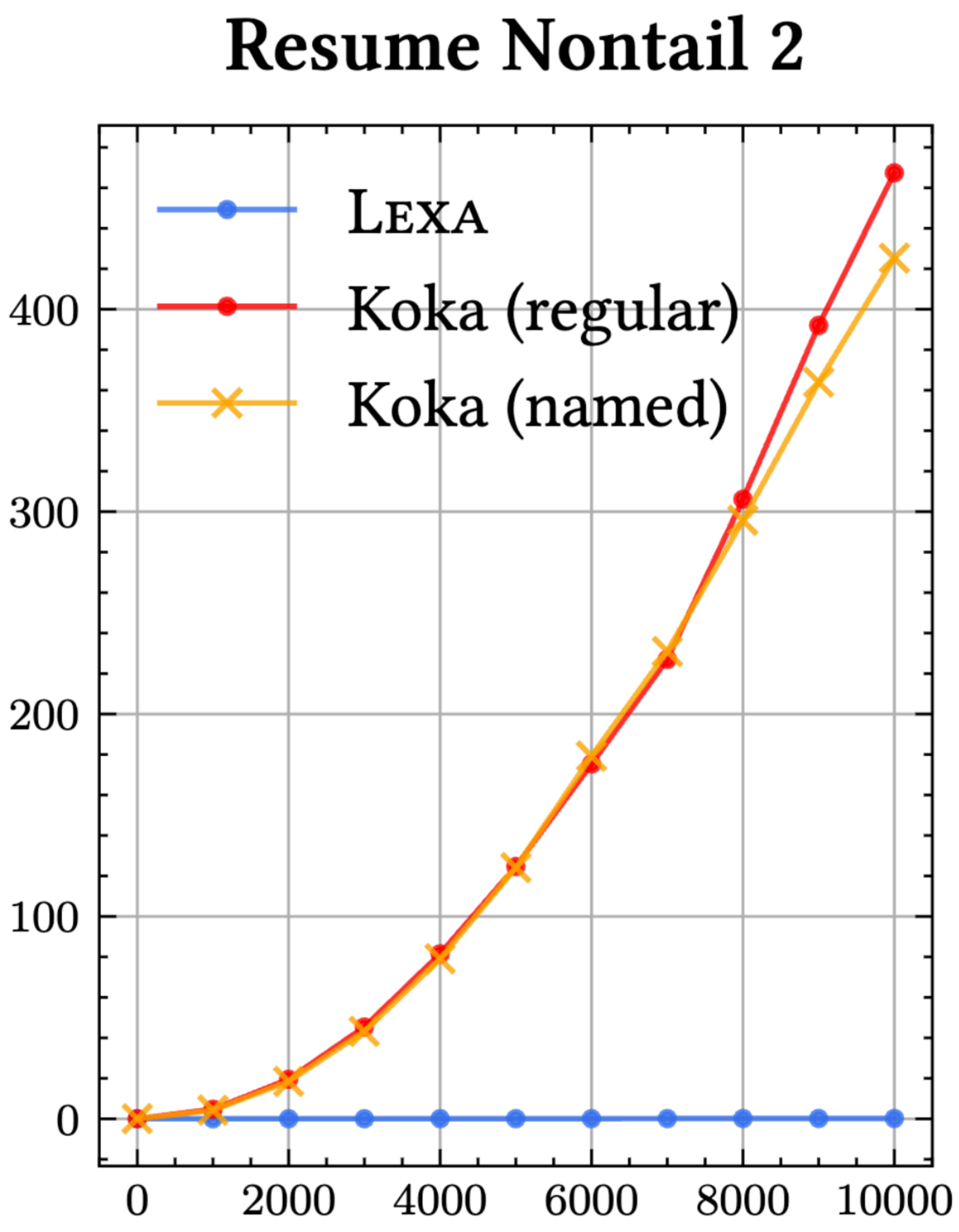


Evaluation: Resume Nontail 2

This benchmark checks how efficient a language captures continuations.

Effekt compiles by CPS, so no extra effort is required to capture a continuation.

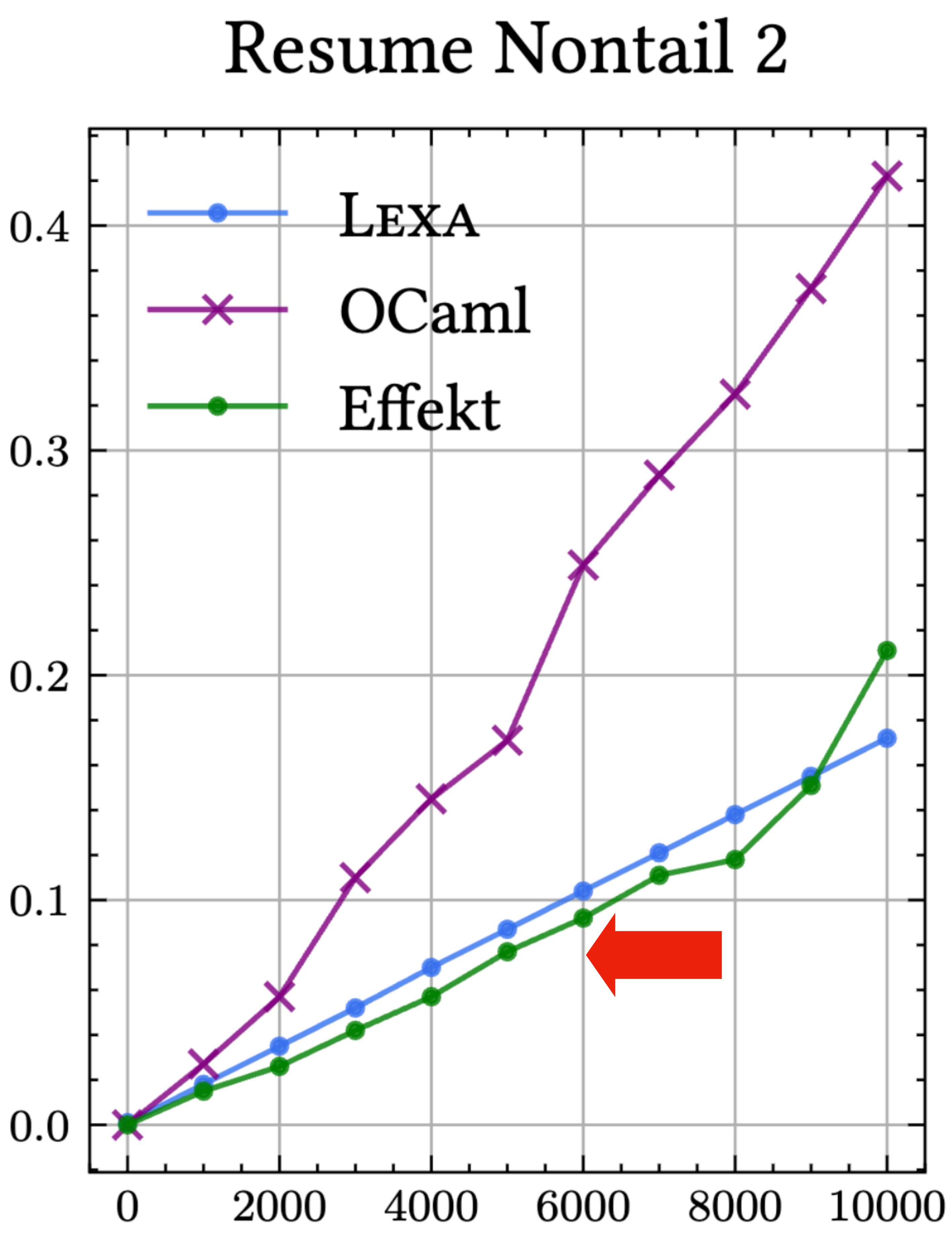


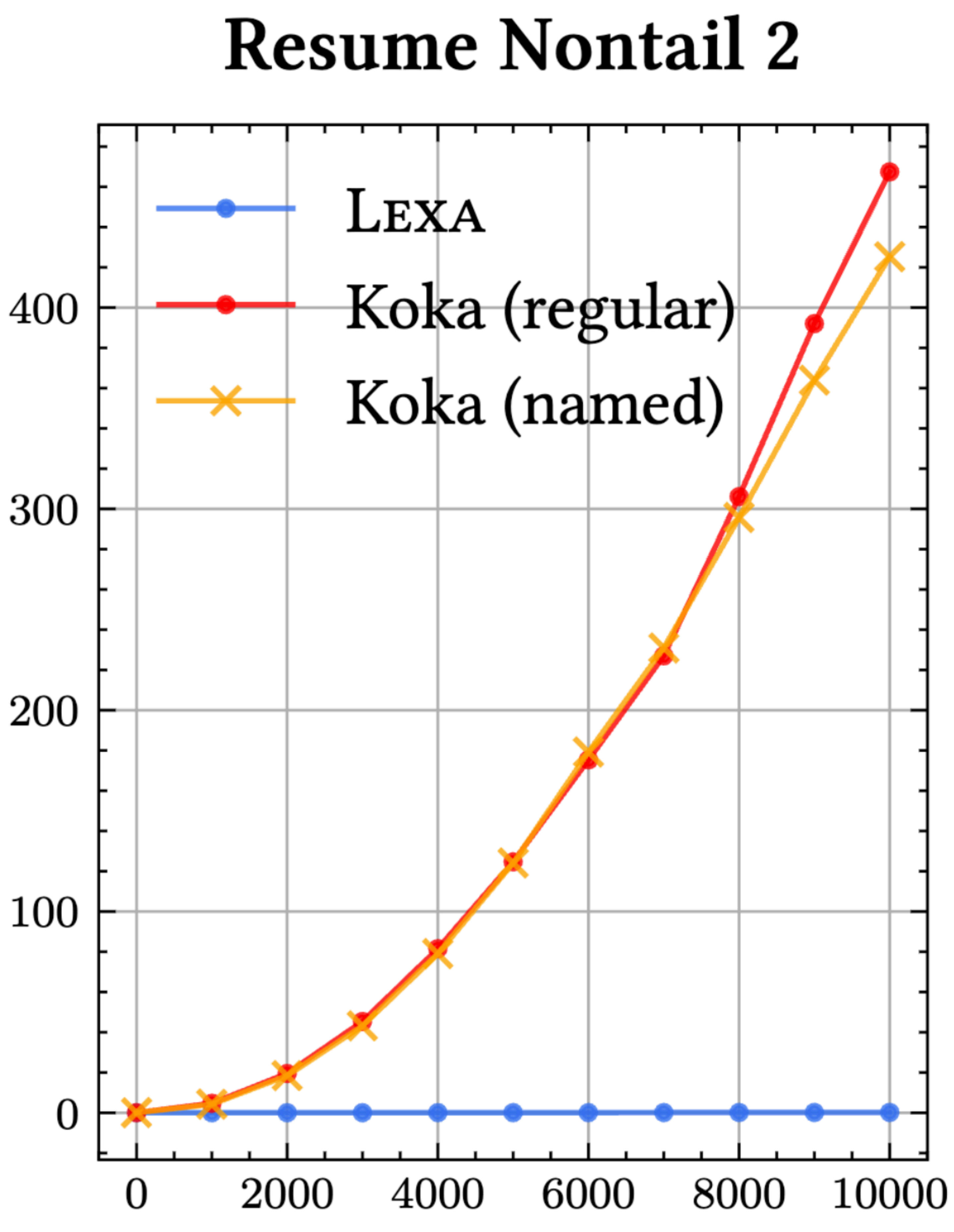


Evaluation: Resume Nontail 2

This benchmark checks how efficient a language captures continuations.

Lexa uses segmented stacks, so no extra effort is required to capture a continuation.

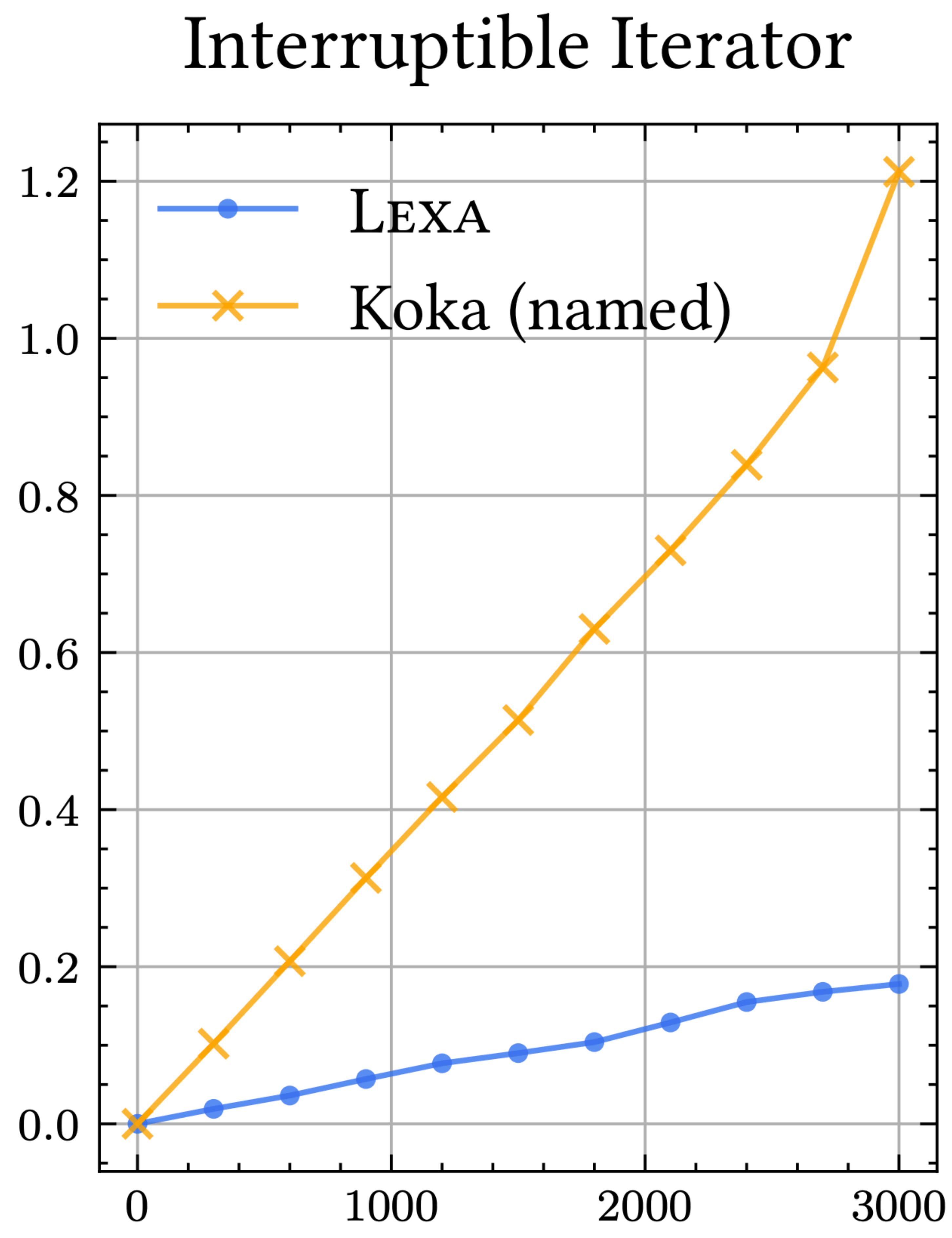


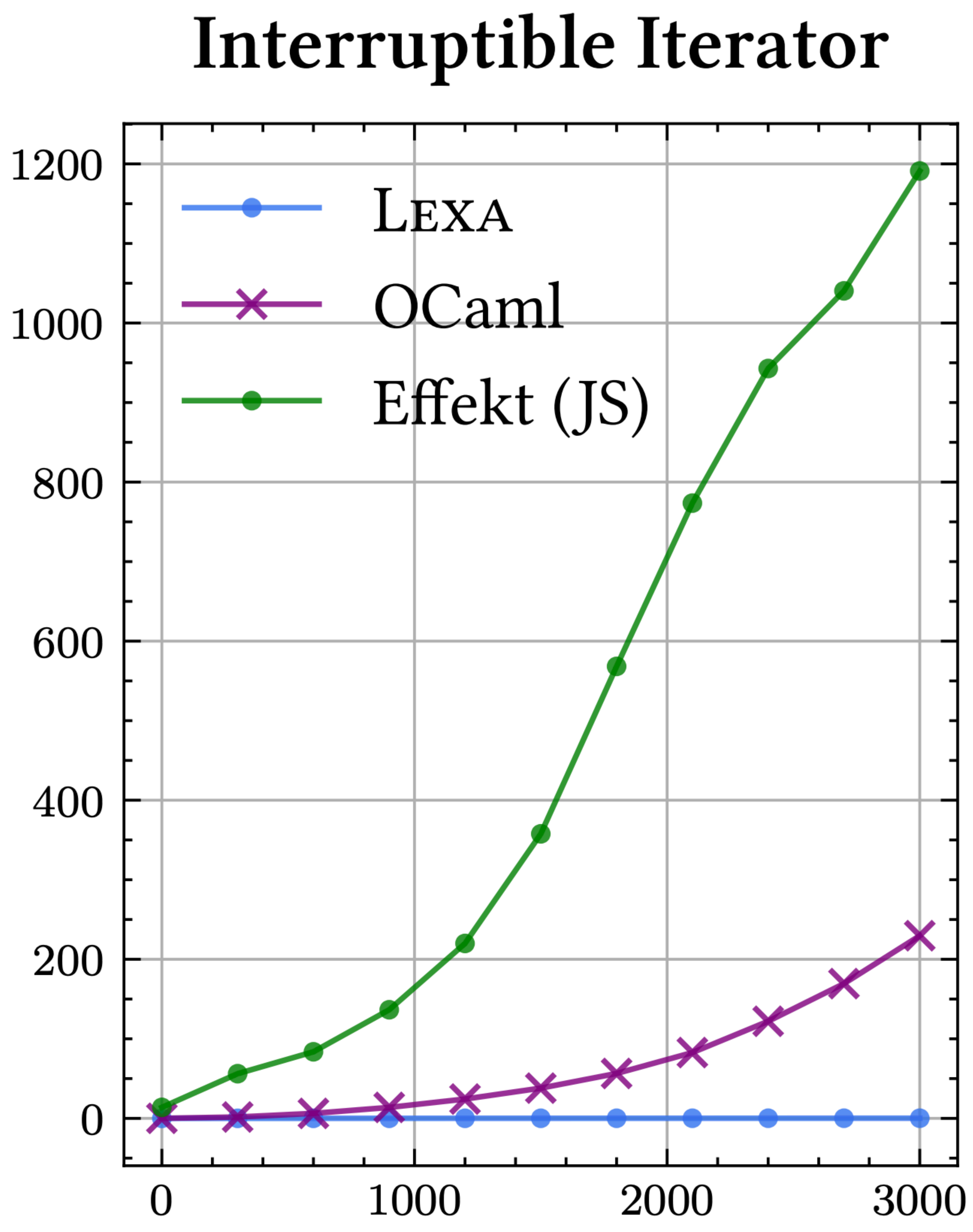


Evaluation: Interruptible Iterator

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. Handling bidirectional control flow. OOPSLA'20
Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. POPL'06

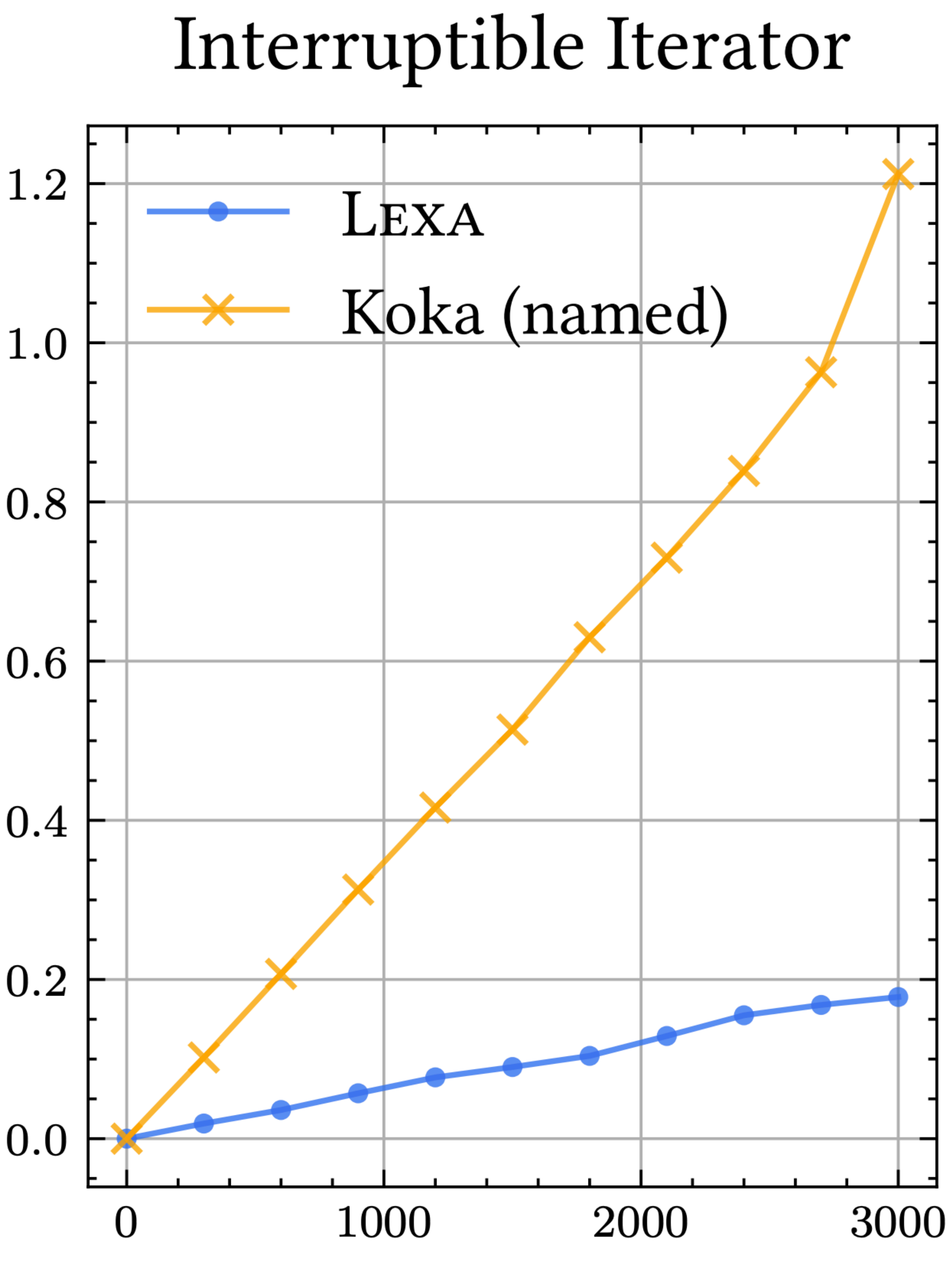
This benchmark installs deeply nested effect handlers.

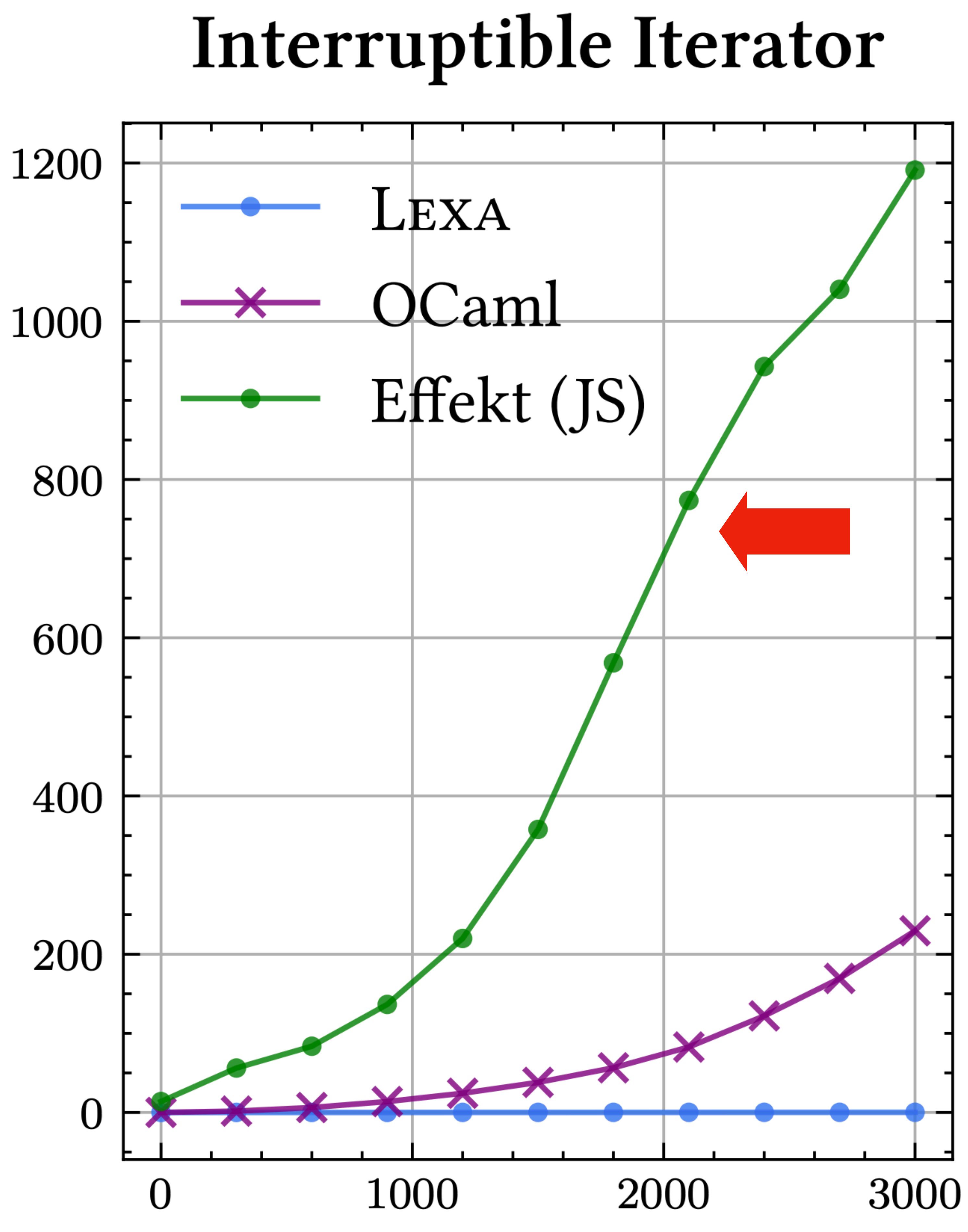




Evaluation: Interruptible Iterator

This benchmark installs deeply nested effect handlers.

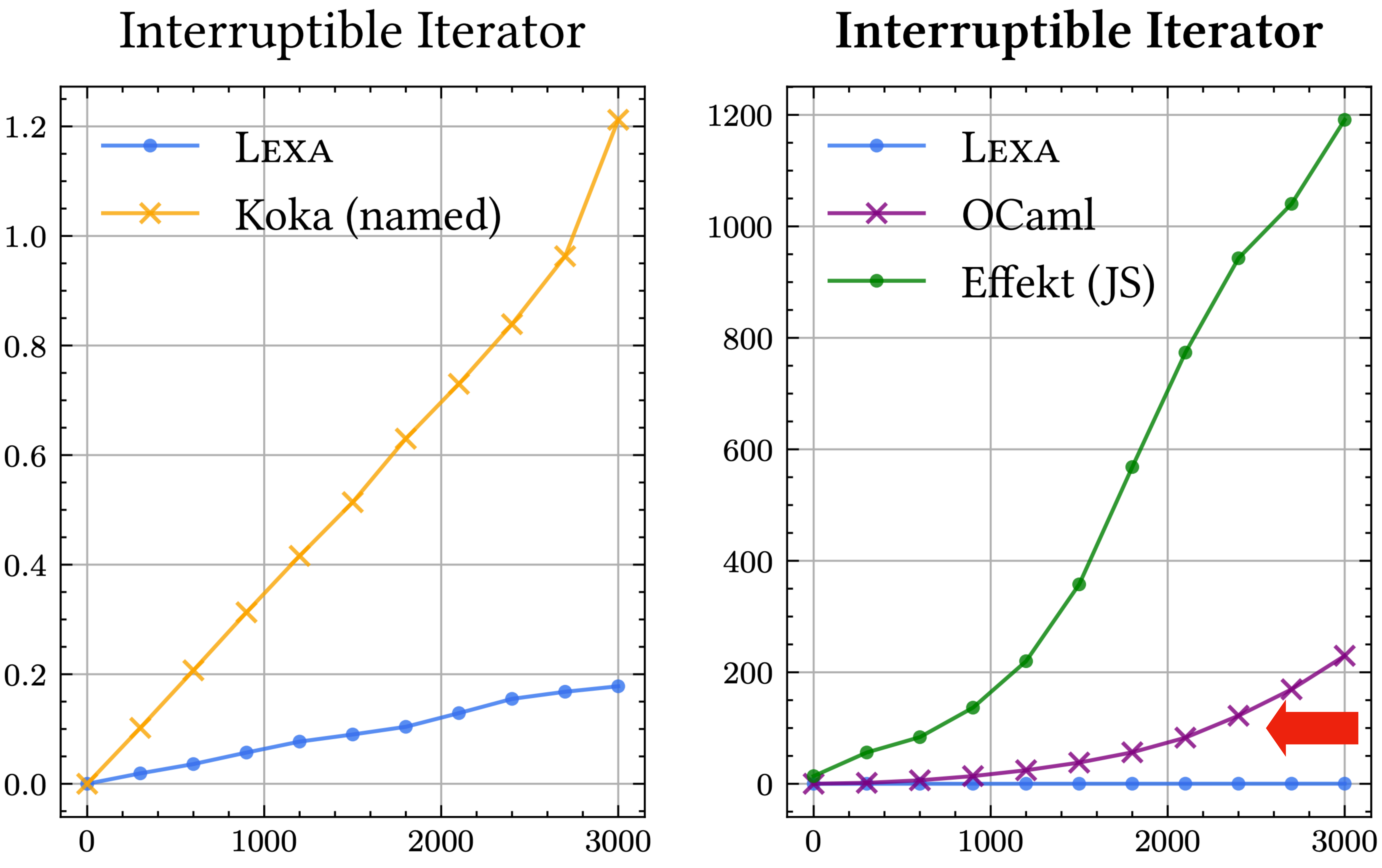




Effekt has a polynomial scaling trend as it performs a linear search on every raise.

Evaluation: Interruptible Iterator

This benchmark installs deeply nested effect handlers.

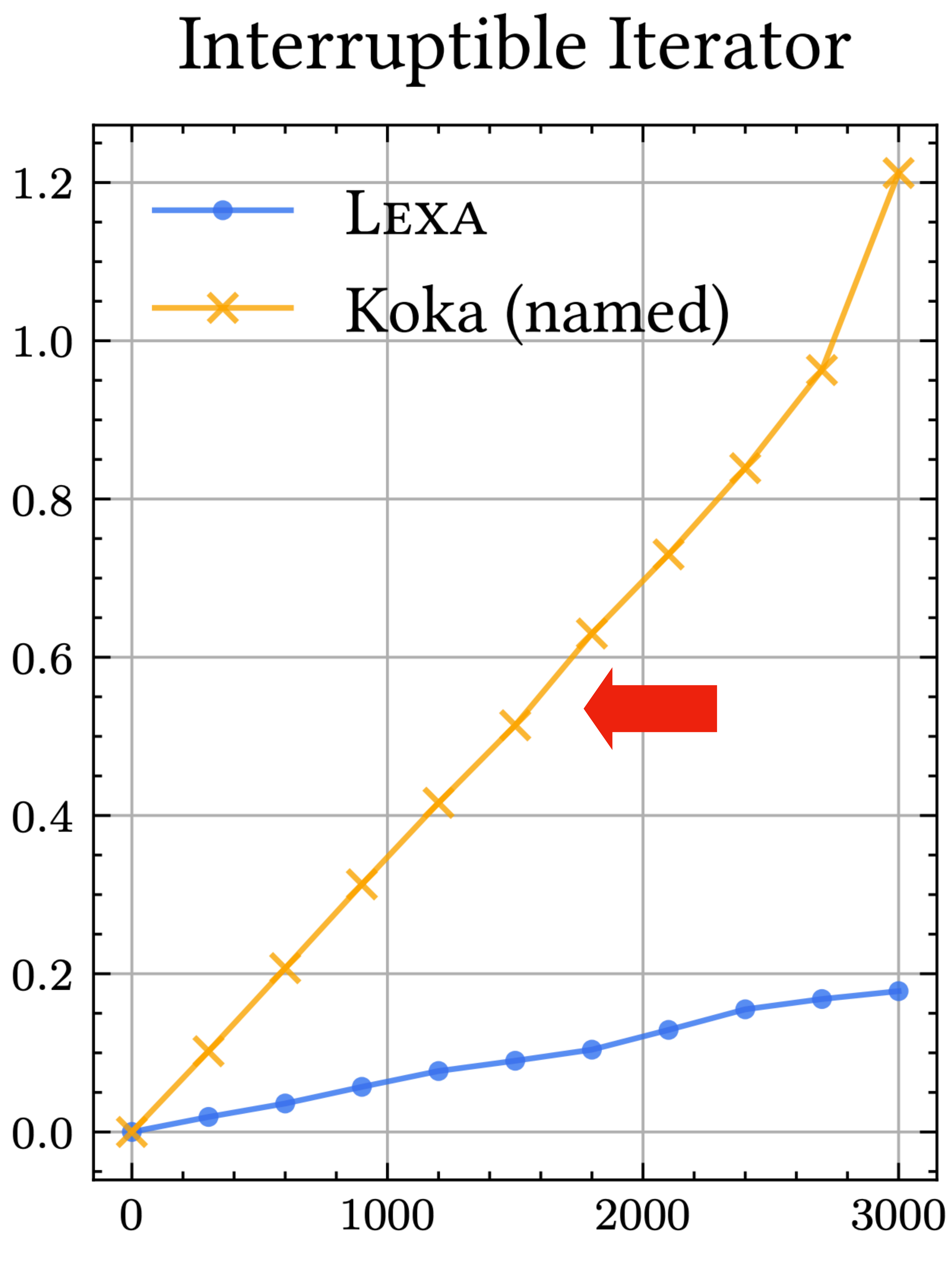


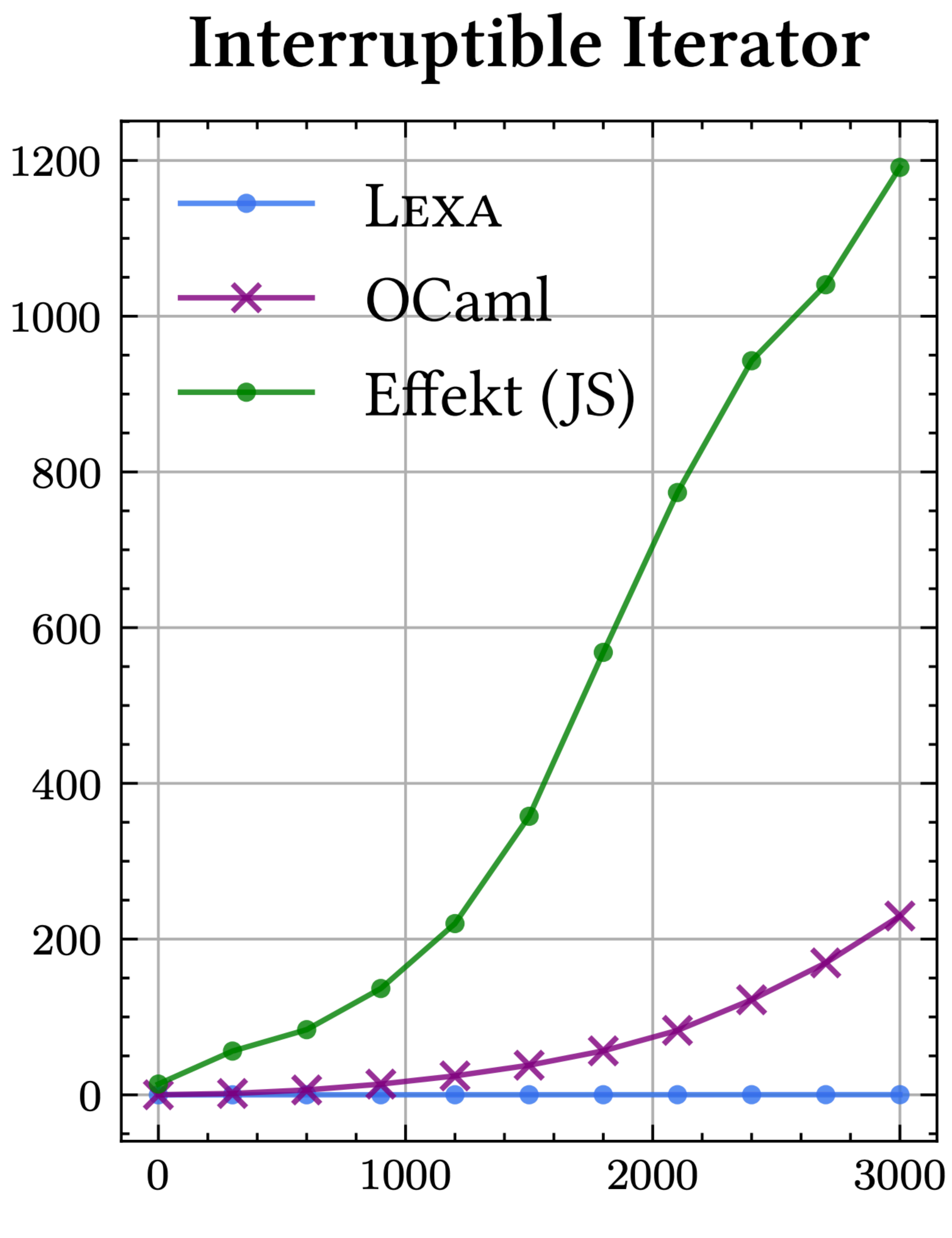
OCaml has a polynomial scaling trend as it performs a linear search on every raise.

Evaluation: Interruptible Iterator

This benchmark installs deeply nested effect handlers.

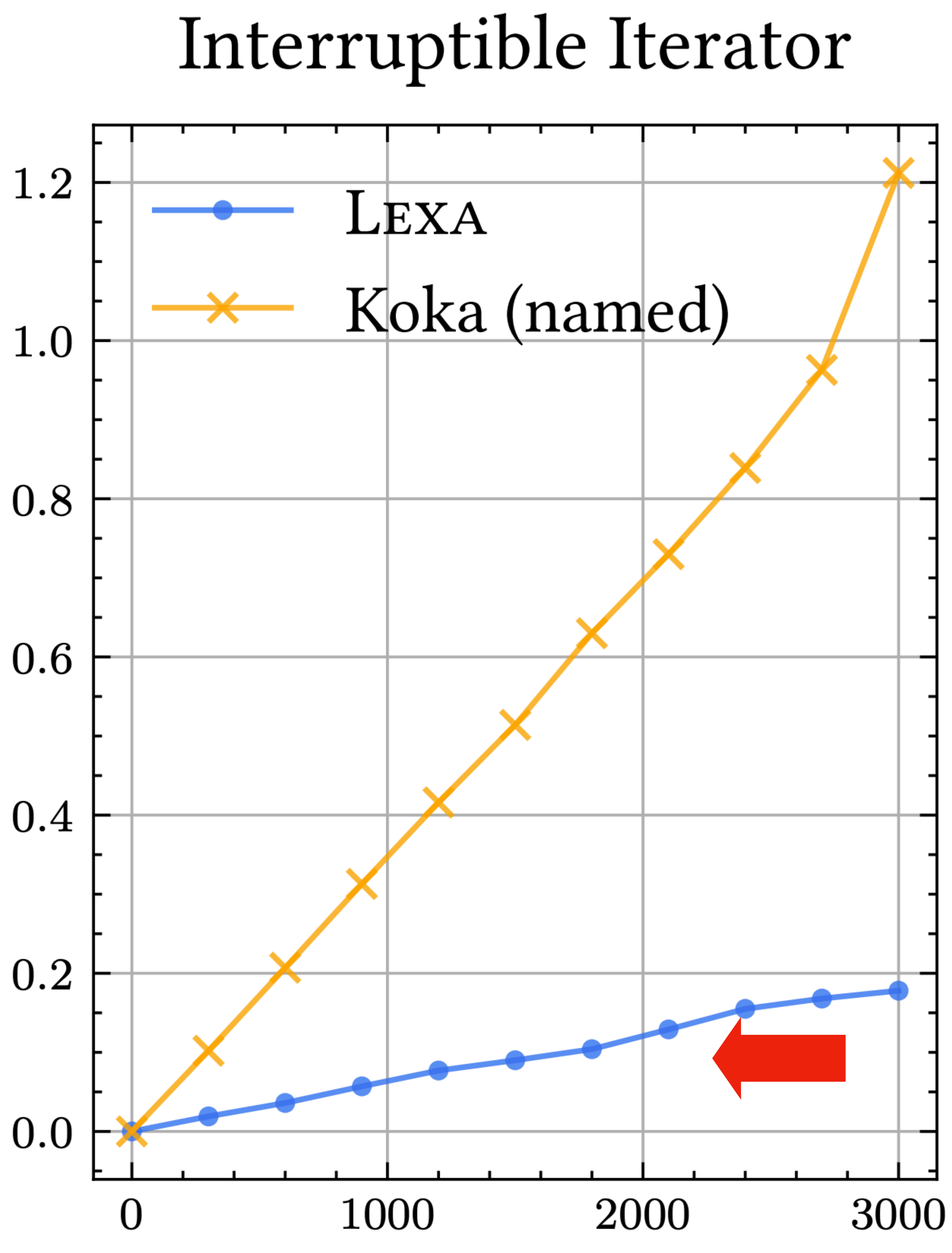
Koka scales linearly as it avoids the handler search because of tail-resumptive optimization.

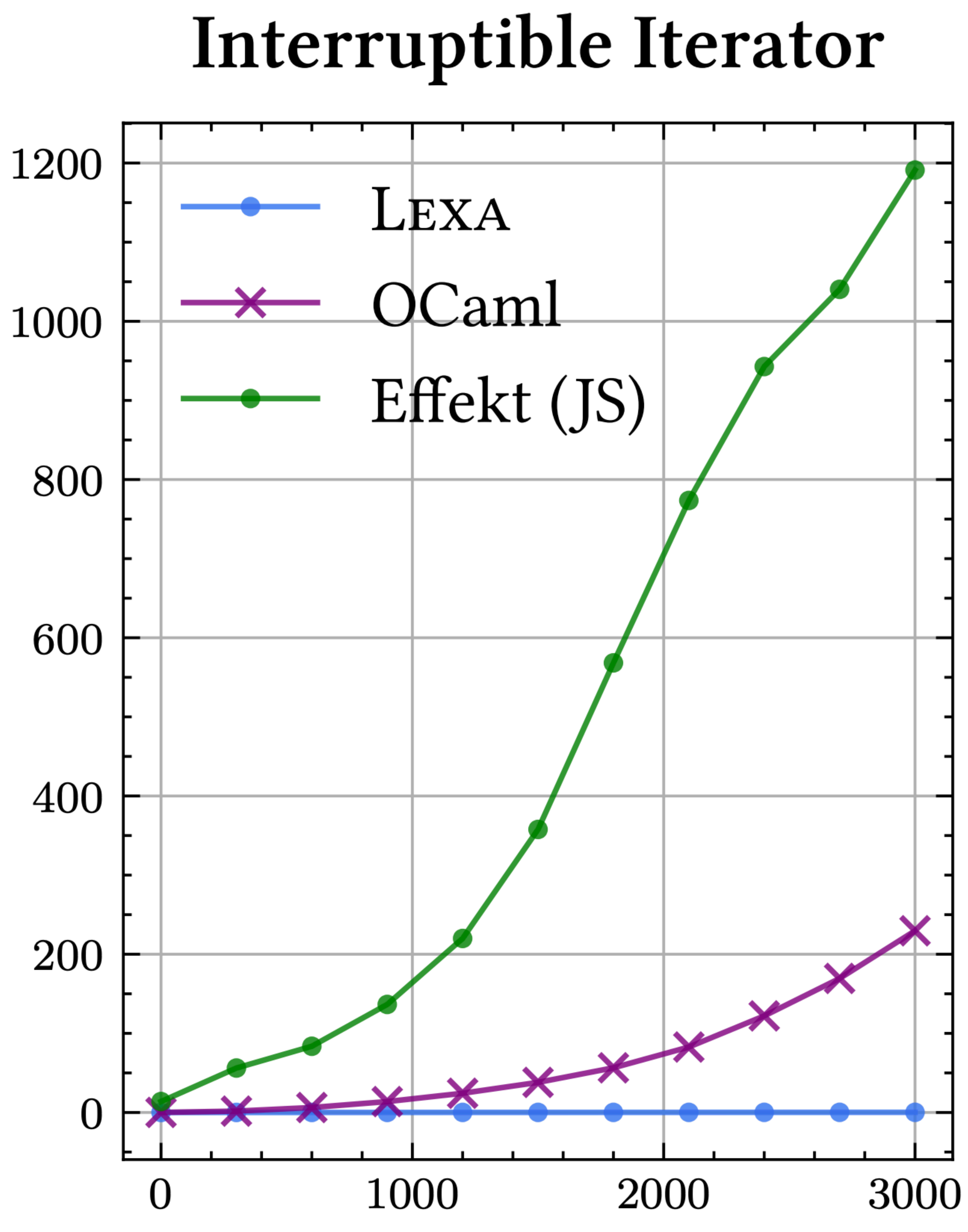




Evaluation: Interruptible Iterator

This benchmark installs deeply nested effect handlers.





Lexa achieves linear scaling because it raises effect in constant time.

high-level, modular algebraic effects in Lexa

low-level, swift stack switching in assembly

Time to find the handler: O(1)

Time to capture the continuation: O(1)

⟦S⟧ = ⟦T⟧

this paper →

```
handle E with H
raise …
resume …
```

```
ENTER
RAISE
RESUME
```

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