An Introduction to **Algebraic Effects** and **Handlers**

Matija Pretnar, University of Ljubljana, Slovenia

What are

algebraic effects?

effects arise from

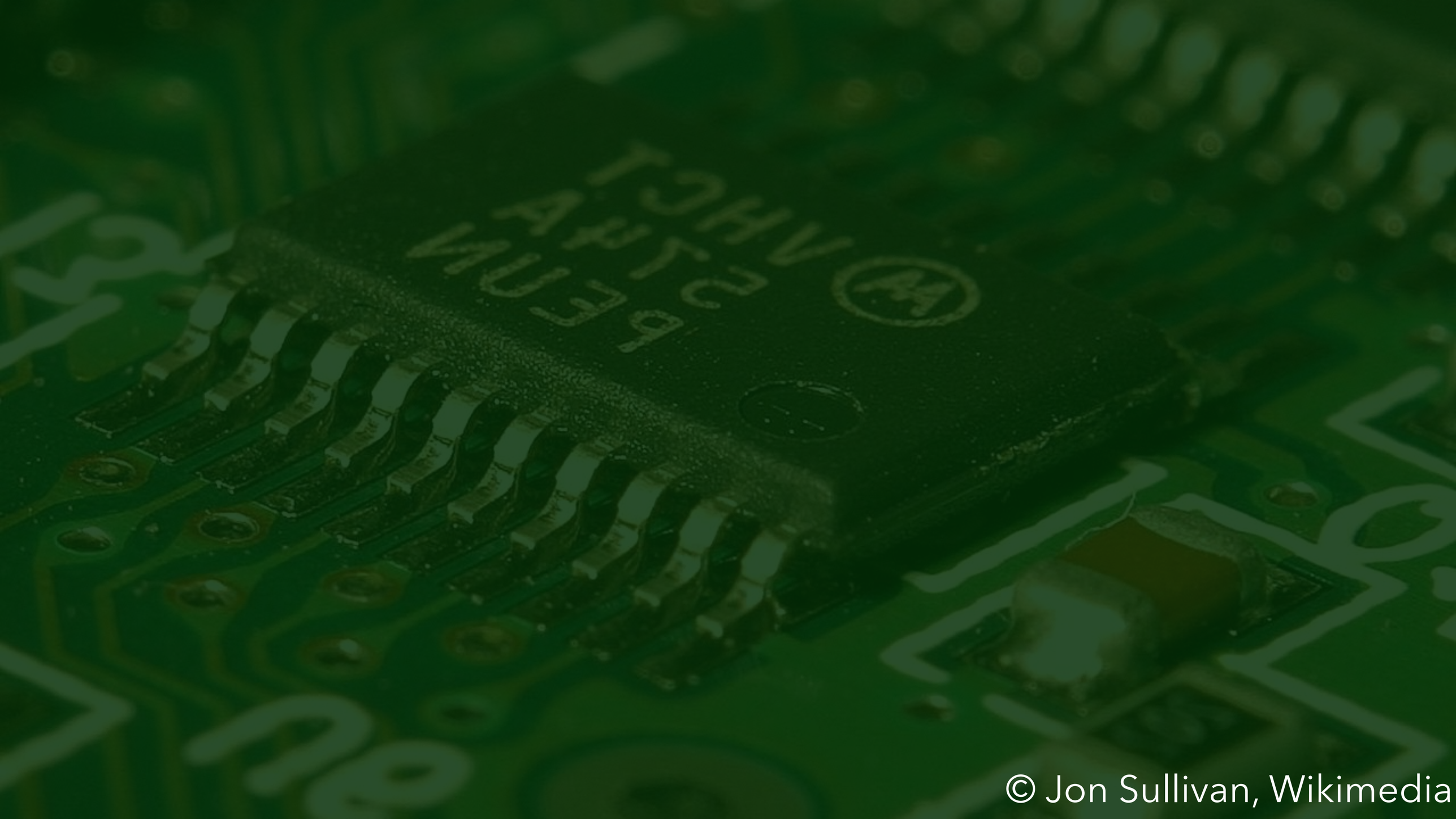
operations

&

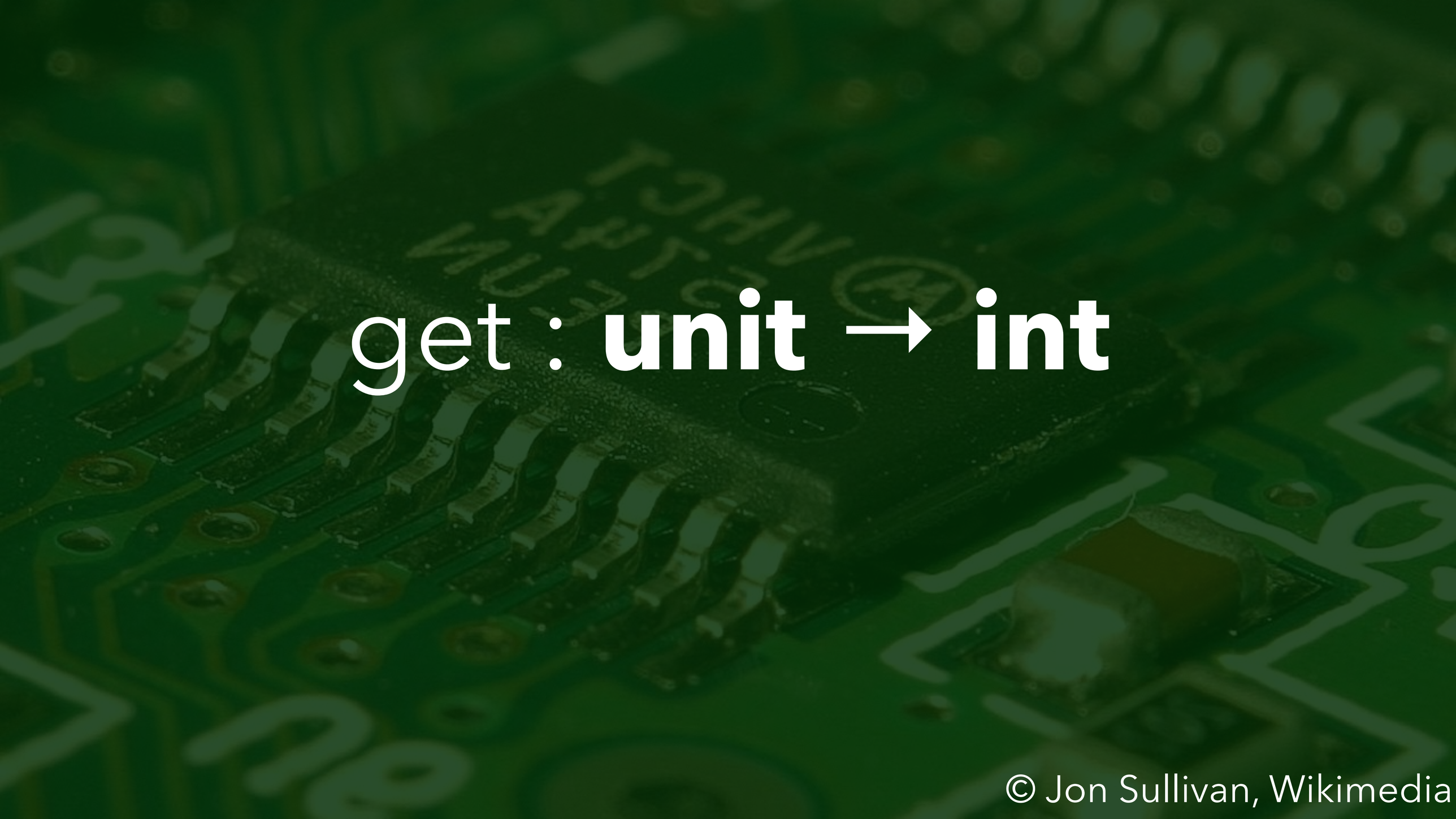
equations

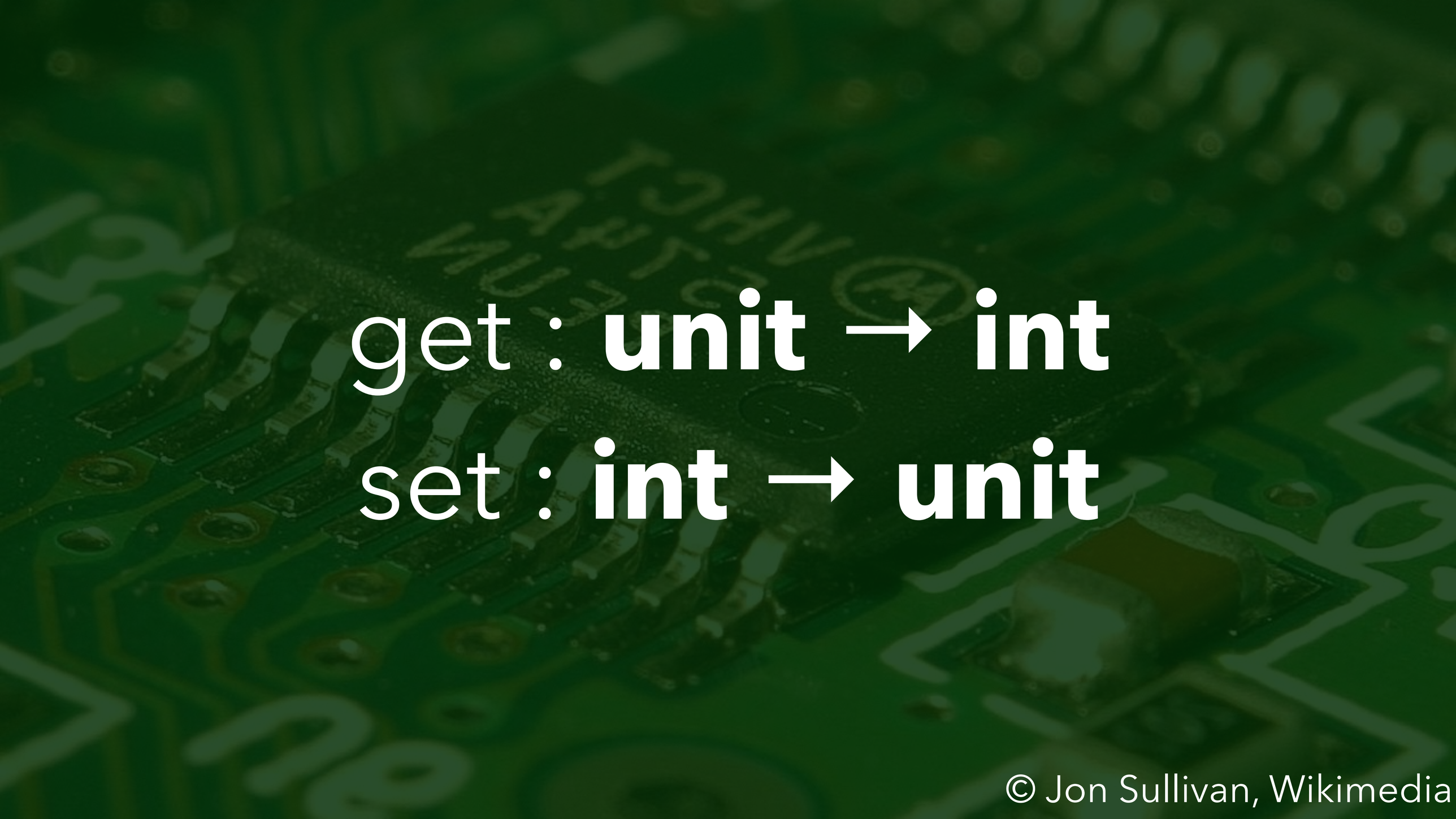
effects arise from

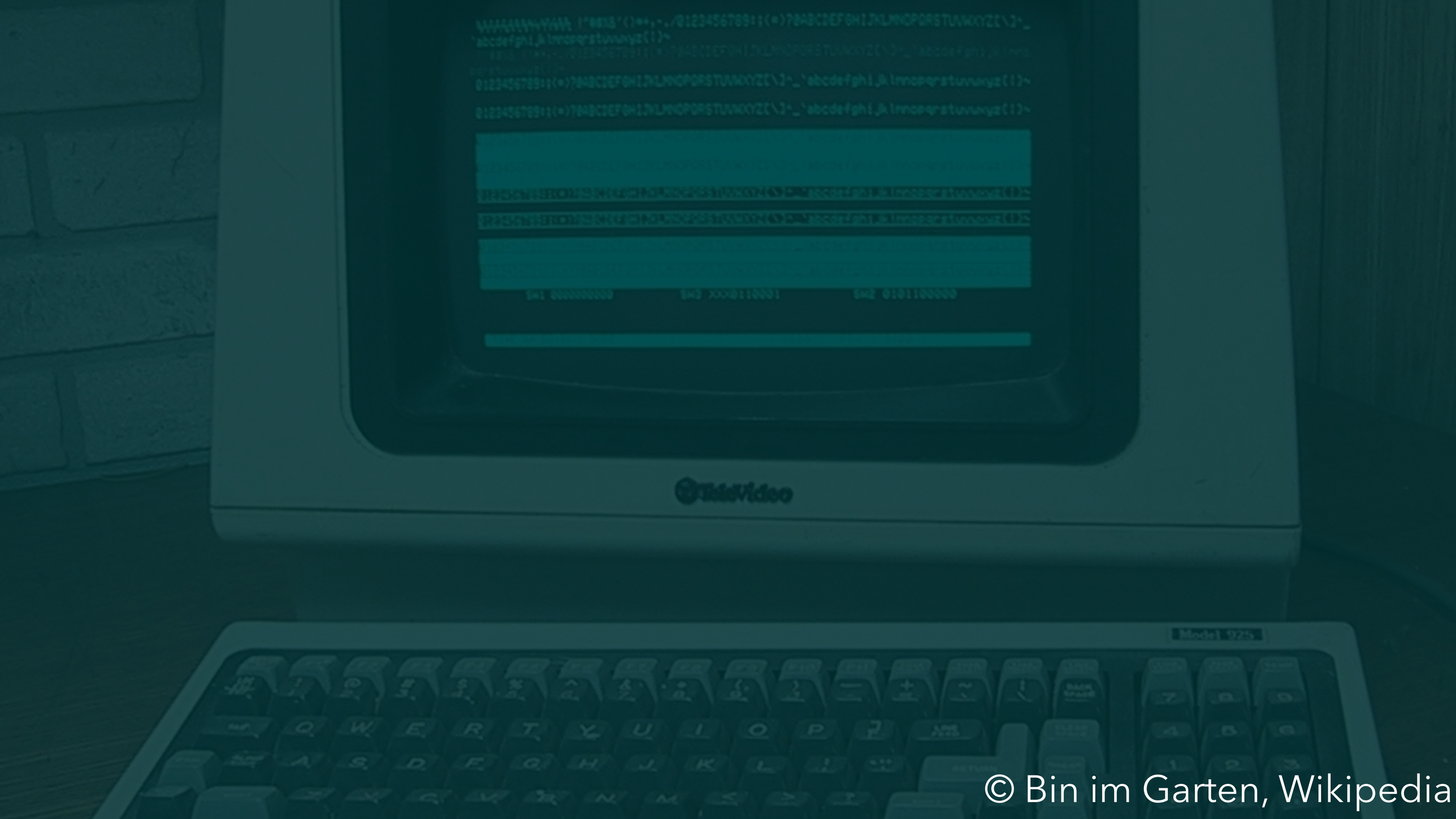
operations



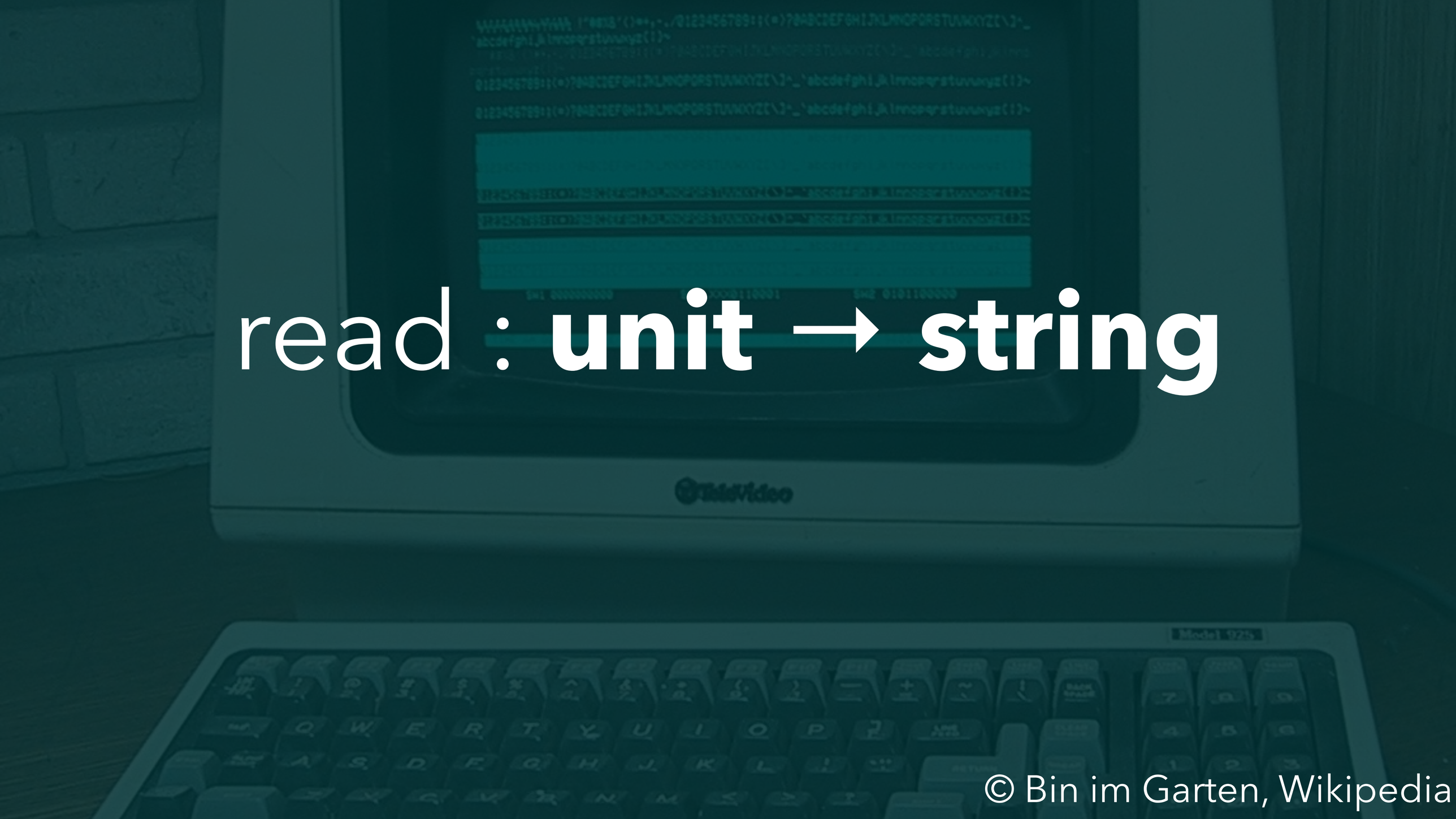
© Jon Sullivan, Wikimedia



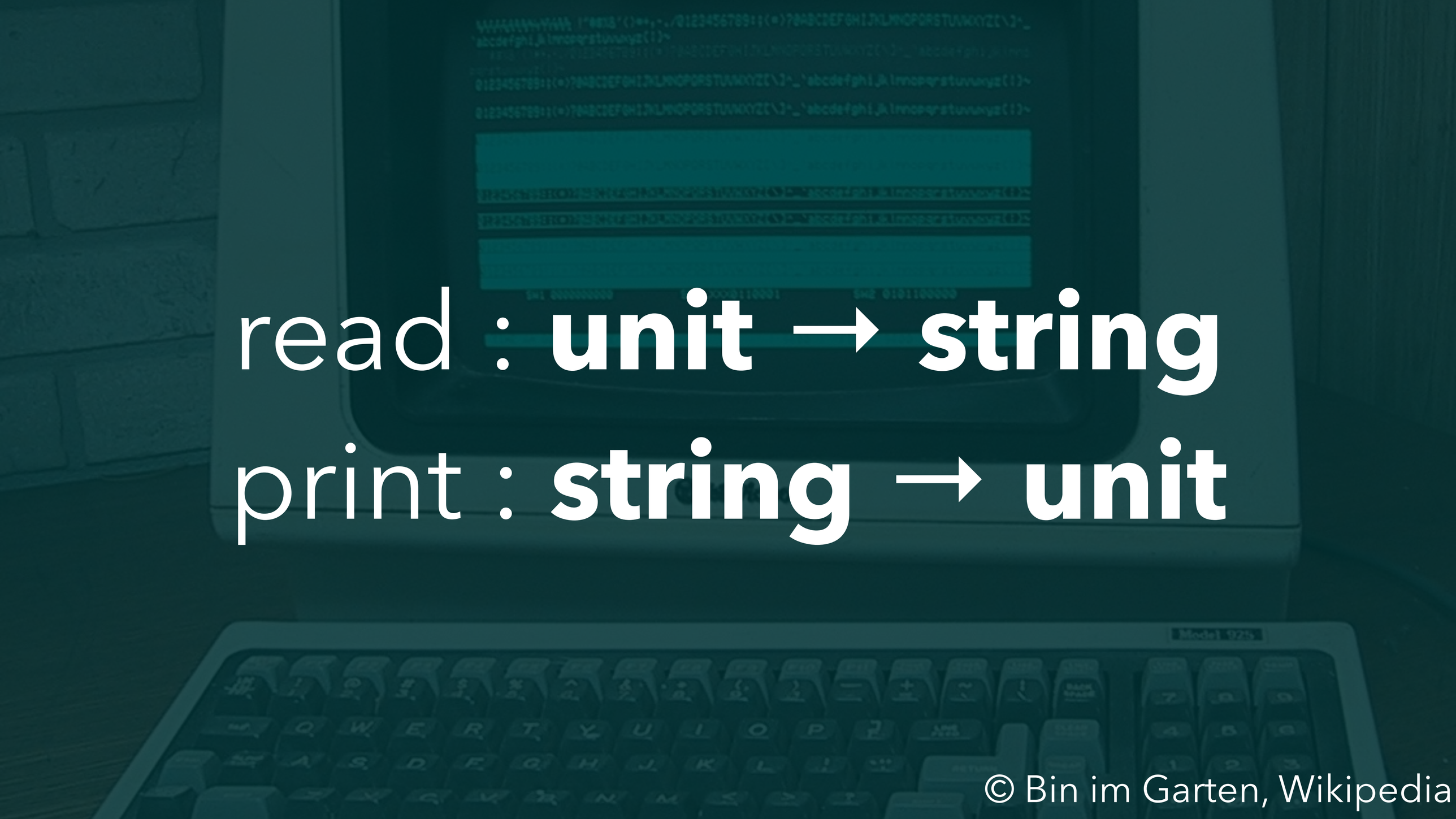




© Bin im Garten, Wikipedia



read : **unit → string**

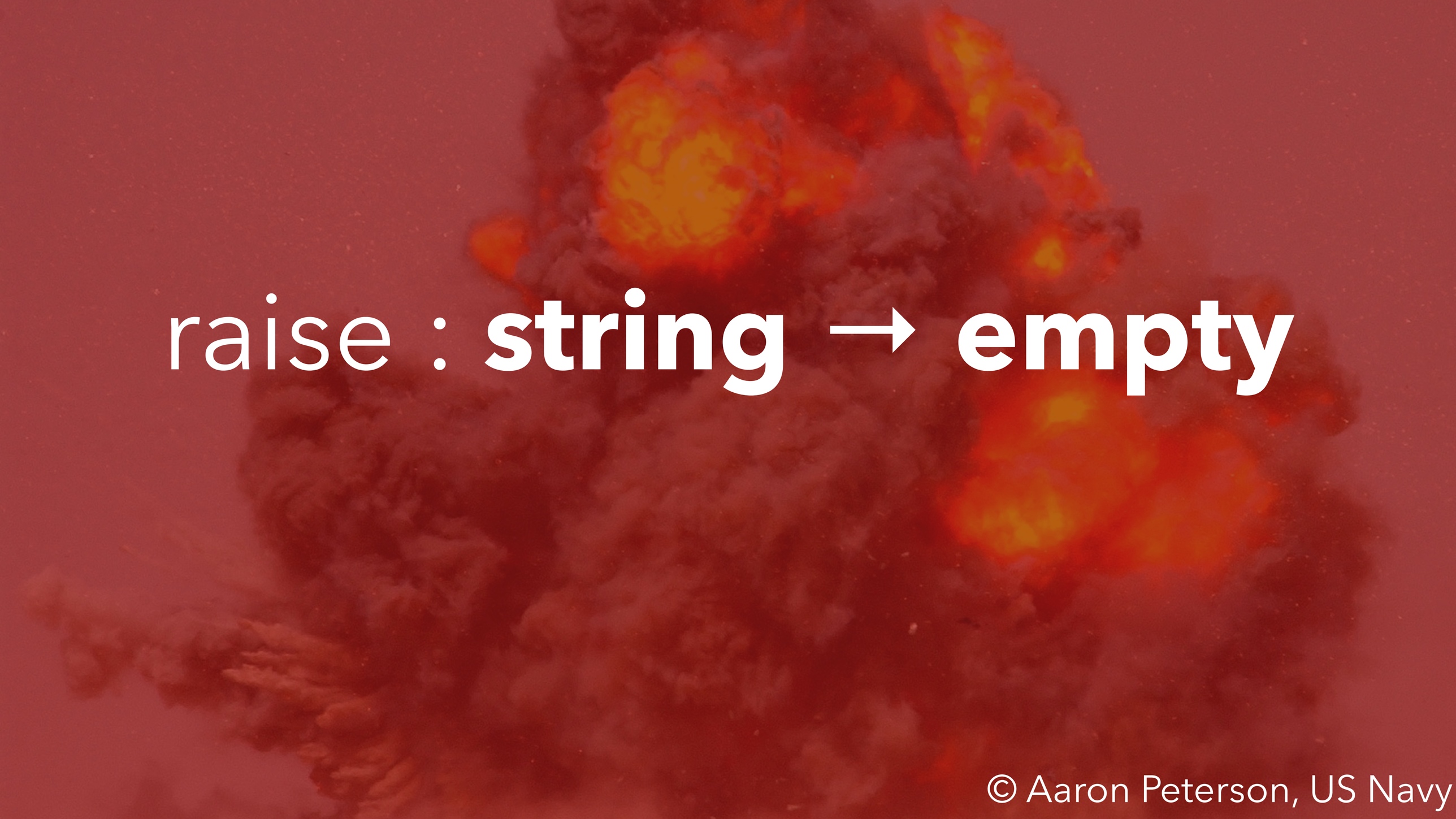


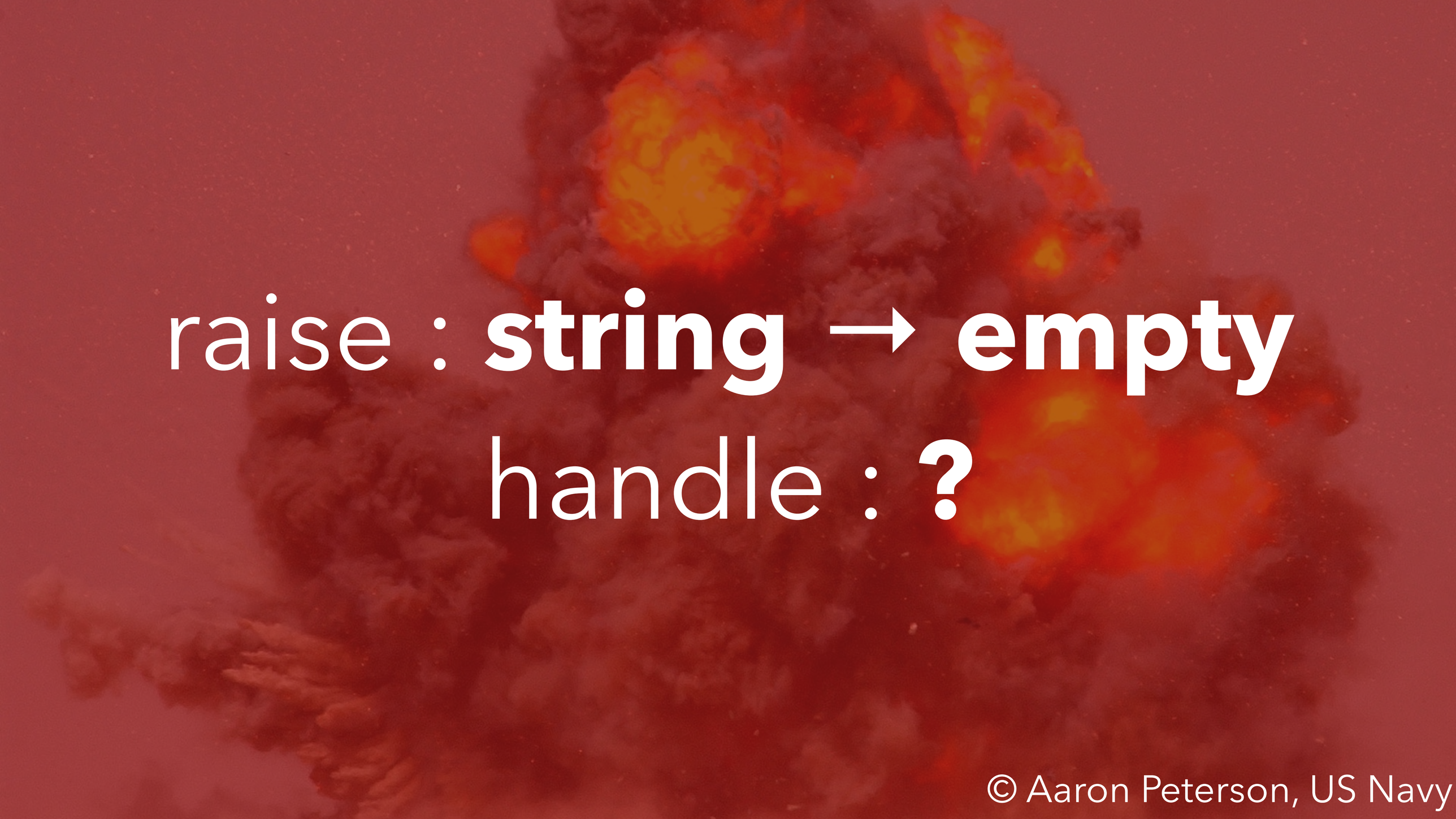
read : **unit → string**

print : **string → unit**

© Bin im Garten, Wikipedia


© Aaron Peterson, US Navy





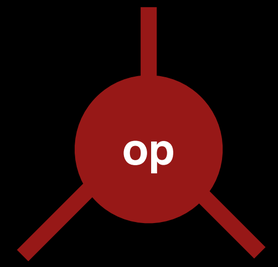
every computation either
calls an **operation**
or returns a **value**

every computation either

calls an **operation** 

or returns a **value**

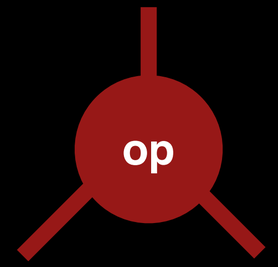
every computation either

calls an **operation** 

or returns a **value**

every computation either

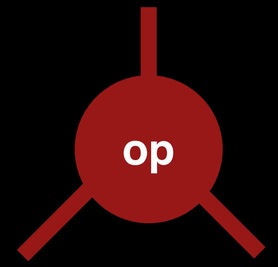
calls an **operation**



or returns a **value**

every computation either

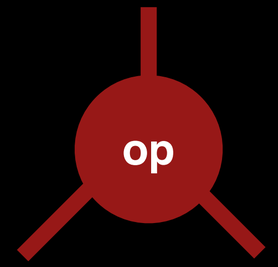
calls an **operation**



or returns a **value**

or diverges

every computation either

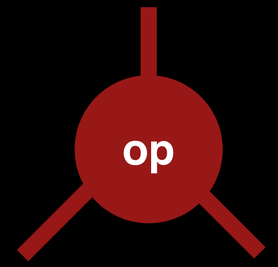
calls an **operation** 

or returns a **value**

or diverges 

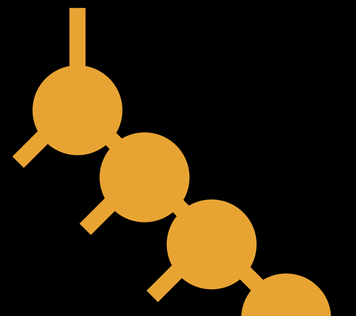
every computation either

calls an **operation**



or returns a **value**

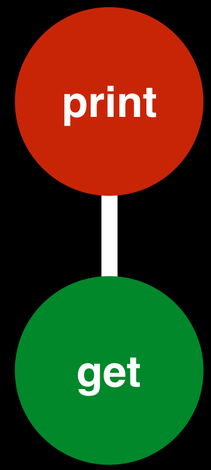
or diverges



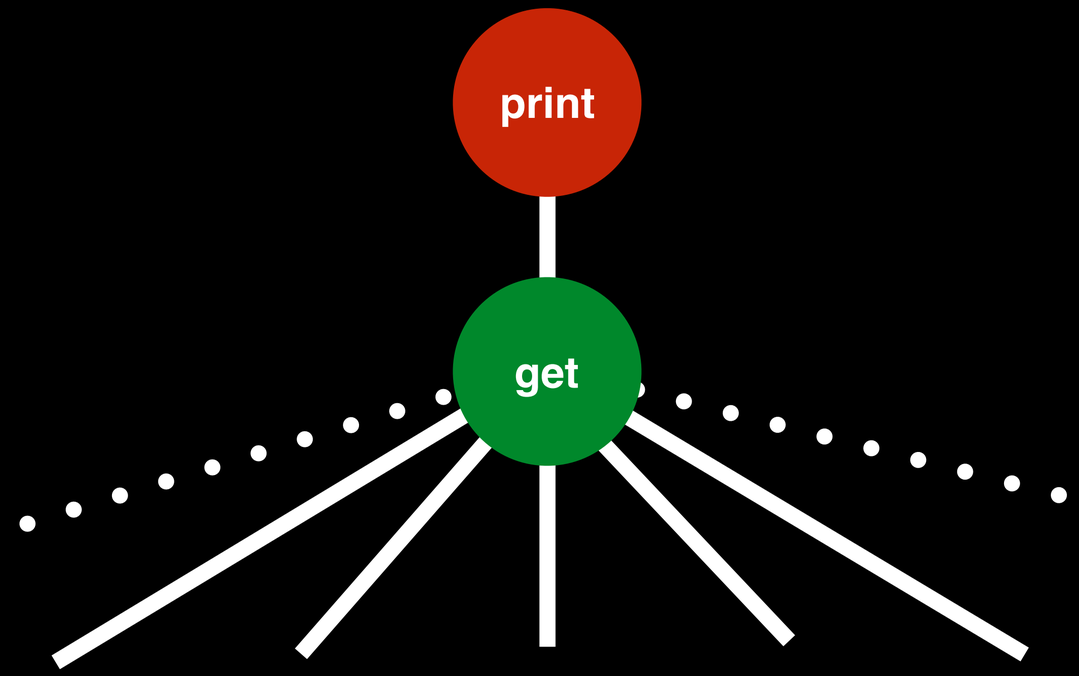
```
print(“A”);
do n ← get() in
if n < 0 then
    print(“B”);
    return −n²
else
    return n + 1
```

```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```

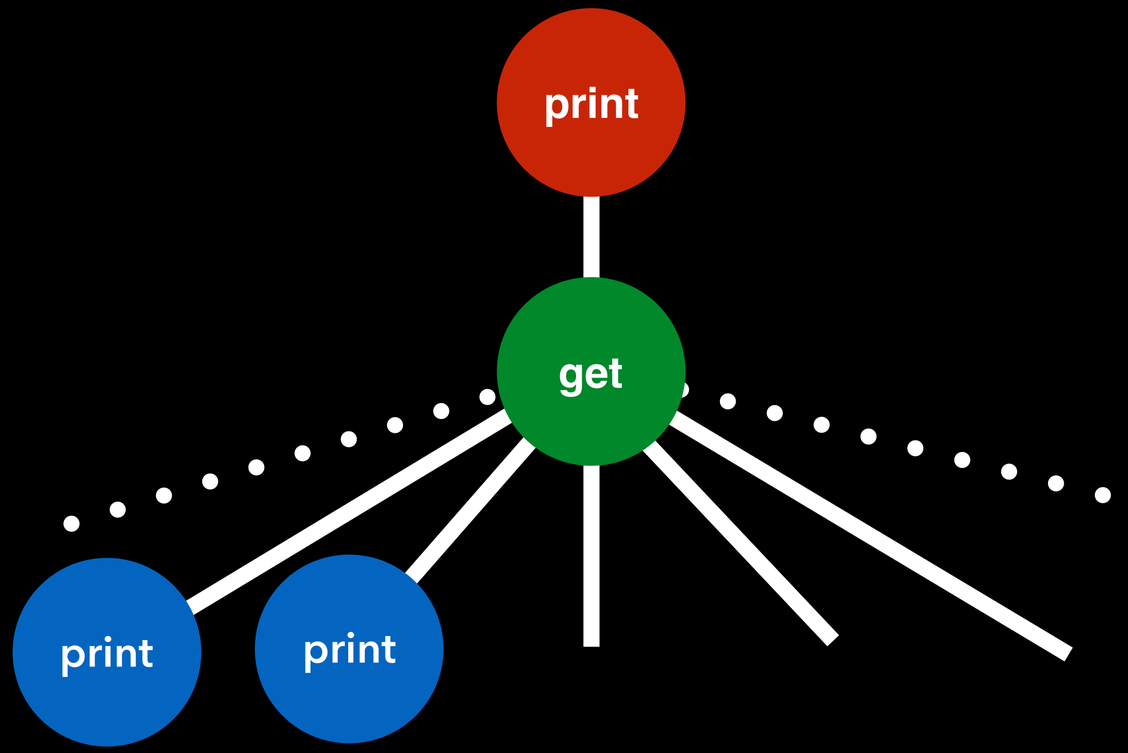
```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```



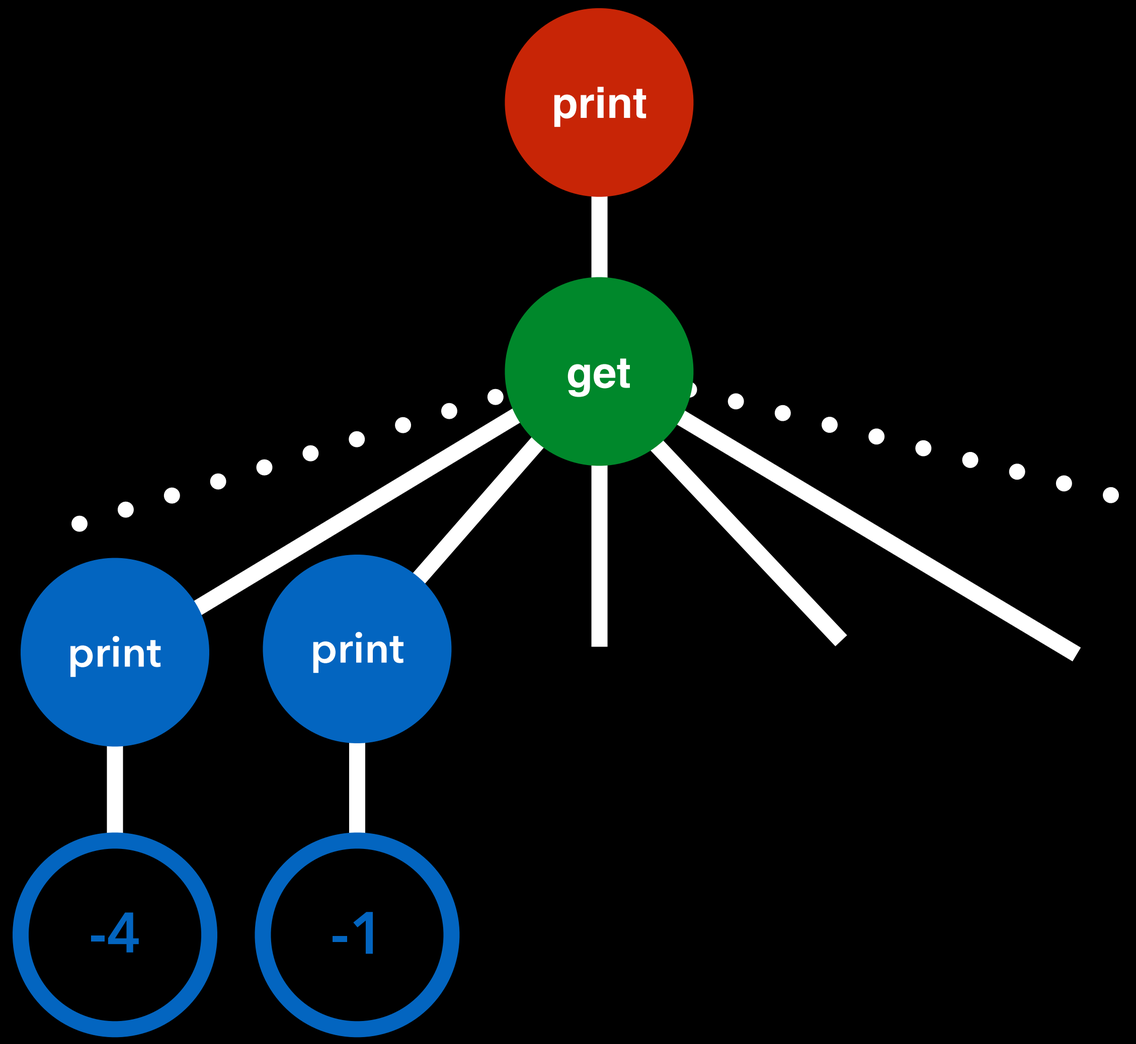
```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```



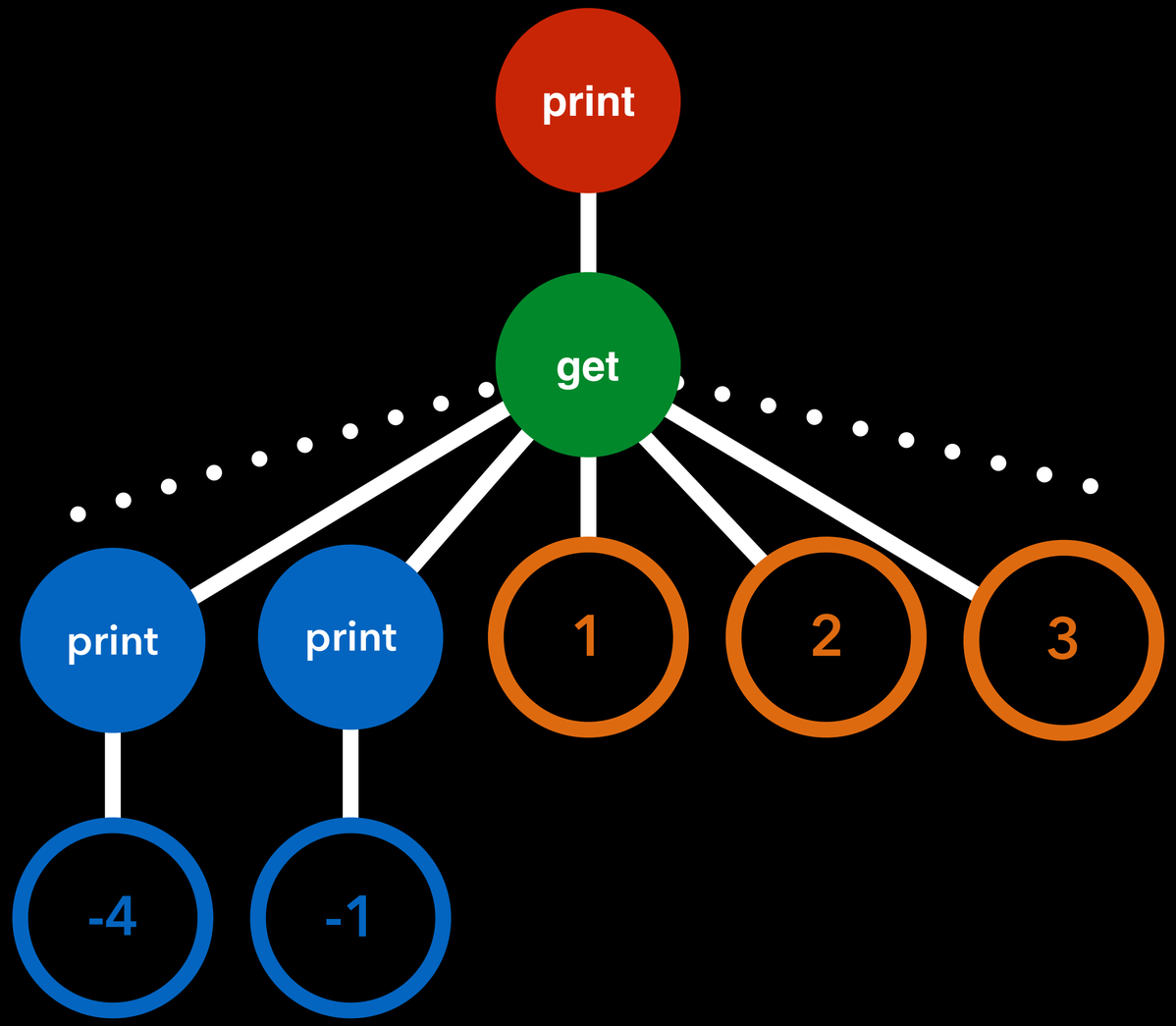
```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```



```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```



```
print("A");
do n ← get() in
if n < 0 then
    print("B");
    return −n²
else
    return n + 1
```



model

carrier M

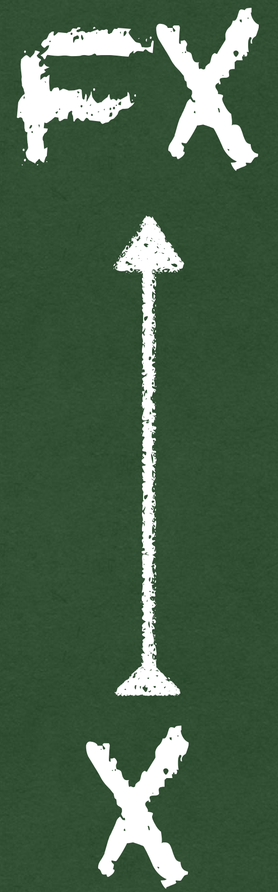
maps $op_M : A \times M^B \to M$

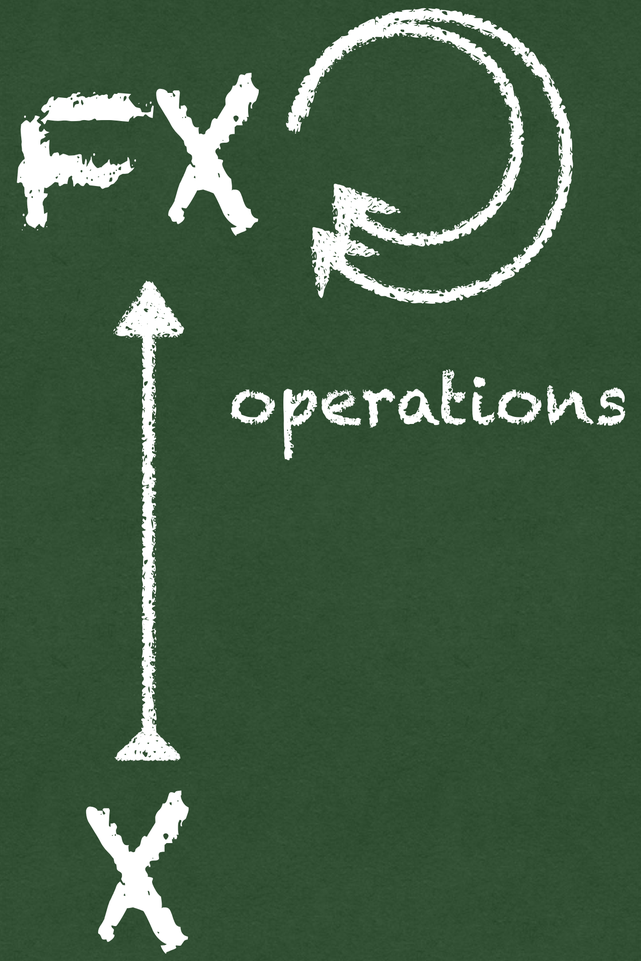
for each $op : A \to B$

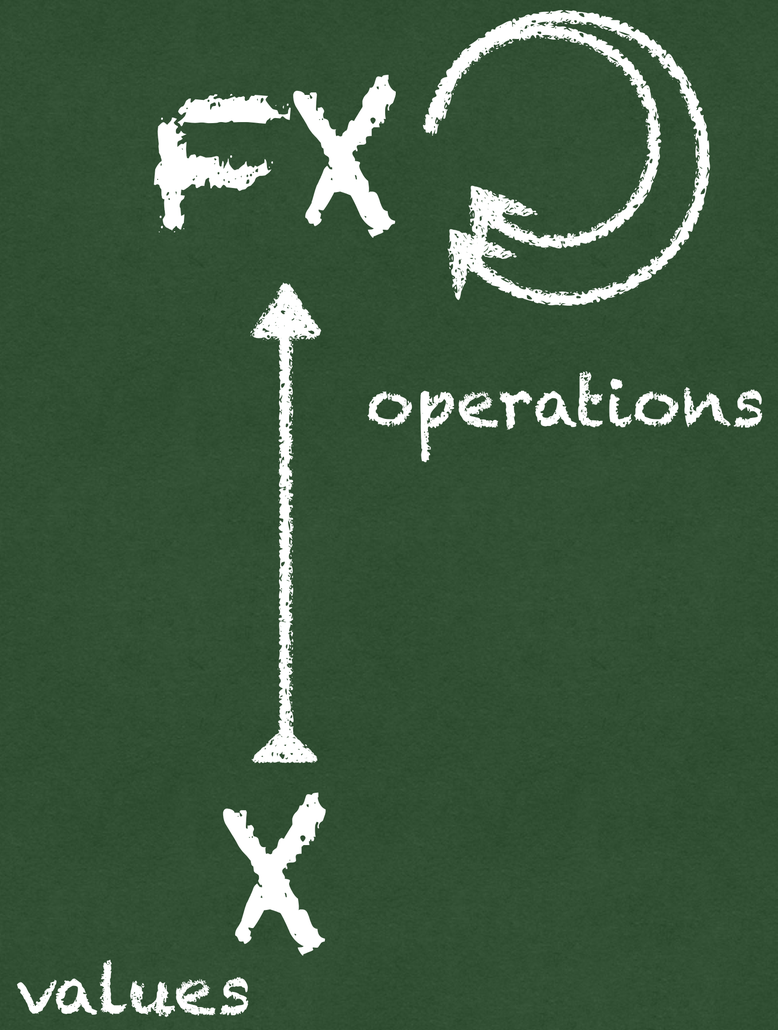
X

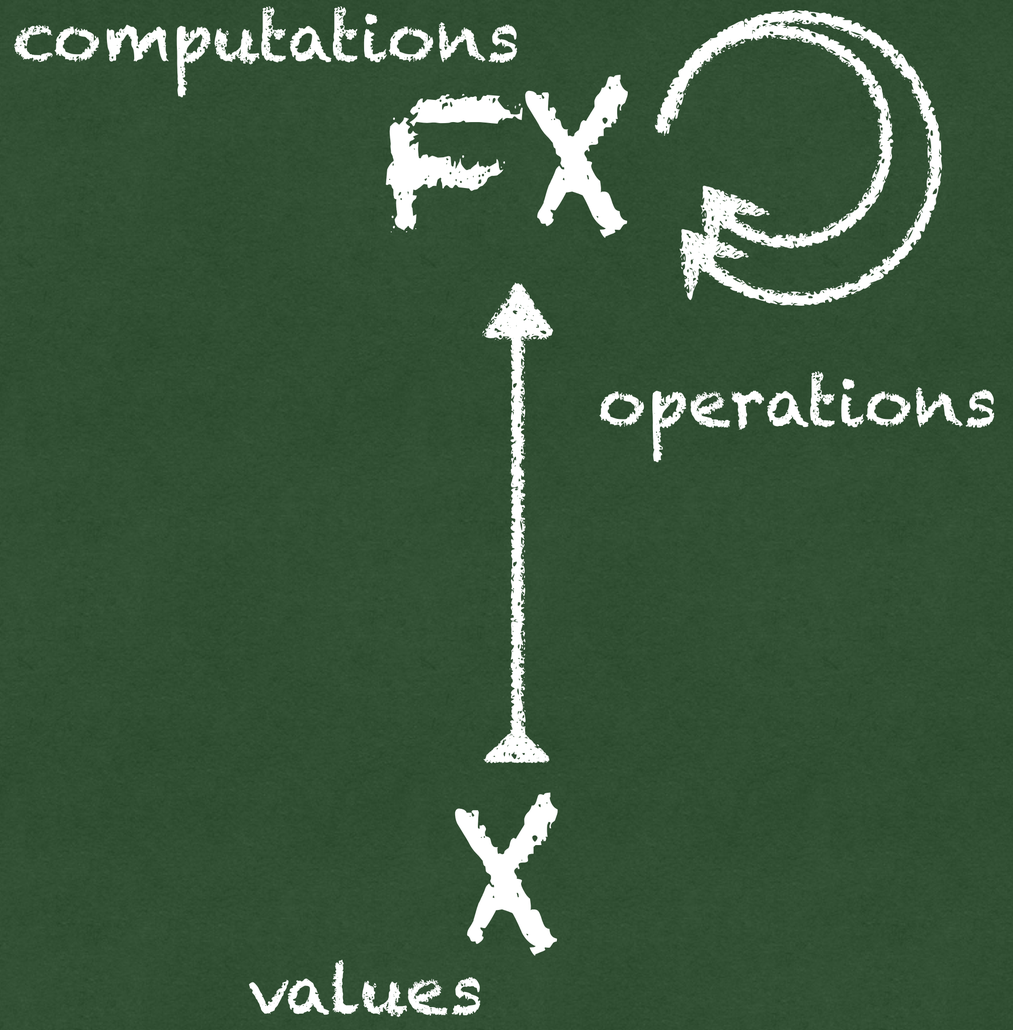
FX

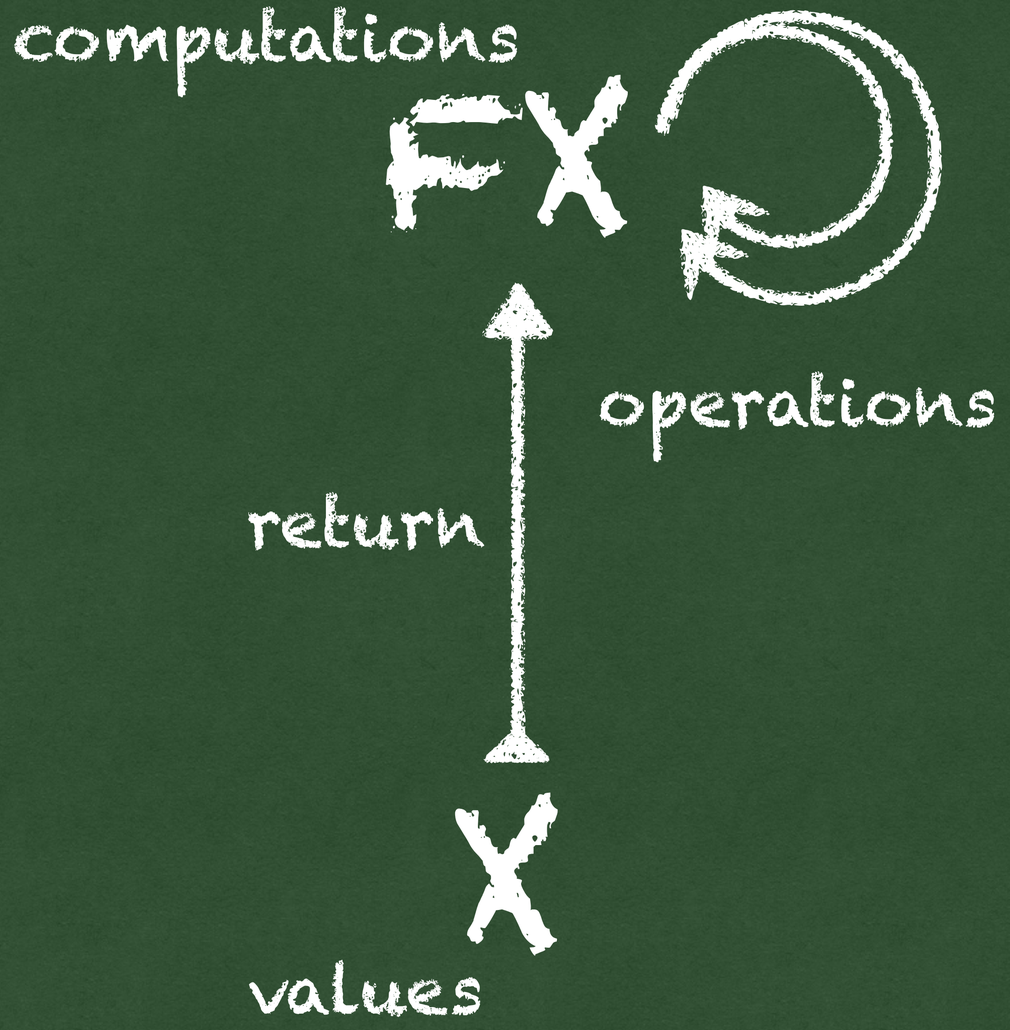
X


FX
X


FX
operations
X


FX
operations
X
values


computations
FX
operations
X
values


computations
FX
operations
return
X
values

What are

algebraic effects?

What are

algebraic effects?

What are

handlers?

```
print(“Hello!”);
raise(“Boom!”);
return 1001
```

```
print(“Hello!”);
raise(“Boom!”);
return 1001
```

```
Hello!
```

```
print(“Hello!”);
raise(“Boom!”);
return 1001
```

```
Hello!
Uncaught exception Boom!
```

```
handle
    print(“Hello!”);
    raise(“Boom!”);
    return 1001
with
    raise(err) → return 10
```

```
Hello!
Uncaught exception Boom!
```

```
handle
    print(“Hello!”);
    raise(“Boom!”);
    return 1001
with
    raise(err) → return 10
```

```
Hello!
- 10 : int
```

```
handle
    print(“Hello!”);
    raise(“Boom!”);
    return 1001
with
    raise(err) → return 10
    print(msg)→ return 20
```

```
Hello!
- 10 : int
```

```
handle
    print(“Hello!”);
    raise(“Boom!”);
    return 1001
with
    raise(err) → return 10
    print(msg)→ return 20
```

```
- 20 : int
```

```
handle
  print(“Hello!”);
  raise(“Boom!”);
  return 1001
with
  raise(err) → return 10
  print(msg; k) → return (1 + k ())
```

```
- 20 : int
```

```
handle
    print(“Hello!”);
    raise(“Boom!”);
    return 1001
with
    raise(err) → return 10
    print(msg; k) → return (1 + k ())
```

```
- 14 : int
```

```
handle
  print("Hello!");
  raise("Boom!");
  return 1001
with
  raise(err) → return 10
  print(msg; k) → return (1 + k ())
  return x → return (x - 1)
```

```
- 14 : int
```

```
handle
  print("Hello!");

  return 1001
with
  raise(err) → return 10
  print(msg; k) → return (1 + k ())
  return x → return (x - 1)
```

```
- 14 : int
```

```
handle
  print("Hello!");

  return 1001
with
  raise(err) → return 10
  print(msg; k) → return (1 + k ())
  return x → return (x - 1)
```

```
- 1004 : int
```

```
abc :=
    print("A");
    print("B");
    print("C")
```

```
abc :=
    print("A");
    print("B");
    print("C")
```

```
> abc
```

```
abc :=
    print("A");
    print("B");
    print("C")
```

```
> abc
A
B
C
- () : unit
```

```
repeat :=
  handler
    print(msg; k) →
      print(msg);
      print(msg);
      k ()
```

```
> with repeat handle abc
A
A
B
B
C
C
- () : unit
```

```
silence :=
  handler
    print(msg; k) →
      k ()
```

```
> with silence handle abc
- () : unit
```

```
reverse :=
  handler
    print(msg; k) →
      k ();
      print(msg)
```

```
> with reverse handle abc
C
B
A
- () : unit
```

```
mirror :=
 handler
  print(msg; k) →
   print(msg);
   k ();
   print(msg)
```

```
> with mirror handle abc
A
B
C
C
B
A
- () : unit
```

```
amplify :=
  handler
    print(msg; k) →
      print(msg);
      k ();
      k ()
```

```
> with amplify handle abc
A
B
C
C
B
C
C
- () : unit
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int

> with count handle
    with repeat handle abc
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int

> with count handle
    with repeat handle abc
- 6 : int
```

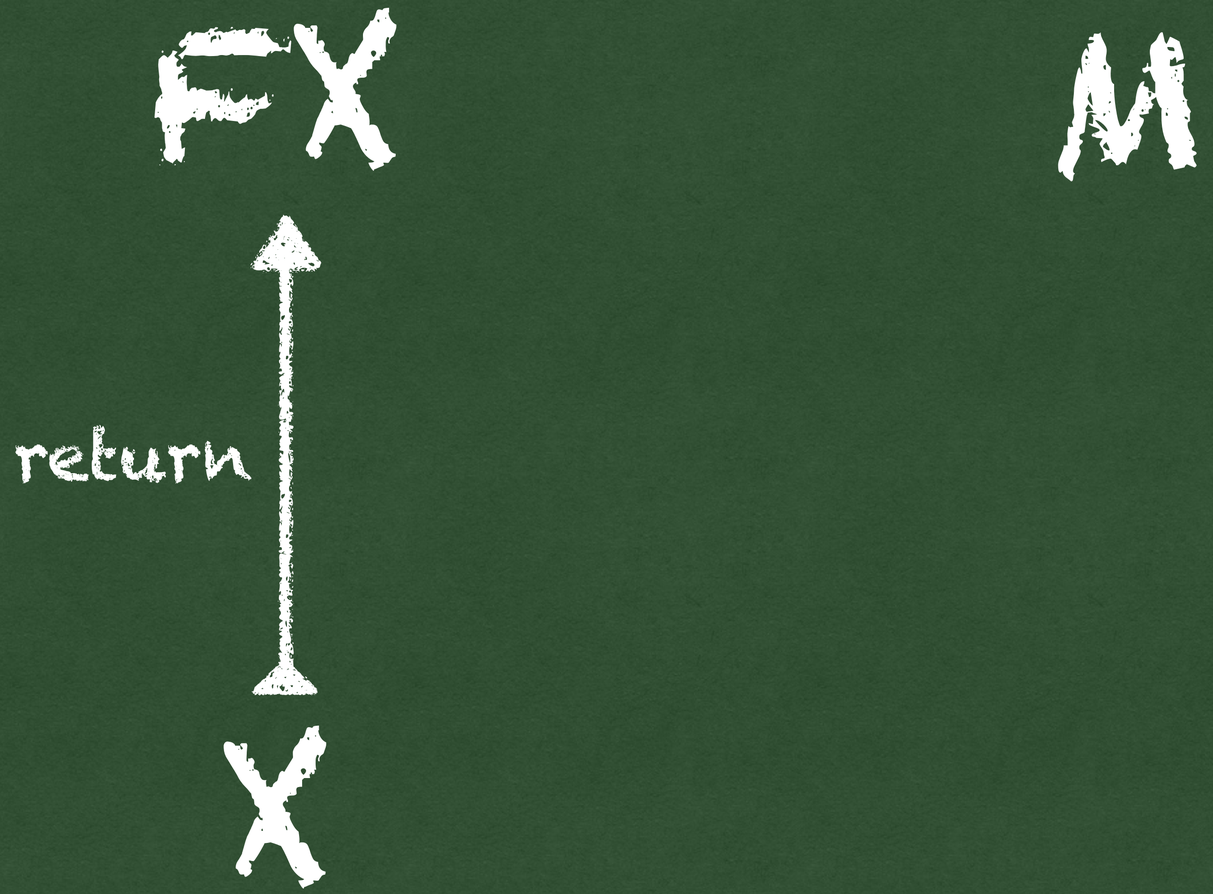
```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int

> with count handle
    with repeat handle abc
- 6 : int

> with silence handle
    with count handle abc
```

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int

> with count handle
    with repeat handle abc
- 6 : int

> with silence handle
    with count handle abc
- 3 : int
```


FX
return
X


FX
M
return
X


FX
return
X


M
operation
clauses

operation
clauses

op : A -> B

operation clauses

op : A -> B

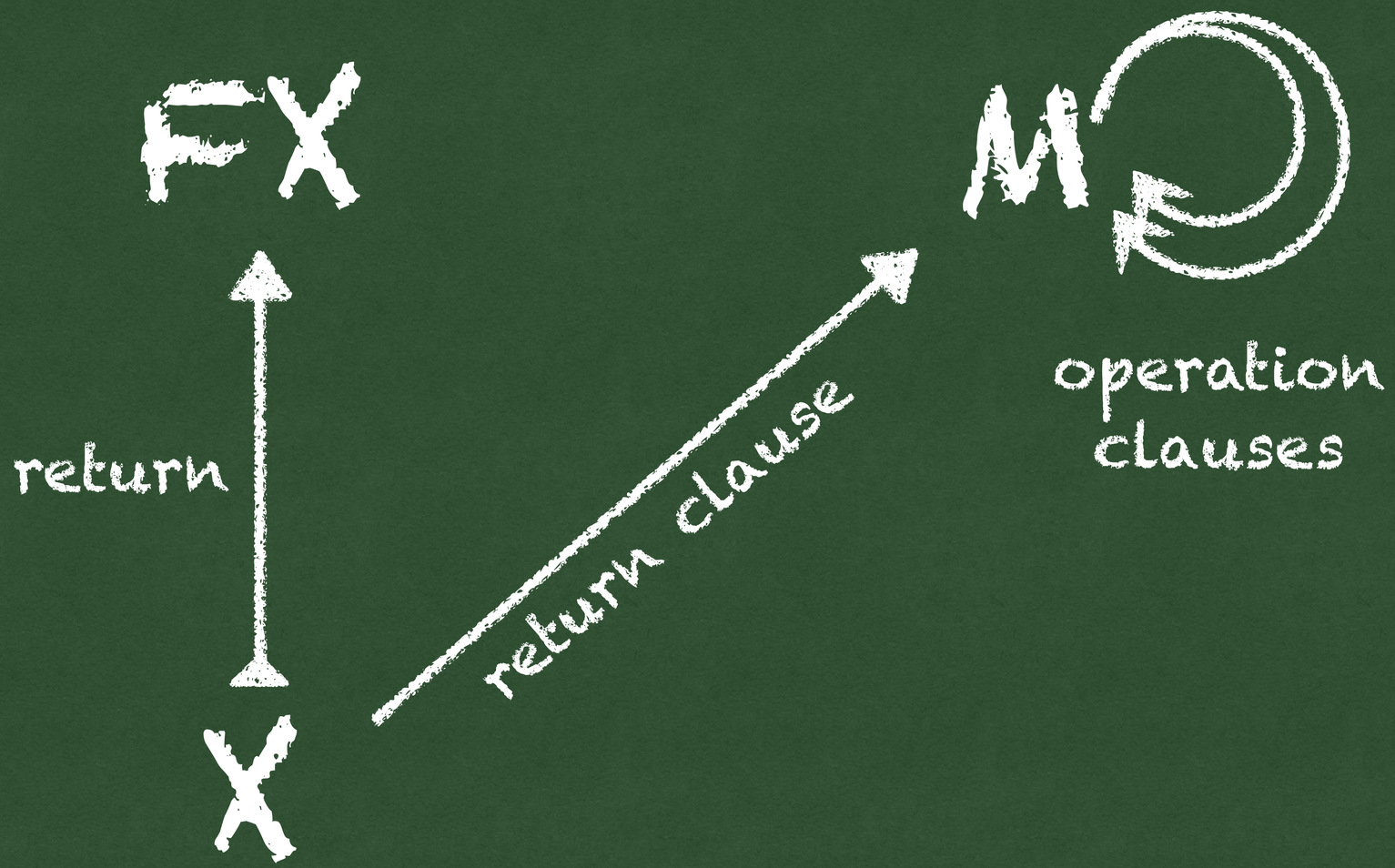
op(y; k) -> c_{op}

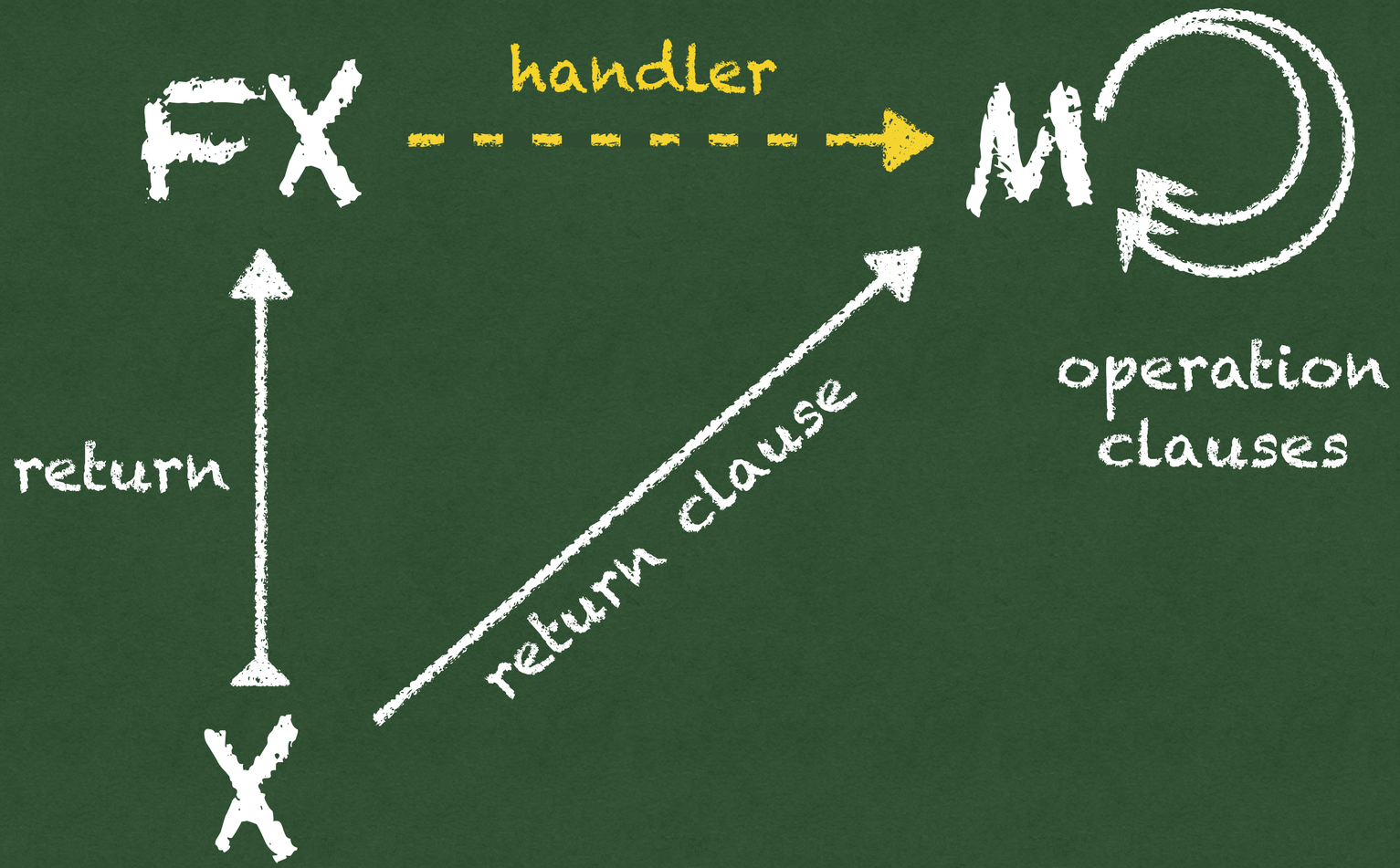
operation clauses

$op : A \to B$

$op(y; k) \to c_{op}$

$op_M = [\![c_{op}]\!] : A \times M^B \to M$


FX
M
return
return clause
operation
clauses
X


FX
handler
M
operation
clauses
return
return clause
X

What are

handlers?

What are

handlers?

Are effects & handers

useful?

reason 1:
overriding existing effects

```
count :=
  handler
    return x →
      return 0
    print(msg; k) →
      return (1 + k ())
```

```
> with count handle abc
- 3 : int
```

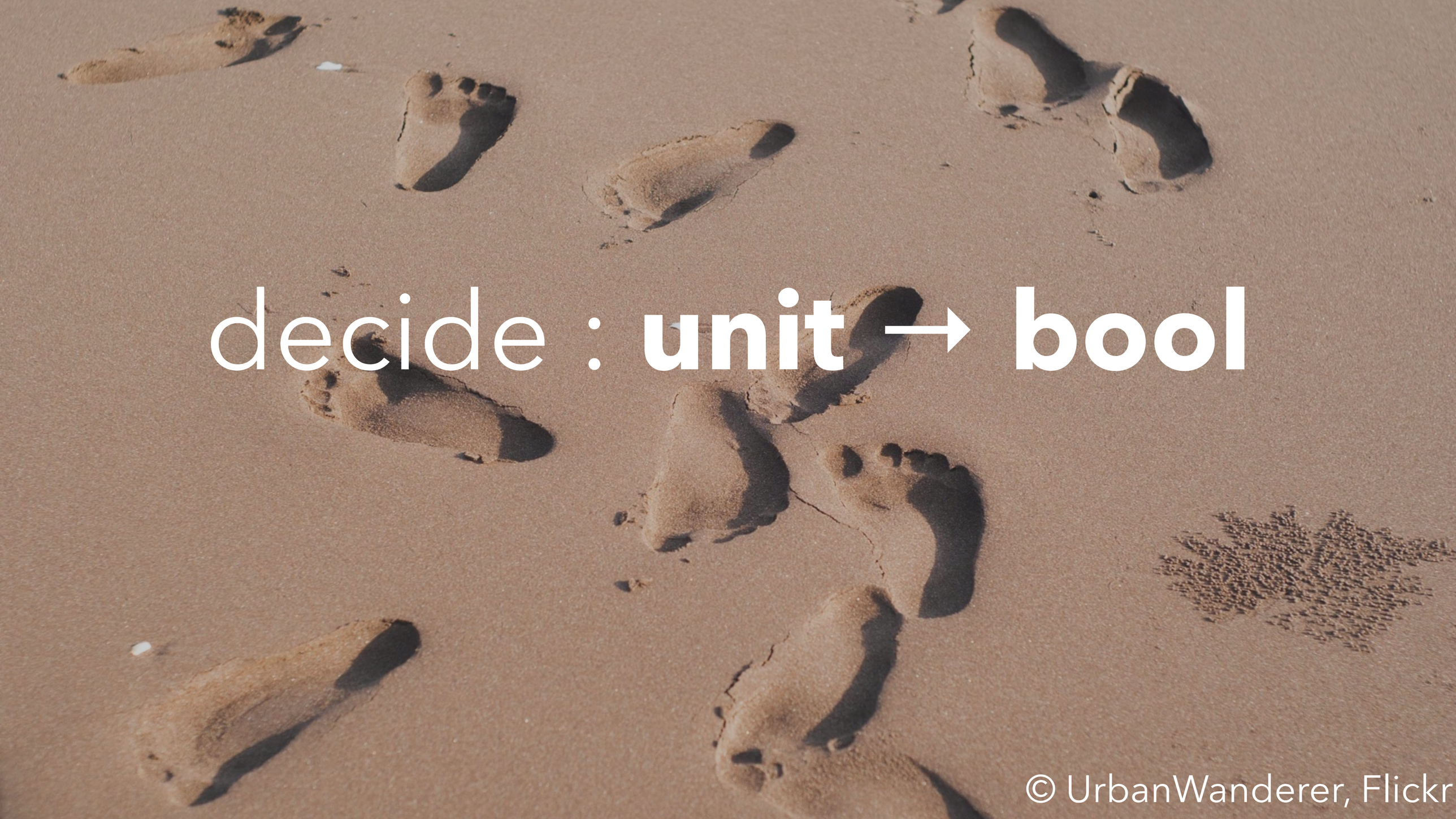
```
collect :=
  handler
    return x →
      return ""
    print(msg; k) →
      return (msg ^ k ())
```

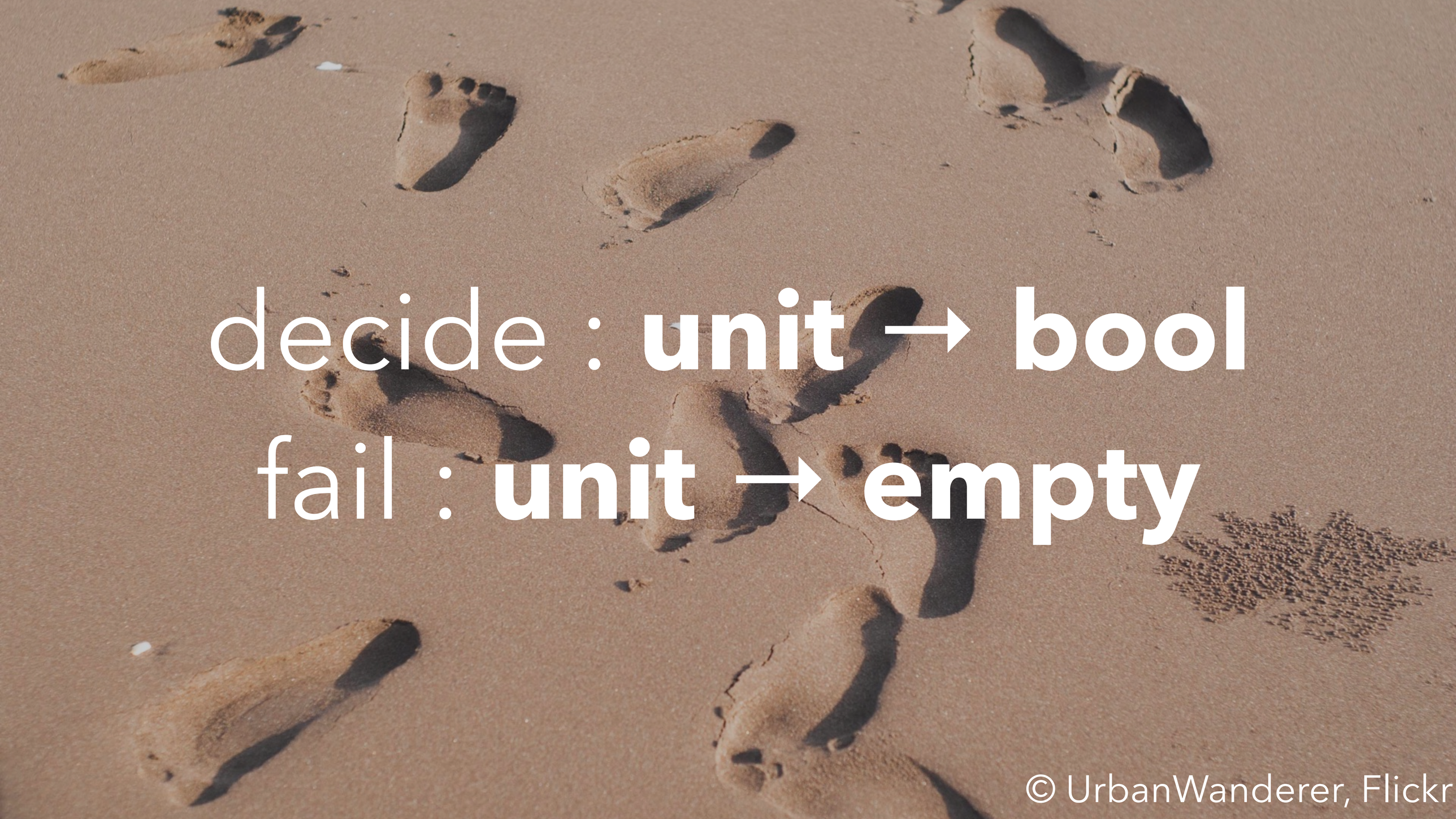
```
> with collect handle abc
- "ABC" : string
```

testing
logging
transactional memory
...

reason 2:

defining new effects





pythagorean (m, n)

$$m \leq a < b \leq n$$

$$a^2 + b^2 = c^2$$

```
chooseInt (m, n) :=
    if m > n then
        fail()
    else if decide() then
        return m
    else
        chooseInt (m + 1, n)
```

```
pythagorean (m, n) ≔
  do a ← chooseInt (m, n – 1) in
  do b ← chooseInt (a + 1, n) in
  if isSquare(a² + b²) then
    return (a, b, sqrt(a² + b²))
  else
    fail ()
```

```
pythagorean (m, n) ≔
  do a ← chooseInt (m, n – 1) in
  do b ← chooseInt (a + 1, n) in
  if isSquare(a² + b²) then
    return (a, b, sqrt(a² + b²))
  else
    fail ()
```

```
> pythagorean(3,4)
```

```
pythagorean (m, n) ≔
  do a ← chooseInt (m, n – 1) in
  do b ← chooseInt (a + 1, n) in
  if isSquare(a² + b²) then
    return (a, b, sqrt(a² + b²))
  else
    fail ()
```

```
> pythagorean(3,4)
Uncaught operation decide!
```

```
backtrack := handler
  decide(_; k) →
    handle k true with
      fail(_; _) → k false
```

```
backtrack := handler
  decide(_; k) →
    handle k true with
      fail(_; _) → k false
```

```
> with backtrack handle
    pythagorean(3,4)
```

```
backtrack ≔ handler
  decide(_; k) → 
    handle k true with
      fail(_; _) → k false
```

```
> with backtrack handle
    pythagorean(3,4)
- (3,4,5) : int×int×int
```

```
backtrack := handler
  decide(_; k) →
    handle k true with
      fail(_; _) → k false
```

```
> with backtrack handle
    pythagorean(3,4)
- (3,4,5) : int×int×int

> with backtrack handle
    pythagorean(5,15)
```

```
backtrack := handler
  decide(_; k) →
    handle k true with
      fail(_; _) → k false
```

```
> with backtrack handle
    pythagorean(3,4)
- (3,4,5) : int×int×int

> with backtrack handle
    pythagorean(5,15)
- (5,12,13) : int×int×int
```

```
backtrack ≔ handler
  decide(_; k) → 
    handle k true with
      fail(_; _) → k false
```

```
> with backtrack handle
    pythagorean(3,4)
- (3,4,5) : int×int×int

> with backtrack handle
    pythagorean(5,15)
- (5,12,13) : int×int×int

> with backtrack handle
    pythagorean(5,7)
```

```
backtrack ≔ handler
  decide(_; k) → 
    handle k true with
      fail(_; _) → k false
```

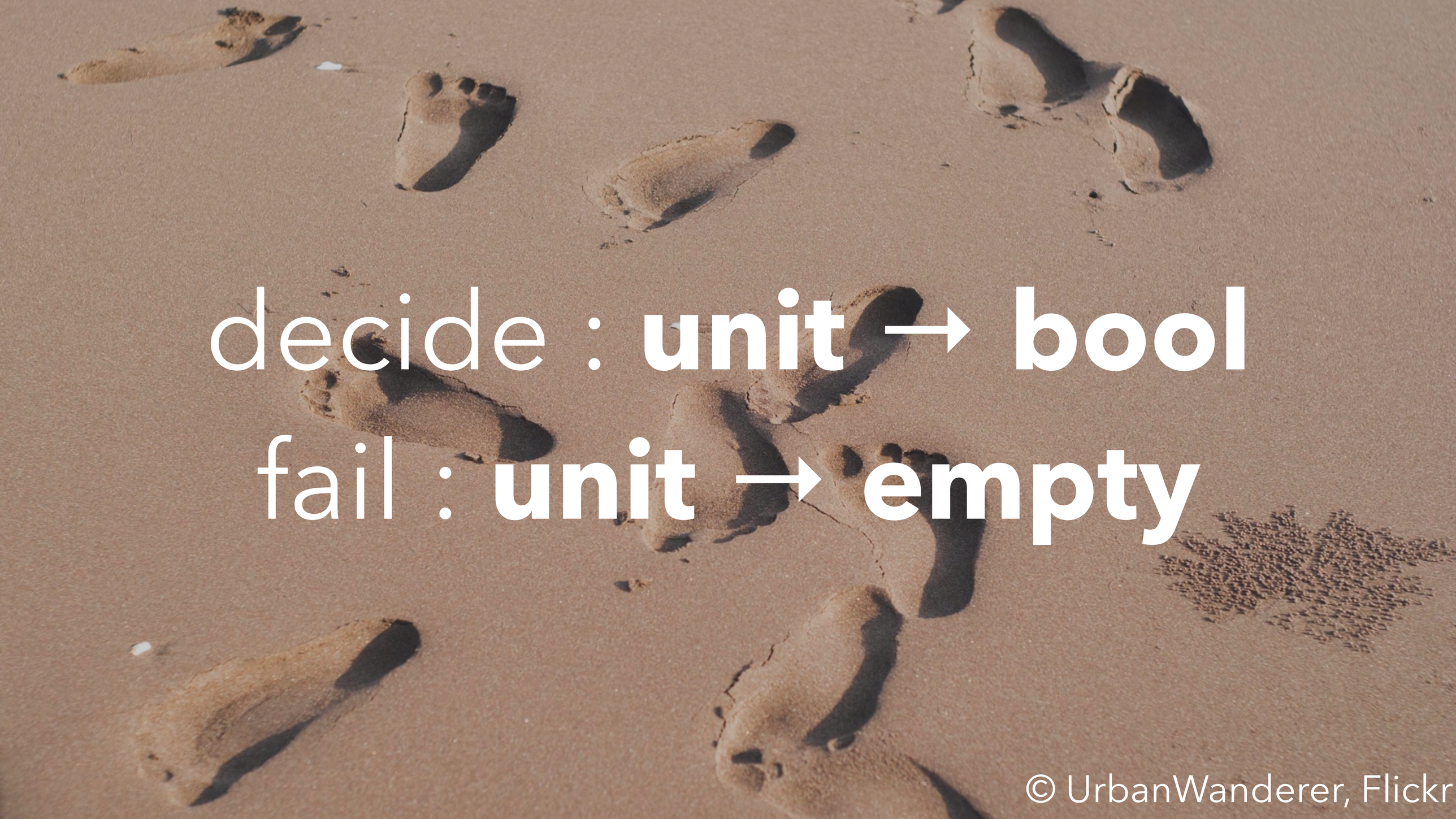
```
> with backtrack handle
    pythagorean(3,4)
- (3,4,5) : int×int×int

> with backtrack handle
    pythagorean(5,15)
- (5,12,13) : int×int×int

> with backtrack handle
    pythagorean(5,7)
Uncaught operation fail!
```

```
trackback := handler
  decide(_; k) →
    handle k false with
      fail(_; _) → k true
```

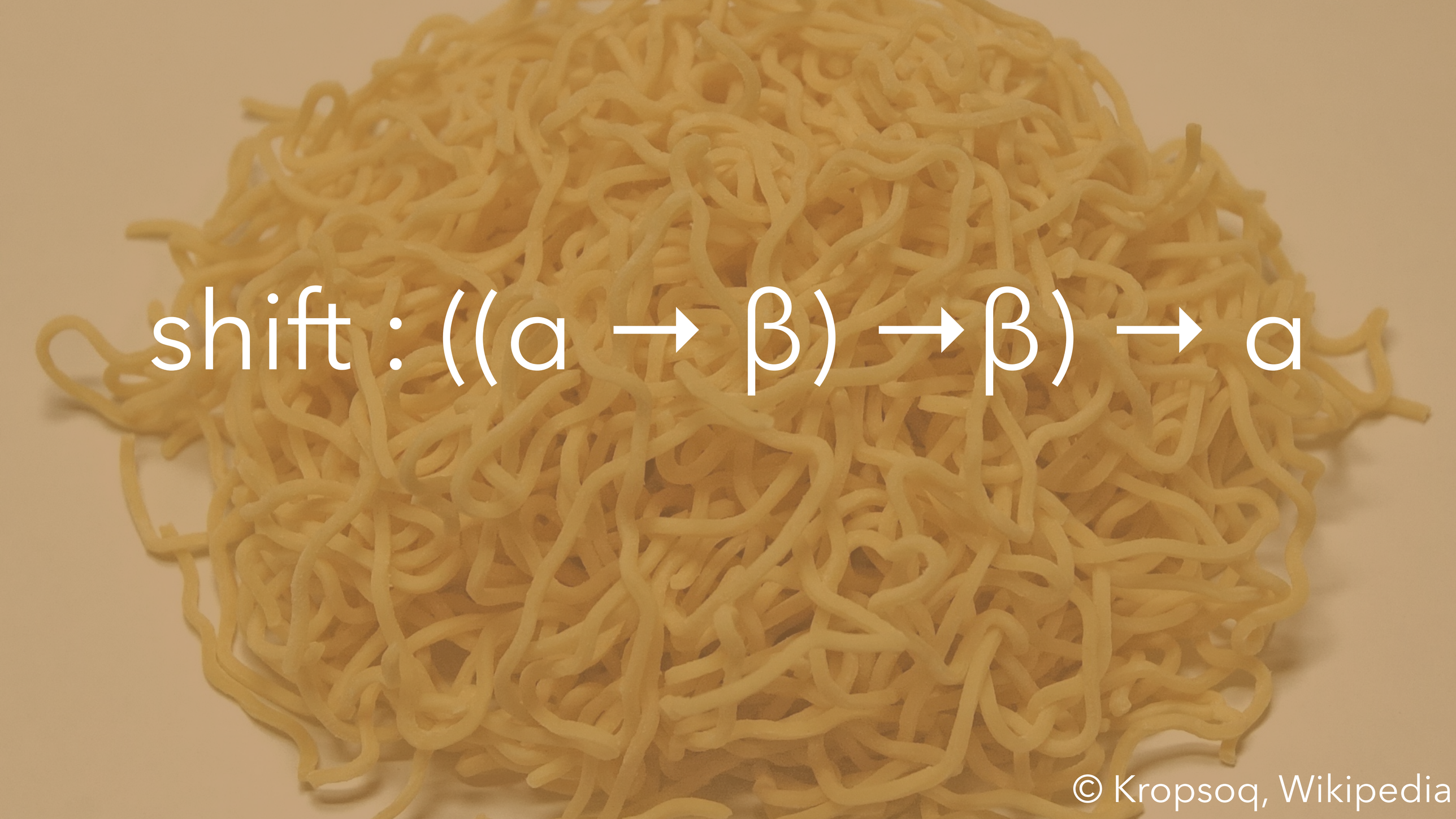
```
> with trackback handle
    pythagorean(3,4)
- (3,4,5) : int×int×int

> with trackback handle
    pythagorean(5,15)
- (9,12,15) : int×int×int

> with backtrack handle
    pythagorean(5,7)
Uncaught operation fail!
```

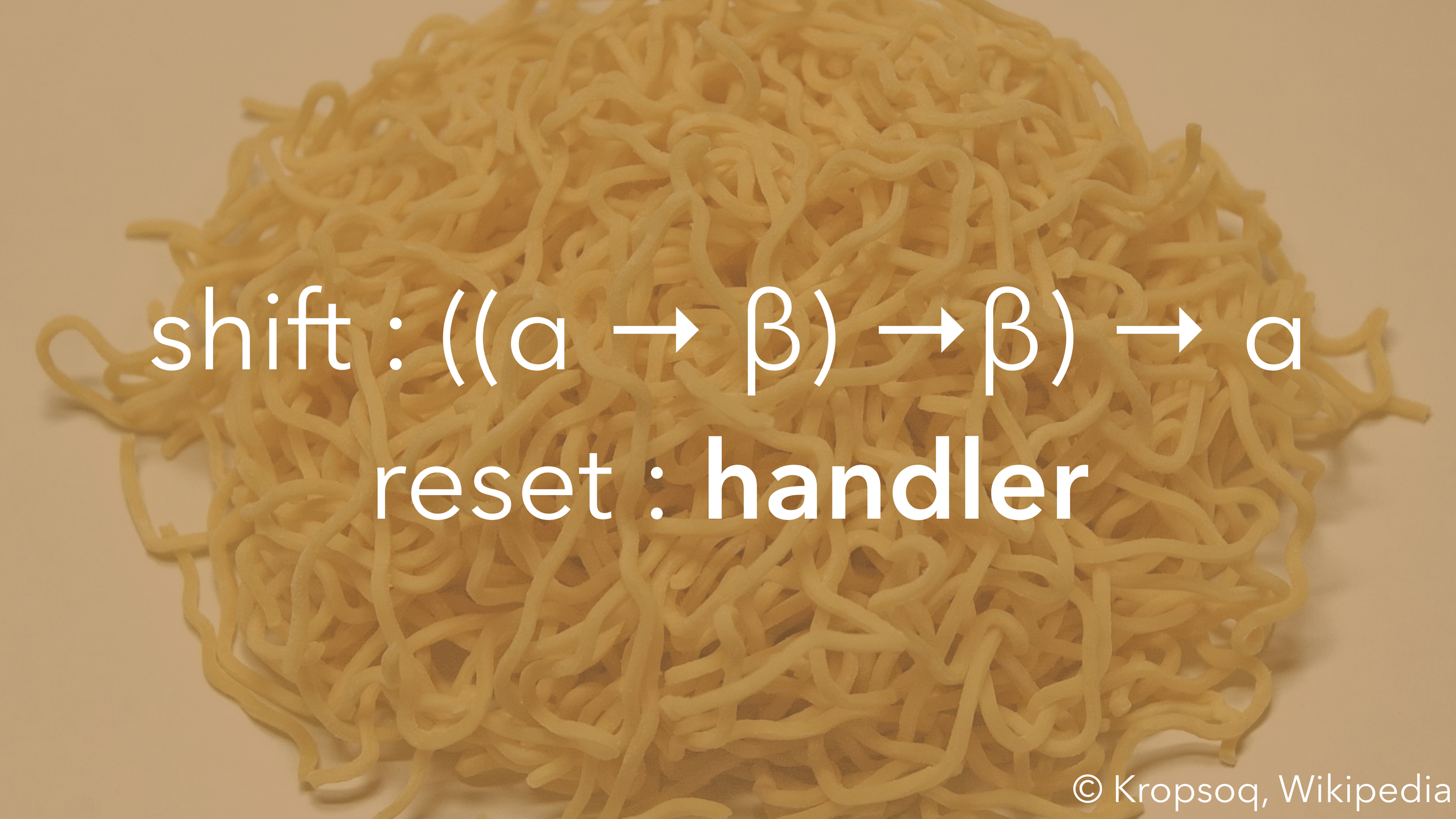
```
findAll ≔ handler
  return x → [x]
  fail(_; k) → []
  decide(_; k) →
    do lst1 ← k true in
    do lst2 ← k false in
    return (lst1 + lst2)
```

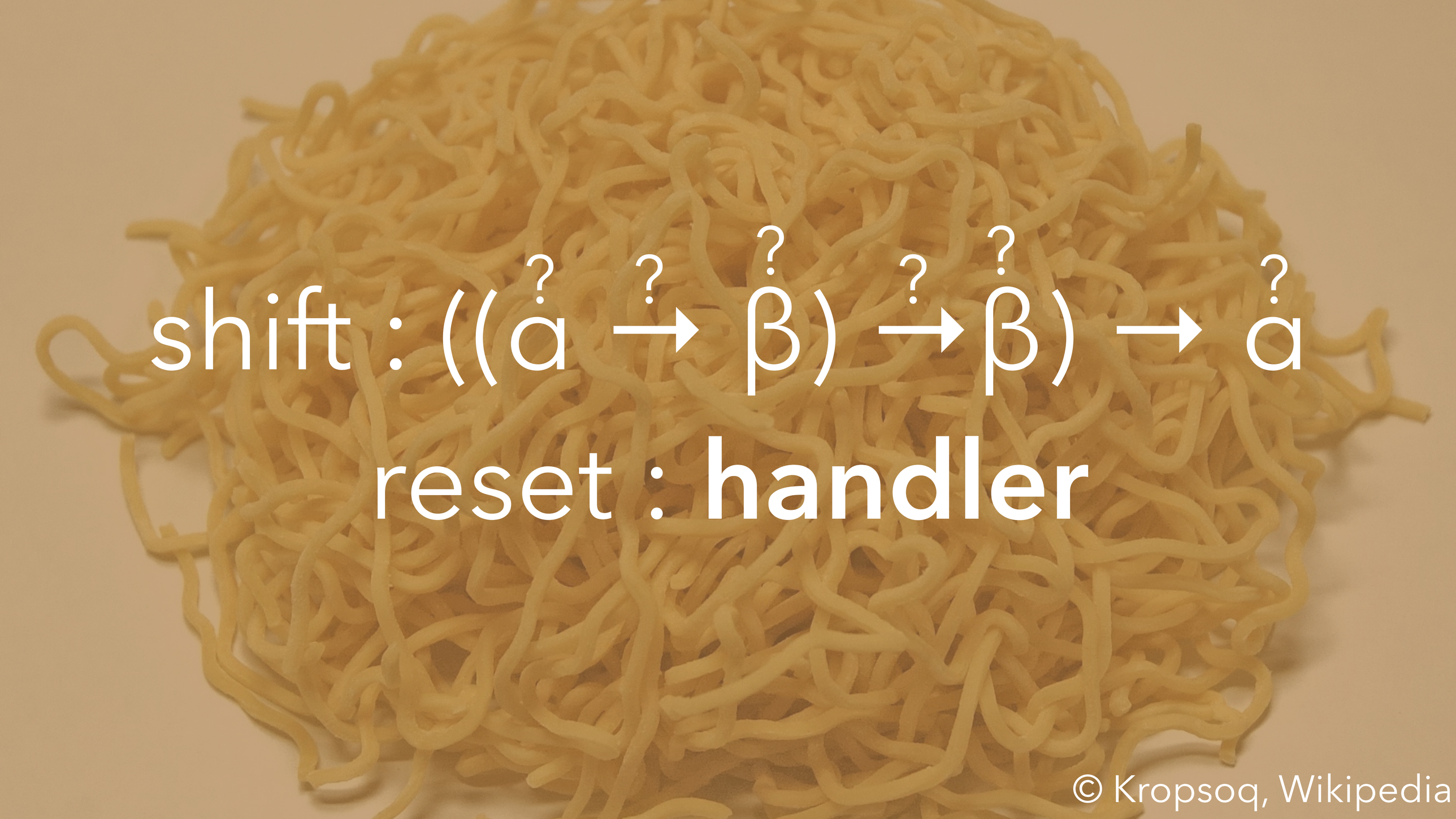
```
> with findAll handle
    pythagorean(3,4)
- [(3,4,5)]

> with findAll handle
    pythagorean(5,15)
- [[(5,12,13);(6,8,10);...]

> with findAll handle
    pythagorean(5,7)
- []
```




© Randen Pederson, Flickr







Are effects & handers

useful?

Are effects & handers

useful?

Can I **write a paper**
on effects & handers?

effect system

call-by-push-value
Haskell implementation

effect inference
polymorphic

bi-directional
polymorphic

values v

constructors

functions

handlers

computations c

deconstructors

control flow

handling

value types A

bool, **int**, …

$A \to \underline{C}$

$\underline{C}_1 \Rightarrow \underline{C}_2$

computation types $\underline{C}$

$A \,!\, \{\mathsf{op}_1, \ldots, \mathsf{op}_n\}$

$$\Gamma \vdash v : A$$

$$\Gamma \vdash c : \underline{C}$$

$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \textbf{fun}\ x \to c : A \to \underline{C}}$$

$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{return}\ v : A\ !\ \Delta}$$

$$\frac{\Gamma \vdash c_1 : A_1\ !\ \Delta \qquad \Gamma, x : A_1 \vdash c_2 : A_2\ !\ \Delta}{\Gamma \vdash \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : A_2\ !\ \Delta}$$

$$\frac{\mathsf{op} : A \to B \qquad \Gamma \vdash v : A \qquad \mathsf{op} \in \Delta}{\Gamma \vdash \mathsf{op}(v) : B \,!\, \Delta}$$

$$\frac{\Gamma \vdash \mathsf{v} : \underline{C}_1 \Rightarrow \underline{C}_2 \qquad \Gamma \vdash \mathsf{c} : \underline{C}_1}{\Gamma \vdash \textbf{with}\ v\ \textbf{handle}\ c : \underline{C}_2}$$

h ≔ **handler**

return x → c_{ret}

[op(x; k) → c_{op}]$_{op}$

$$\frac{?}{\Gamma \vdash h : A_1 \,!\, \Delta_1 \Rightarrow A_2 \,!\, \Delta_2}$$

$$\frac{(1) \qquad (2) \qquad (3)}{\Gamma \vdash h : A_1 \,!\, \Delta_1 \Rrightarrow A_2 \,!\, \Delta_2}$$

(1) $\Gamma, x : A_1 \vdash c_{ret} : A_2 \,!\, \Delta_2$

$$\frac{(1) \qquad (2) \qquad (3)}{\Gamma \vdash h : A_1 \,!\, \Delta_1 \Rightarrow A_2 \,!\, \Delta_2}$$

(1) $\Gamma, x : A_1 \vdash c_{ret} : A_2 \,!\, \Delta_2$

(2) *for each listed* op : $A_{op} \to B_{op}$:

$\Gamma, x : A_{op}, k : B_{op} \to A_2 \,!\, \Delta_2 \vdash c_{op} : A_2 \,!\, \Delta_2$

$$\frac{(1) \qquad (2) \qquad (3)}{\Gamma \vdash h : A_1 \,!\, \Delta_1 \Rightarrow A_2 \,!\, \Delta_2}$$

(1) $\Gamma, x : A_1 \vdash c_{ret} : A_2 \,!\, \Delta_2$

(2) *for each listed* op : $A_{op} \to B_{op}$:

$\Gamma, x : A_{op}, k : B_{op} \to A_2 \,!\, \Delta_2 \vdash c_{op} : A_2 \,!\, \Delta_2$

(3) $\Delta_1 \setminus \{\mathrm{op}\}_{\mathrm{op}} \subseteq \Delta_2$

$$\frac{(1) \qquad (2) \qquad (3)}{\Gamma \vdash h : A_1 \,!\, \Delta_1 \Rightarrow A_2 \,!\, \Delta_2}$$

every computation either
calls an **operation**
or returns a **value**

every $\vdash c : A \,!\, \Delta$ either

calls an $\mathbf{op} \in \Delta$

or returns a $\vdash v : A$

every $\vdash c : A \,!\, \Delta$ either

calls an **op** $\in \Delta$

or returns a $\vdash v : A$

or diverges

effect system

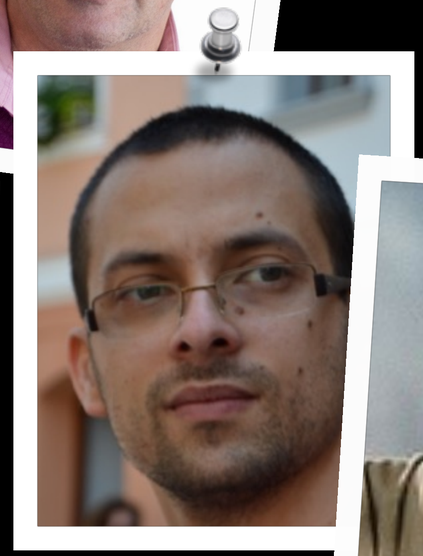
call-by-push-value
Haskell implementation

effect inference
polymorphic

bi-directional
polymorphic

reasoning

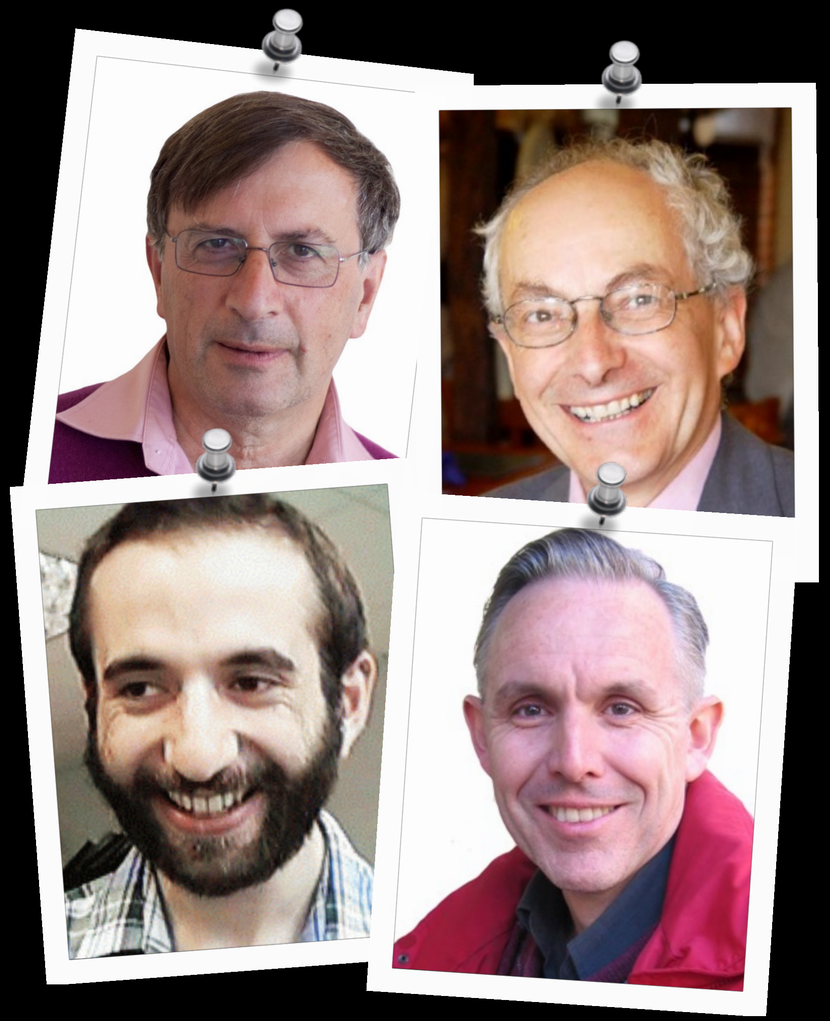
operational semantics, equations & induction



refinement types

observational equivalence

combining effects



tensors & sums

factoring to
tensors & sums

composing
handlers

modelling actual effects

comodels
tensors with models

resources

runners

$$op : A \to B$$

$$A \times M^B \to M$$

$$op : A \to B$$

$$M^B \to M^A$$

$$op : A \to B$$

$$A \times W \to B \times W$$

```
read : unit -> string
```

```
1 x W -> str x W
```

```
print : string -> unit

str x W -> 1 x W
```

decide : unit -> bool

1 x W -> 2 x W

raise : unit -> empty

$$1 \times W \to 0 \times W$$

modelling actual effects

comodels
tensors with models

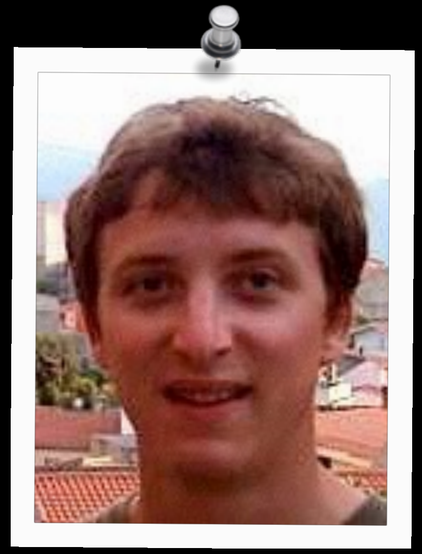
resources

runners

local effects

local state

local state



local state
instances

binary handlers

concurrency
via higher-order

multi-handlers

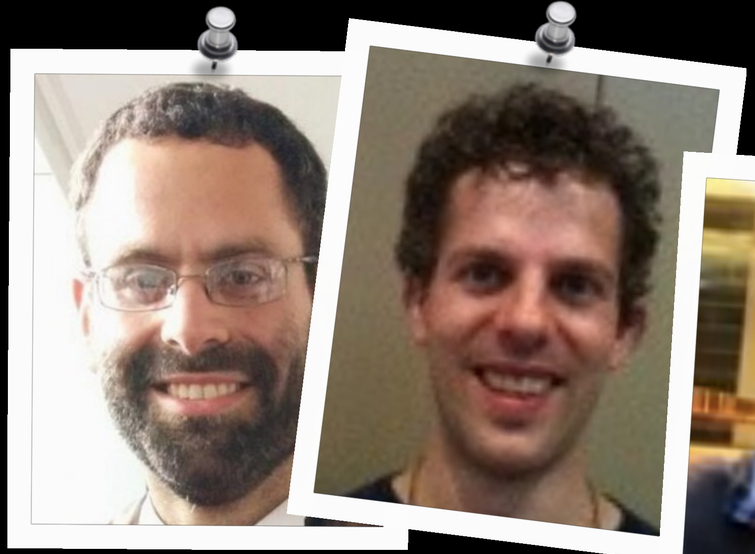
examples

Eff

Frank

jumps

quantum computing
predicate logic



handlers in action

Idris

extensible
effects

Can I **write a paper**
on effects & handers?

Can I **write a paper**
on effects & handers?

What **next**?

MFPS 2015

An Introduction to Algebraic Effects and Handlers

Invited tutorial paper

Matija Pretnar[1]

Faculty of Mathematics and Physics
University of Ljubljana
Slovenia

Abstract

This paper is a tutorial on algebraic effects and handlers. In it, we explain what algebraic effects are, give ample examples to explain how handlers work, define an operational semantics and a type & effect system, show how one can reason about effects, and give pointers for further reading.

Keywords: algebraic effects, handlers, effect system, semantics, logic, tutorial

Algebraic effects are an approach to computational effects based on a premise that impure behaviour arises from a set of *operations* such as get & set for mutable store, read & print for interactive input & output, or raise for exceptions [16,18]. This naturally gives rise to *handlers* not only of exceptions, but of any other effect, yielding a novel concept that, amongst others, can capture stream redirection, backtracking, co-operative multi-threading, and delimited continuations [21,22,5].

I keep hearing from people that they are interested in algebraic effects and handlers, but do not know where to start. This is what this tutorial hopes to fix. We will look at how to program with algebraic effects and handlers, how to model them, and how to reason about them. The tutorial requires no special background knowledge except for a basic familiarity with the theory of programming languages (a good introduction can be found in [8,15]).

1 Language

Before we dive into examples of handlers, we need to fix a language in which to work. As the order of evaluation is important when dealing with effects, we split language terms (Figure 1) into inert *values* and potentially effectful *computations*,

[1] The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

MFPS 2015

An Introduction to Algebraic Effects and Handlers

Invited tutorial paper

Matija Pretnar[1]

Faculty of Mathematics and Physics
University of Ljubljana
Slovenia

Abstract

This paper is a tutorial on algebraic effects and handlers. In it, we explain what algebraic effects are, give ample examples to explain how handlers work, define an operational semantics and a type & effect system, show how one can reason about effects, and give pointers for further reading.

Keywords: algebraic effects, handlers, effect system, semantics, logic, tutorial

Algebraic effects are an approach to computational effects based on a premise that impure behaviour arises from a set of *operations* such as get & set for mutable store, read & print for interactive input & output, or raise for exceptions [16,18]. This naturally gives rise to *handlers* not only of exceptions, but of any other effect, yielding a novel concept that, amongst others, can capture stream redirection, backtracking, co-operative multi-threading, and delimited continuations [21,22,5].

I keep hearing from people that they are interested in algebraic effects and handlers, but do not know where to start. This is what this tutorial hopes to fix. We will look at how to program with algebraic effects and handlers, how to model them, and how to reason about them. The tutorial requires no special background knowledge except for a basic familiarity with the theory of programming languages (a good introduction can be found in [8,15]).

1 Language

Before we dive into examples of handlers, we need to fix a language in which to work. As the order of evaluation is important when dealing with effects, we split language terms (Figure 1) into inert *values* and potentially effectful *computations*,

[1] The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

special

session

on

effects

Questions?

Thanks!