Zero-Cost Lexical Effect Handlers

Cong Ma, Zhaoyi Ge, Max Jung, Yizhou Zhang
University of Waterloo

Background

Effect handler

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

Dynamically scoped handler has a modularity problem, and lexically scoped handler restores the modularity.

Effect handler

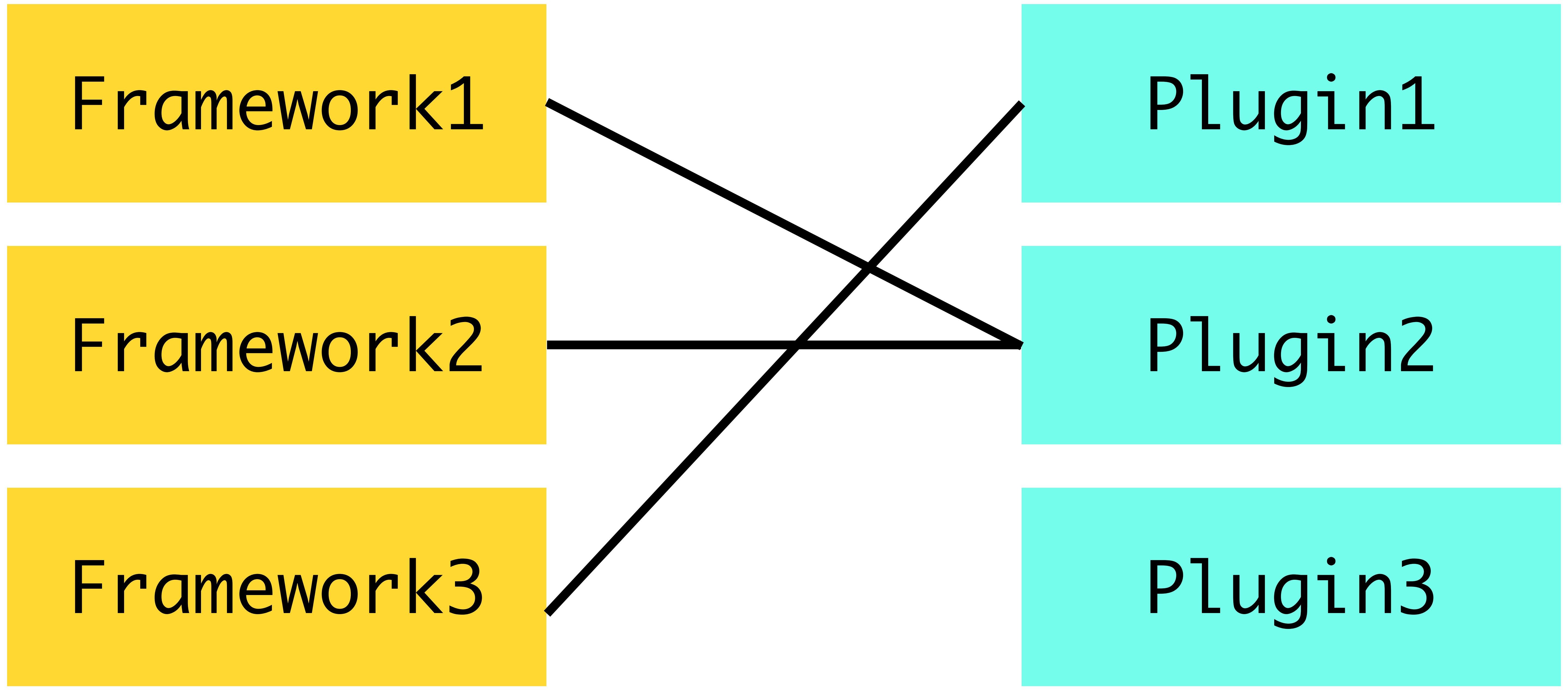
Effect handlers subsume an array of control flow features: async/await, coroutine, generator…

Dynamically scoped handler has a modularity problem, and lexically scoped handler restores the modularity.

Background

Accidental Handling

Modern softwares are built with modular components, which are designed to be interchangeable.



Accidental Handling

Modern softwares are built with modular components, which are designed to be interchangeable.

However, dynamically scoped handler breaks the modularity.

Accidental Handling

However, dynamically scoped handler breaks the modularity.

Accidental Handling

However, dynamically scoped handler breaks the modularity.

Assuming you are an application developer, and you want to monitor the usage of the plugin by the framework.

```
#application
let plugin = λx. … in
let framework = λf. … in
framework(plugin)
```

Background

Accidental Handling

However, dynamically scoped handler breaks the modularity.

Assuming you are an application developer, and you want to monitor the usage of the plugin by the framework.

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Background

Accidental Handling

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

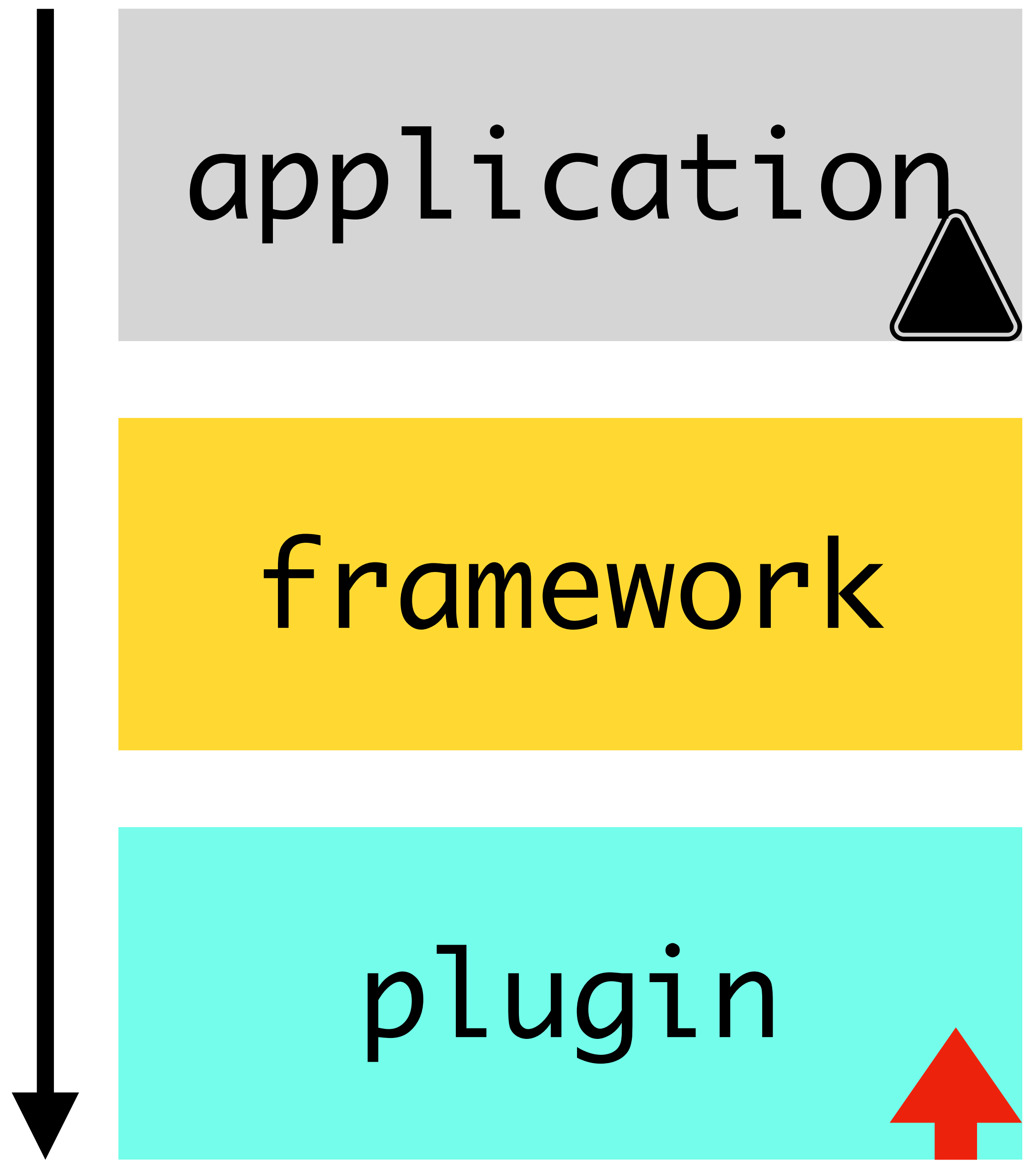
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Black triangle denotes a handler

Red arrow denotes a raised effect

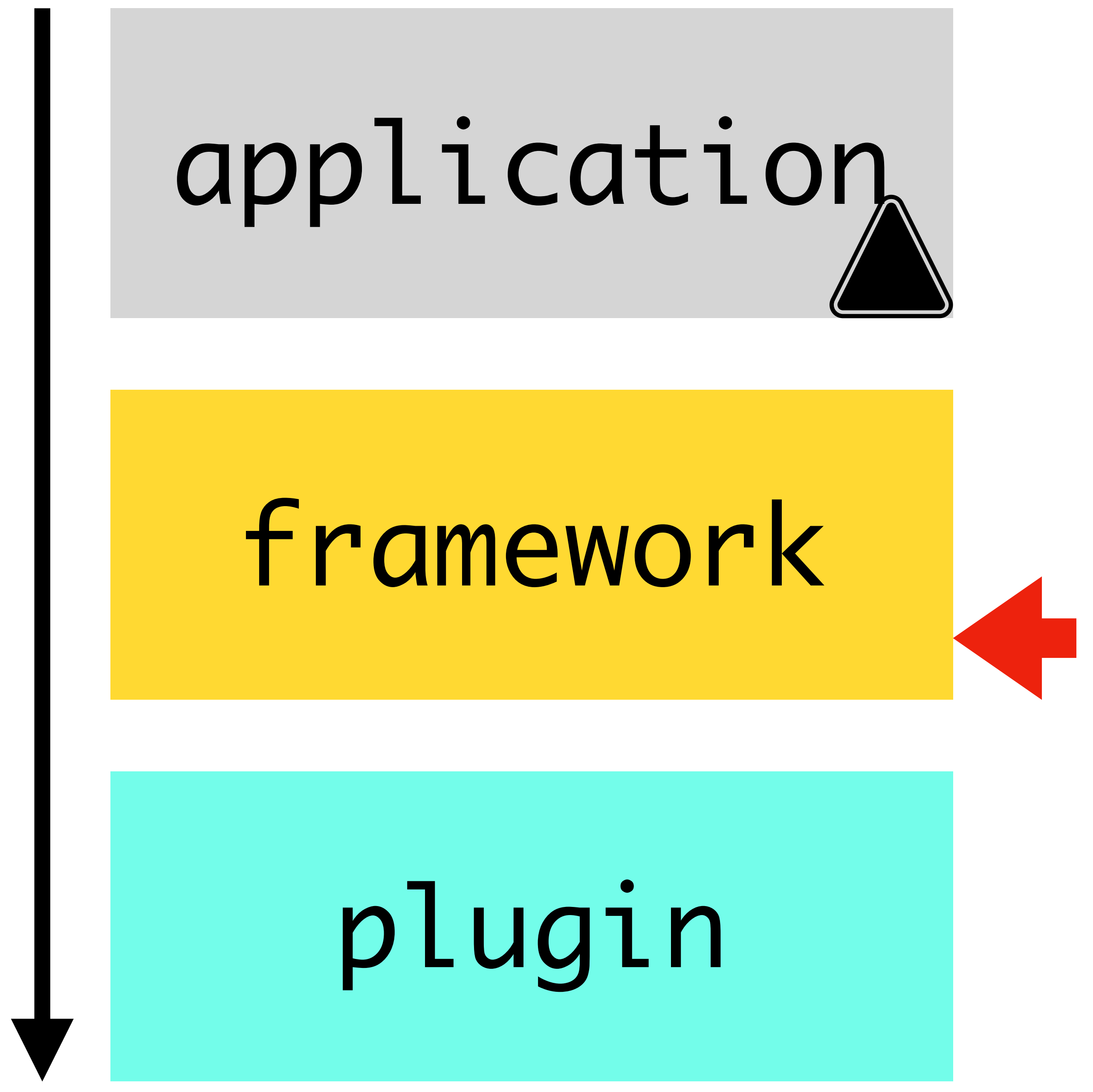
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Black triangle denotes a handler

Red arrow denotes a raised effect

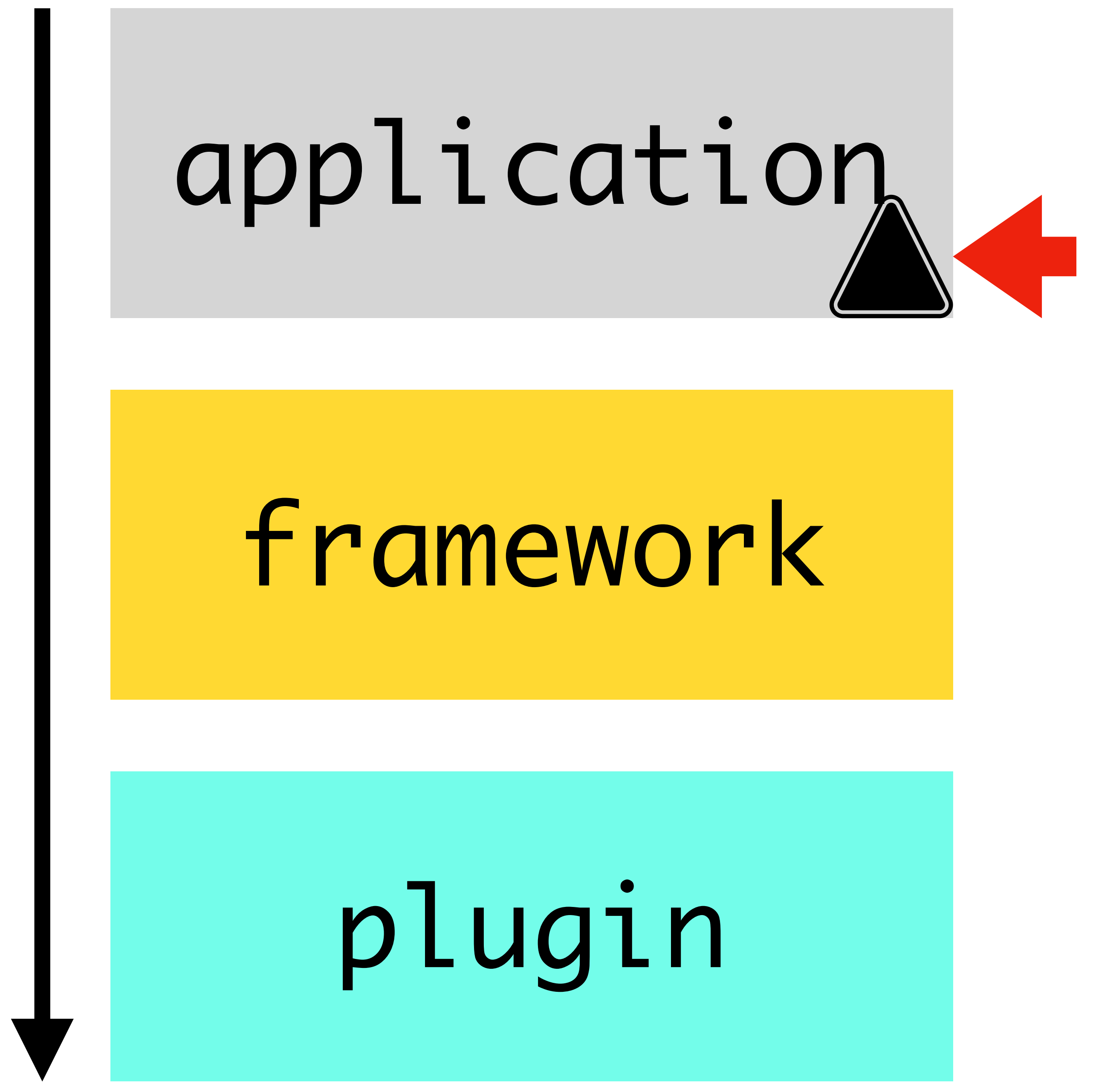
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Black triangle denotes a handler

Red arrow denotes a raised effect

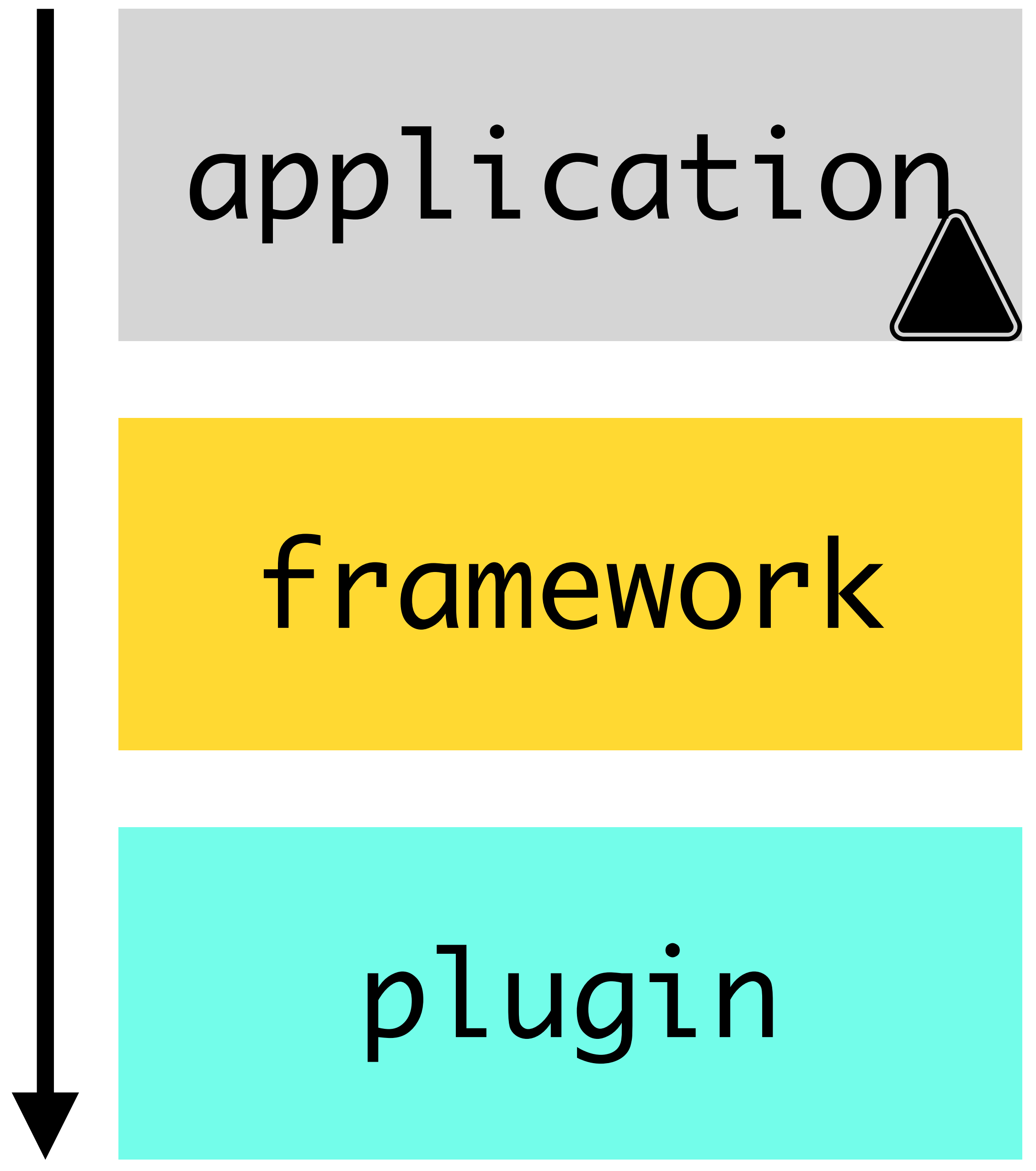
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

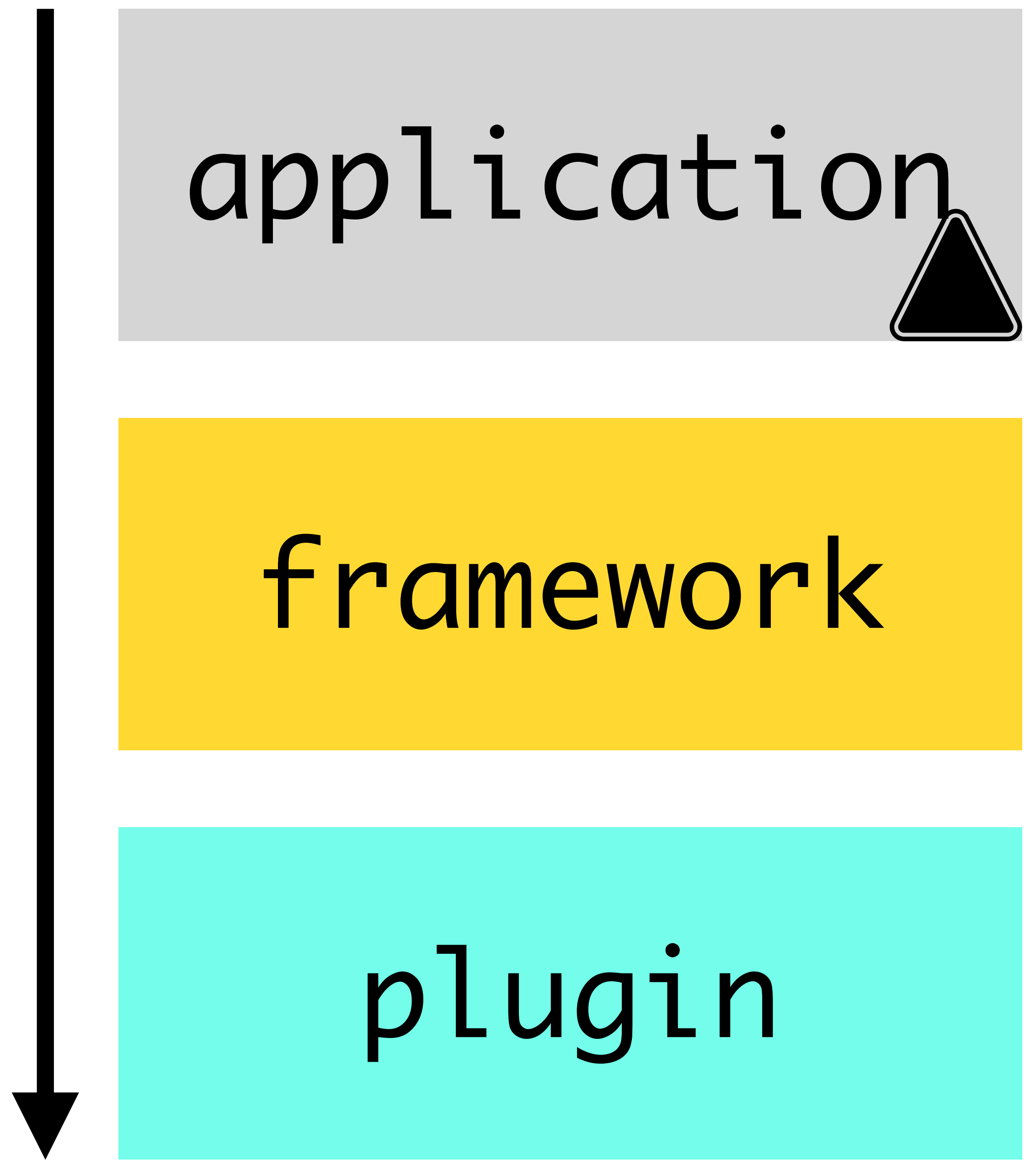
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. handle … with Logging: … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

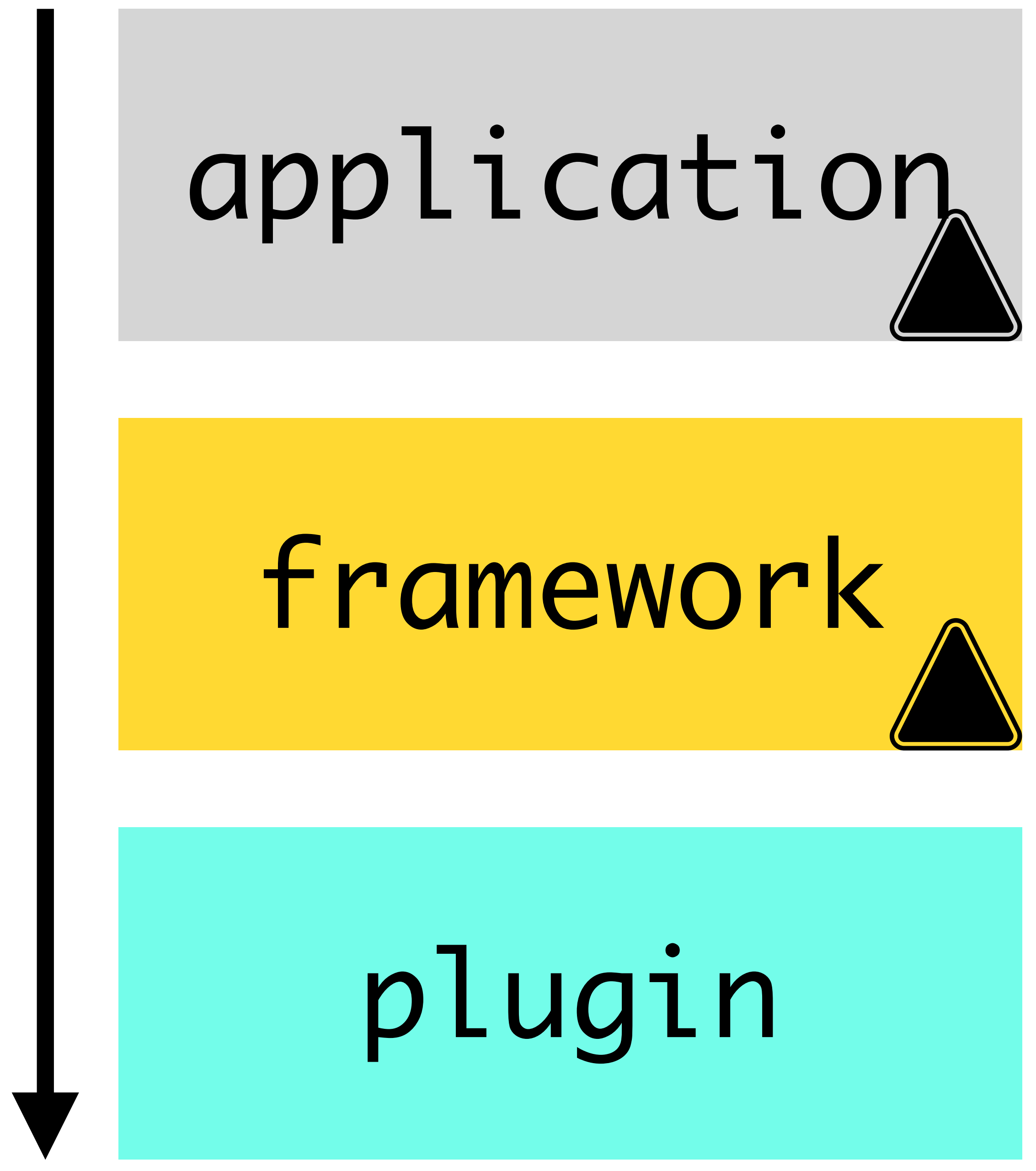
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework = λf. handle … with Logging: … in
handle
    framework(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

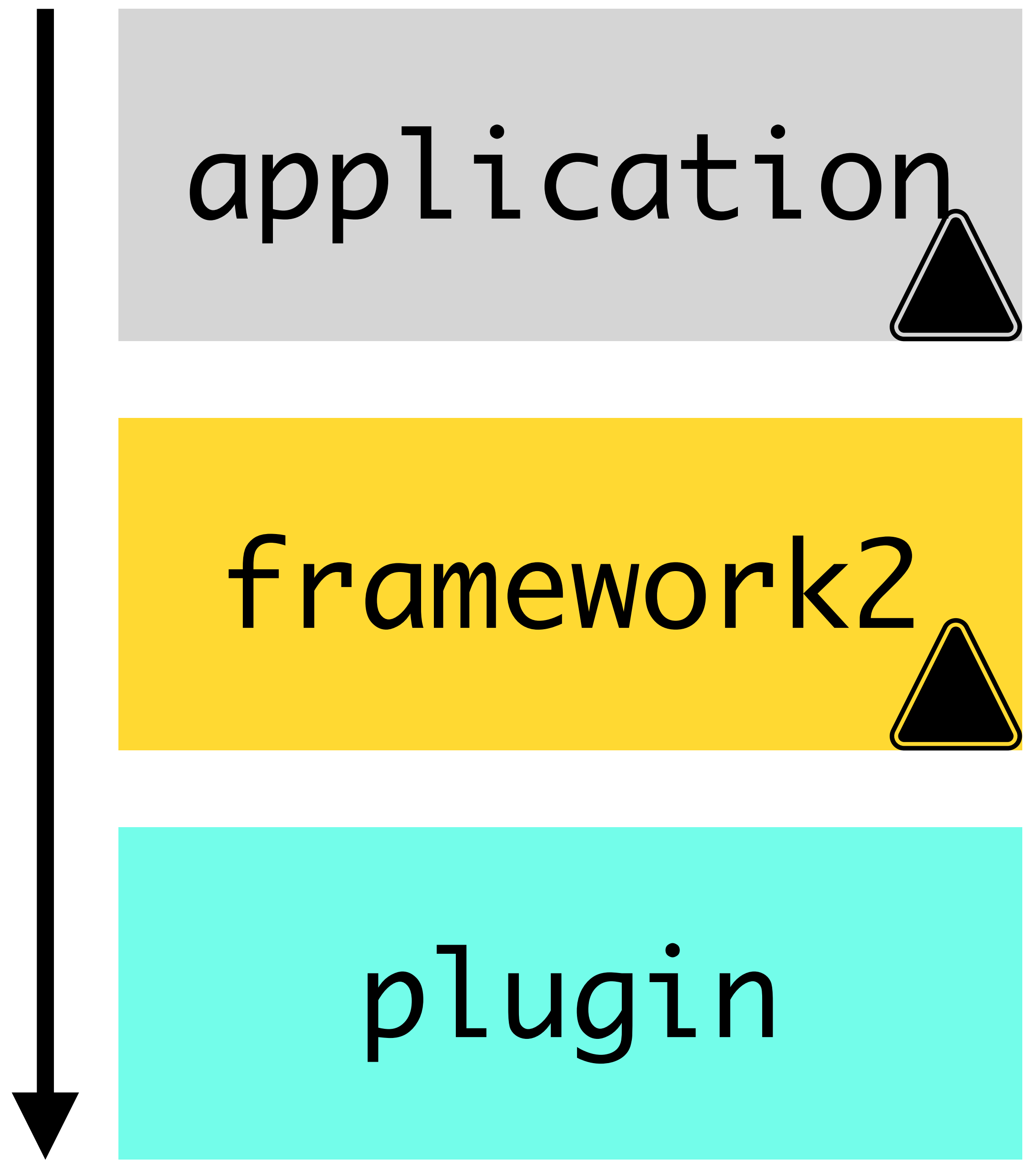
Background

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework2 = λf. handle … with Logging: … in
handle
    framework2(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



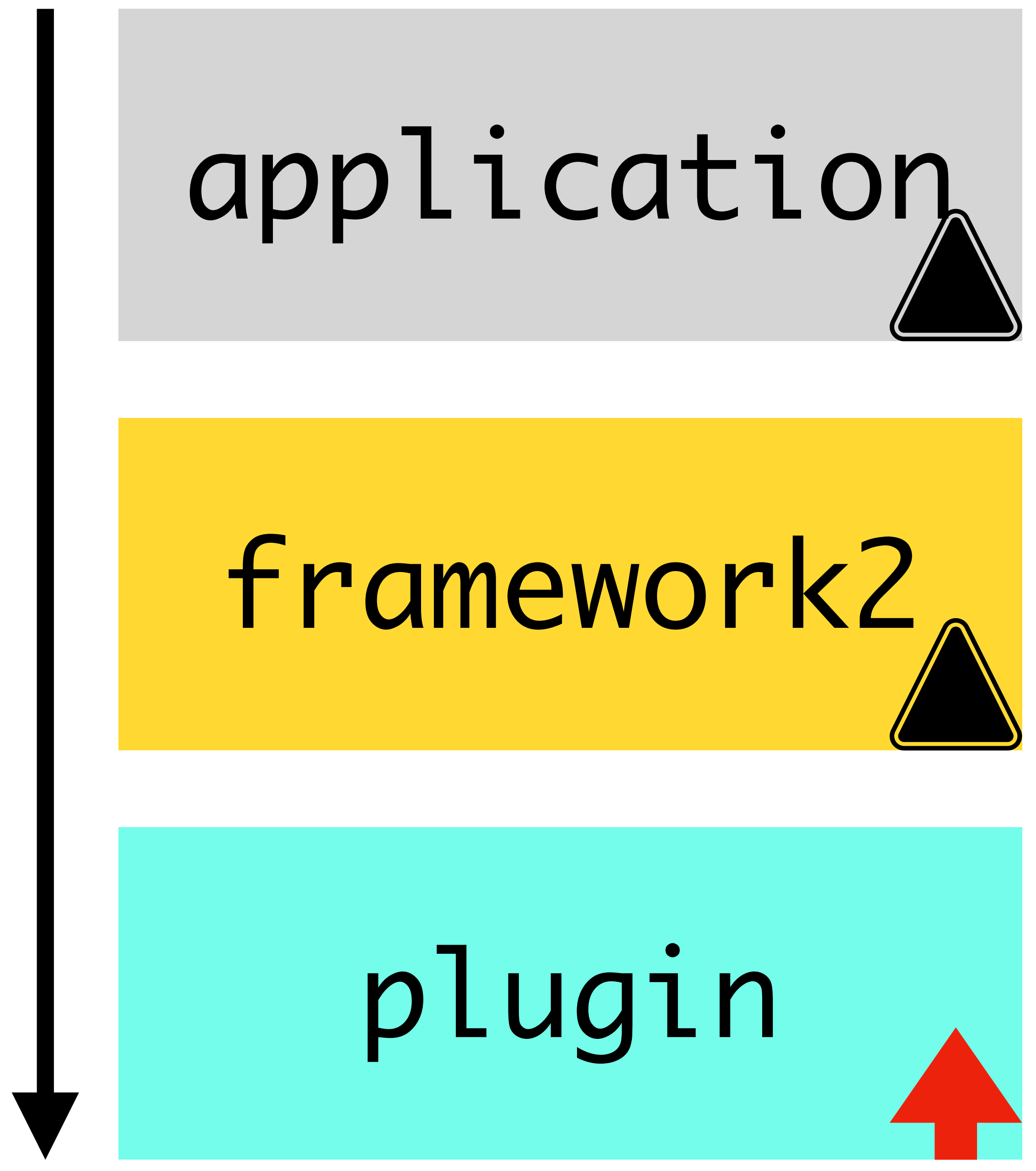
Now, you choose a different framework that install a Logging handler for its own purpose.

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework2 = λf. handle … with Logging: … in
handle
    framework2(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

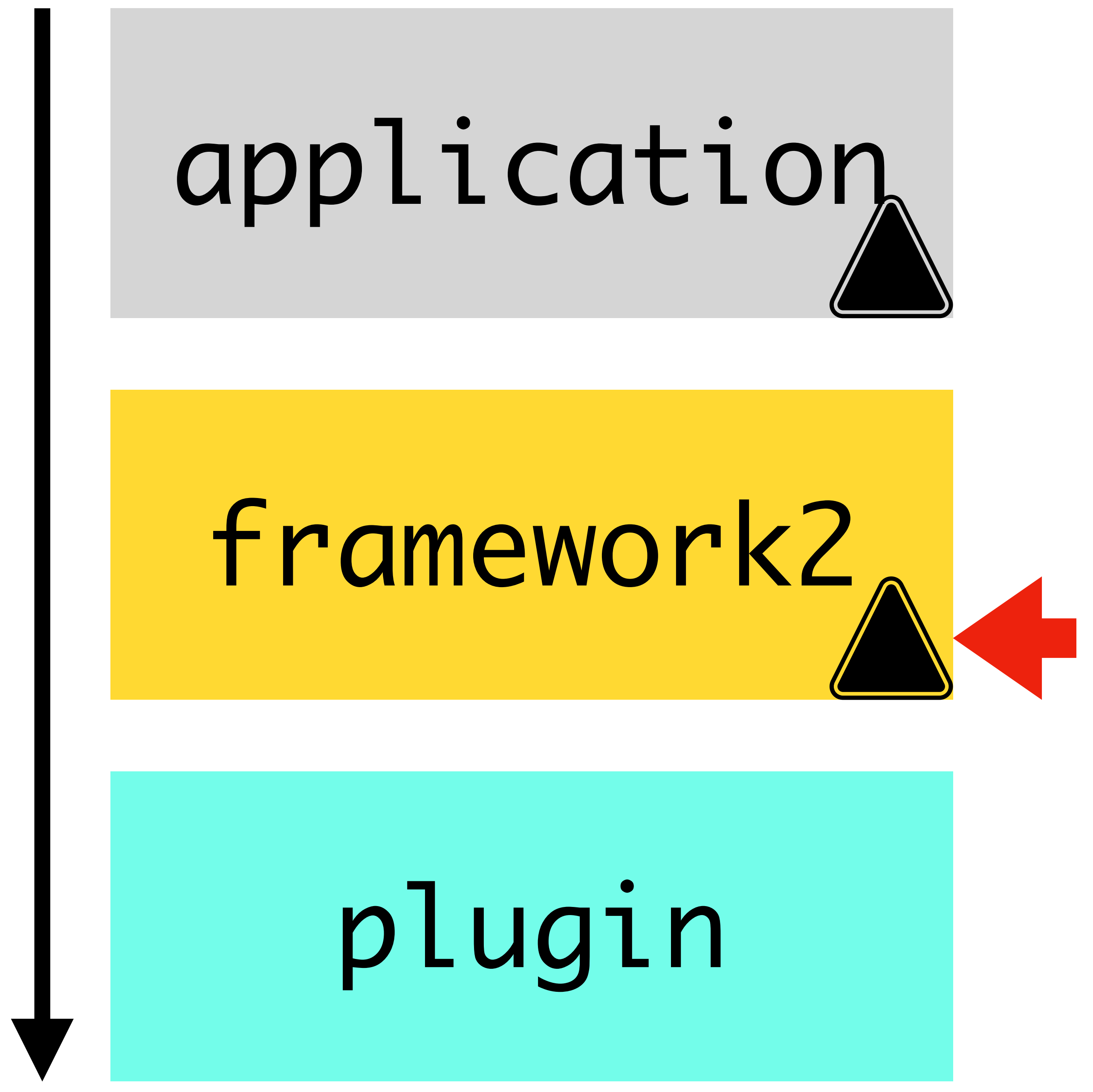
When an effect is raised, it will be **accidentally handled** by the handler in framework2.

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework2 = λf. handle … with Logging: … in
handle
    framework2(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

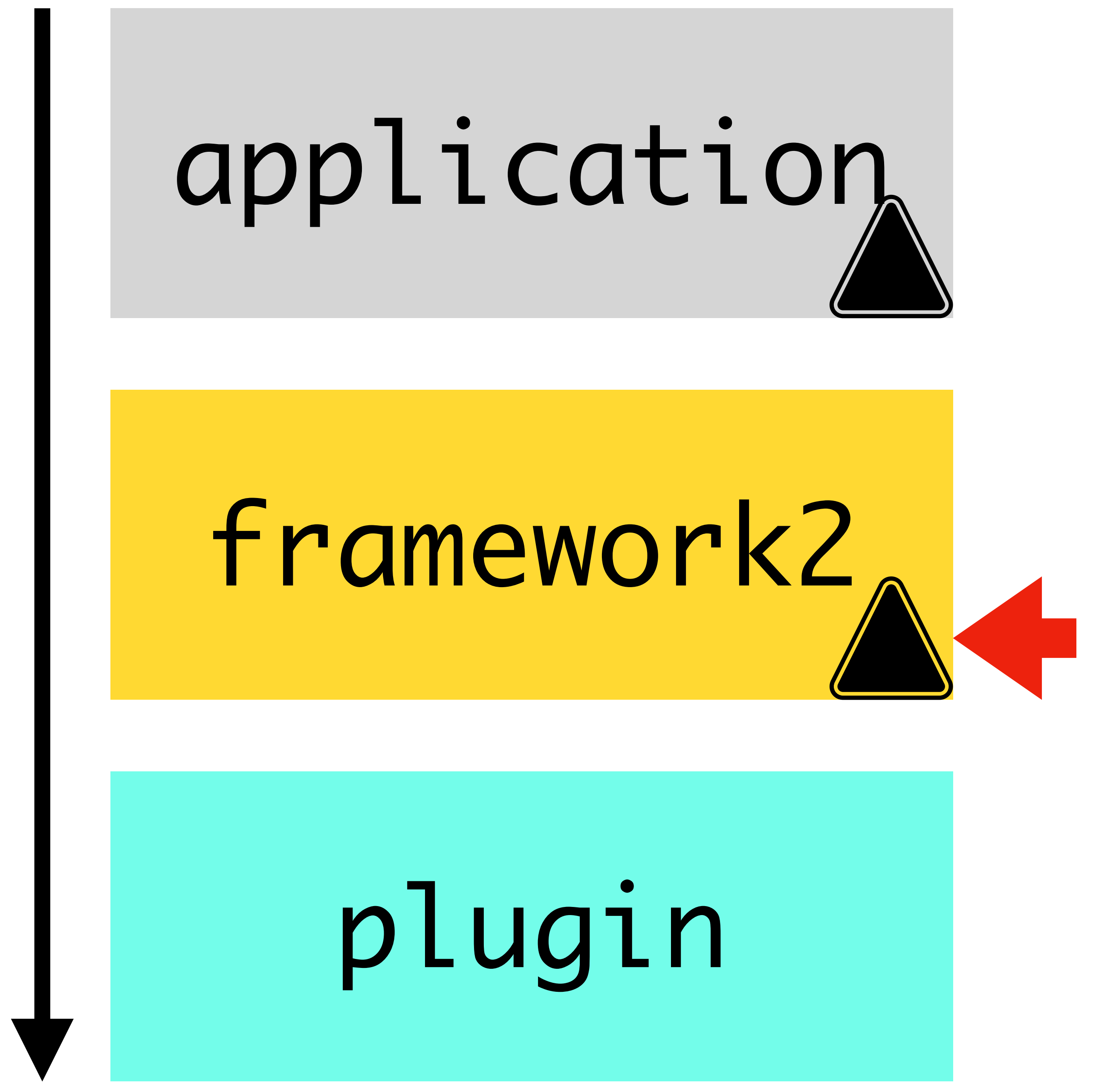
When an effect is raised, it will be **accidentally handled** by the handler in framework2.

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework2 = λf. handle … with Logging: … in
handle
    framework2(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that install a Logging handler for its own purpose.

When an effect is raised, it will be **accidentally handled** by the handler in framework2.

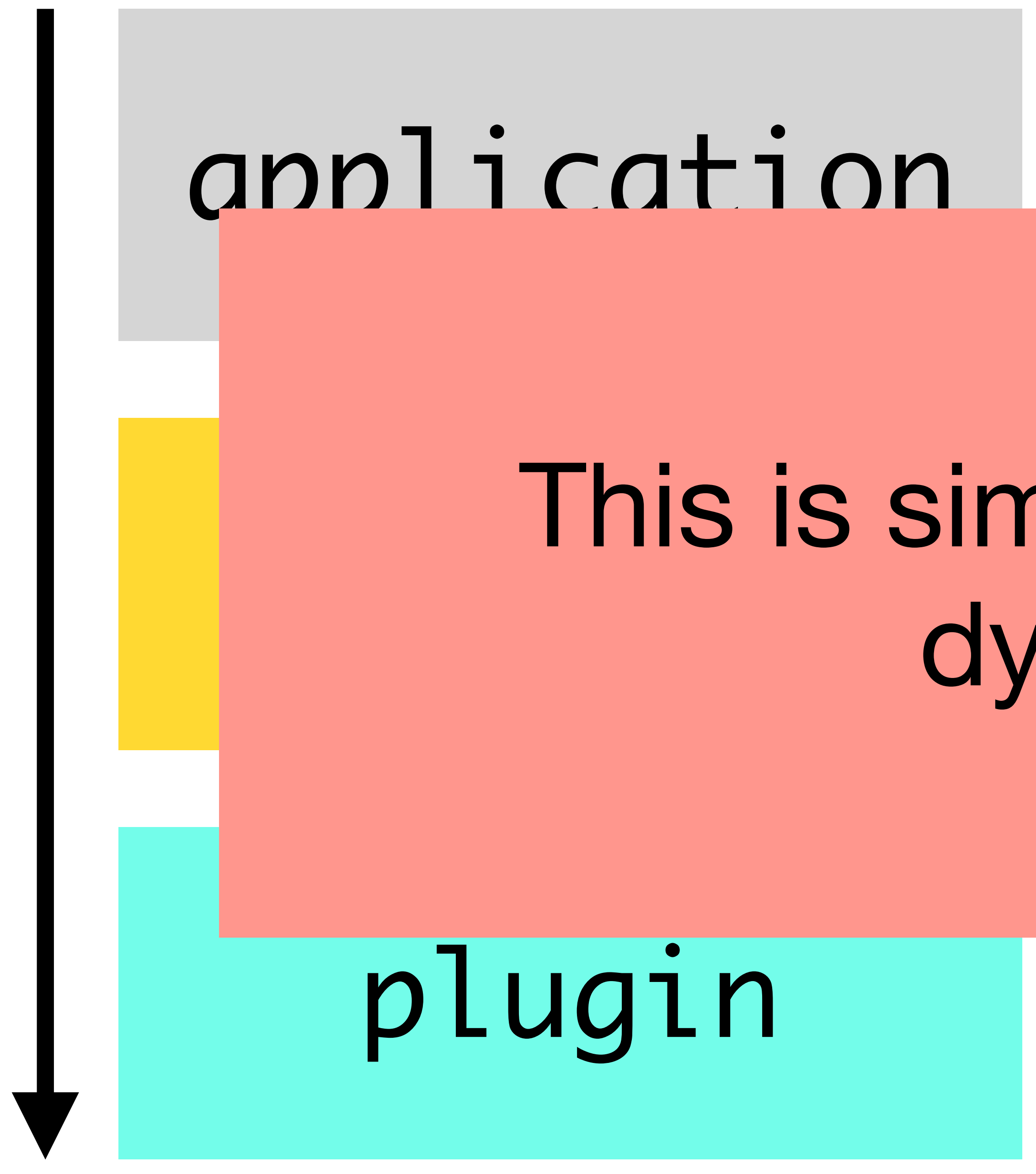
This behavior breaks the abstraction and makes it difficult to reason about the programs.

Accidental Handling

Code

```
#application
let plugin = λx. … in
let framework2 = λf. handle … with Logging: … in
handle
    framework2(λx.plugin x; raise Logging(…))
with Logging:
    λx,k. print(x); resume k
```

Stack



Now, you choose a different framework that

purpose.

er in

n and

makes it difficult to reason about the programs.

This is similar to the abstraction problem with dynamically scoped variables.

Background

Effect handler

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

Dynamically scoped handler has a modularity problem, and lexically scoped handler restores the modularity.

Background

Lexical effect handler 101

An example illustrating its operational semantics.

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

Background

Lexical effect handler 101

An example illustrating its operational semantics.

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

freshly generated label

Background

Lexical effect handler 101

An example illustrating its operational semantics.

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

Background

Lexical effect handler 101

An example illustrating its operational semantics.

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

Background

Lexical effect handler 101

Programs built with lexical effect handlers enjoy modularity.

Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. Proc. of the ACM on Programming Languages (PACMPL), 3(POPL), January 2019

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. Proc. of the ACM on Programming Languages (PACMPL), 4(POPL), January 2020

Background

Lexical effect handler

However, existing implementations impose a runtime cost

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

All effectful functions need to explicitly receive handler labels as arguments.

This imposes a runtime cost even for rarely raised effects.

Background

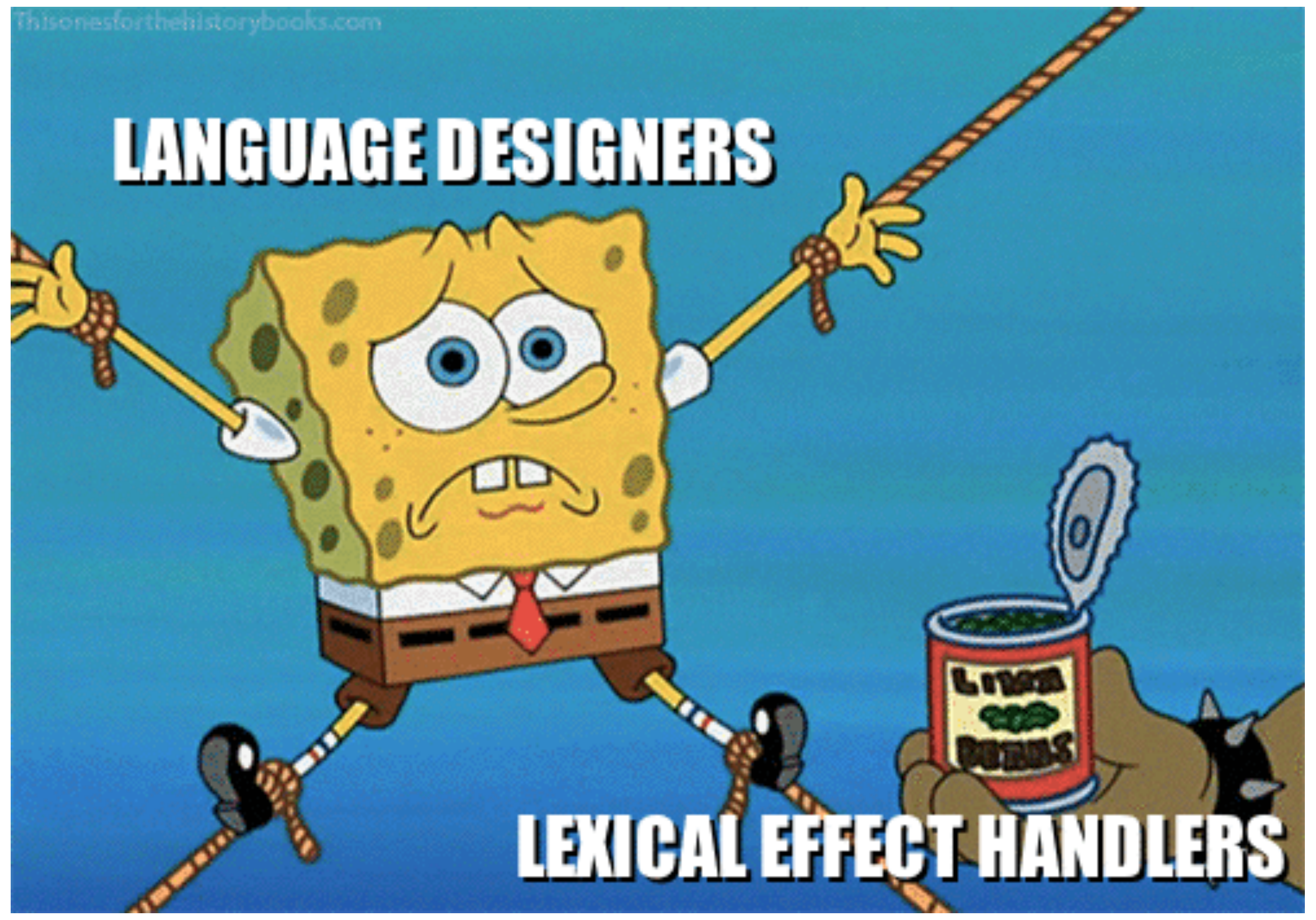
Lexical effect handler

However, existing implementations impose a runtime cost

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

All effectful functions need to explicitly receive handler labels as arguments.

This imposes a runtime cost even for rarely raised effects.



Zero-Cost Lexical Effect Handlers

In this work, we present a type-directed compilation that eliminates the runtime cost for having handlers in the lexical context. This compilation design obeys zero-cost principle.

Zero-Cost Lexical Effect Handlers, Example 1

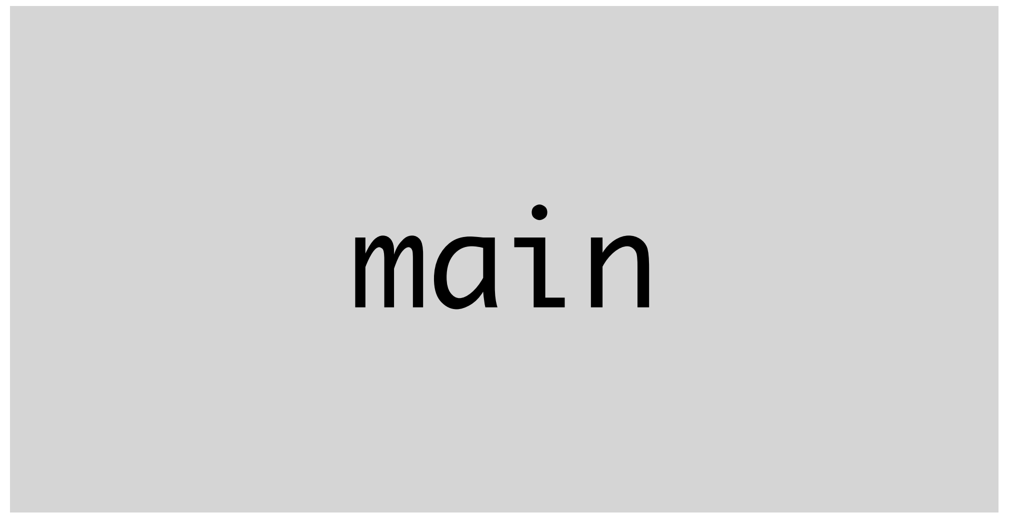
```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

We will first see an execution with the lexical effect handler semantics.

We will then figure out how to make the semantics zero-cost!

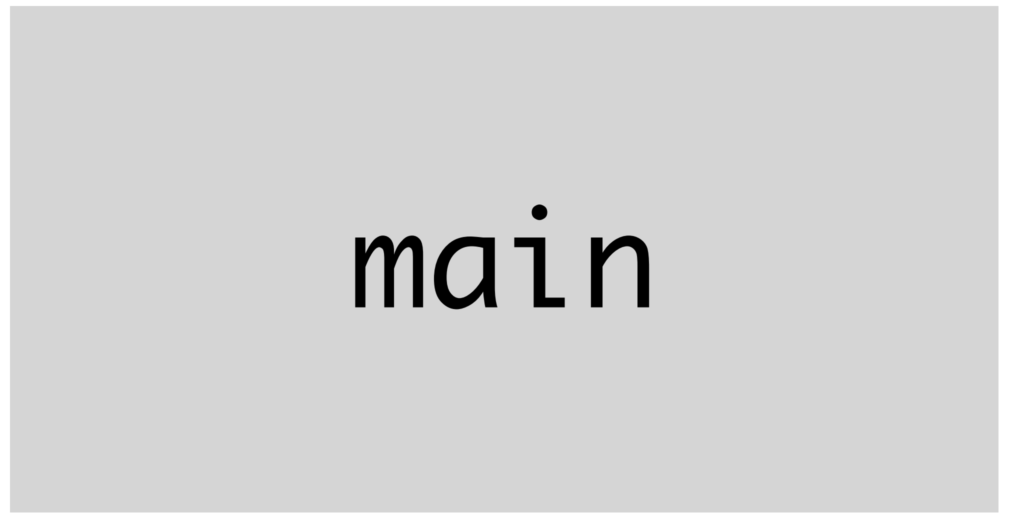
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



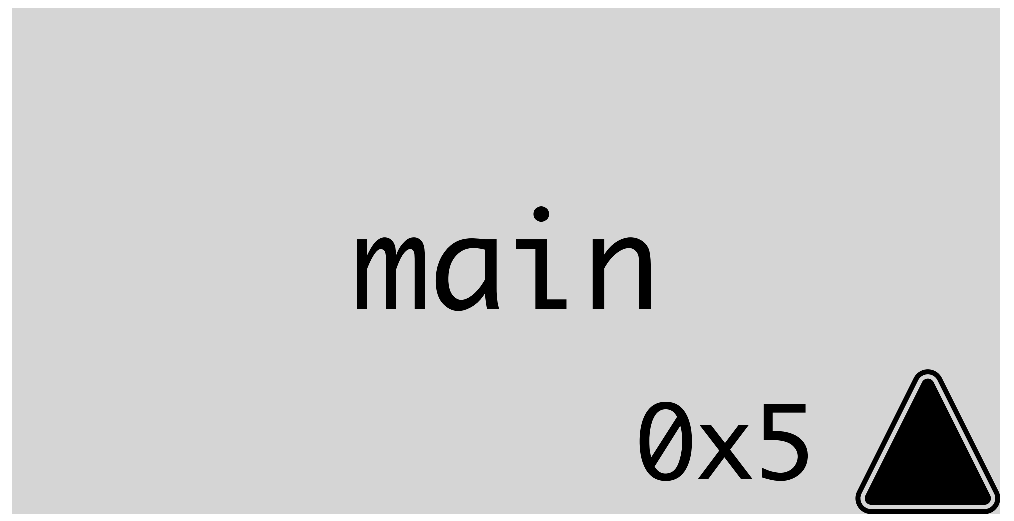
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
          handle
            f(x, h, log)
          with log = ...
in
handle
 g(42, log)
with log = ...
```



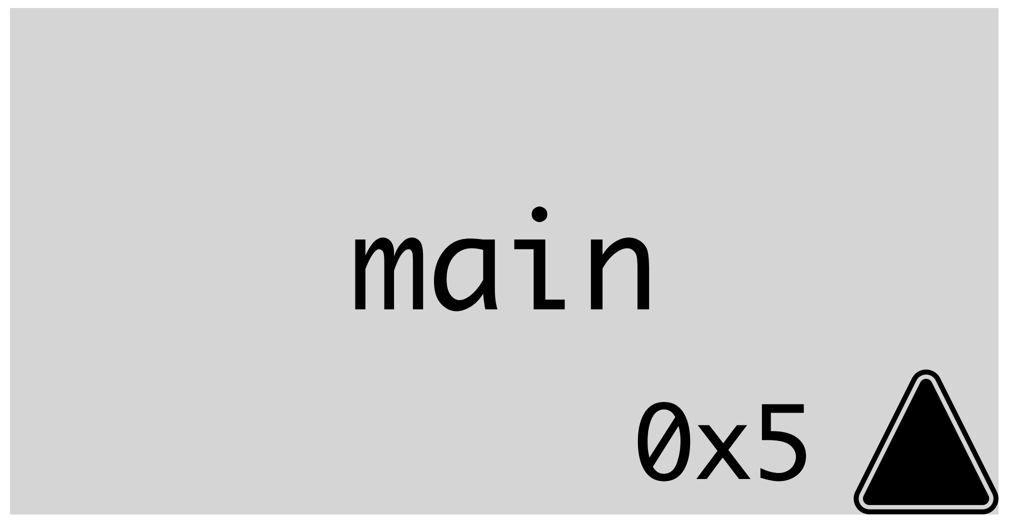
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



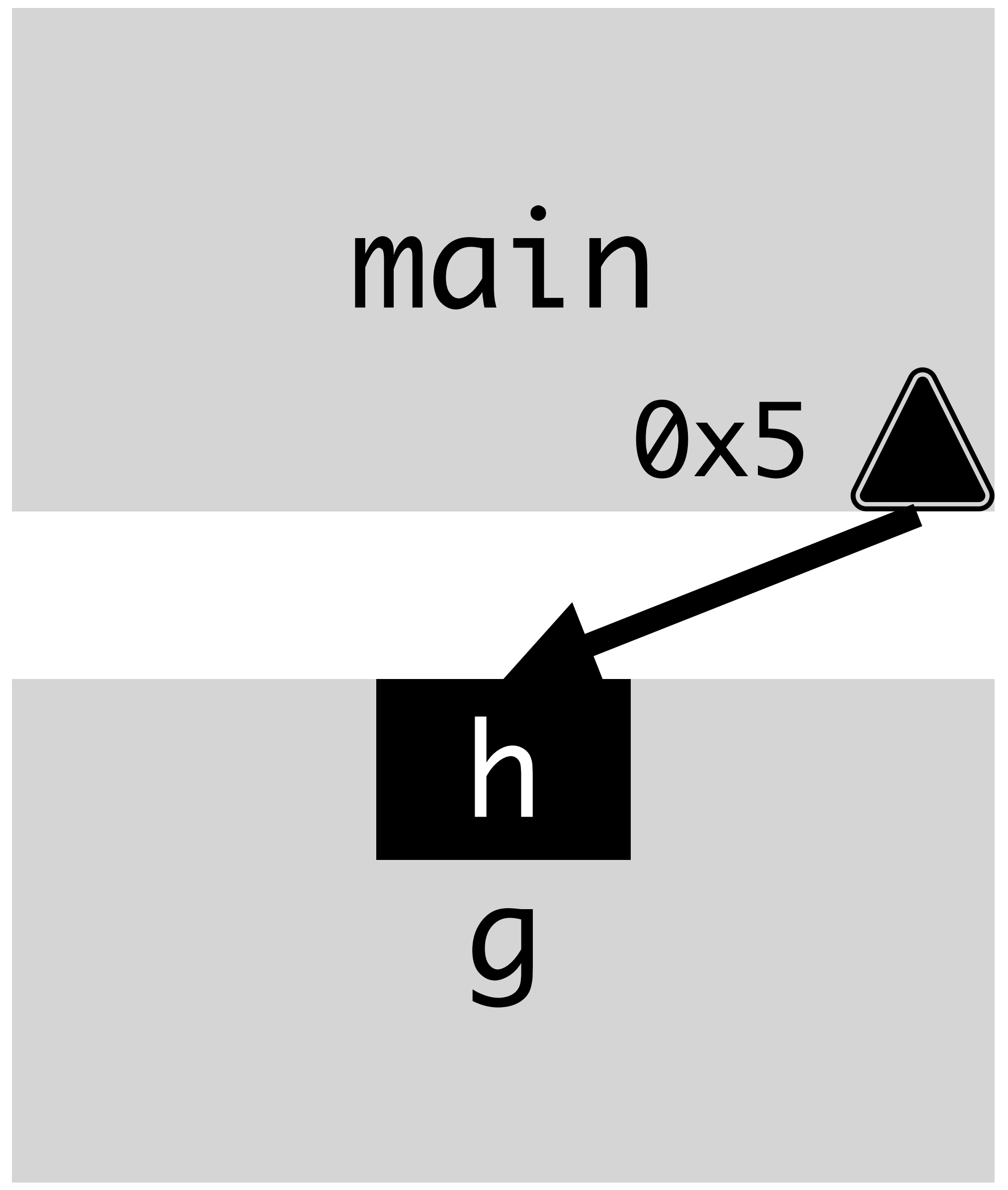
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
         handle
           f(x, h, log)
         with log = ...
in
handle
 g(42, log)
with log = ...
```



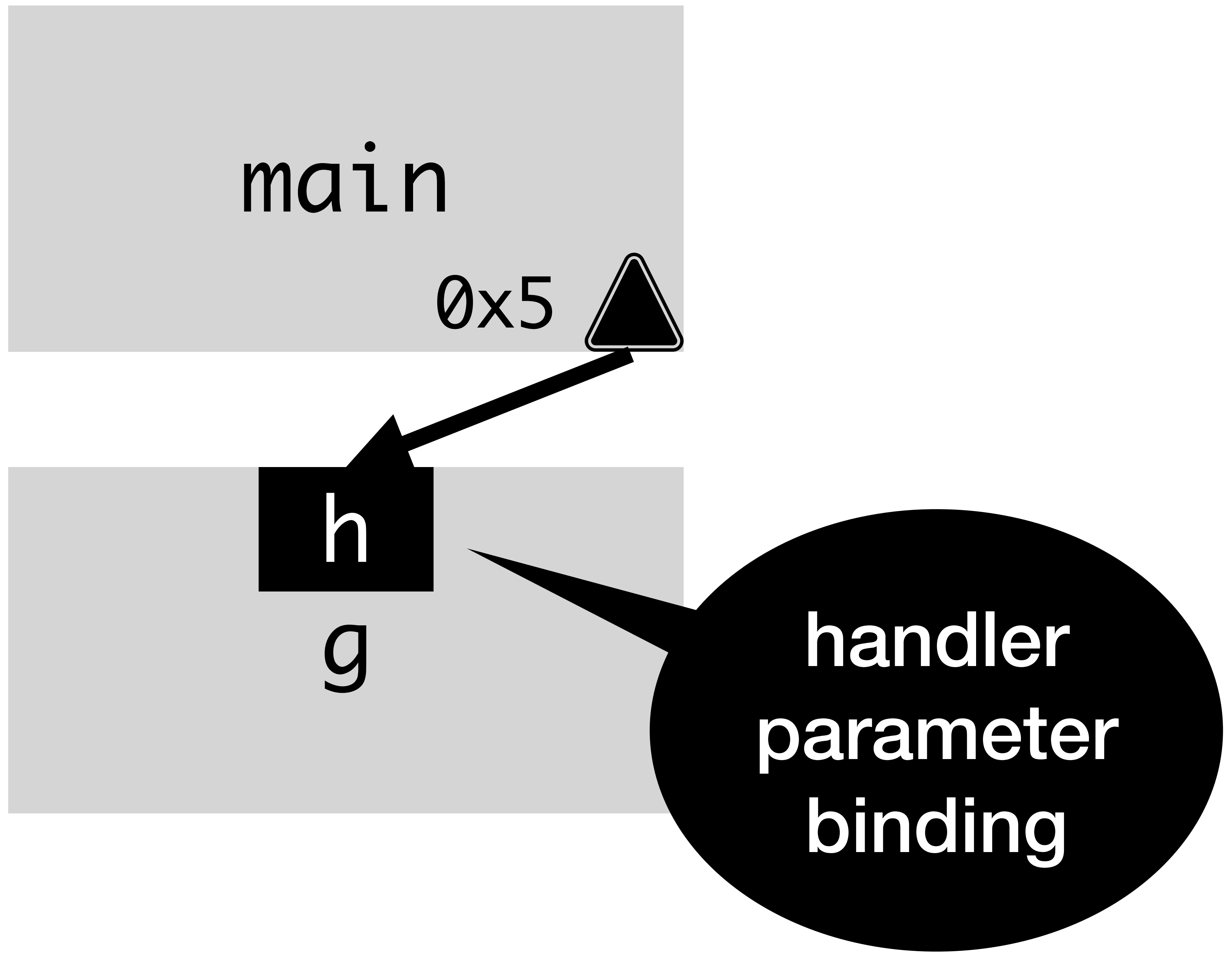
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



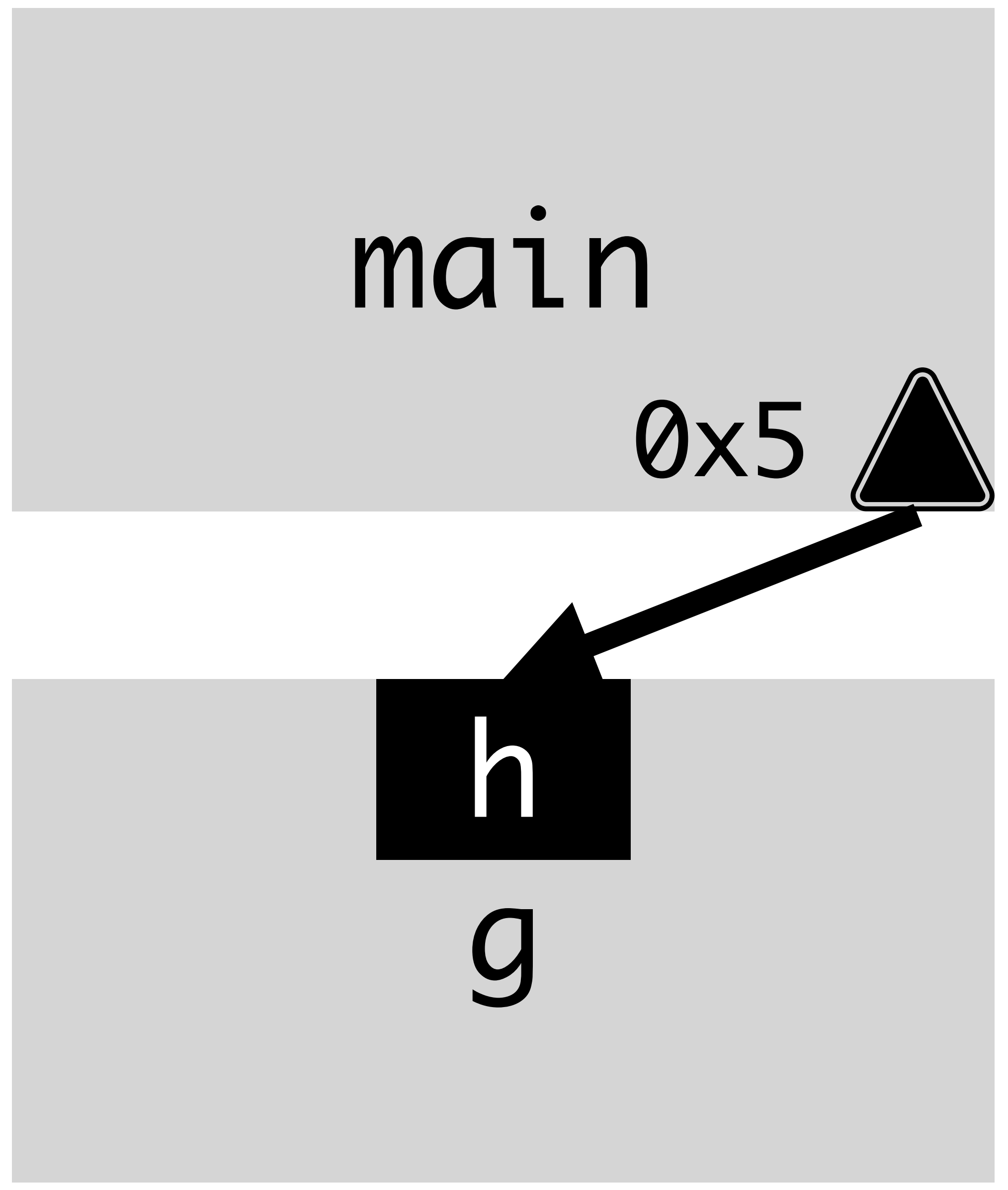
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



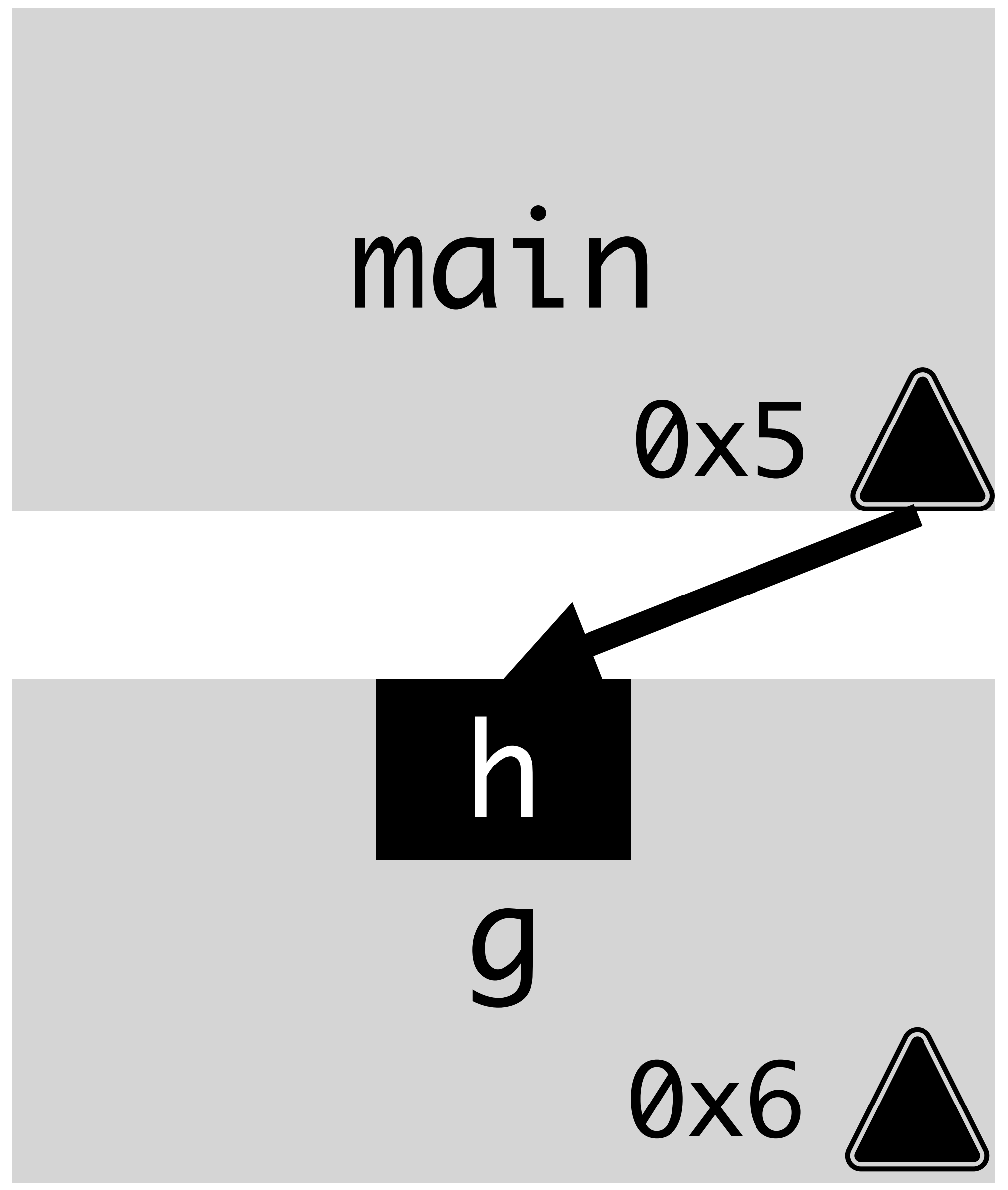
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



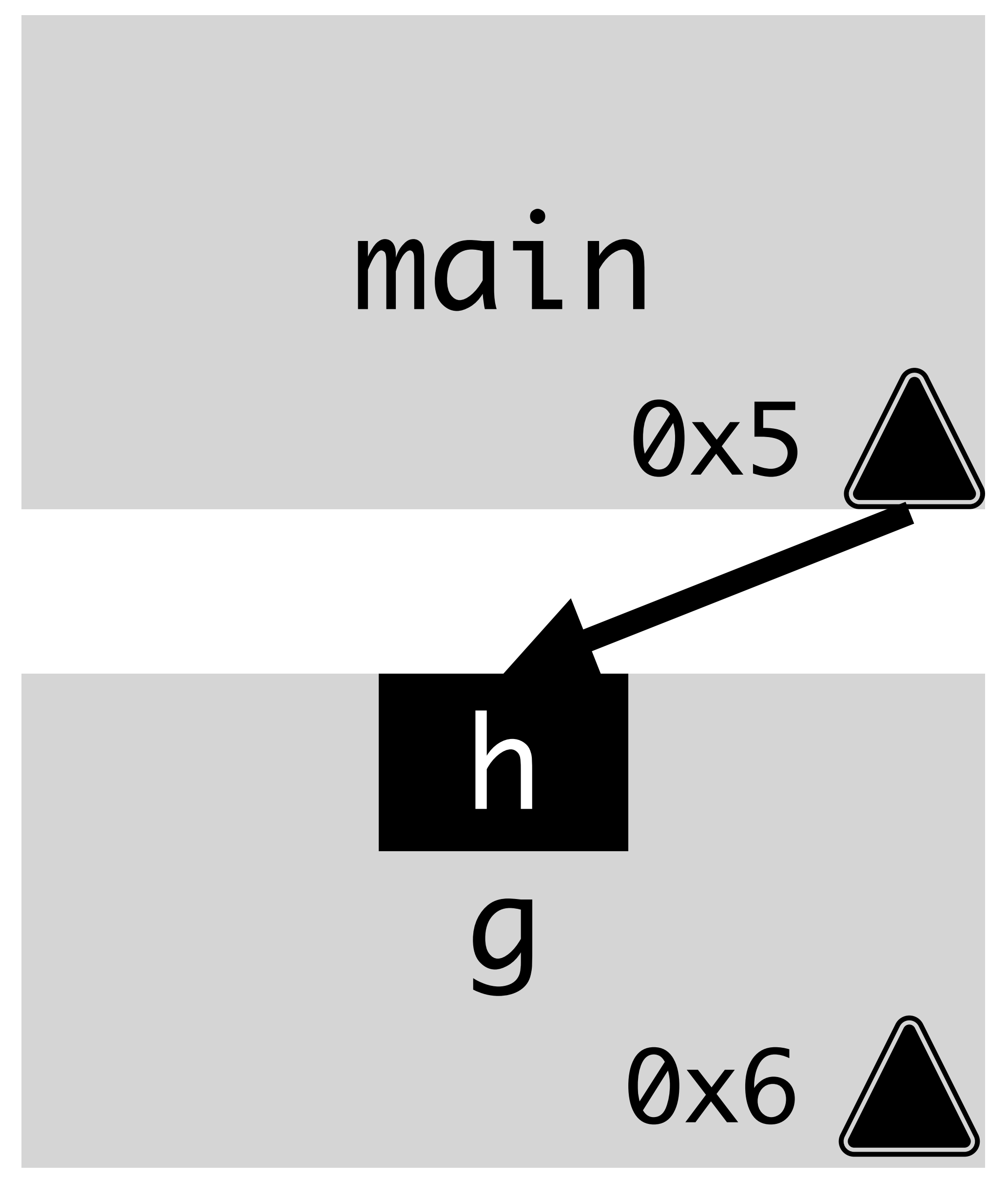
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



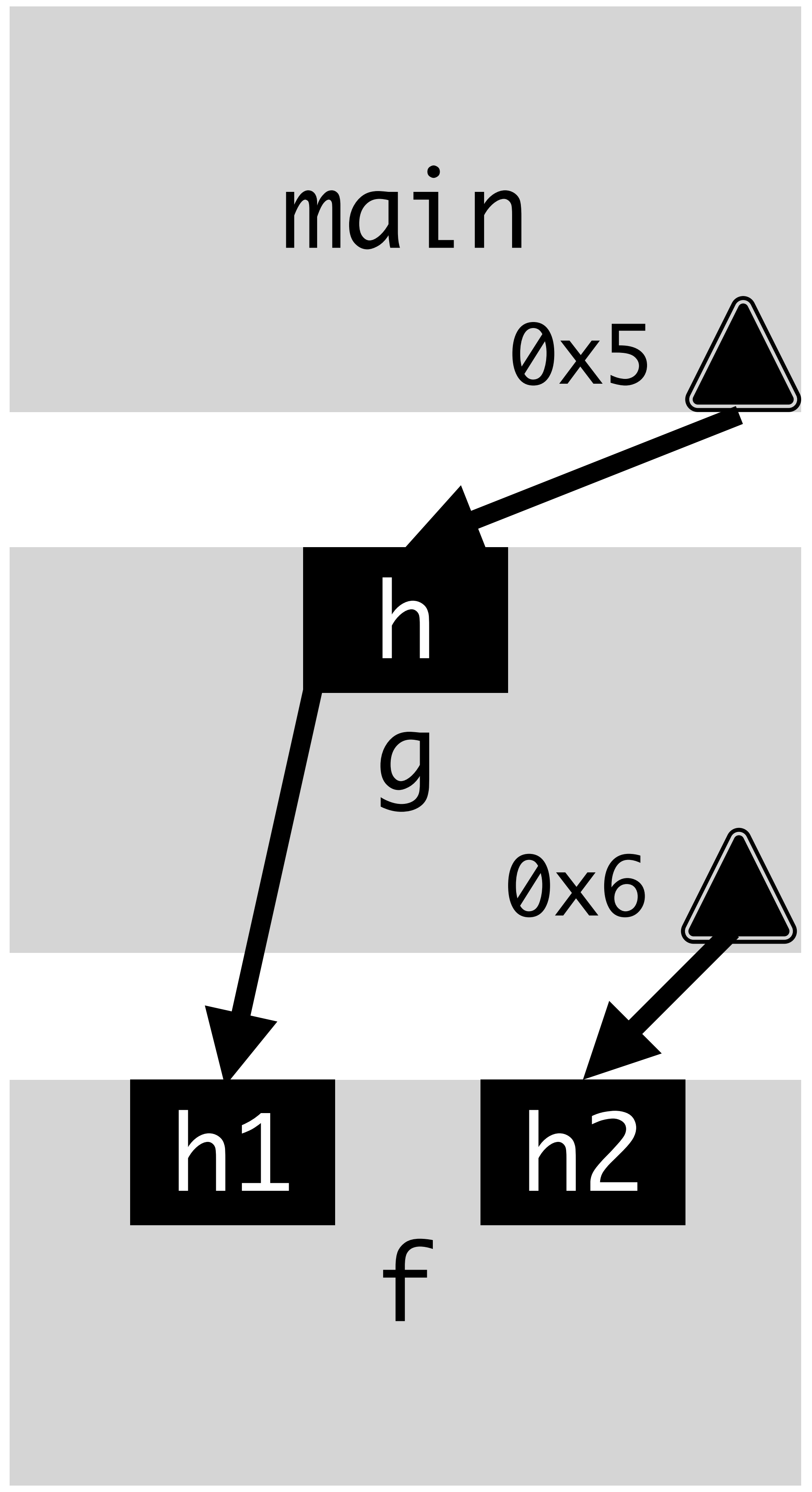
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



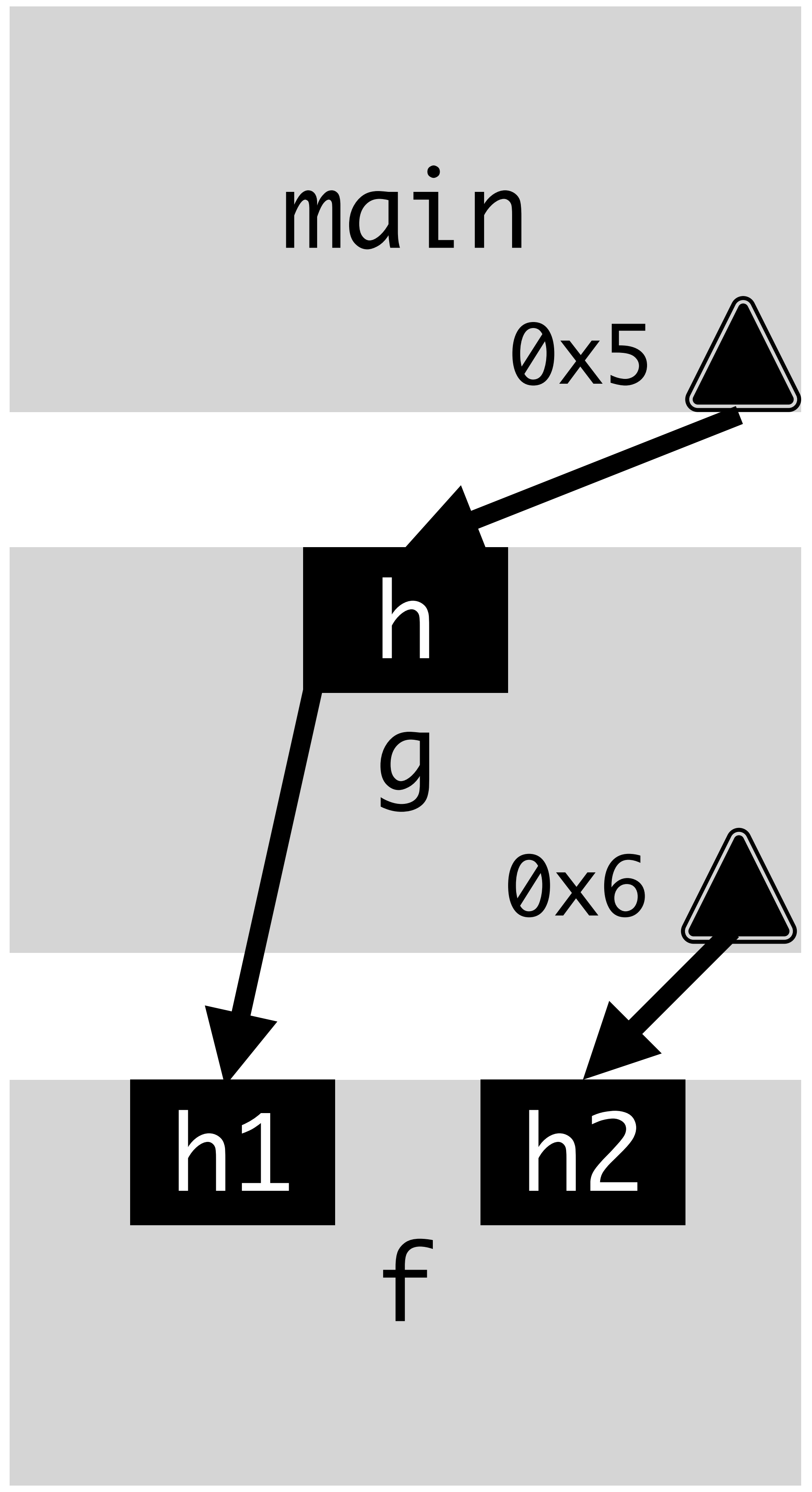
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



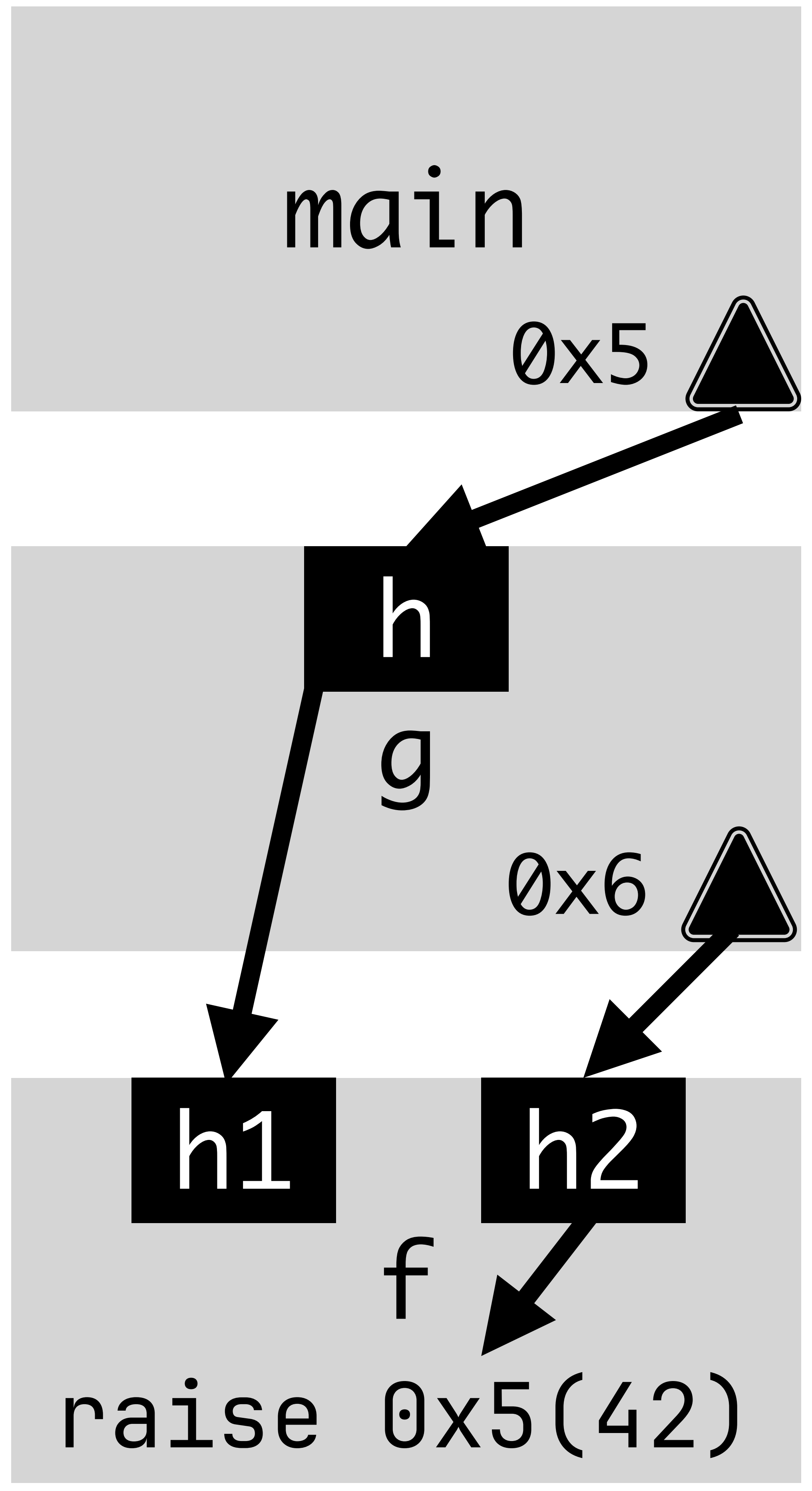
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
        raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



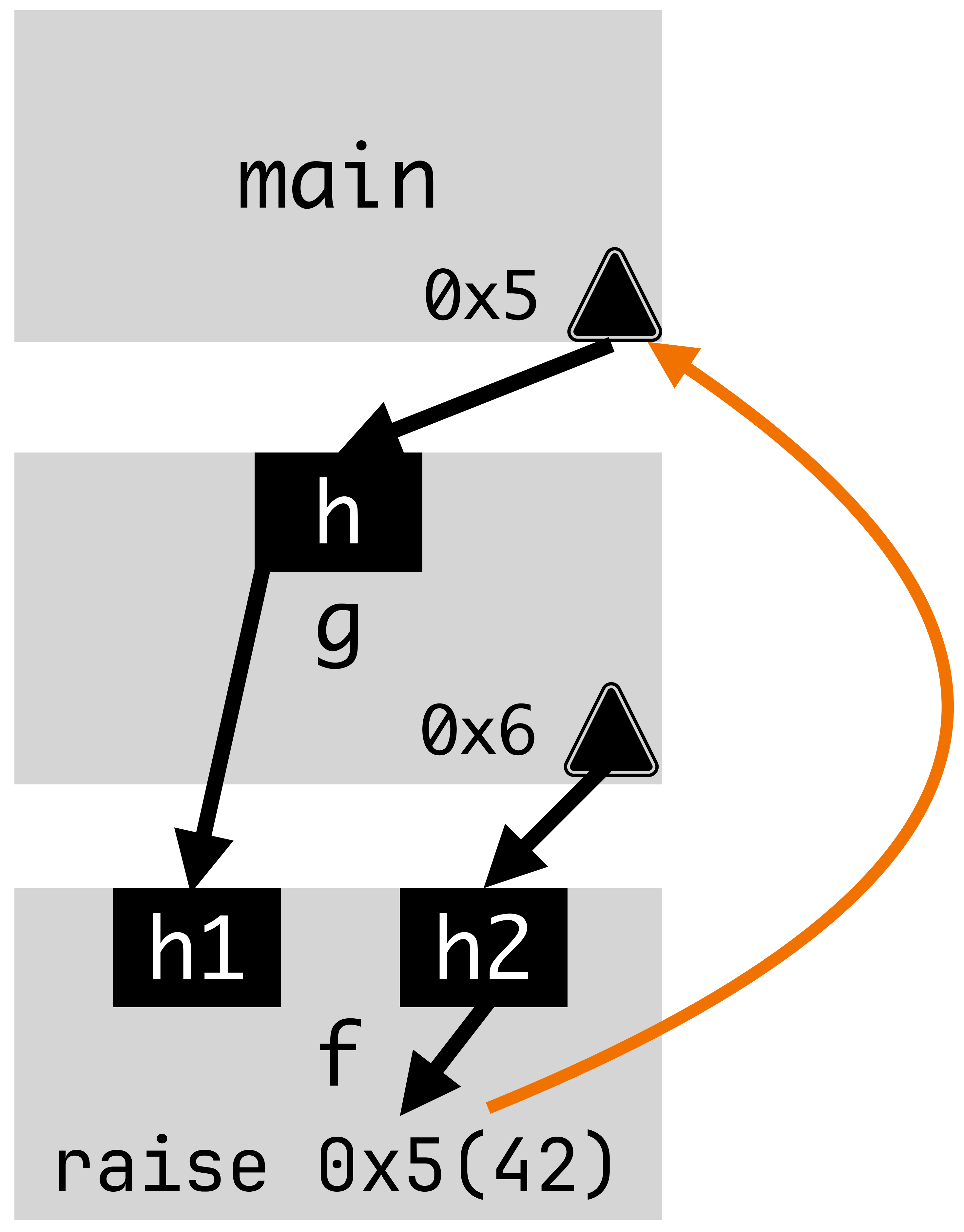
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
        raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



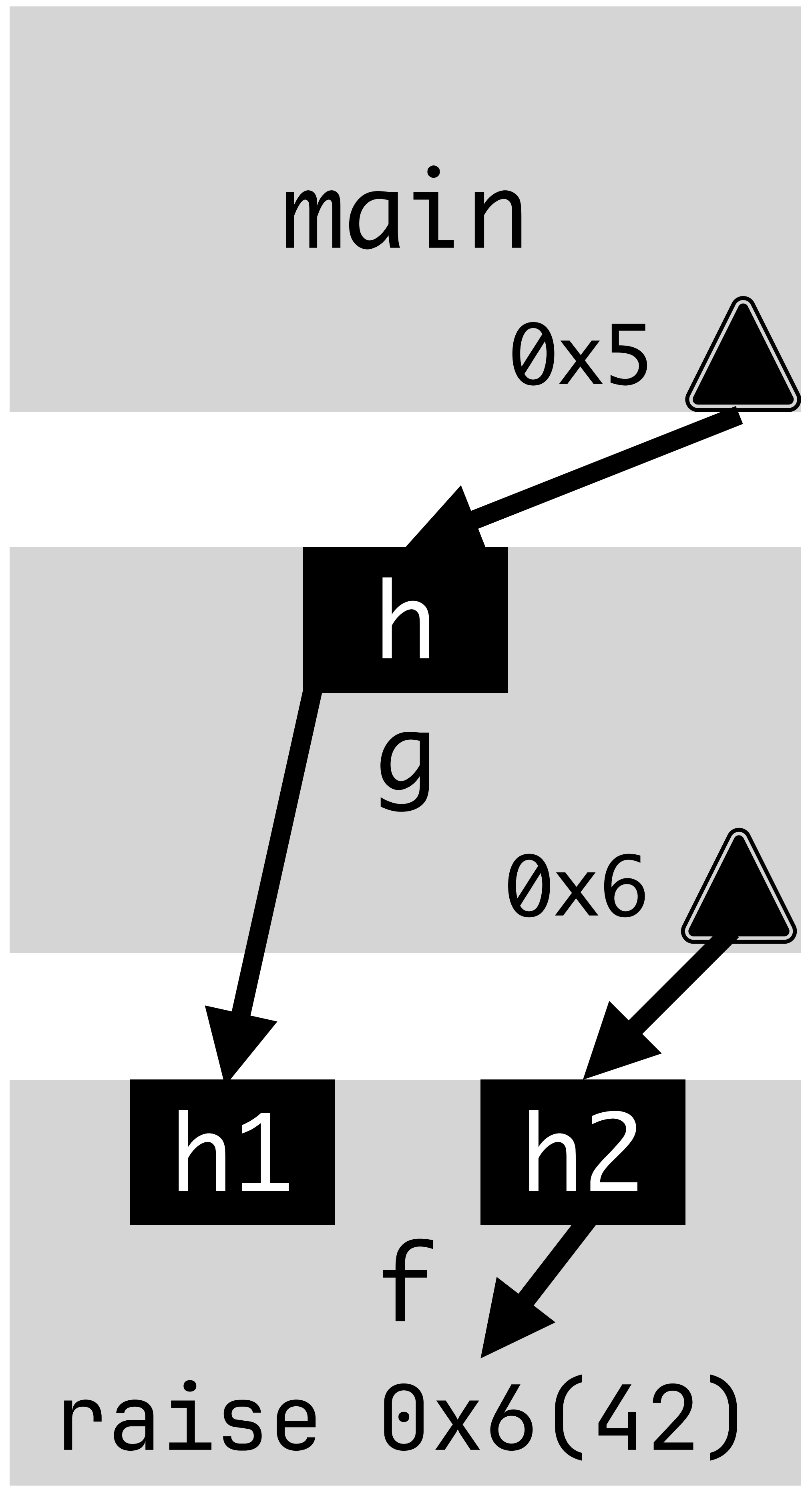
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
        raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



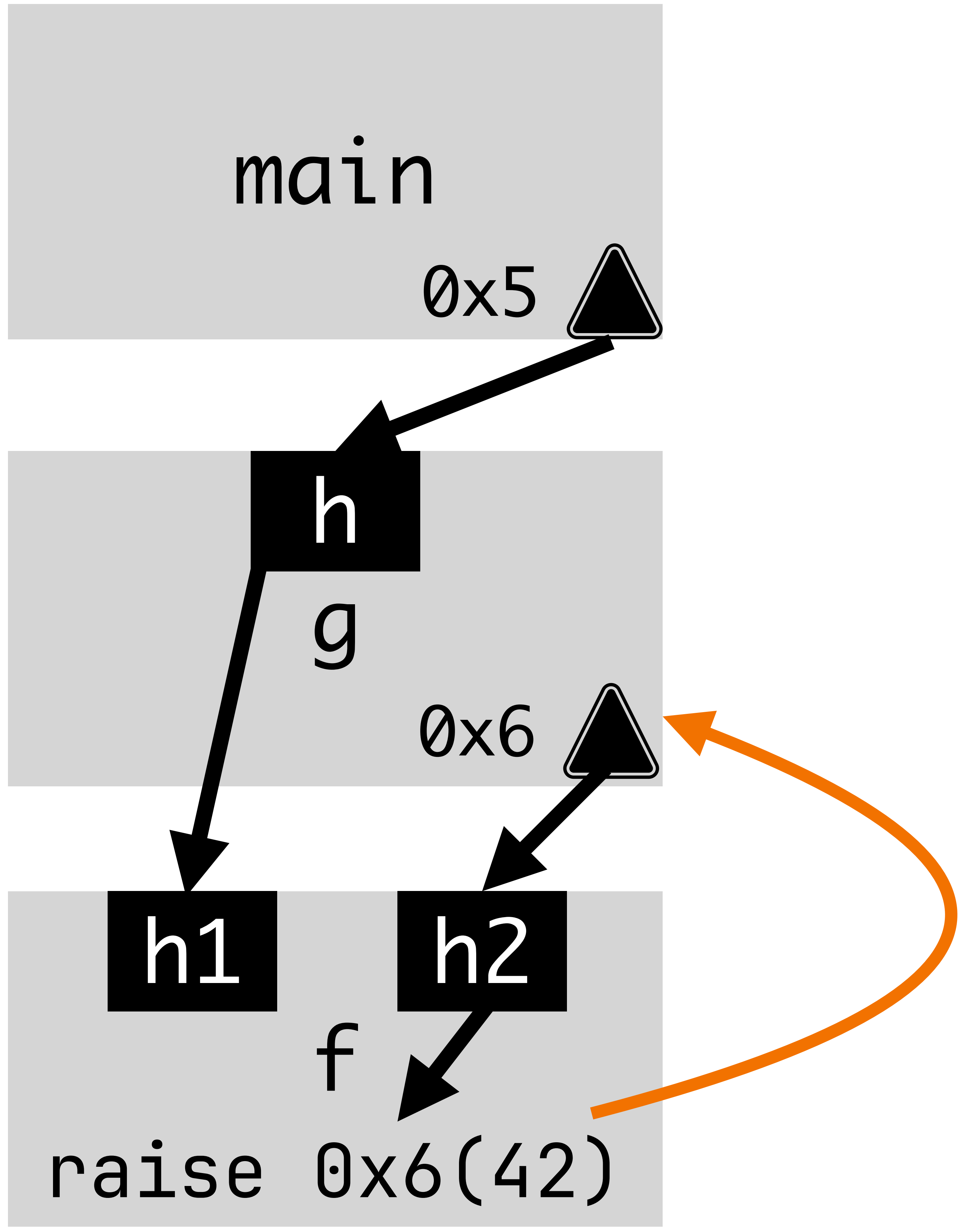
Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Zero-Cost Lexical Effect Handlers, Example 1

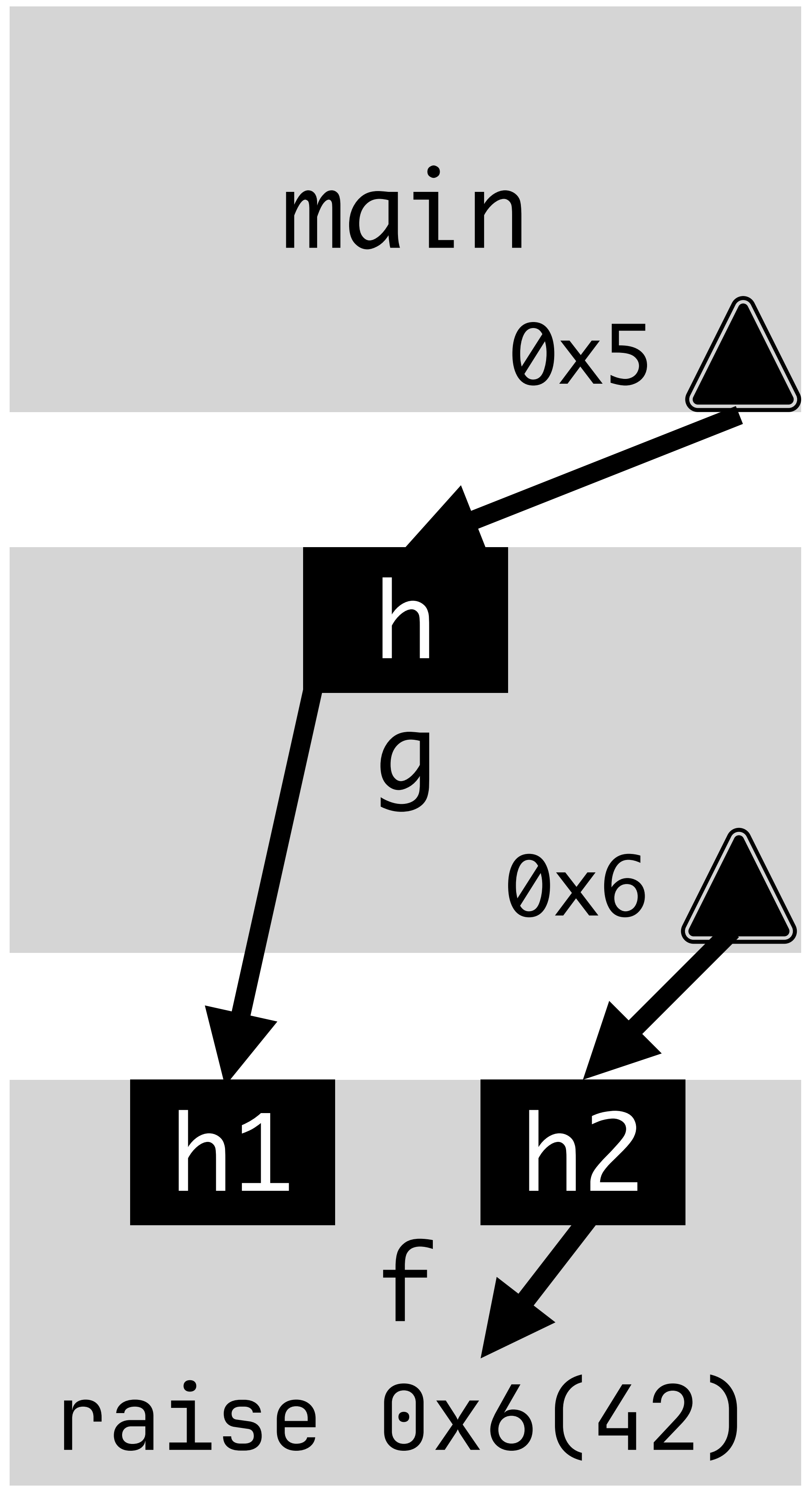
```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

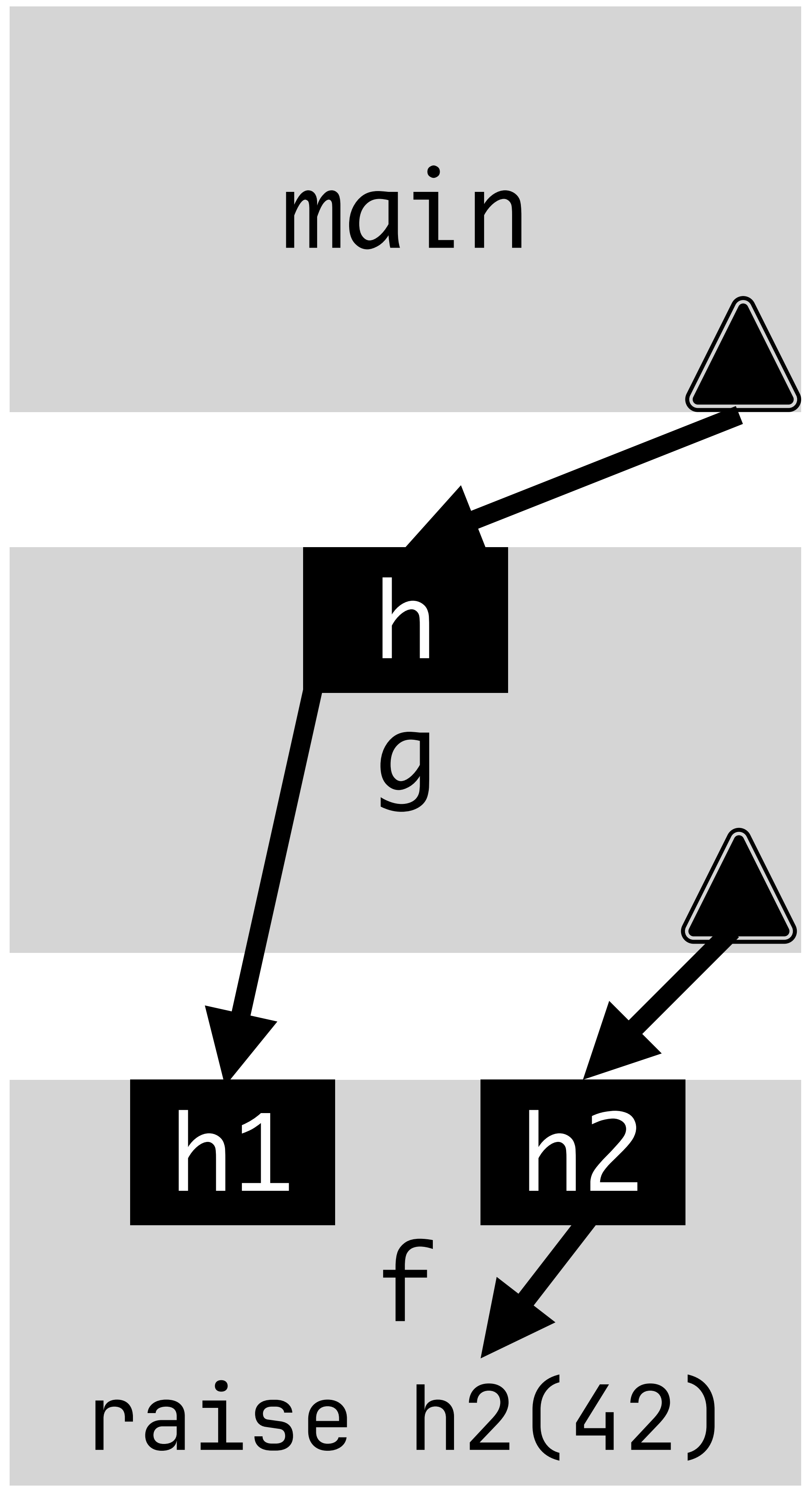
Can we locate the intended handler without passing down labels?



Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

Can we locate the intended handler without passing down labels?

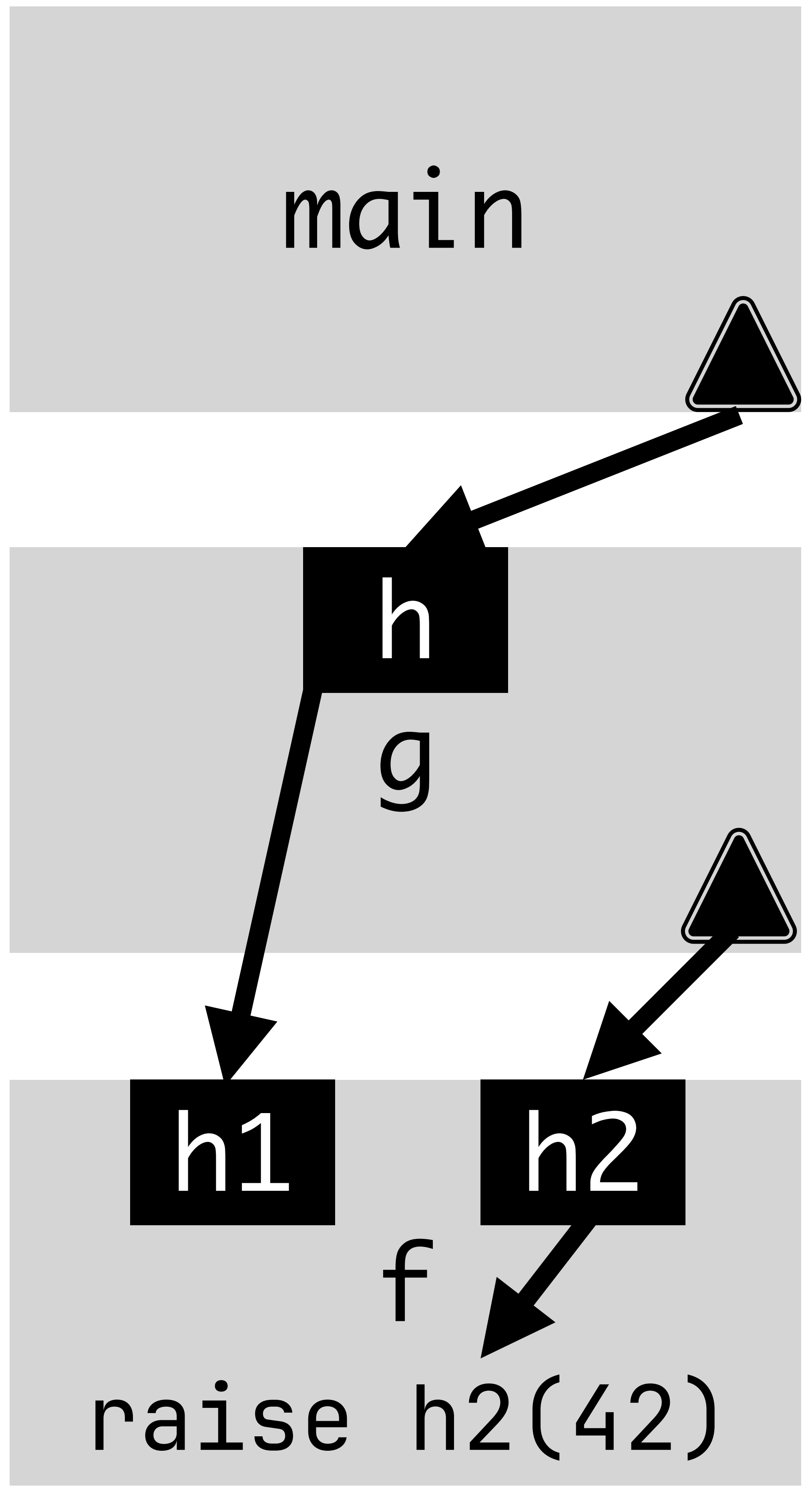


Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

Can we locate the intended handler without passing down labels?

Yes, just reverse the arrows!

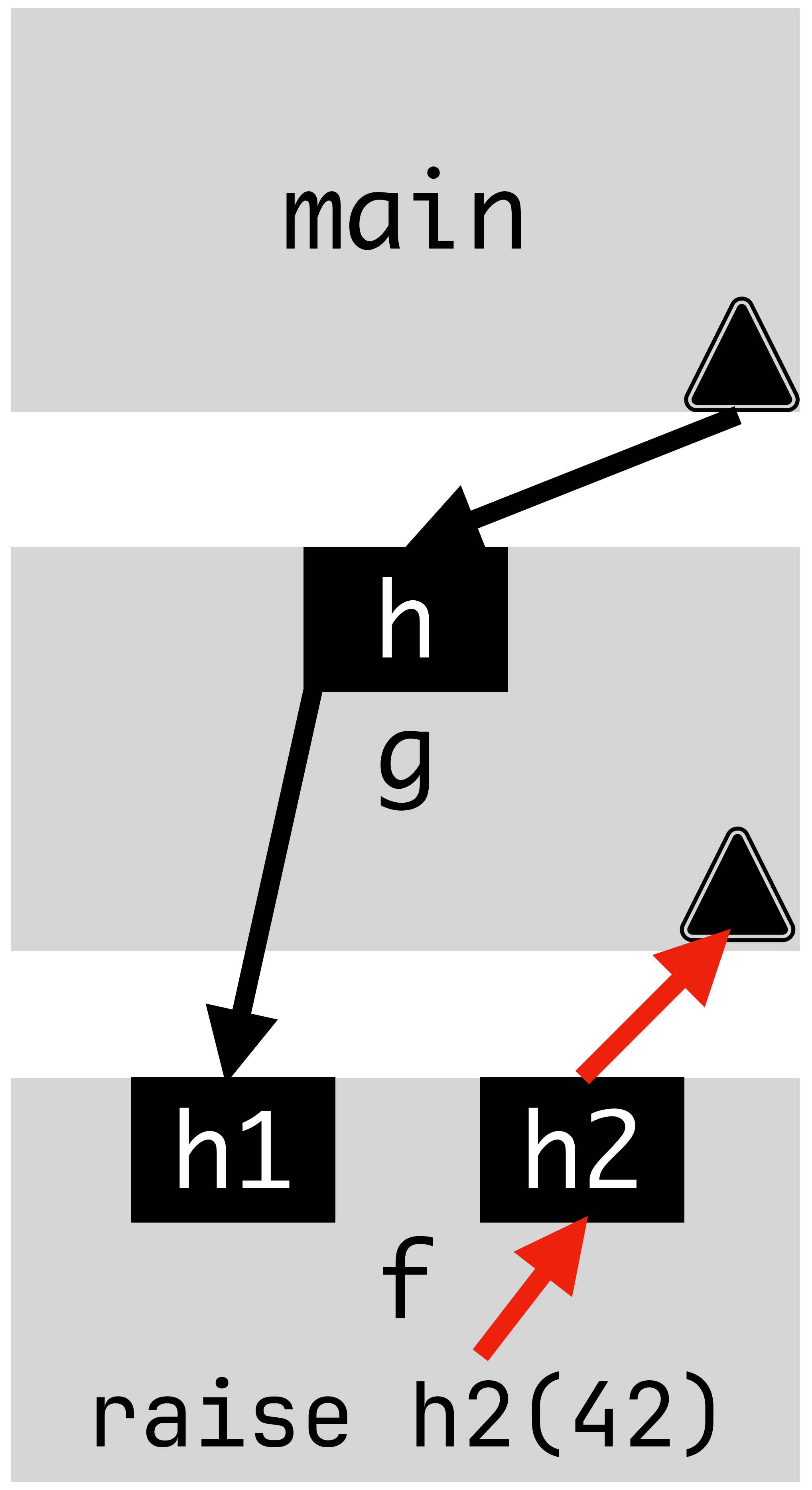


Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
          handle
            f(x, h, log)
          with log = ...
in
handle
 g(42, log)
with log = ...
```

Can we locate the intended handler without passing down labels?

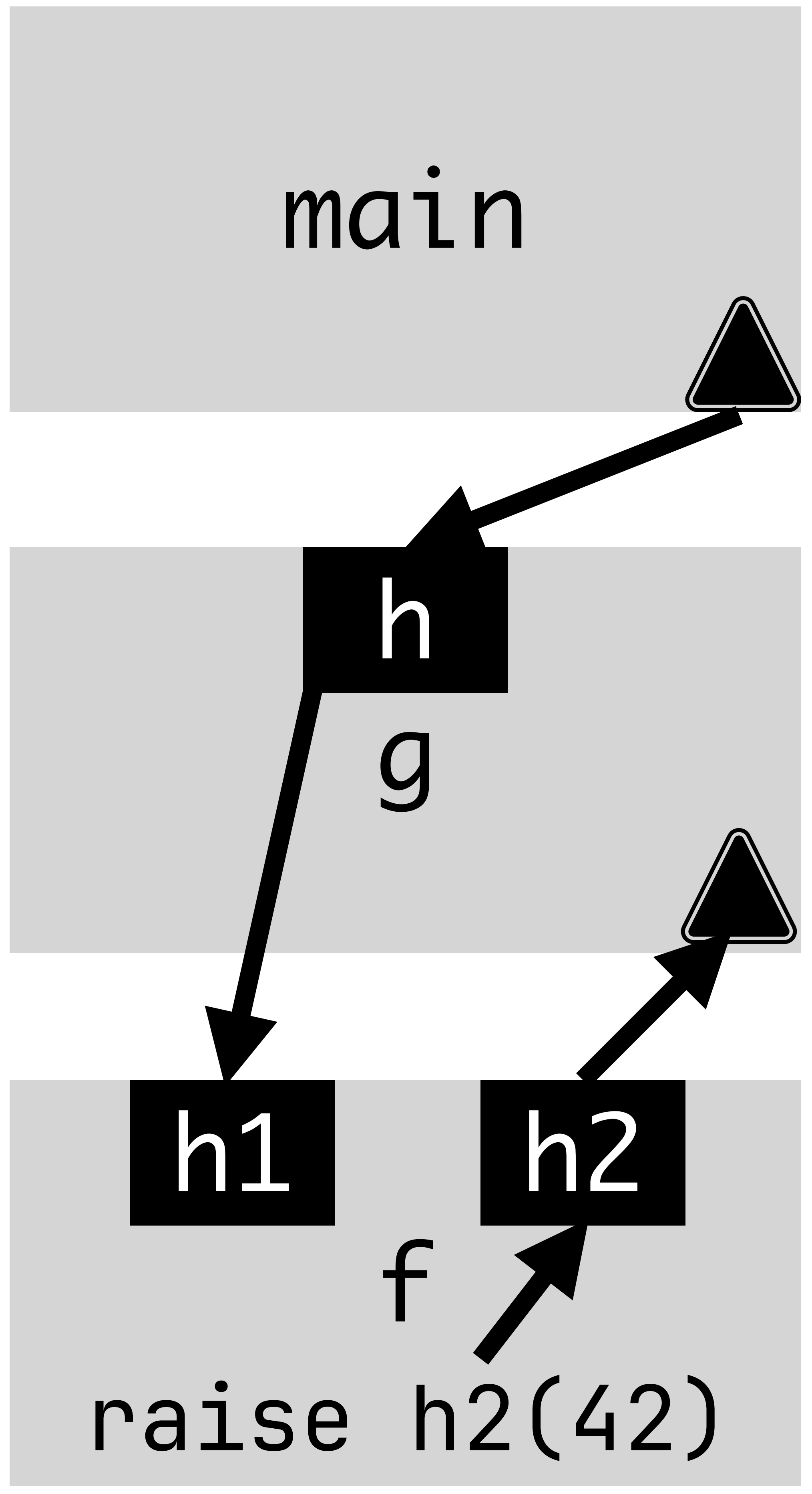
Yes, just reverse the arrows!



Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

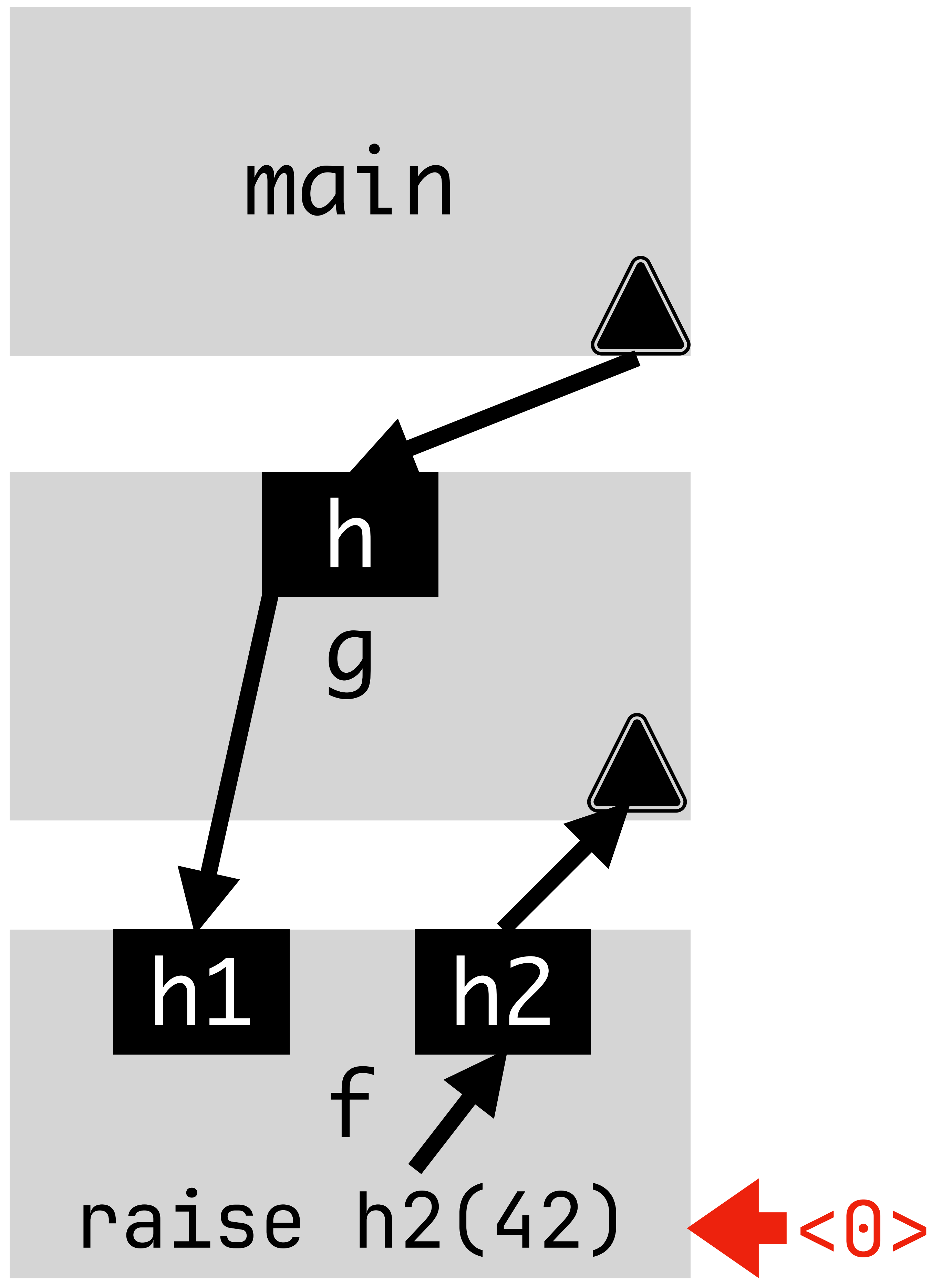
Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.



Zero-Cost Lexical Effect Handlers, Example 1

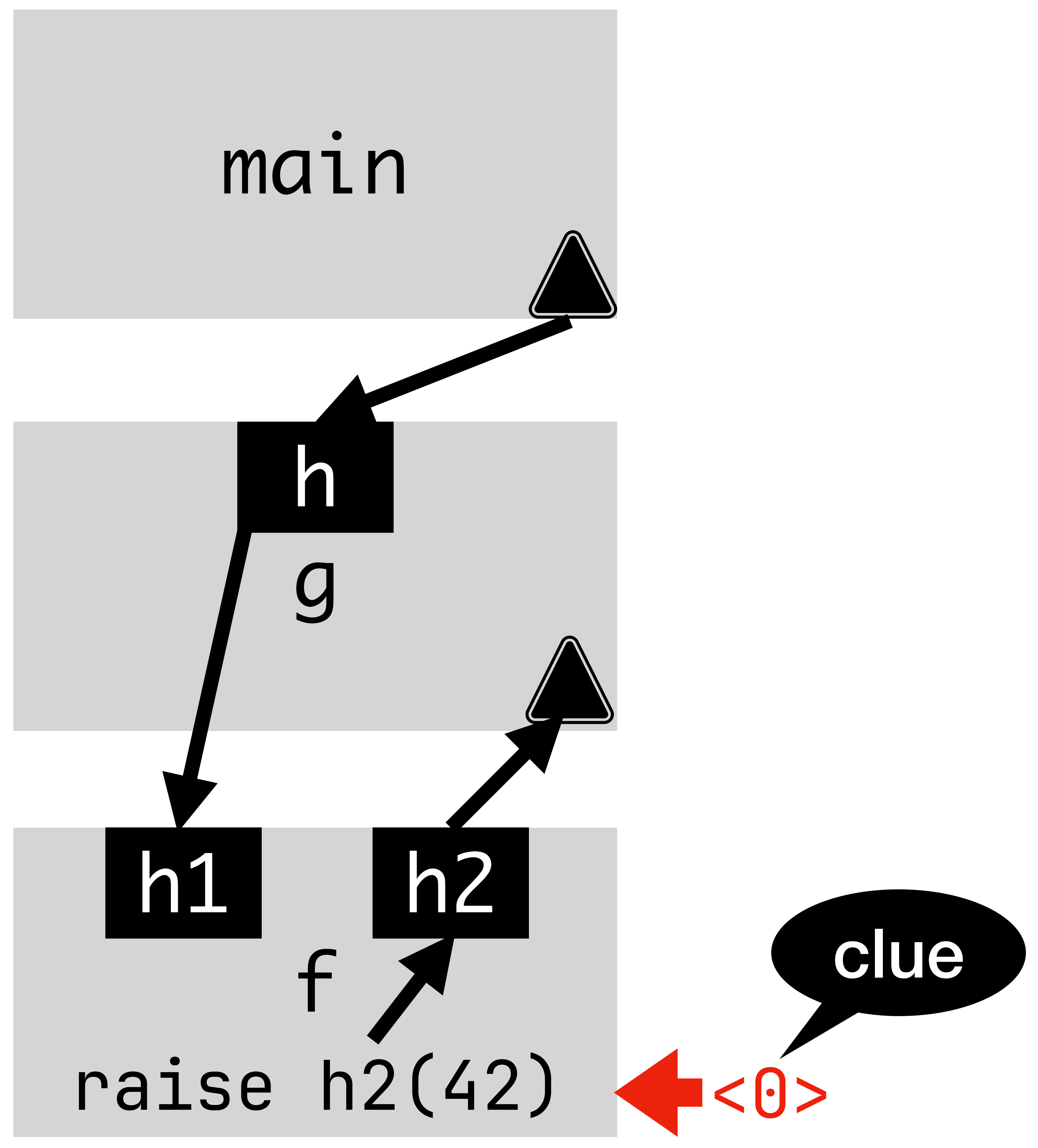
```
#main
let f = λ(x, h1, h2).
          raise h1(x); raise h2(x)
let g = λ(x, h).
         handle
           f(x, h, log)
         with log = ...
in
handle
 g(42, log)
with log = ...
```

Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.



Zero-Cost Lexical Effect Handlers, Example 1

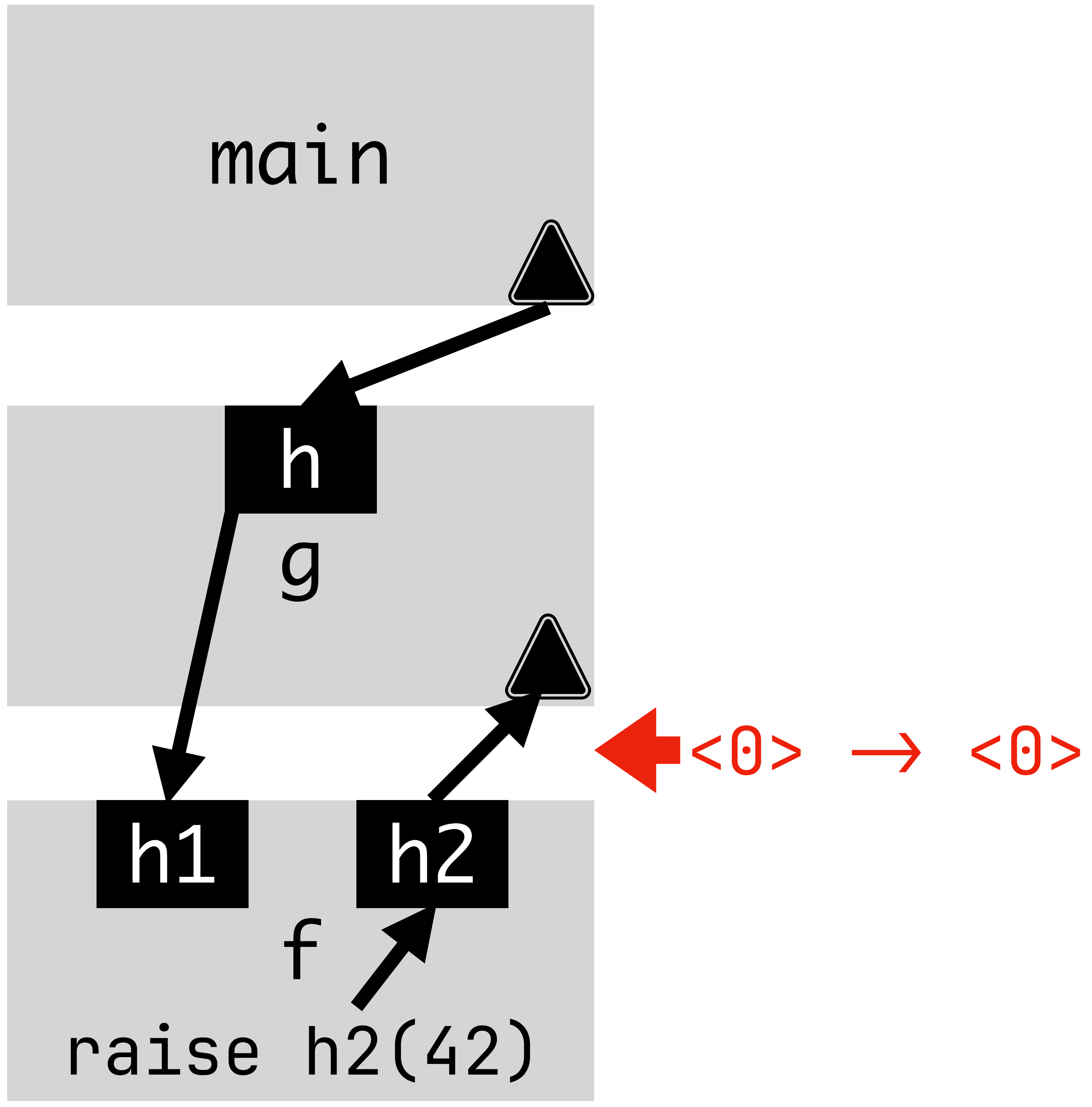
```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.

Zero-Cost Lexical Effect Handlers, Example 1

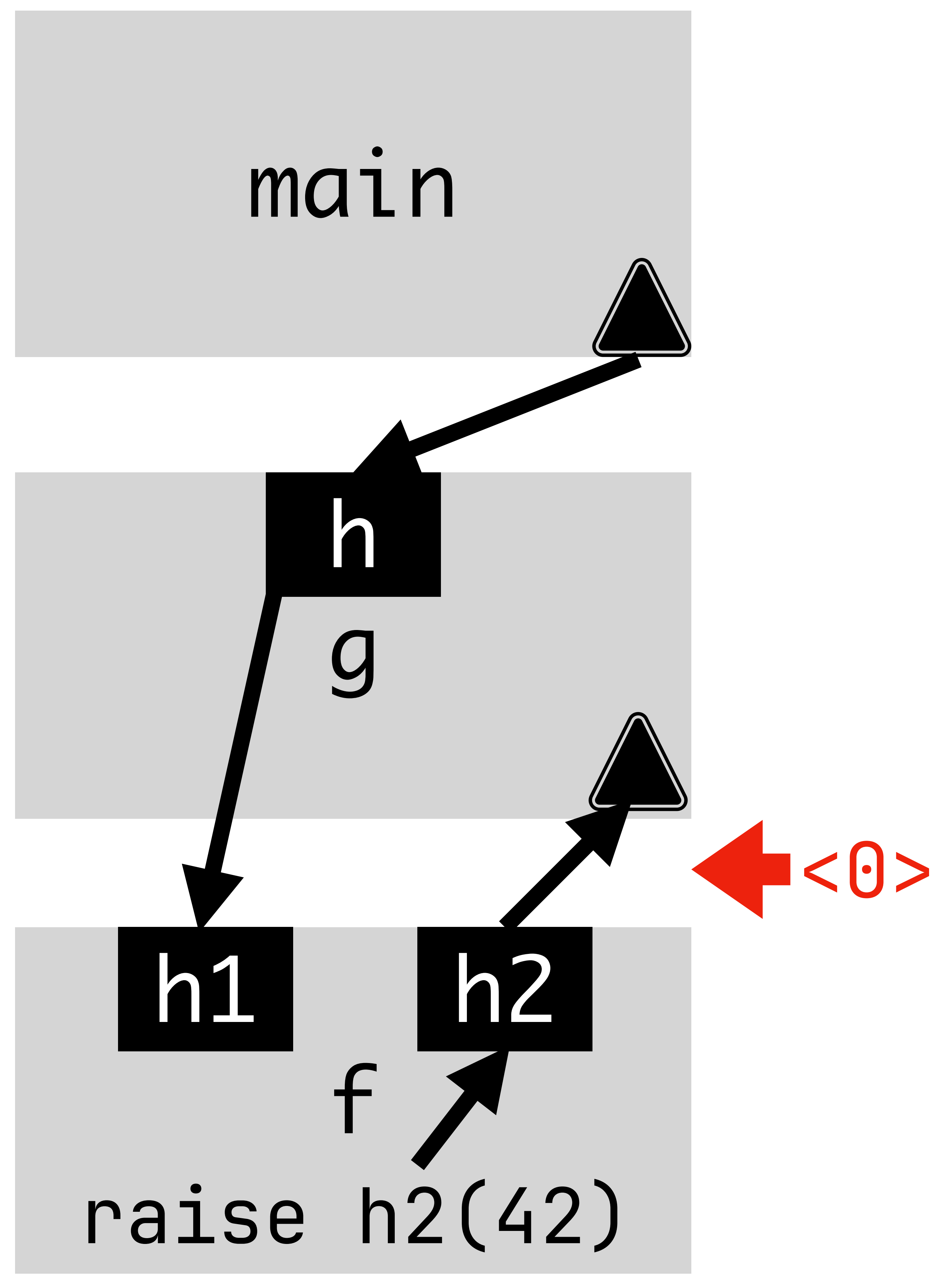
```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.

Zero-Cost Lexical Effect Handlers, Example 1

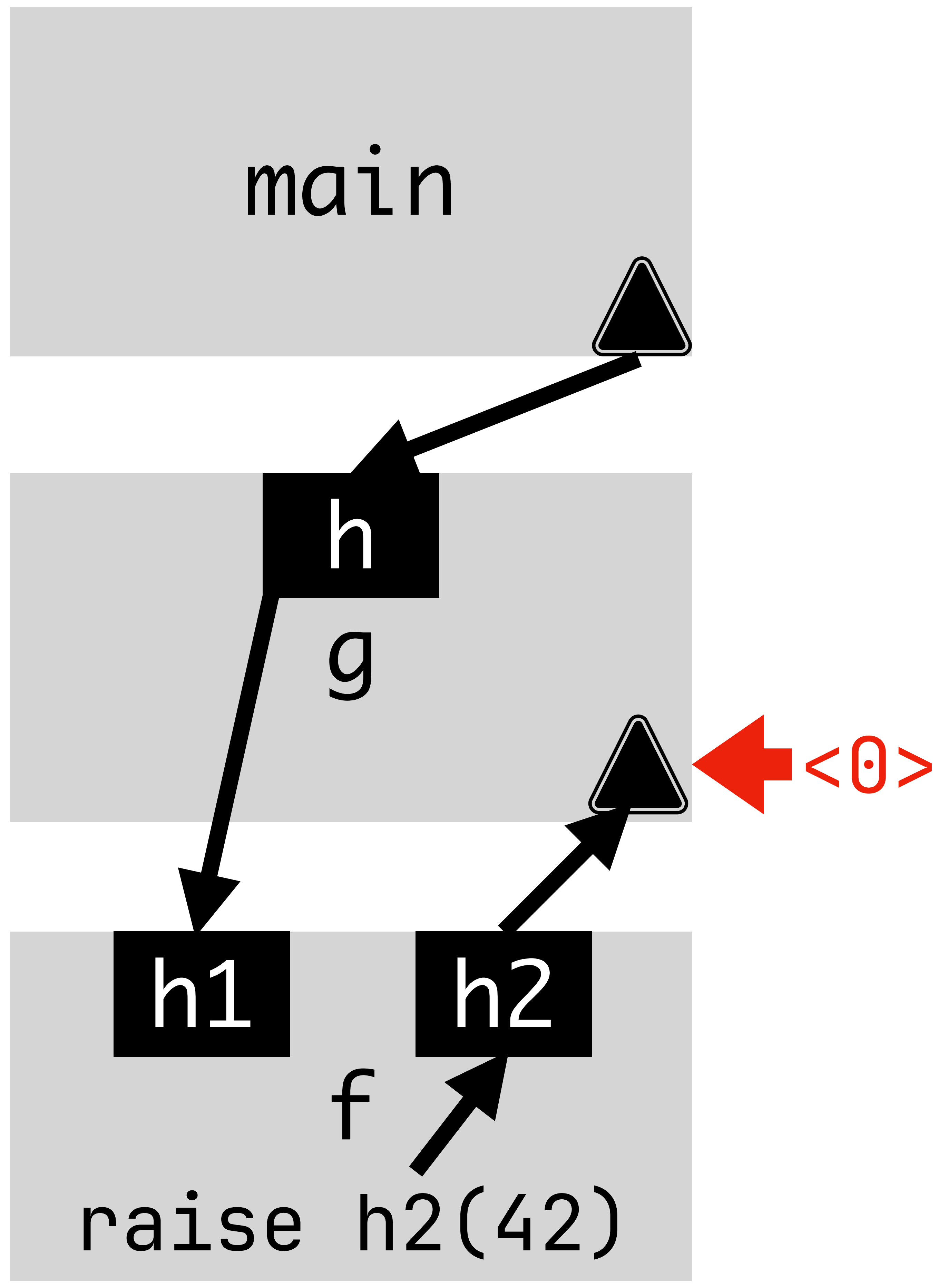
```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.

Zero-Cost Lexical Effect Handlers, Example 1

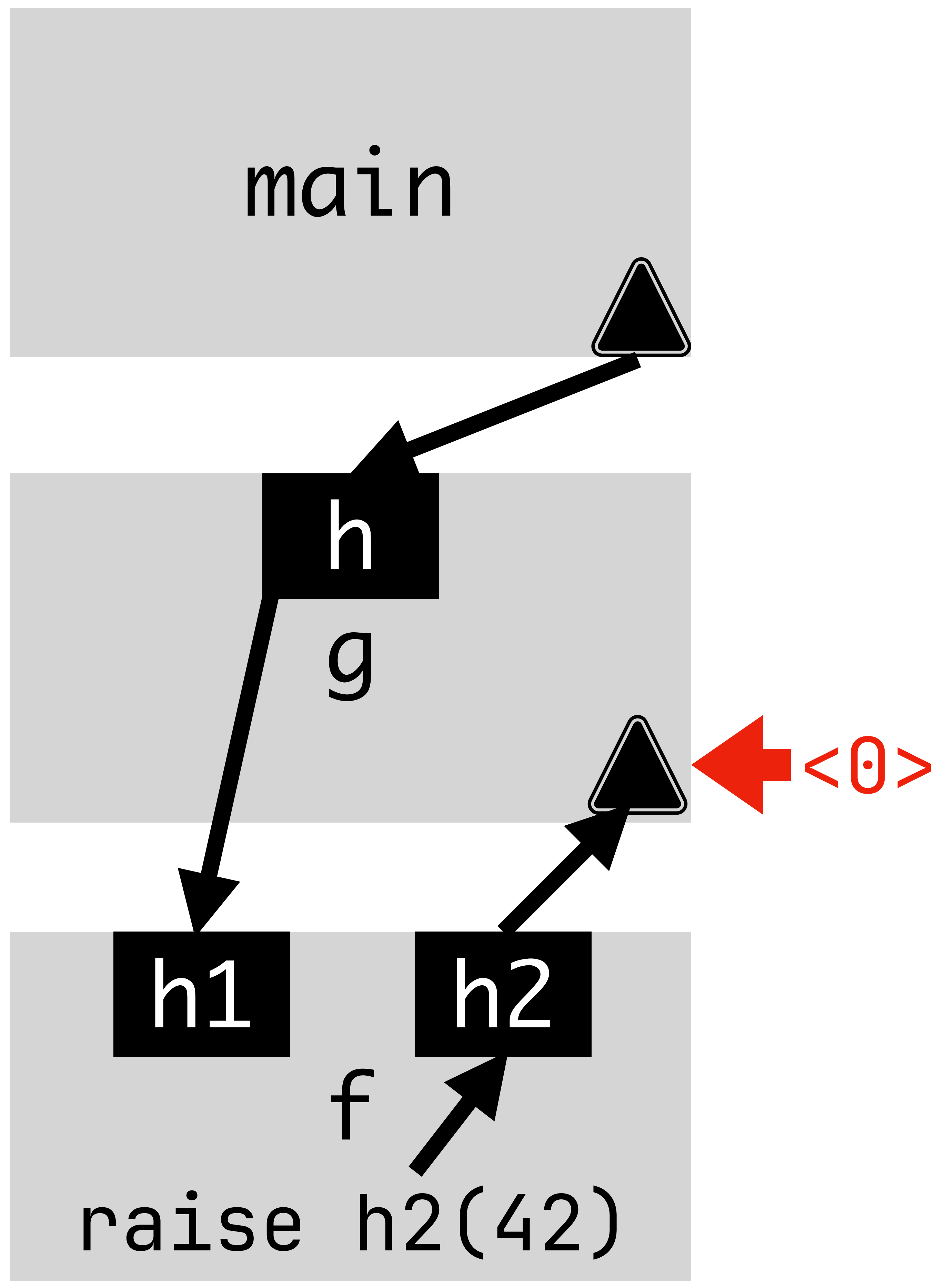
```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```



Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.

Zero-Cost Lexical Effect Handlers, Example 1

```
#main
let f = λ(x, h1, h2).
         raise h1(x); raise h2(x)
let g = λ(x, h).
        handle
          f(x, h, log)
        with log = ...
in
handle
 g(42, log)
with log = ...
```

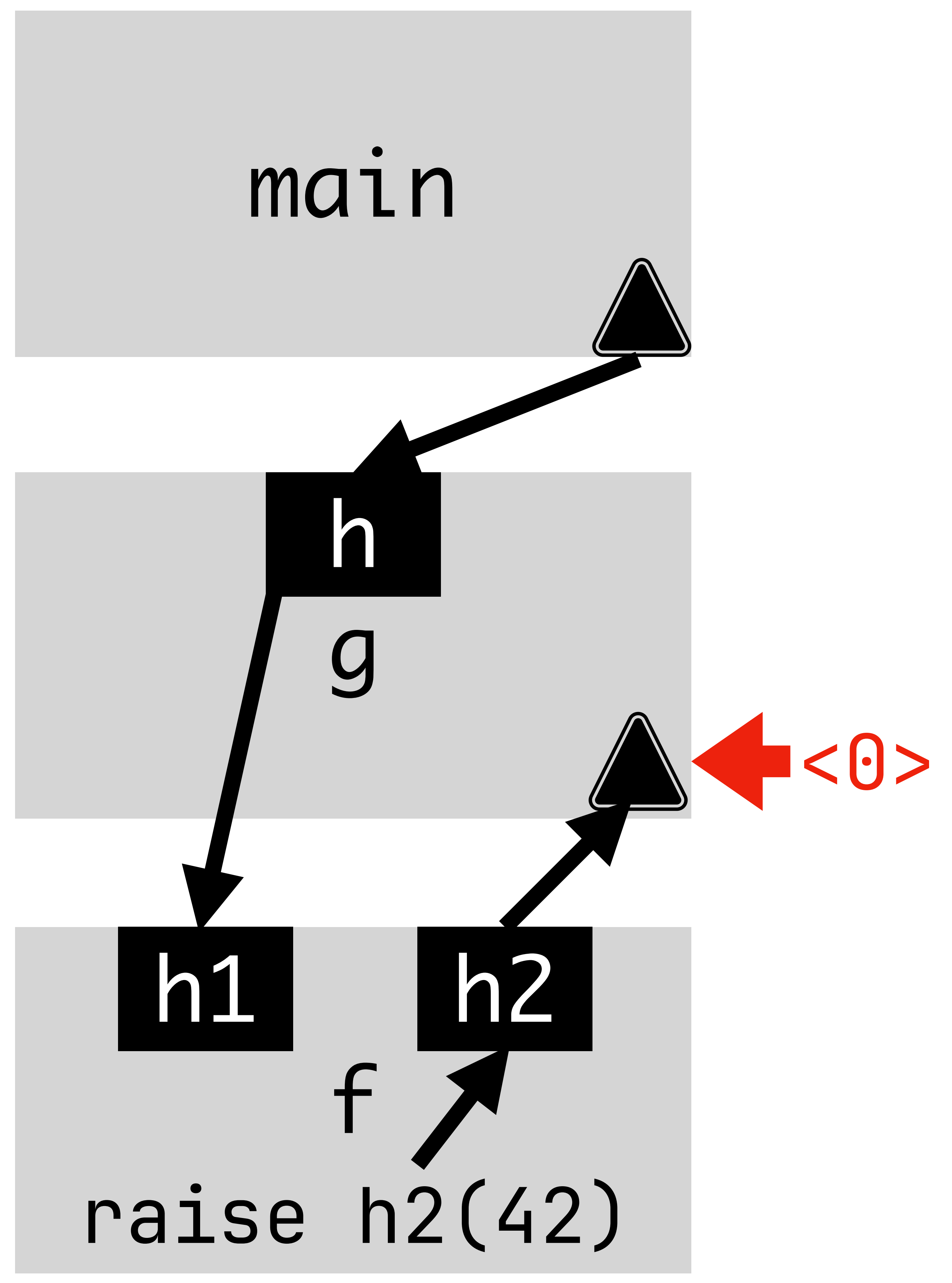


Implementation-wise, a stackwalker walks the stack. It carries the De Bruijn index of the intended handler variable.

Zero-Cost Lexical Effect Handlers, Example 1

This semantics finds the handler by walking the stack, so its performance characteristics is similar to dynamically scoped handlers.

Moreover, it walks the stack more “carefully” and simulates the behavior of lexically scoped handlers, so it enjoys modularity.

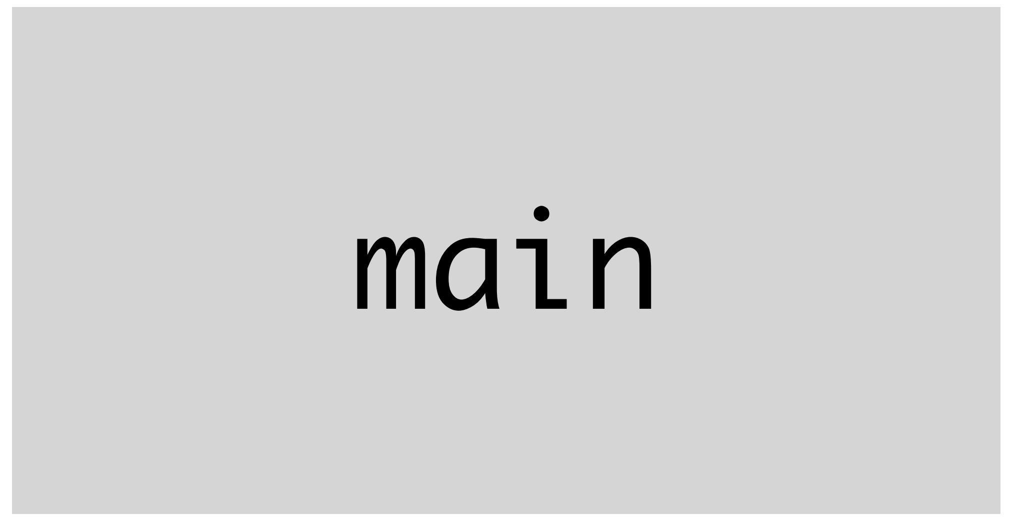


Zero-Cost Lexical Effect Handlers, Example 2

Our compilation can also deal with higher-order functions.

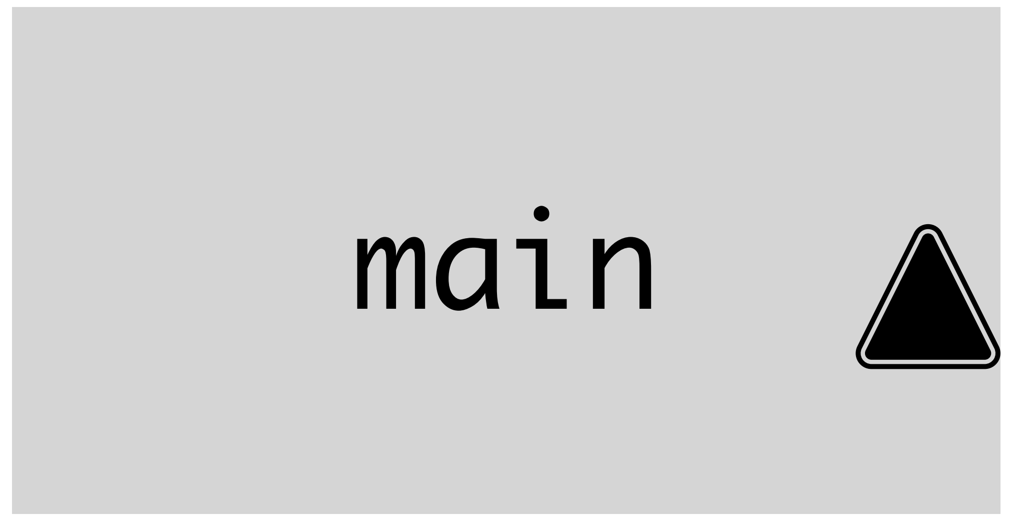
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



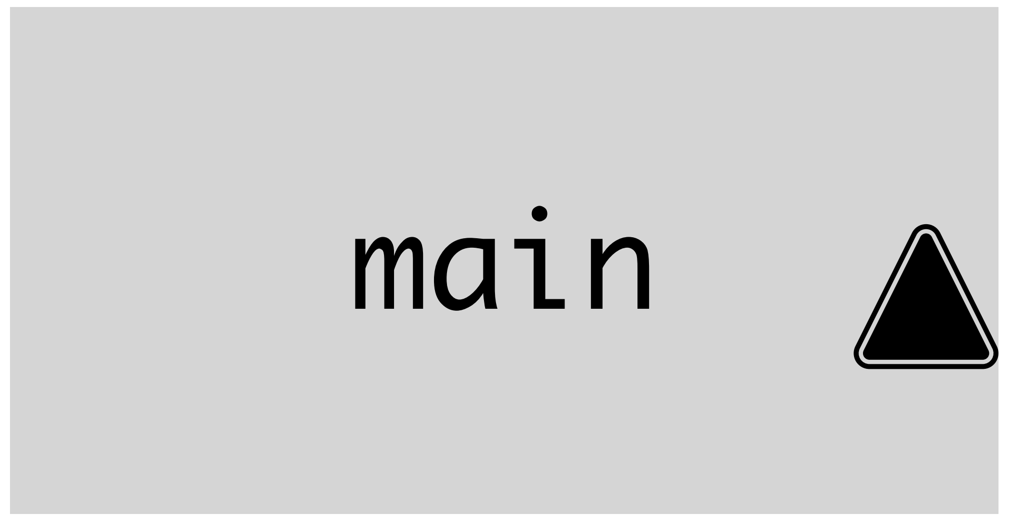
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



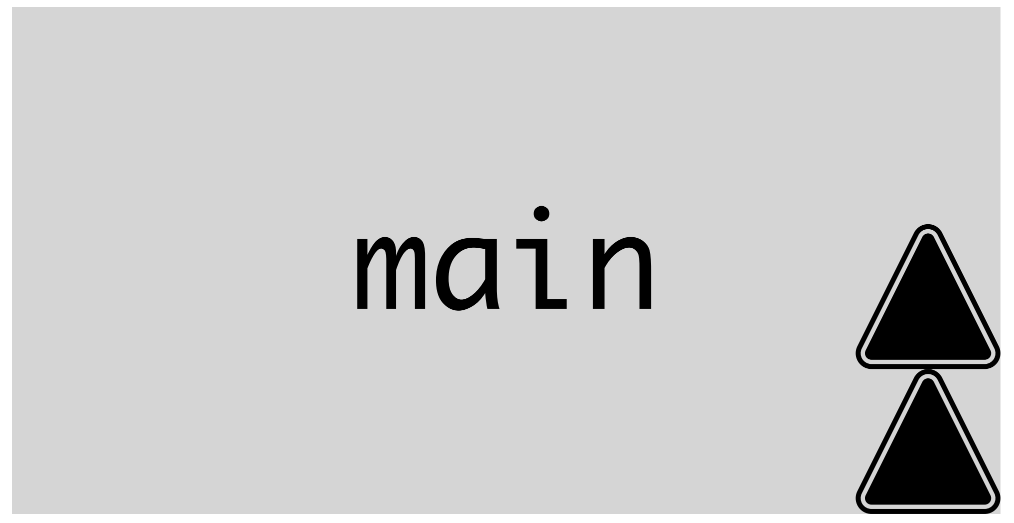
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



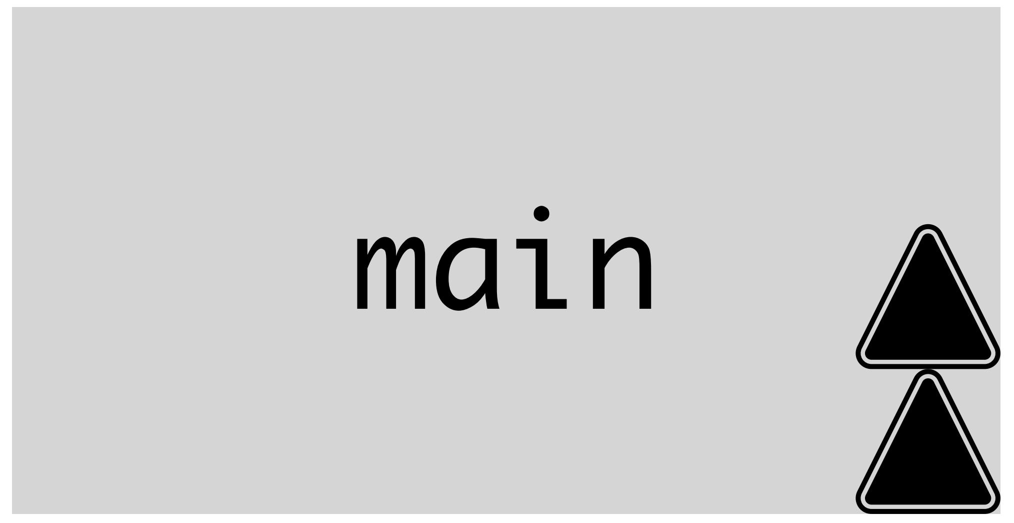
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



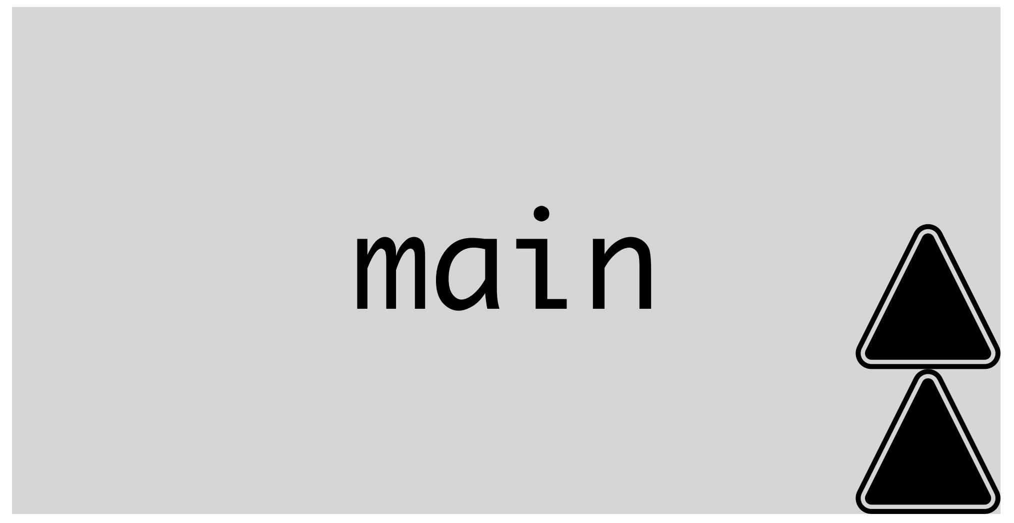
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



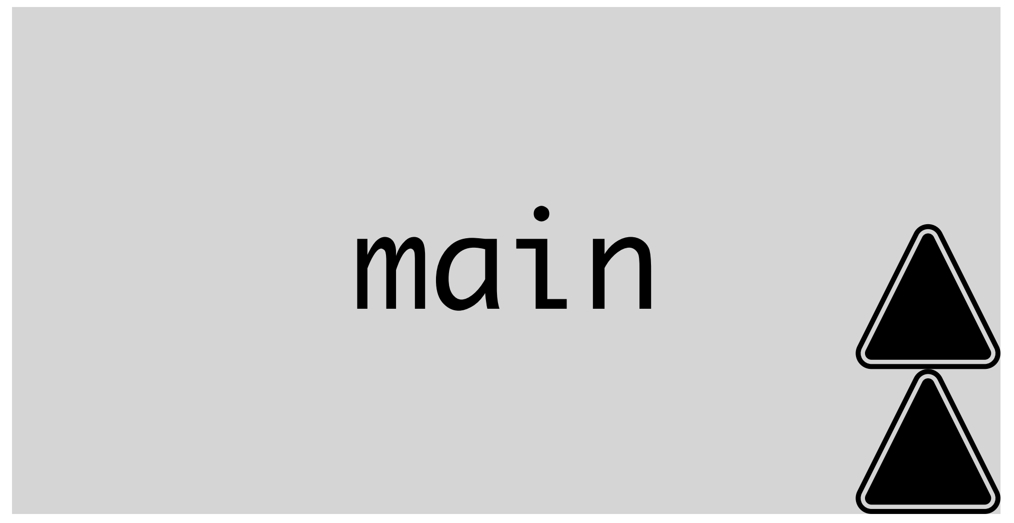
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



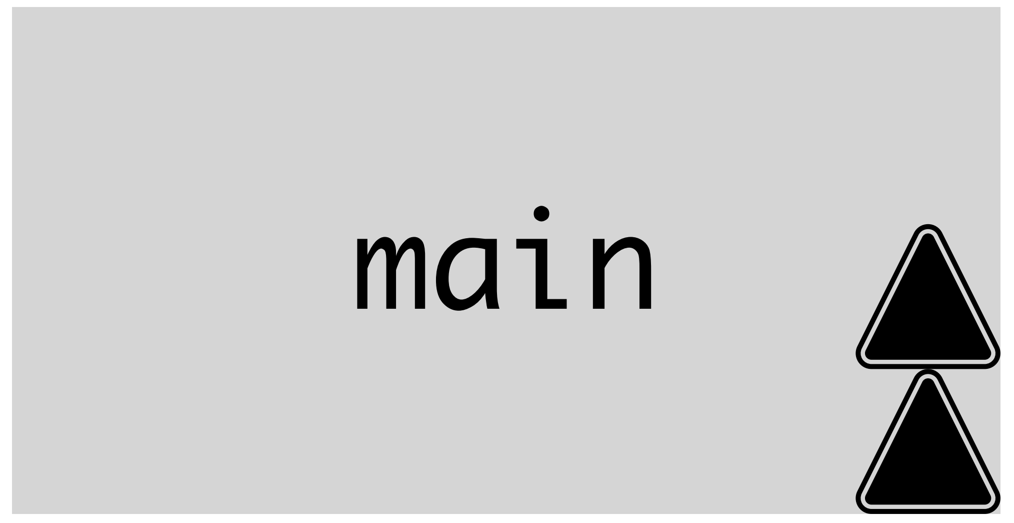
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



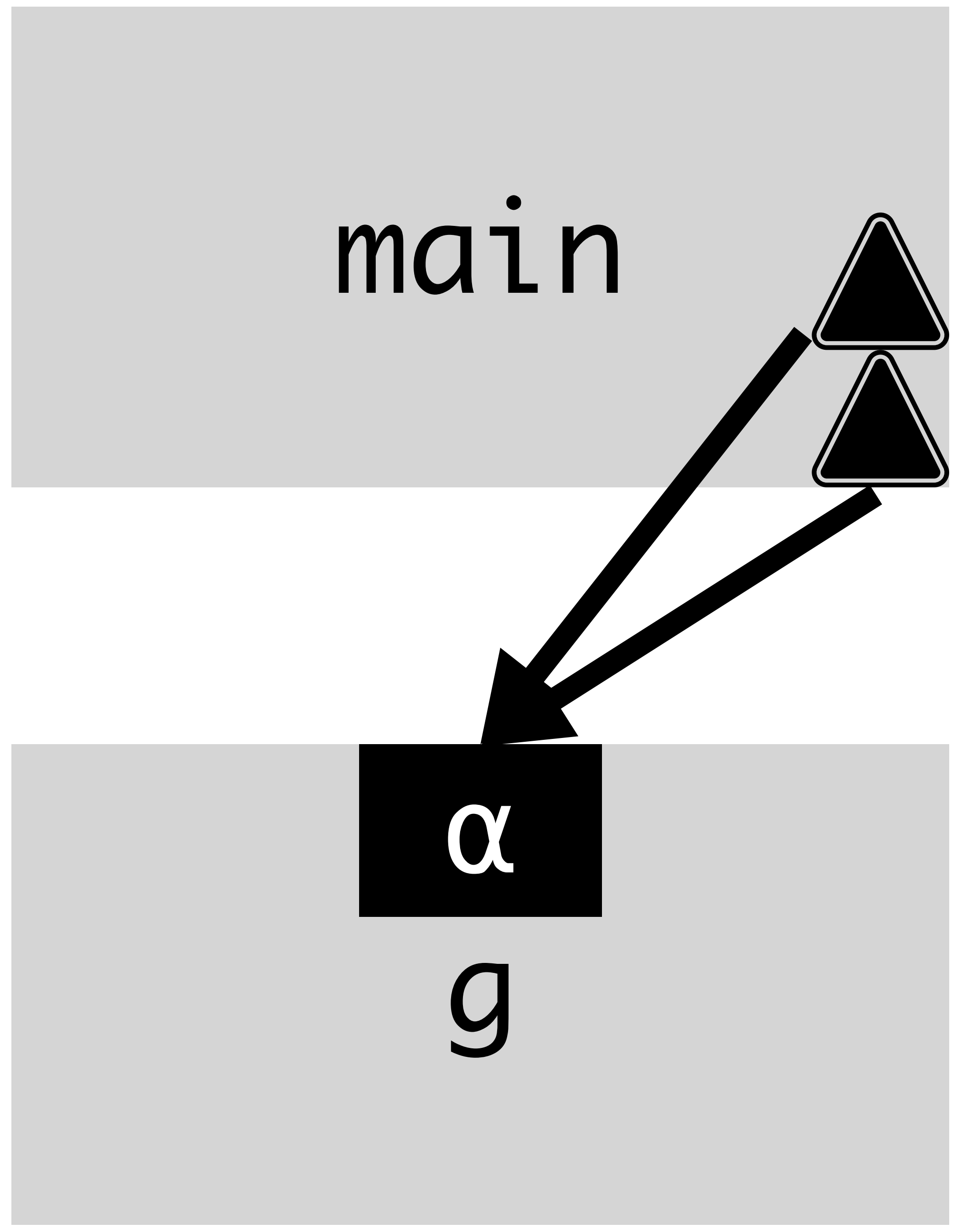
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



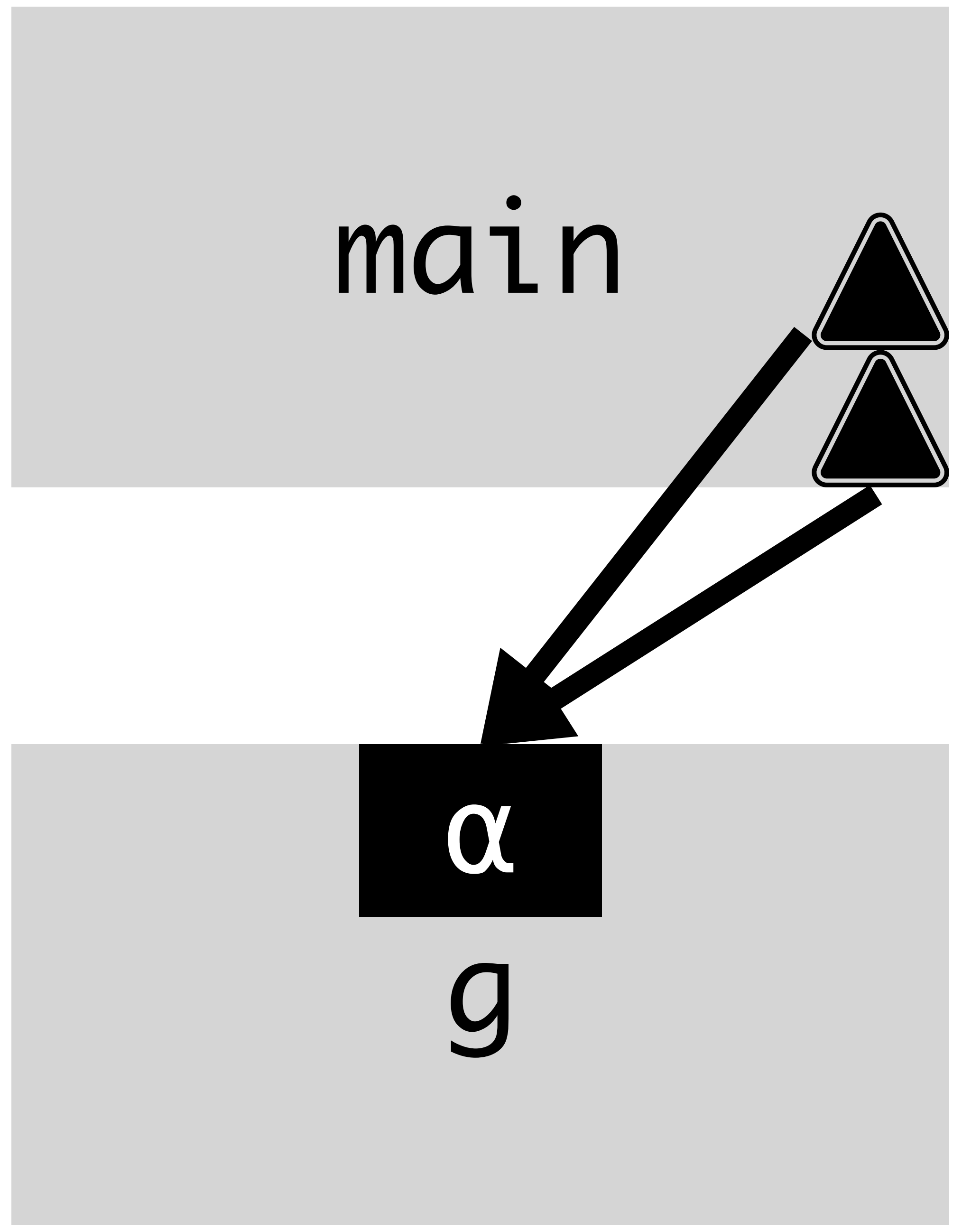
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



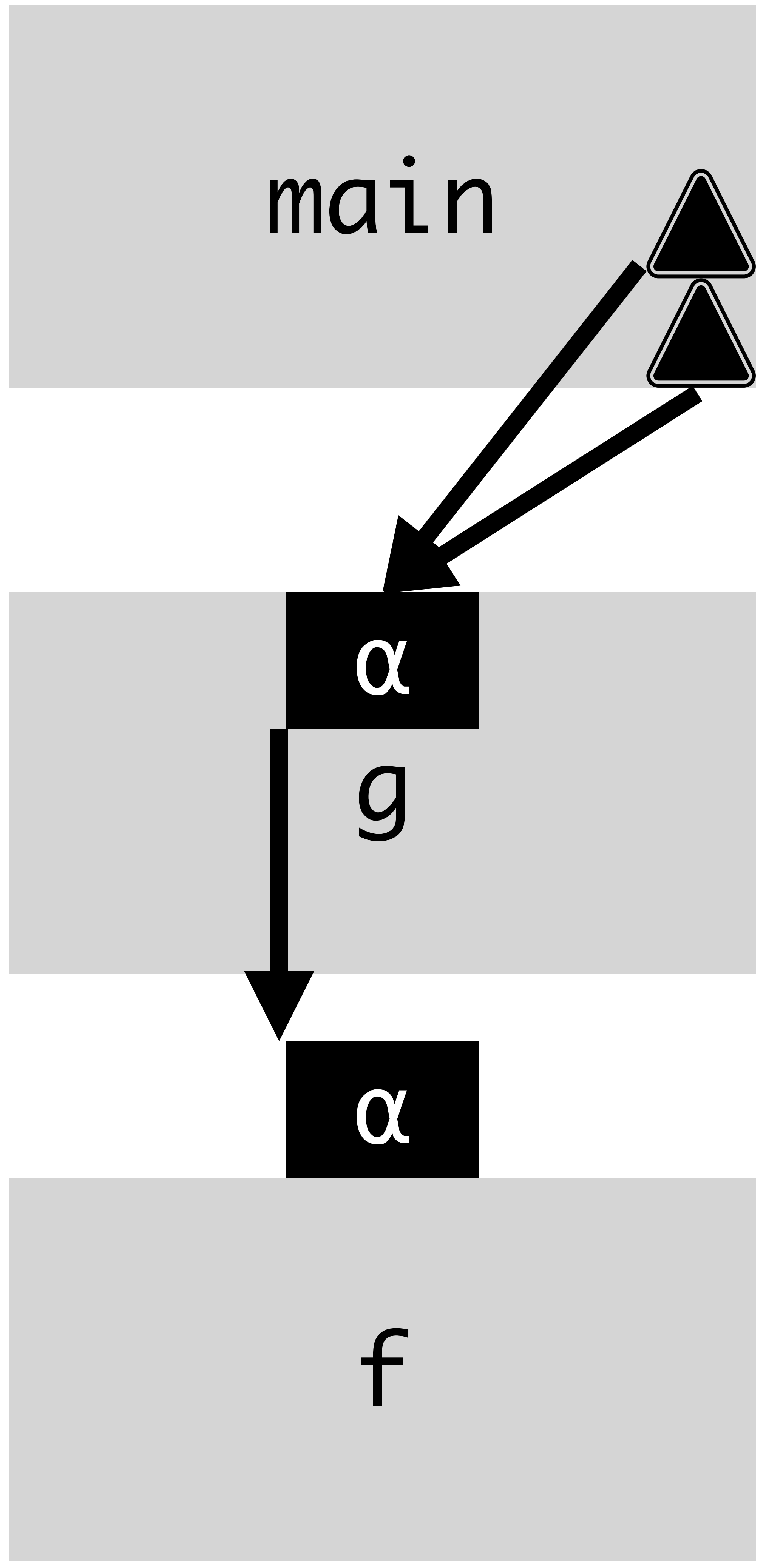
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



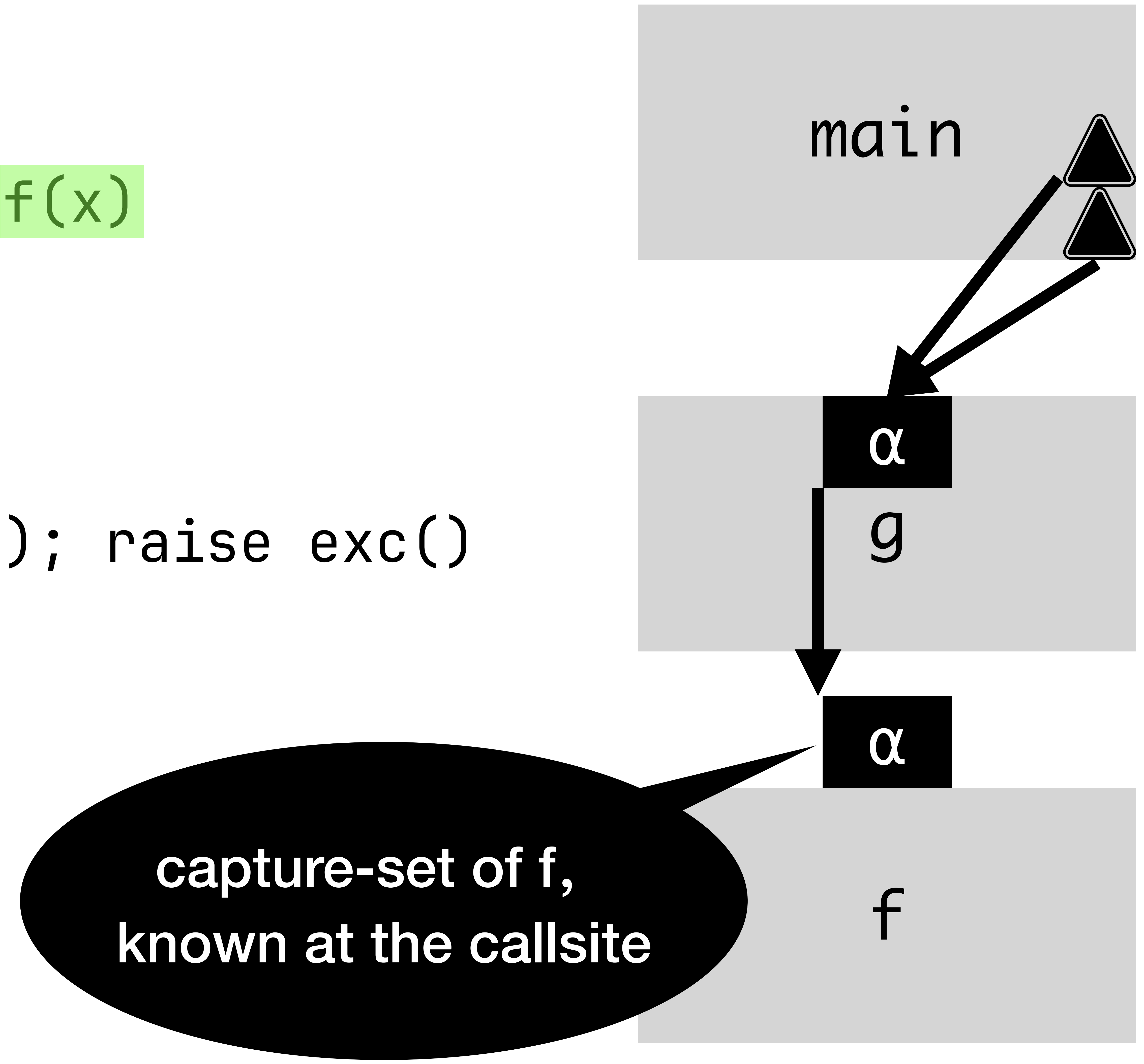
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



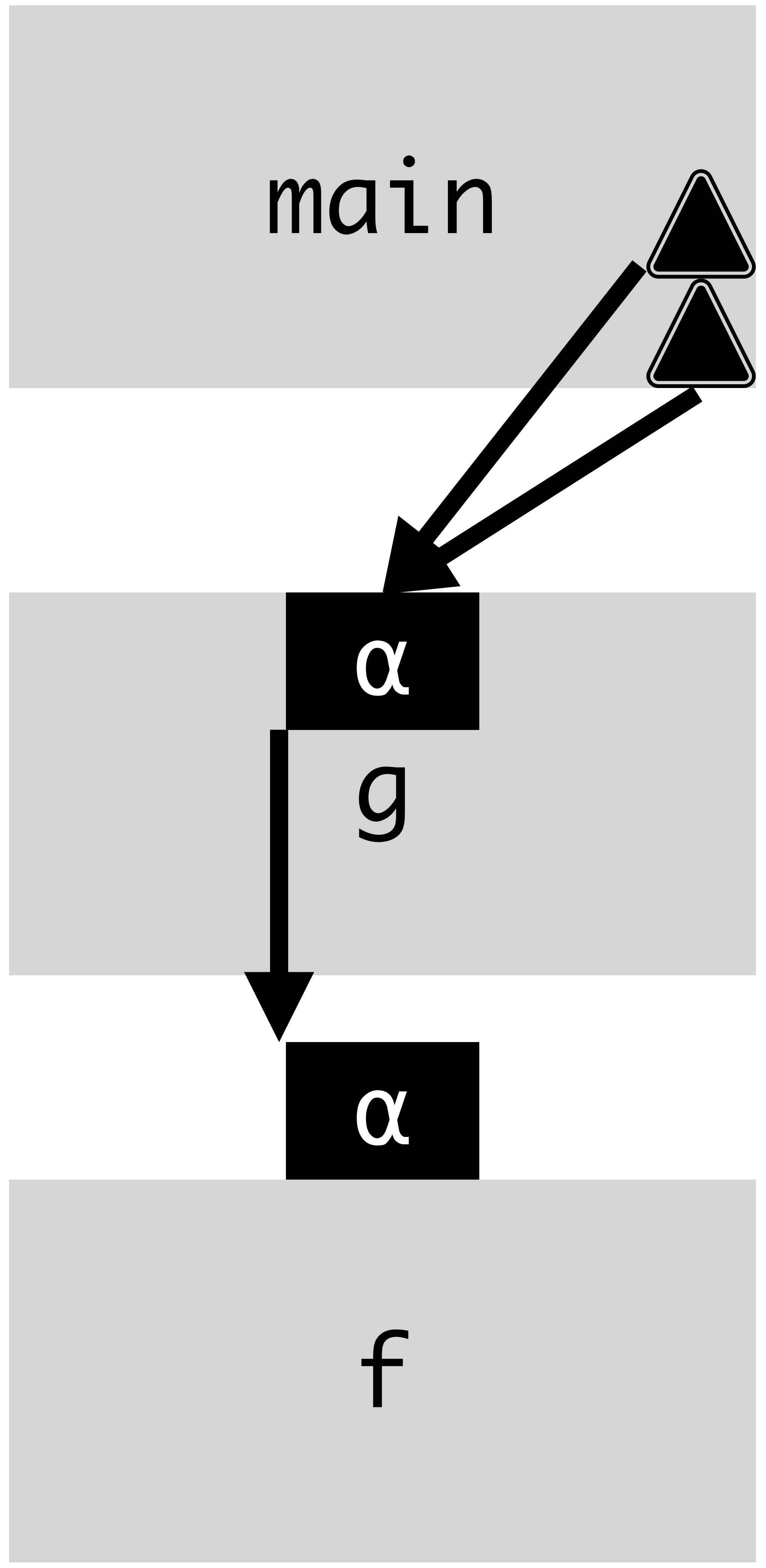
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



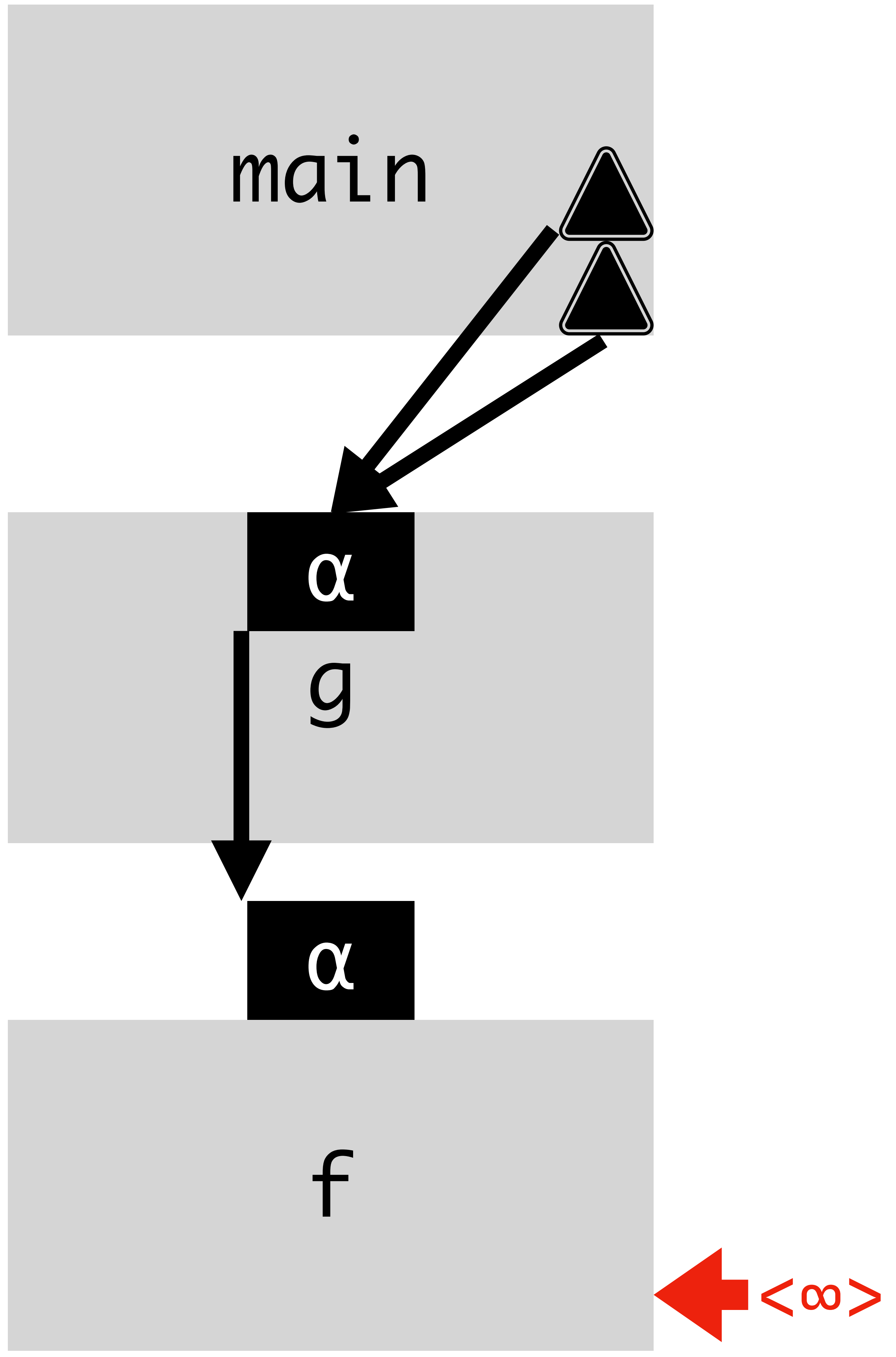
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



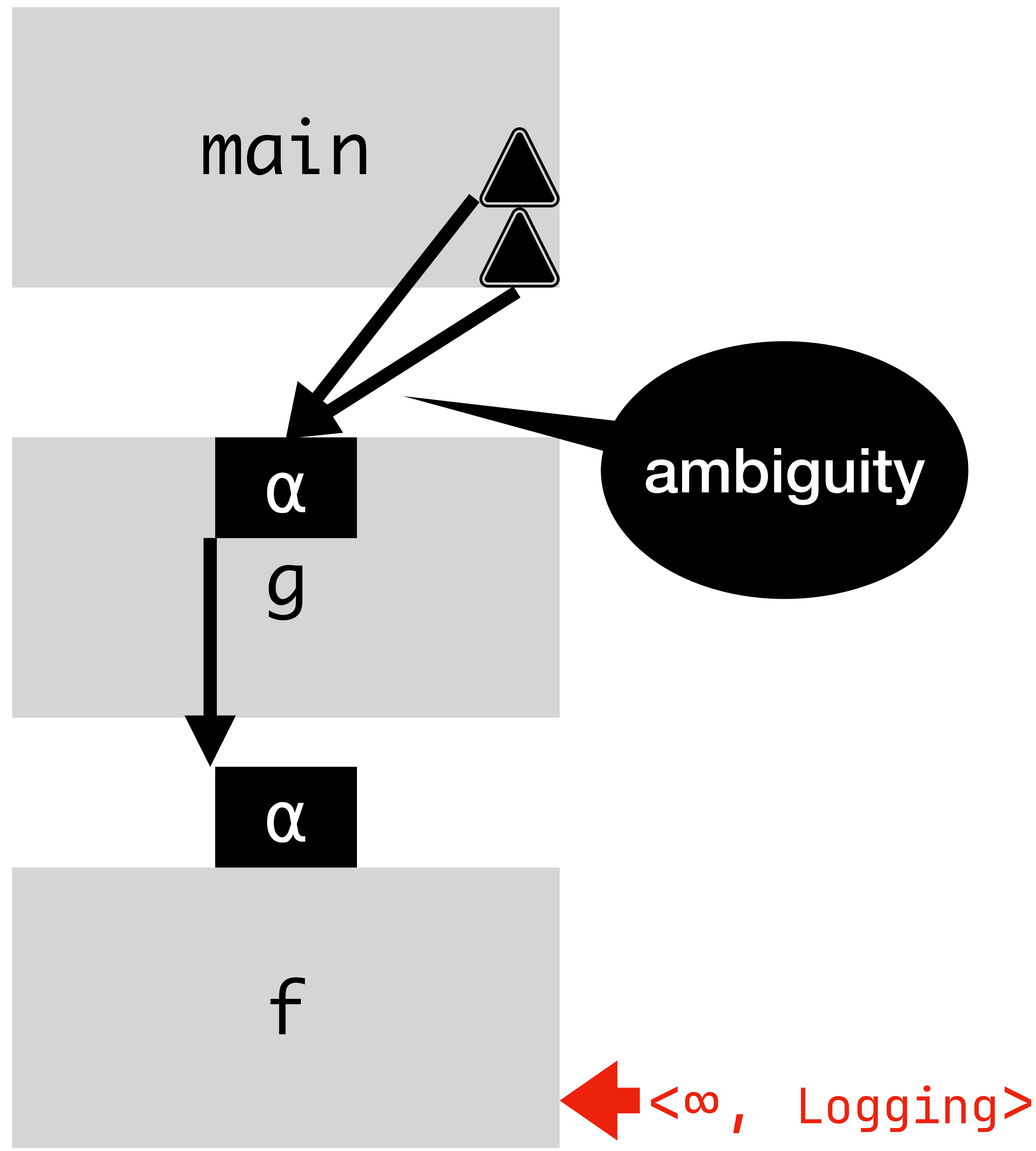
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



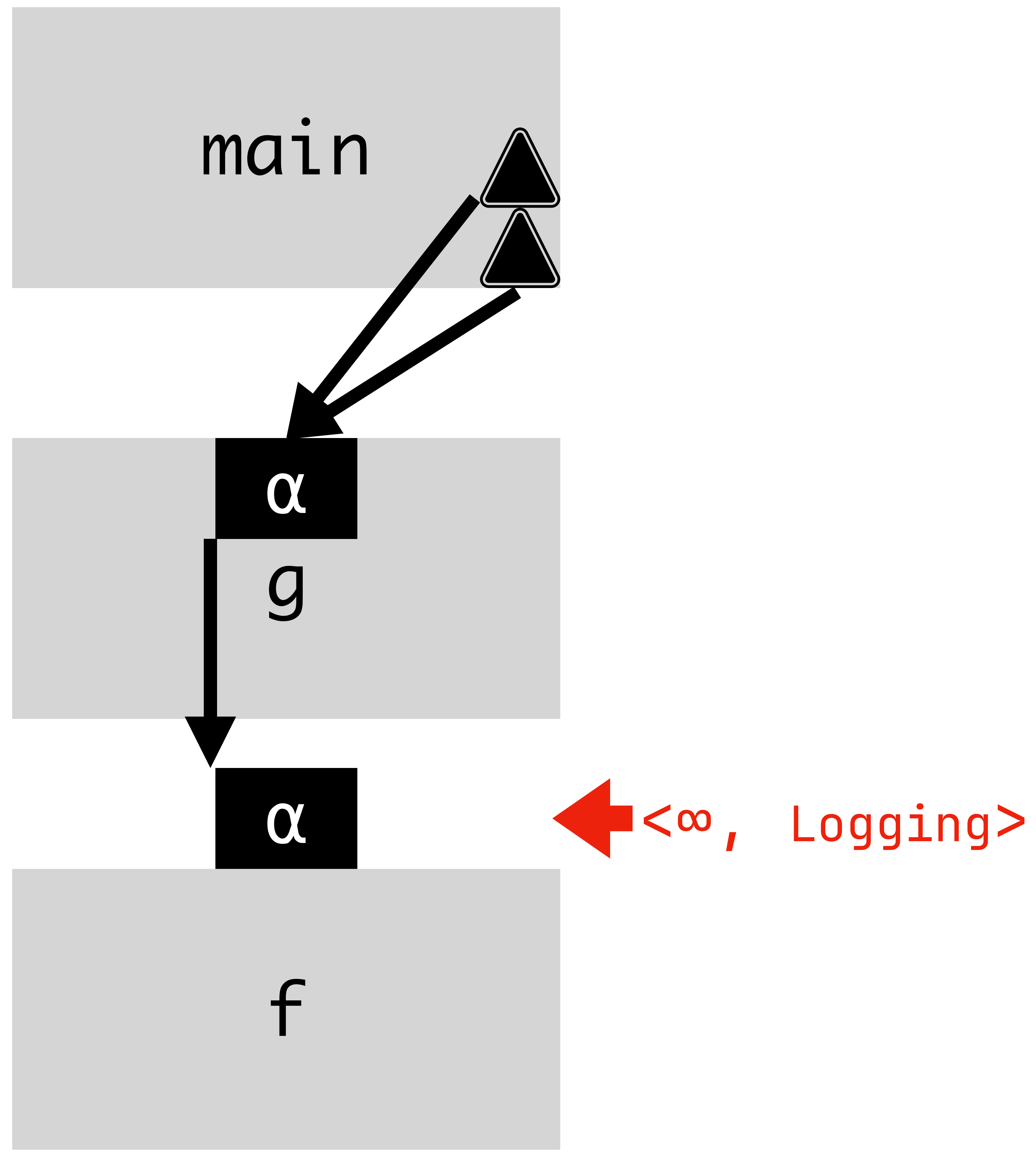
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



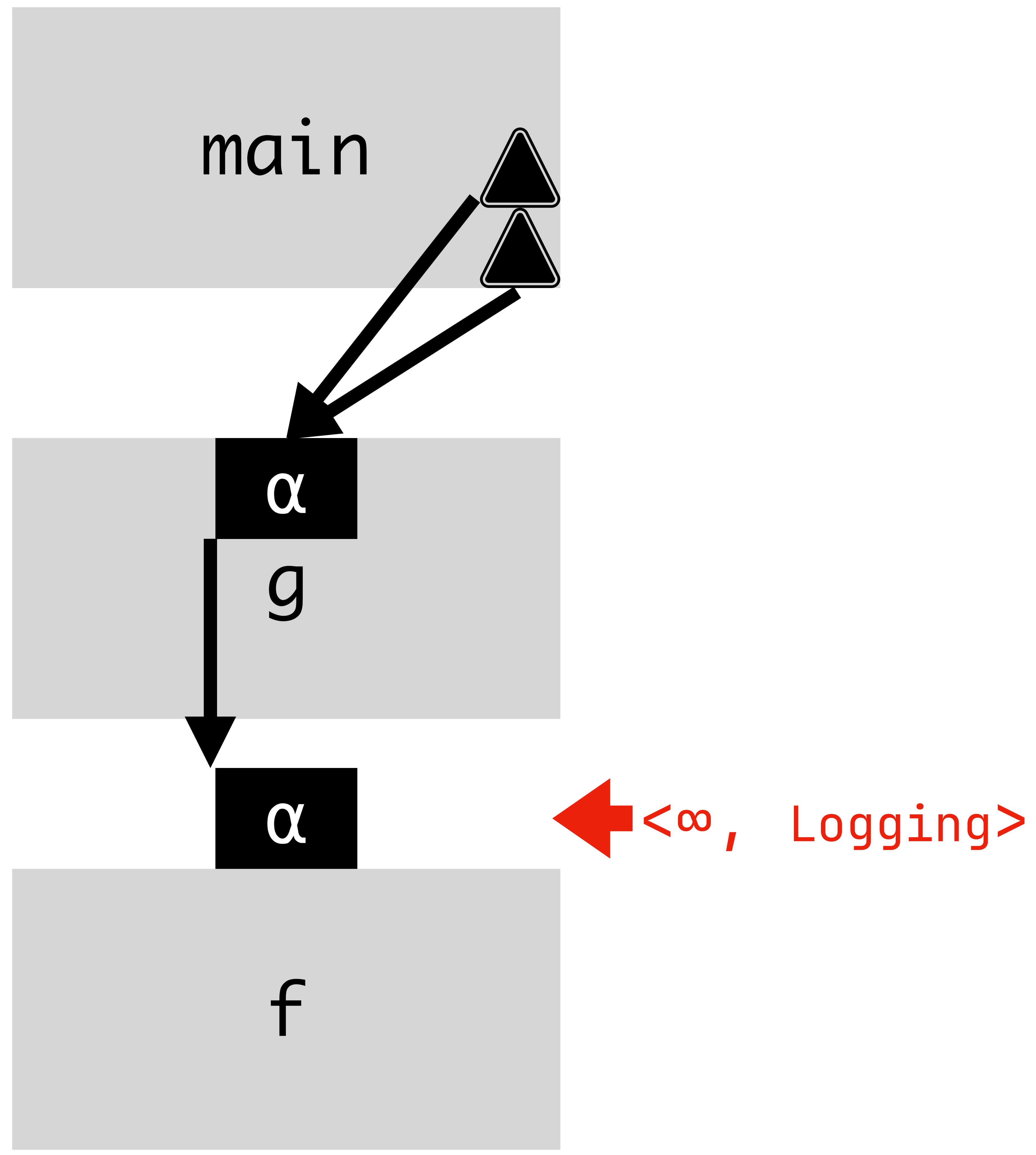
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



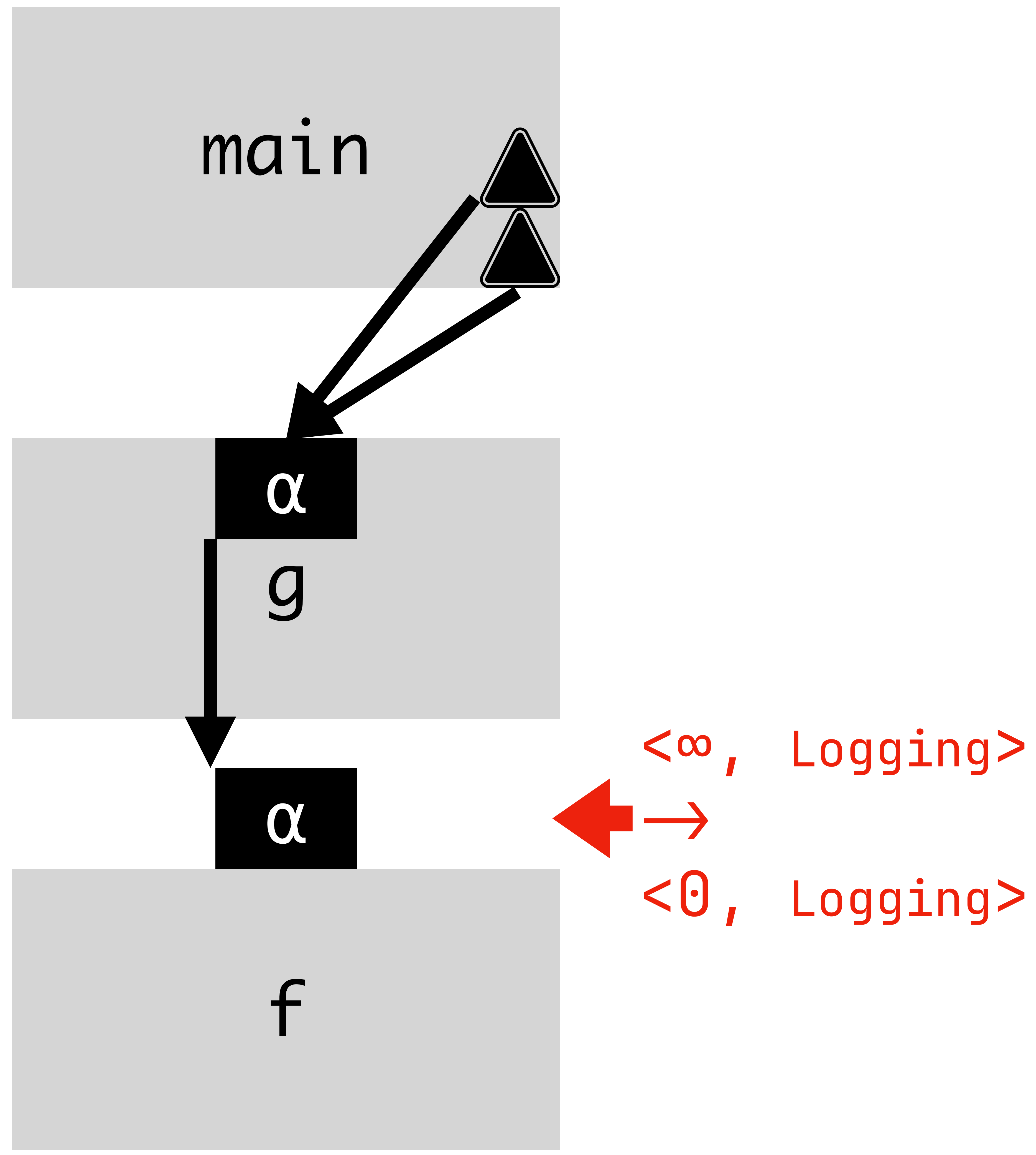
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



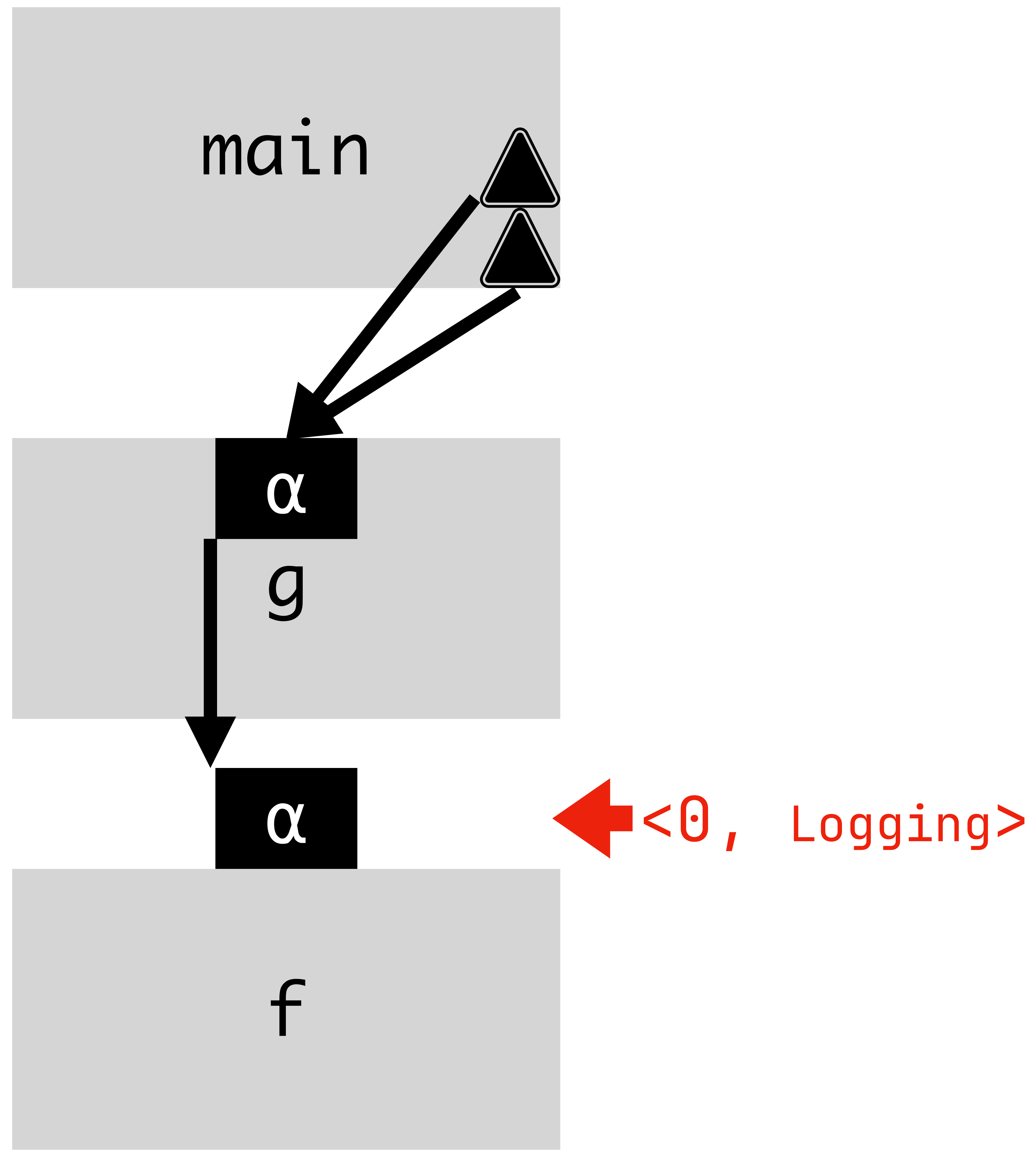
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



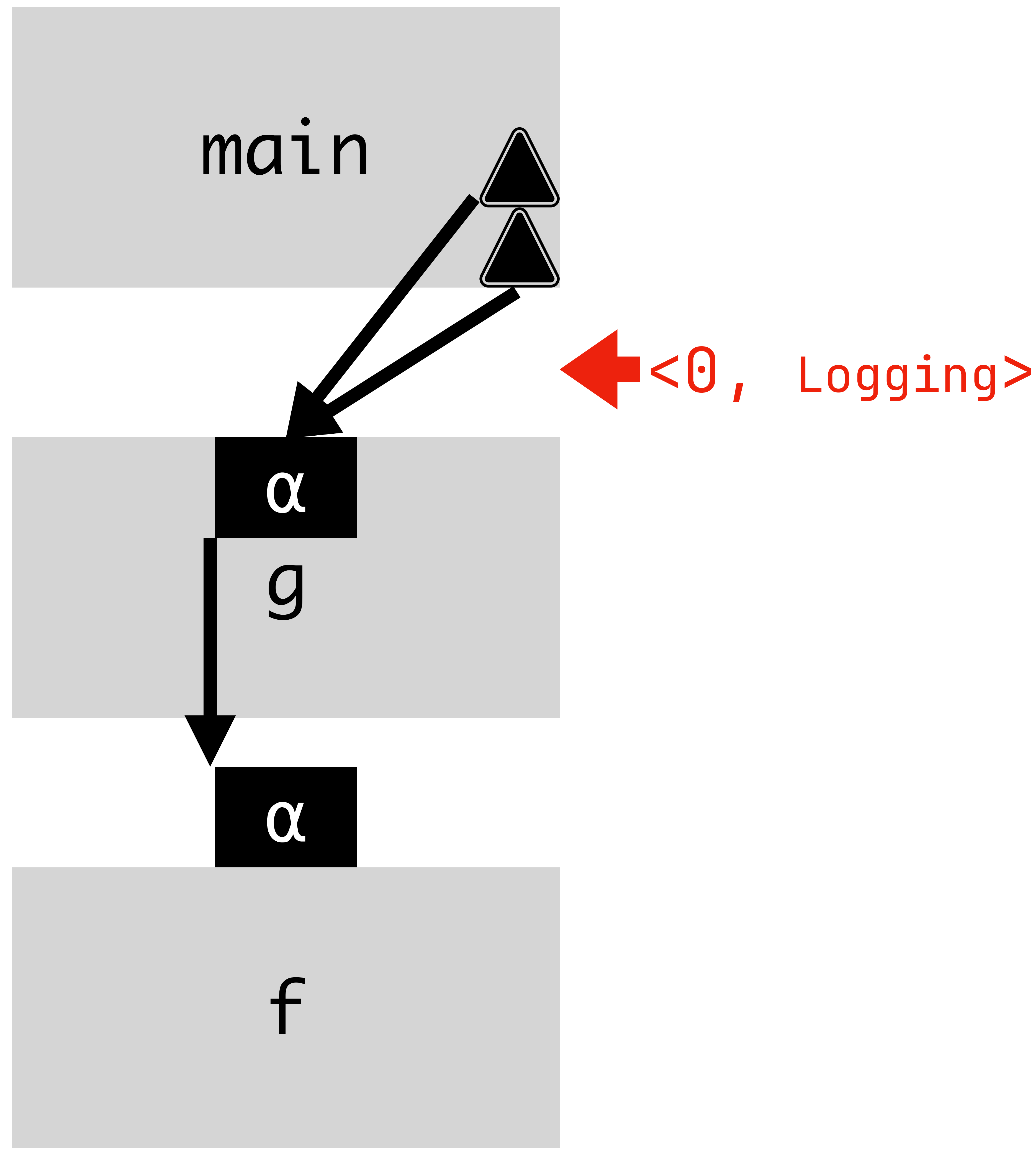
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



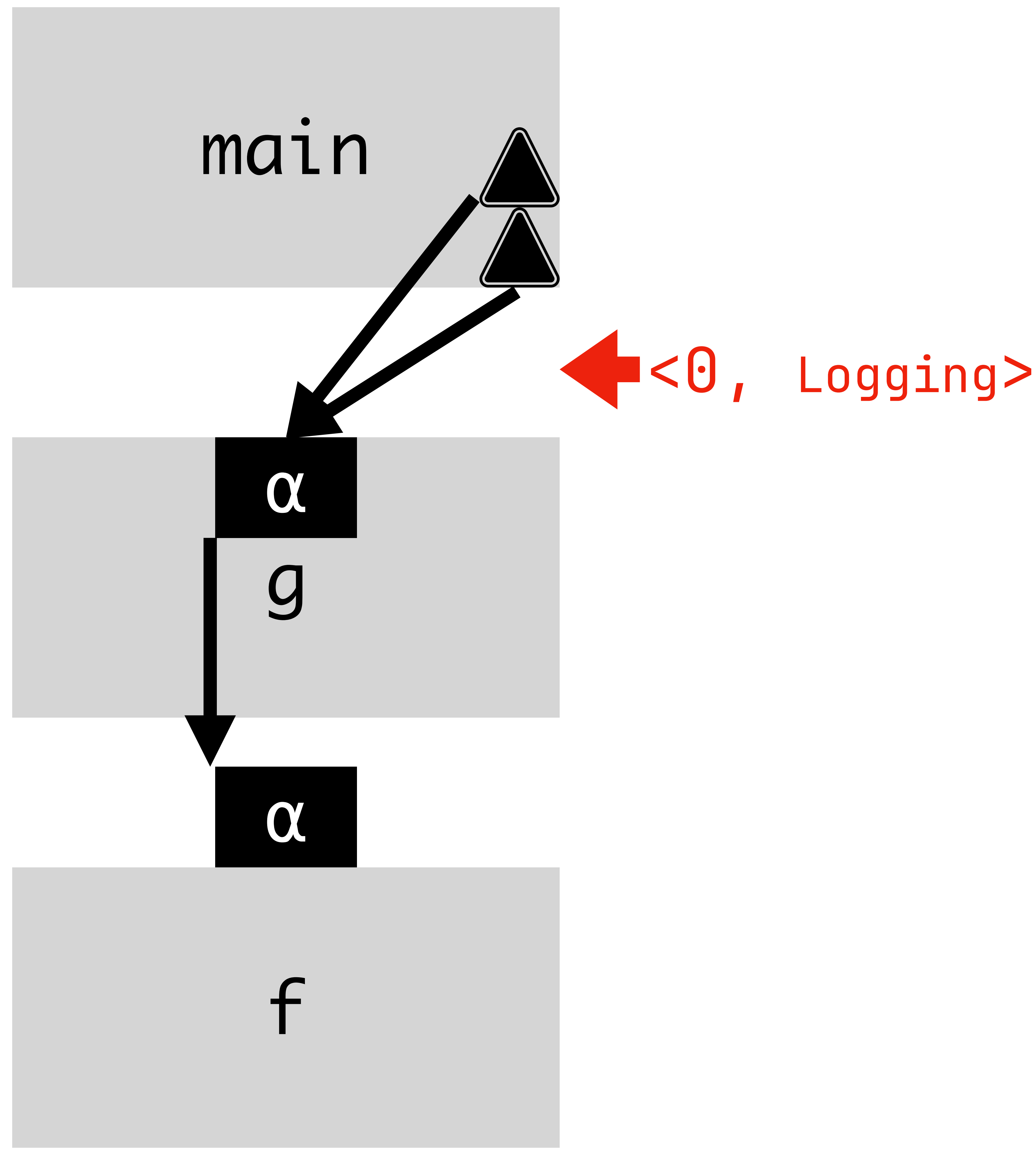
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



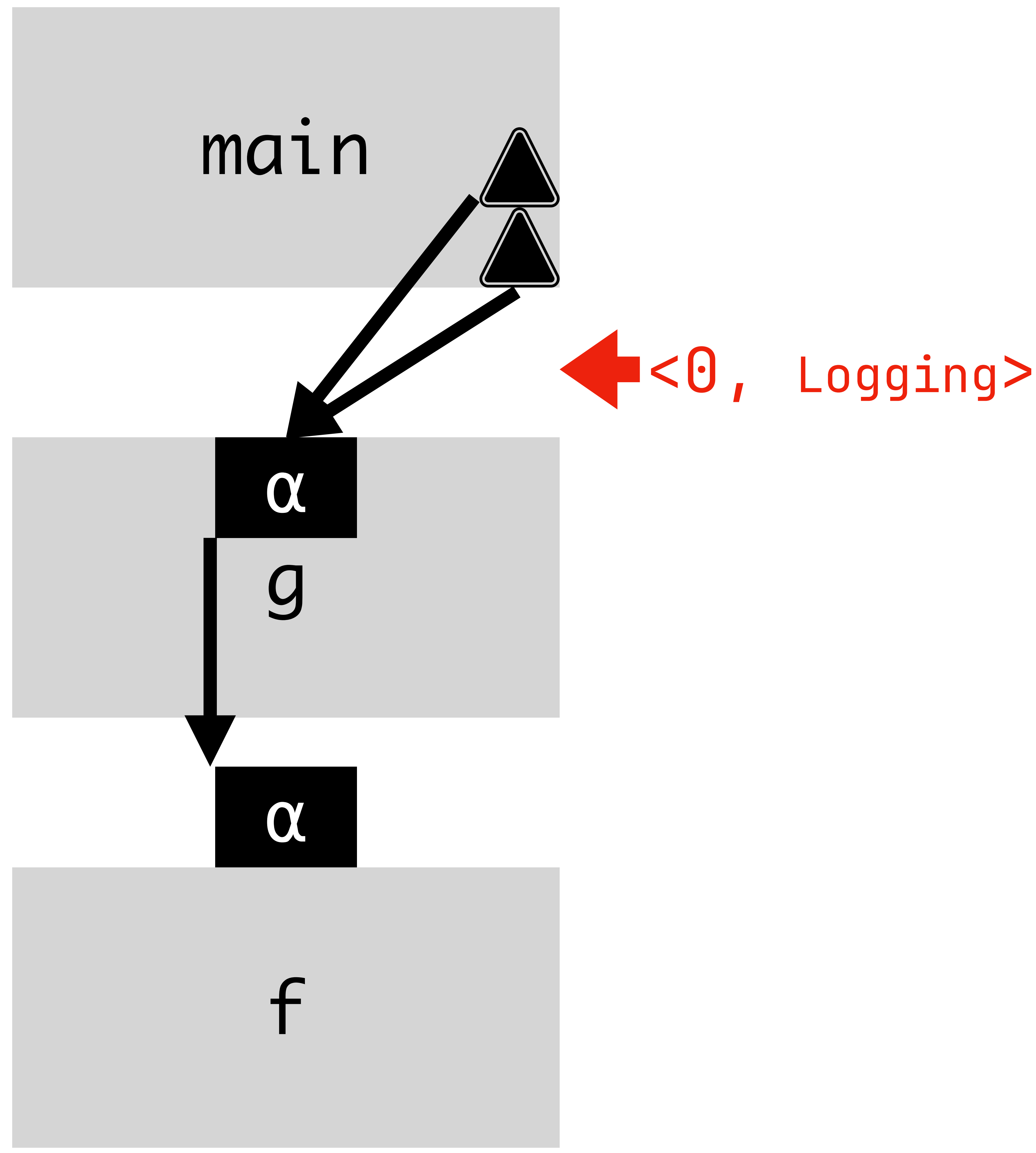
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



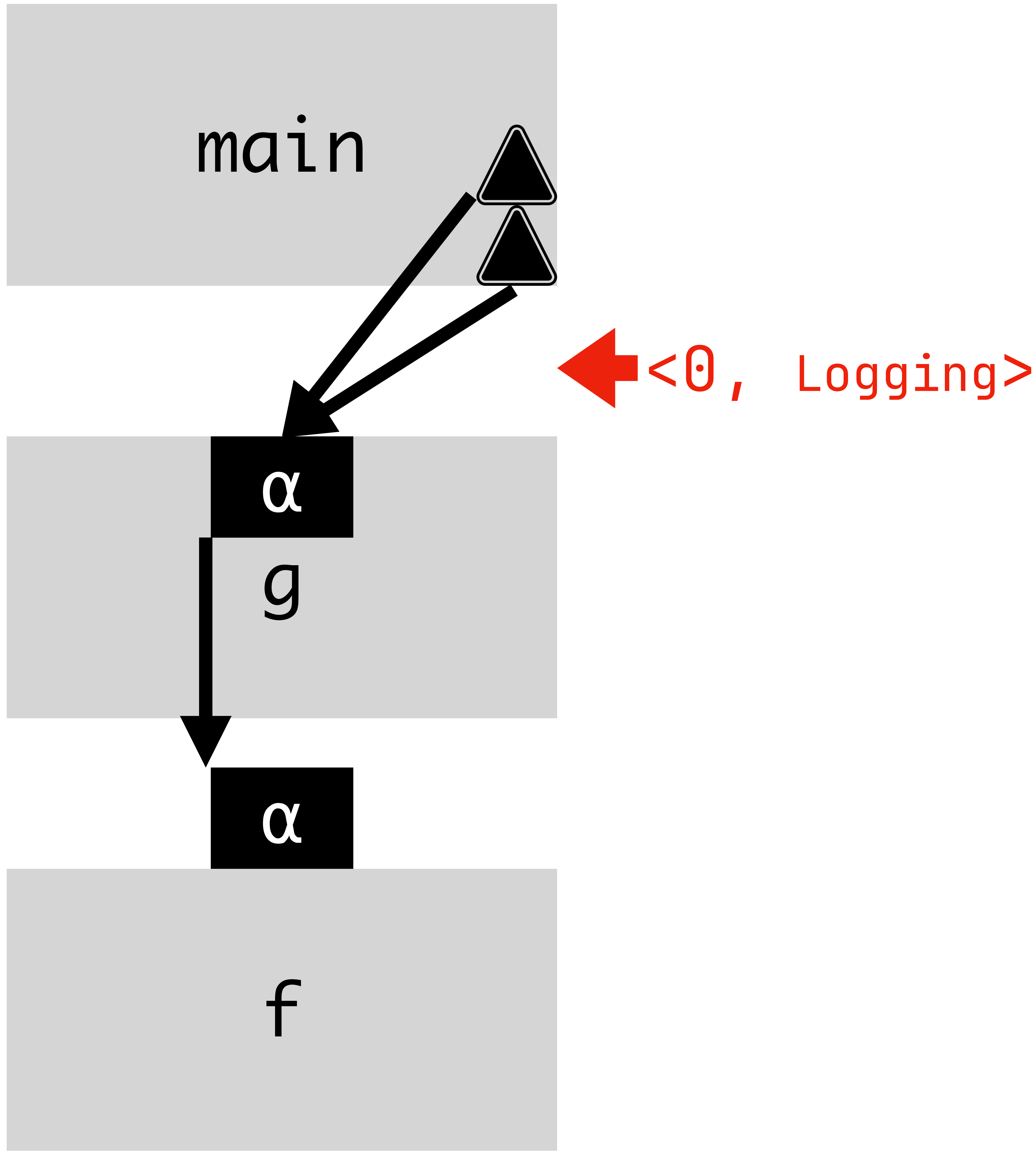
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



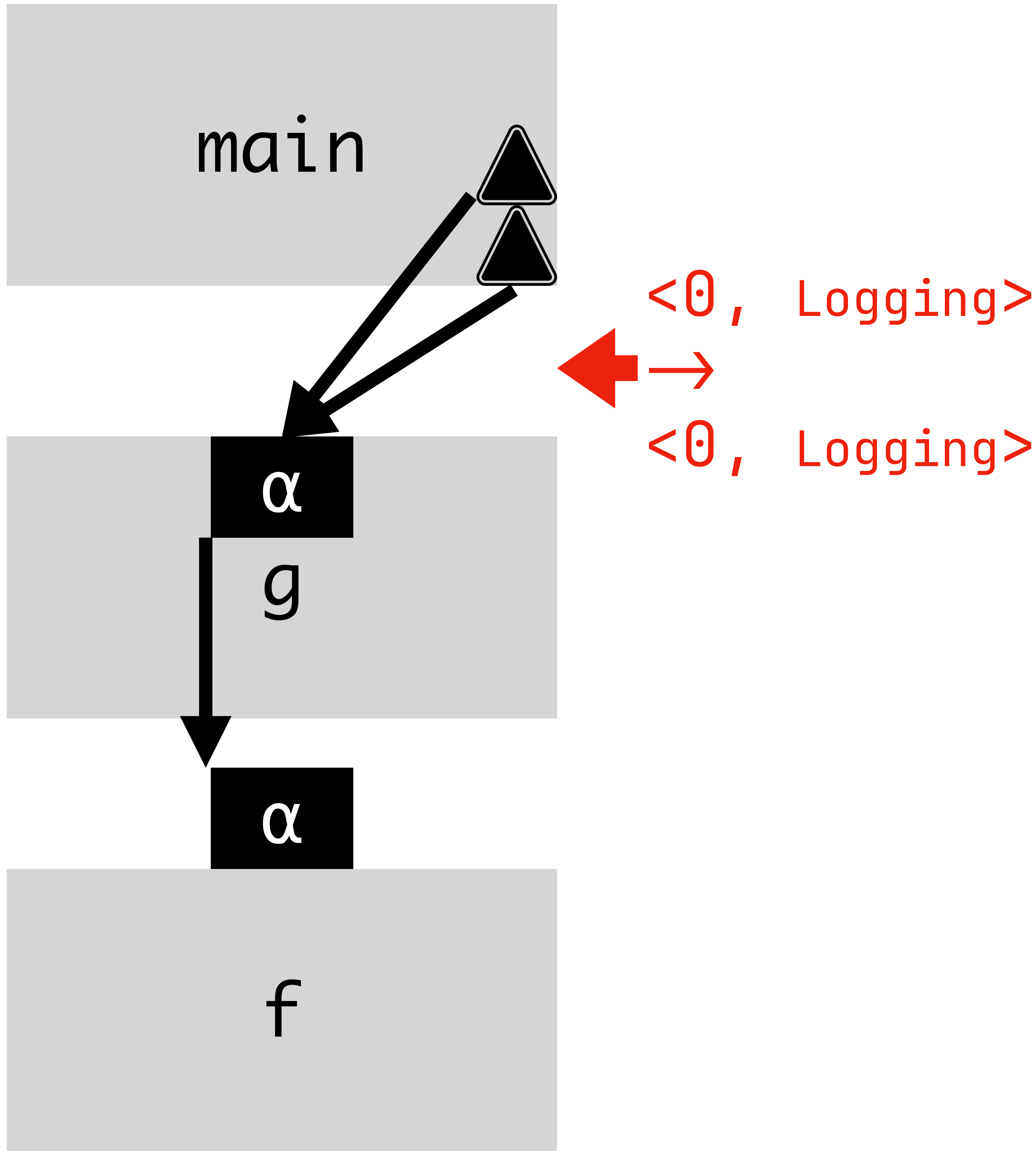
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



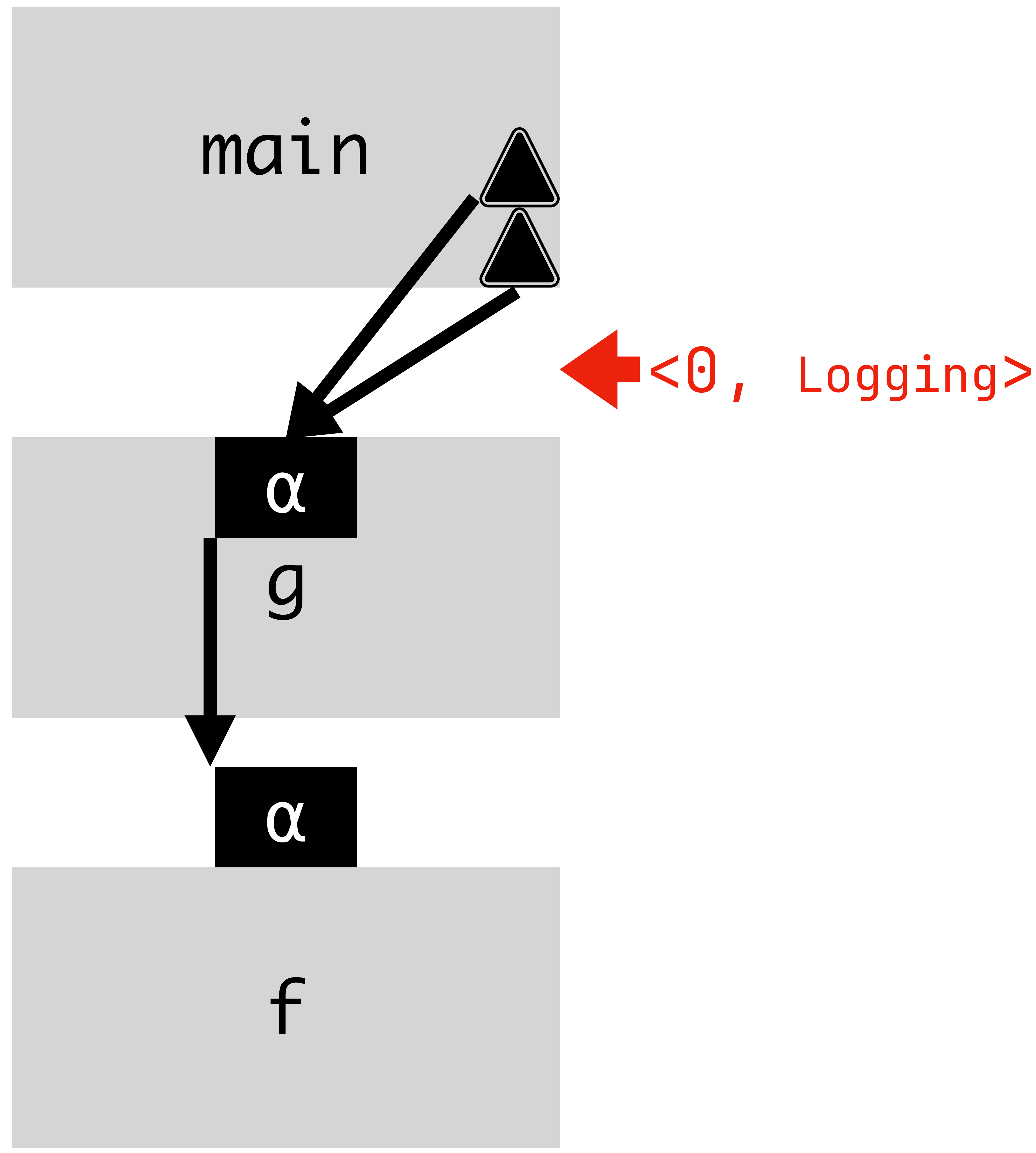
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



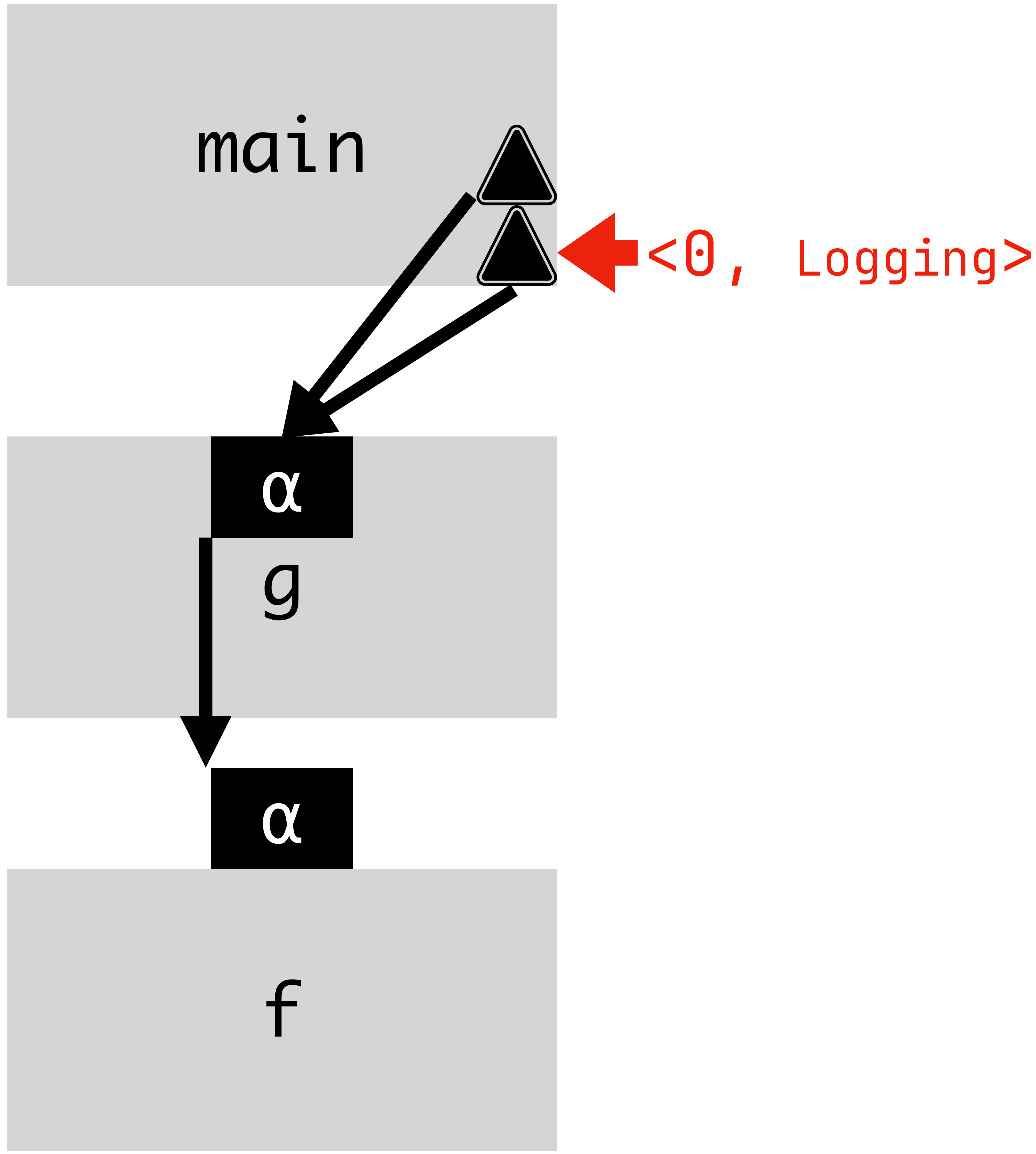
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



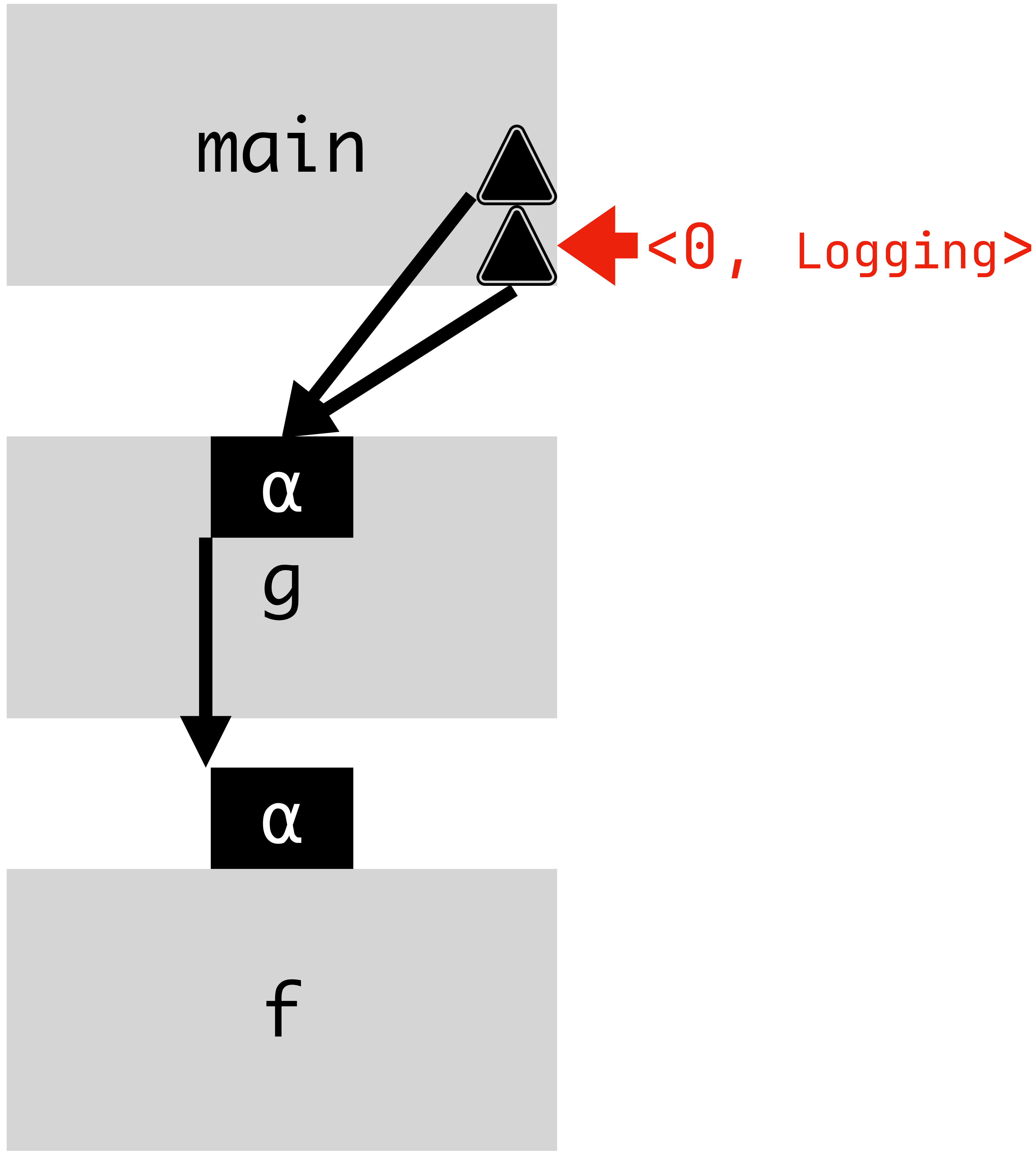
Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N⟶N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



Zero-Cost Lexical Effect Handlers, Example 2

```
#main
let
  g = Λα.λ(x, f: [α]N→N).f(x)
in
handle
  handle
    let
      f = λ(x).raise log(x); raise exc()
    in
      g[log, exc](42, f)
  with log: Logging = …
with exc: Exception = ...
```



Implementation: Lexa

Source Language

```
handle
  g(log)
with log:Logging =
   ...
```

Implementation: Lexa

Source Language → Target Language

```
handle
  g(log)
with log:Logging =
  ...
```

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

callsite metadata

Implementation: Lexa

$$\begin{tikzcd}\text{Source Language} \arrow[r] & \text{Target Language} \arrow[r] & \text{Binary}\end{tikzcd}$$

```
handle
  g(log)
with log:Logging =
  ...
```

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

```
Code Segment:

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

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

Implementation: Lexa

$$\begin{tikzcd}\text{Source Language} \arrow[r] & \text{Target Language} \arrow[r] & \text{Binary}\end{tikzcd}$$

```
handle
  g(log)
with log:Logging =
  ...
```

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

```
Code Segment:

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

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

Implementation: Lexa

$$\text{Source Language} \longrightarrow \text{Target Language} \longrightarrow \text{Binary}$$

Source Language

```
handle
  g(log)
with log:Logging =
  ...
```

Target Language

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

Binary

```
Code Segment:

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

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

We formally defined the source and targe languages and proved that the compilation is semantic-preserving.

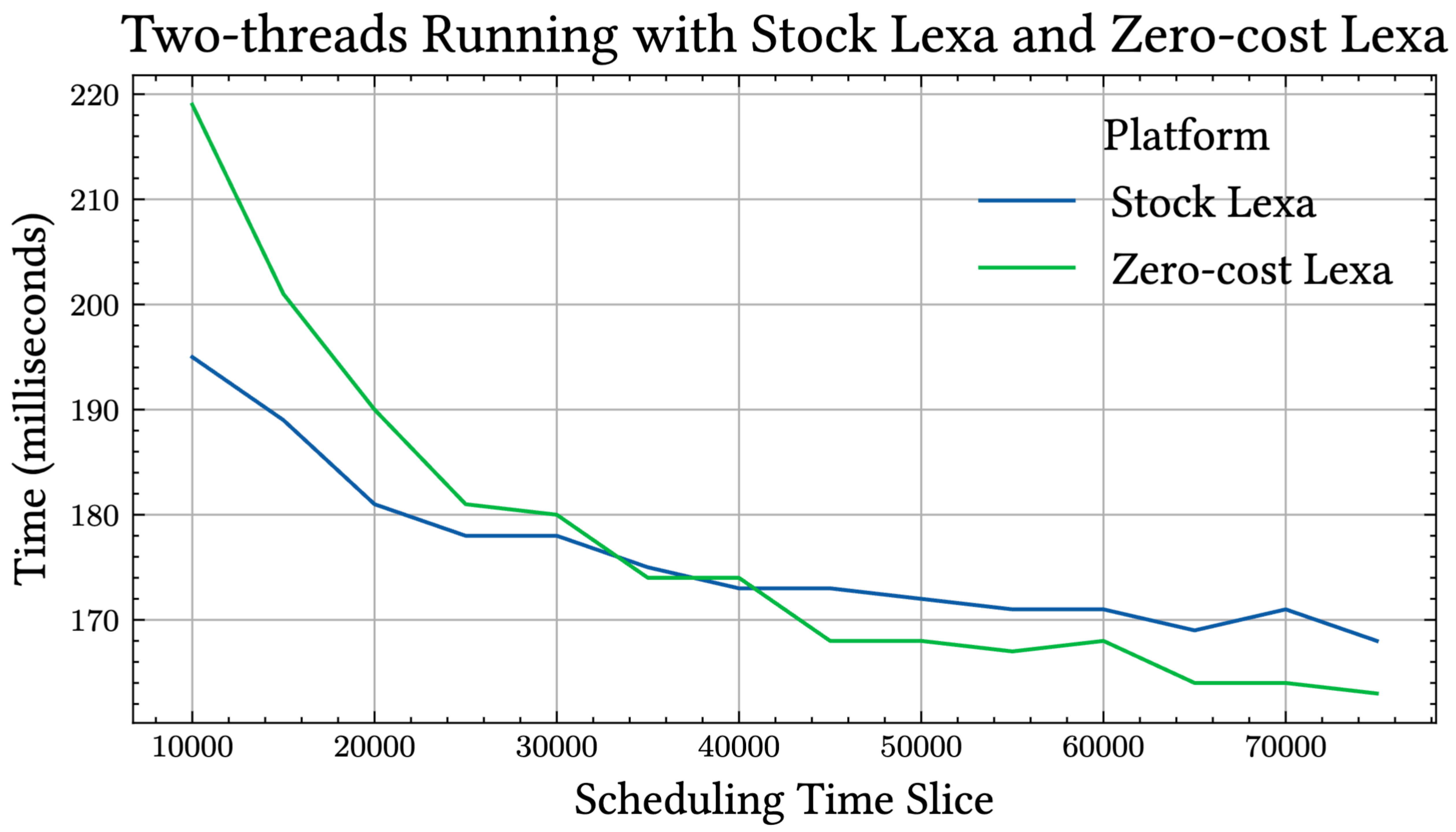
Evaluation

Hypothesis: effect handlers that are rarely used benefit from zero-cost implementation.

Evaluation

A program with two coorperativley scheduled co-routine.

A larger value on x-axis means less frequent yielding, so zero-cost strategy is more efficient.



Lexical Effect Handler

Enjoys modularity, but incurs overhead even on infrequently-used effects.

$\xrightarrow{\text{this work}}$

Enjoys modularity, and obey zero-cost principle.

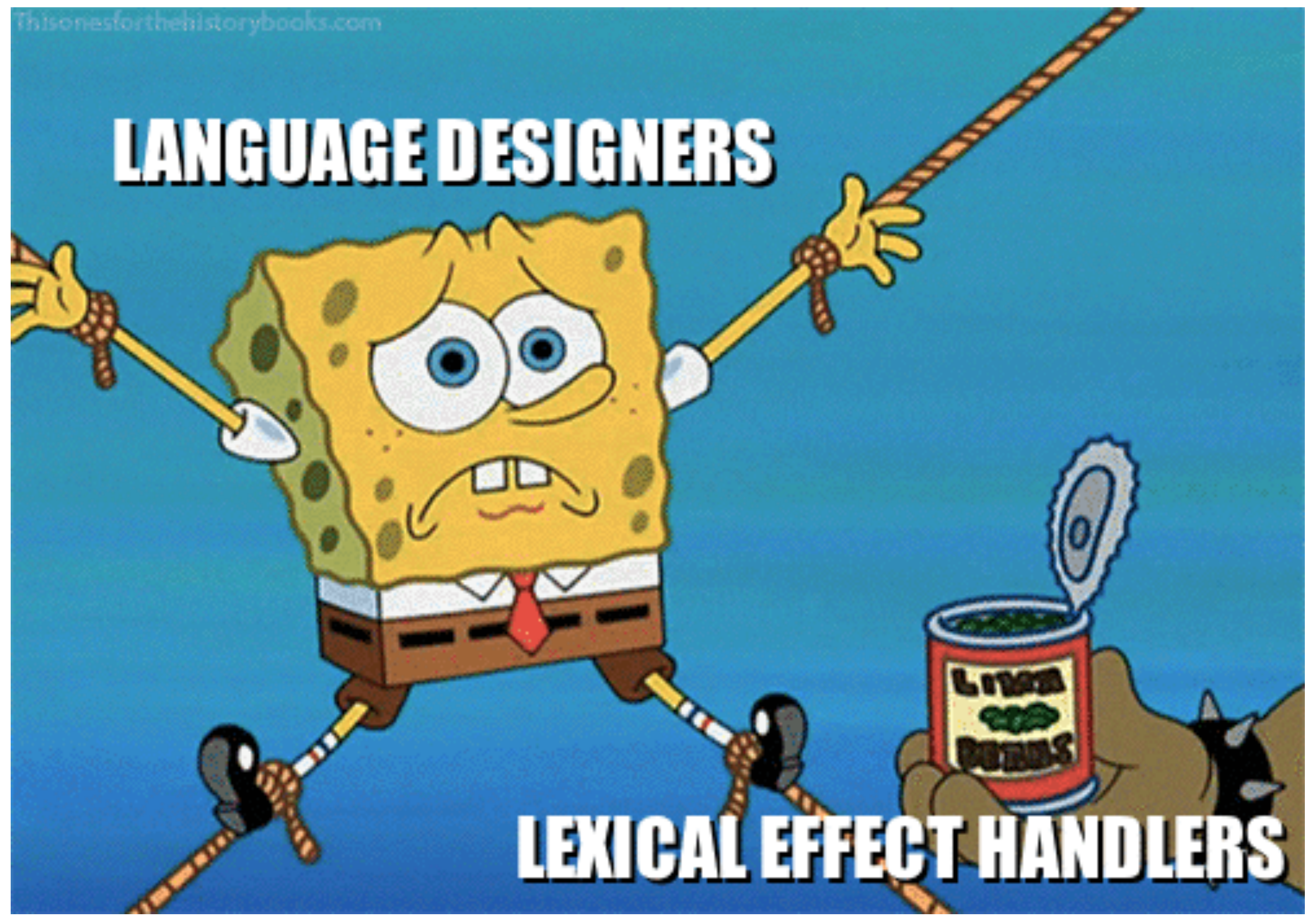
Lexical Effect Handler

Enjoys modularity, but incurs overhead even on infrequently-used effects.

this work

$\longrightarrow$

Enjoys modularity, and obey zero-cost principle.



our hope

$\longrightarrow$