
# Verifying Effectful Programs with Answer-Type Modification

Taro Sekiyama

National Institute of Informatics (NII)
Tokyo, Japan

@ OlivierFest

# Control effects

- Effects influencing the control flow
  - States, exception, async/await, coroutine, nondeterminism, etc.
- Implemented by ***control operators***
  - Able to manipulate continuations in a flexible manner
- A variety of control operators have been proposed, including
  - shift/reset, shift0/reset0 [Danvy and Filinski '89]
  - control/prompt [Felleisen '88]
  - Effect handlers [Plotkin & Pretnar '09, '13]

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

**Handling effects**

Installs handler h to handle **`Tick`** invoked by the term

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
with h handle
  let a = n + m in Tick ();
  let b = a + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

**Handling effects**

Installs handler h to handle **`Tick`** invoked by the term

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
with h handle
                    Tick ();
  let b = N + 1 in Tick ();
  b
```

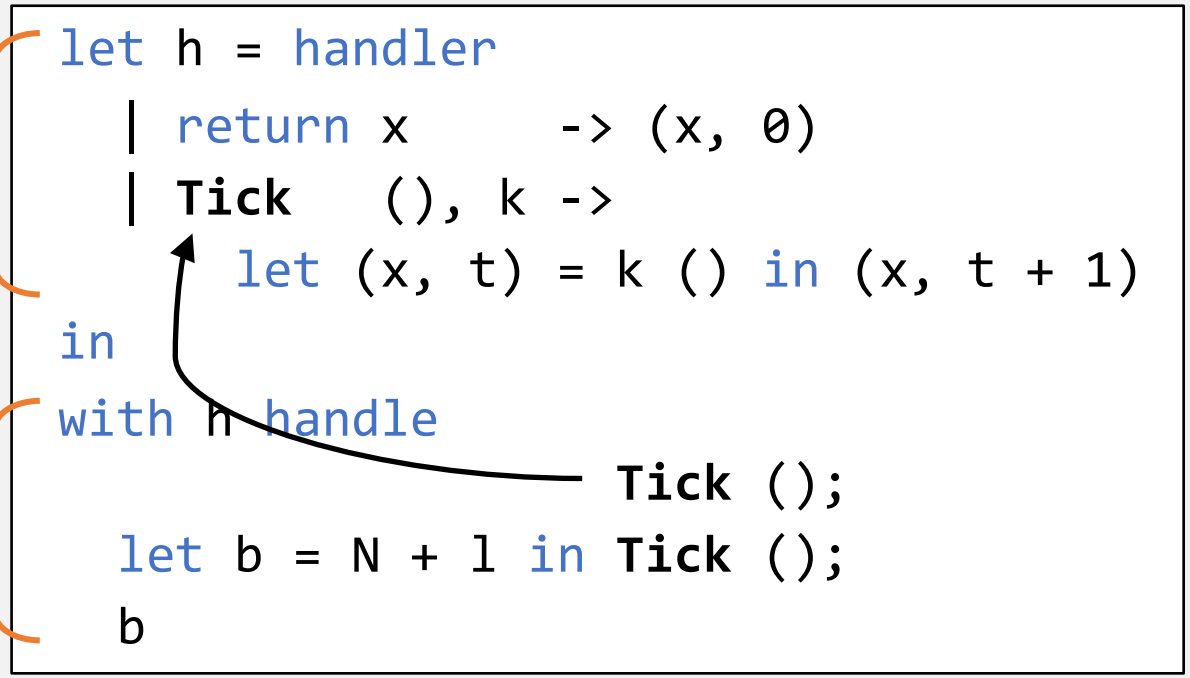
# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

**Handling effects**

Installs handler h to handle **`Tick`** invoked by the term



# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
```

```
with h handle
                                  ;
  let b = N + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
```

```
with h handle
                              ;
  let b = N + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) = k () in (x, t + 1)
```

```
with h handle
                                ;
  let b = N + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) = k () in (x, t + 1)
```

```
with h handle
                                ;
  let b = N + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) = k () in (x, t + 1)
```

```
with h handle
                                  ;
  let b = N + 1 in Tick ();
  b
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  with h handle
                      ()    ;
    let b = N + 1 in Tick ();
    b

in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  with h handle
                          ()   ;
    let b = N + 1 in Tick ();
    b

in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

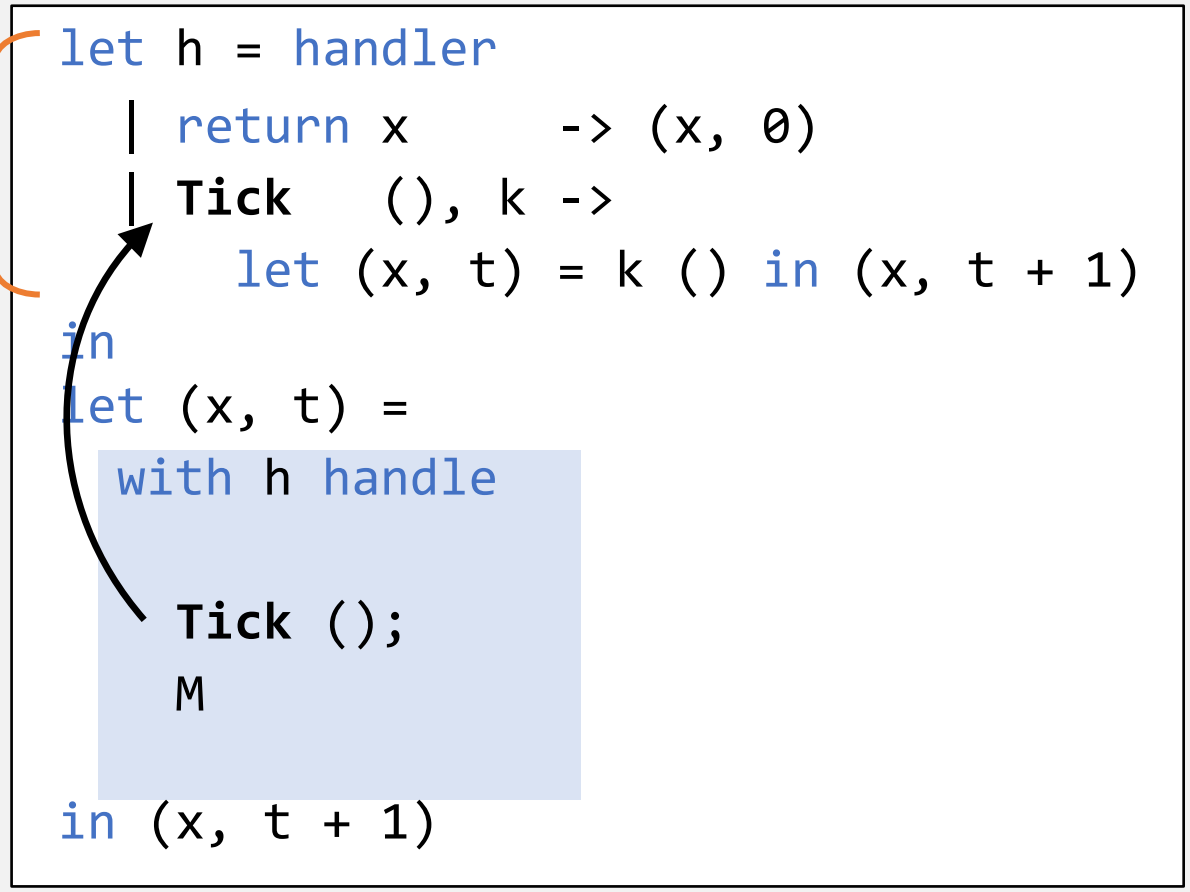
Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  with h handle

    Tick ();
    M

in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked



# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  let (x, t) =
    with h handle
      ();
      M
  in (x, t + 1)
in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  let (x, t) =
    with h handle
      ();
      M
  in (x, t + 1)
in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

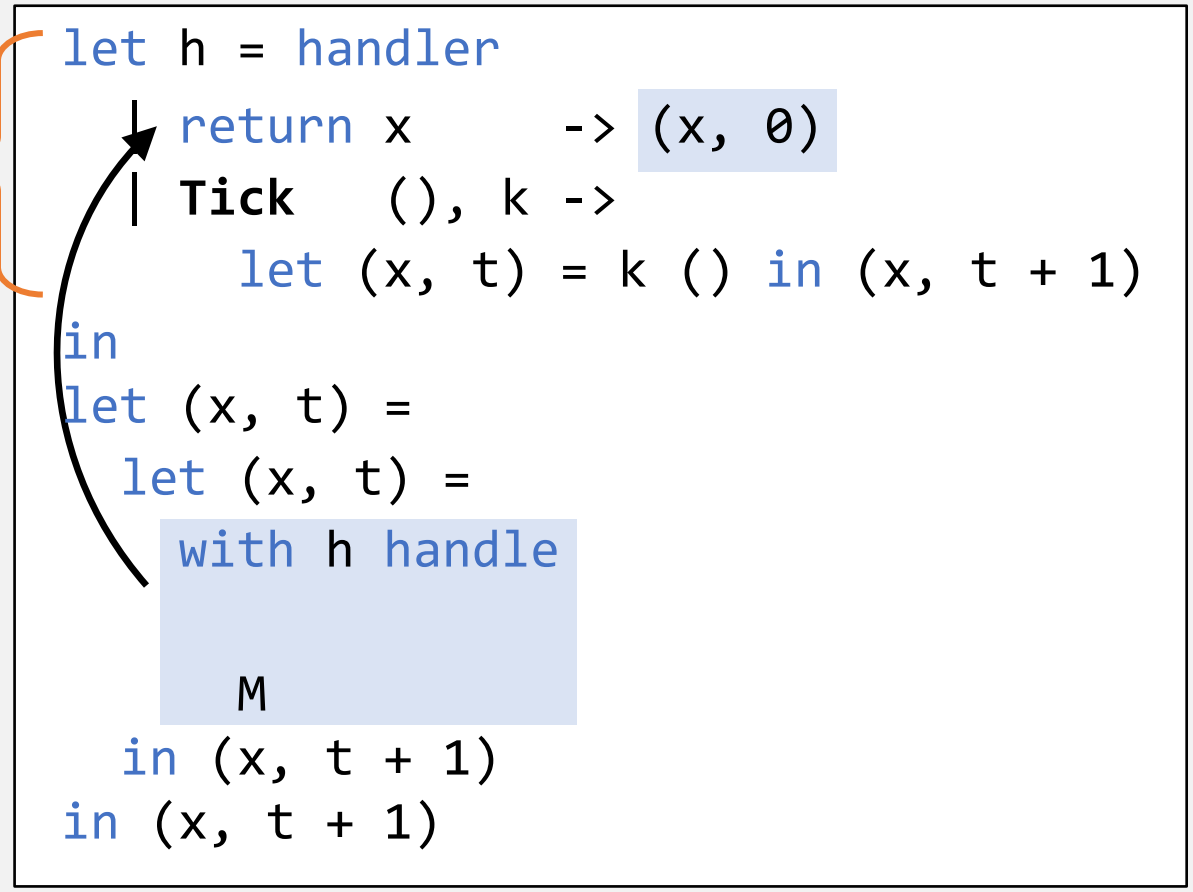
Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  let (x, t) =
    with h handle

      M
  in (x, t + 1)
in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked



# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  let (x, t) =
    (M, 0)


  in (x, t + 1)
in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
let (x, t) =
  (M, 1)



in (x, t + 1)
```

# Example: Cost analysis with effect handlers

**Effect handler**

Defines a handler counting how many times **`Tick`** is invoked

```
let h = handler
  | return x      -> (x, 0)
  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1)
in
(M, 2)
```

**Q. How can we statically reason about the behavior of programs with control effects?**

# A. Answer-type modification (ATM)!

*Draft 4.2 – July 22nd, 1989*

## A Functional Abstraction of Typed Contexts

Olivier Danvy & Andrzej Filinski

**Q. How can we statically reason about the behavior of programs with control effects?**

# A. Answer-type modification (ATM)!

... + typing mechanism adequate for properties to be verified

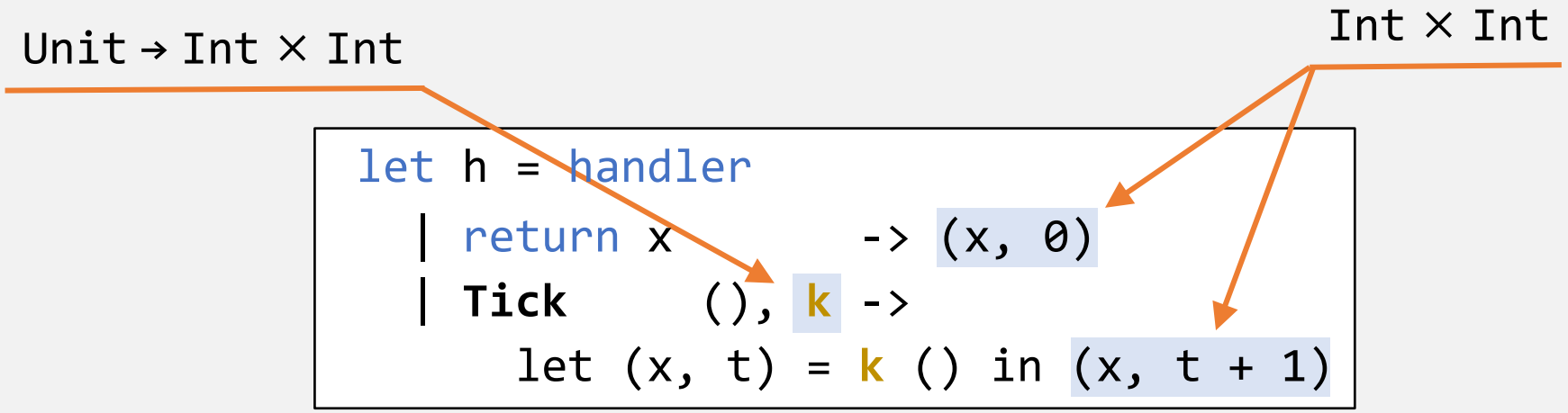
- Refinement types for functional correctness [Kawamata+ '24]

**{ x:B | ϕ }**

- Denoting a set of values of base type B satisfying formula φ
- E.g., 4 : { x:int | x mod 2 = 0 }, [1; 2; 3] : { x:int list | len(x) > 0 }

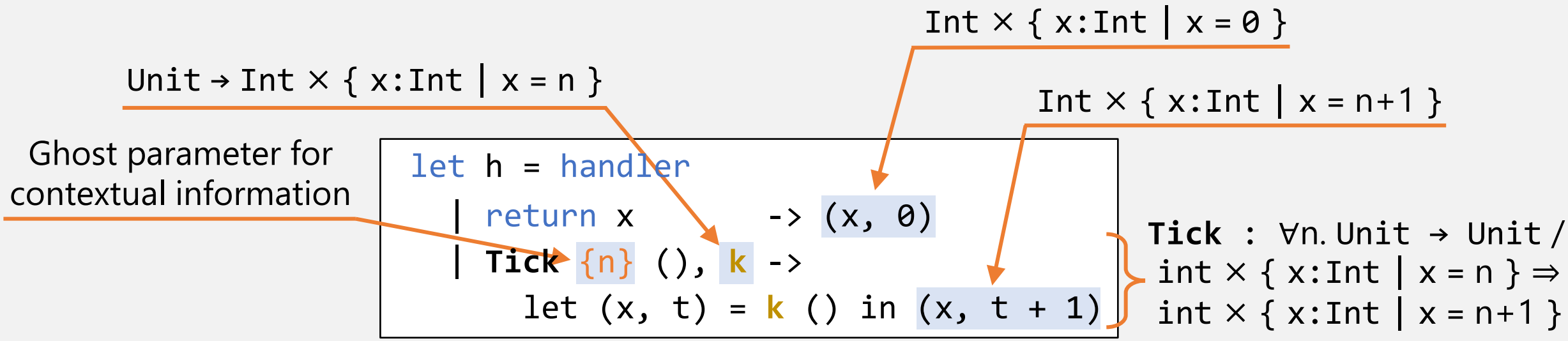
# Answer types

- CPS view: $[\![e]\!] : ([\![\tau]\!] \to \alpha) \to \alpha$ with answer type $\alpha$
- Direct-style view
  - Return types of captured continuations
  - ≈ Types of clauses of effect handlers (continuation delimiters)



# Answer-type modification (ATM)

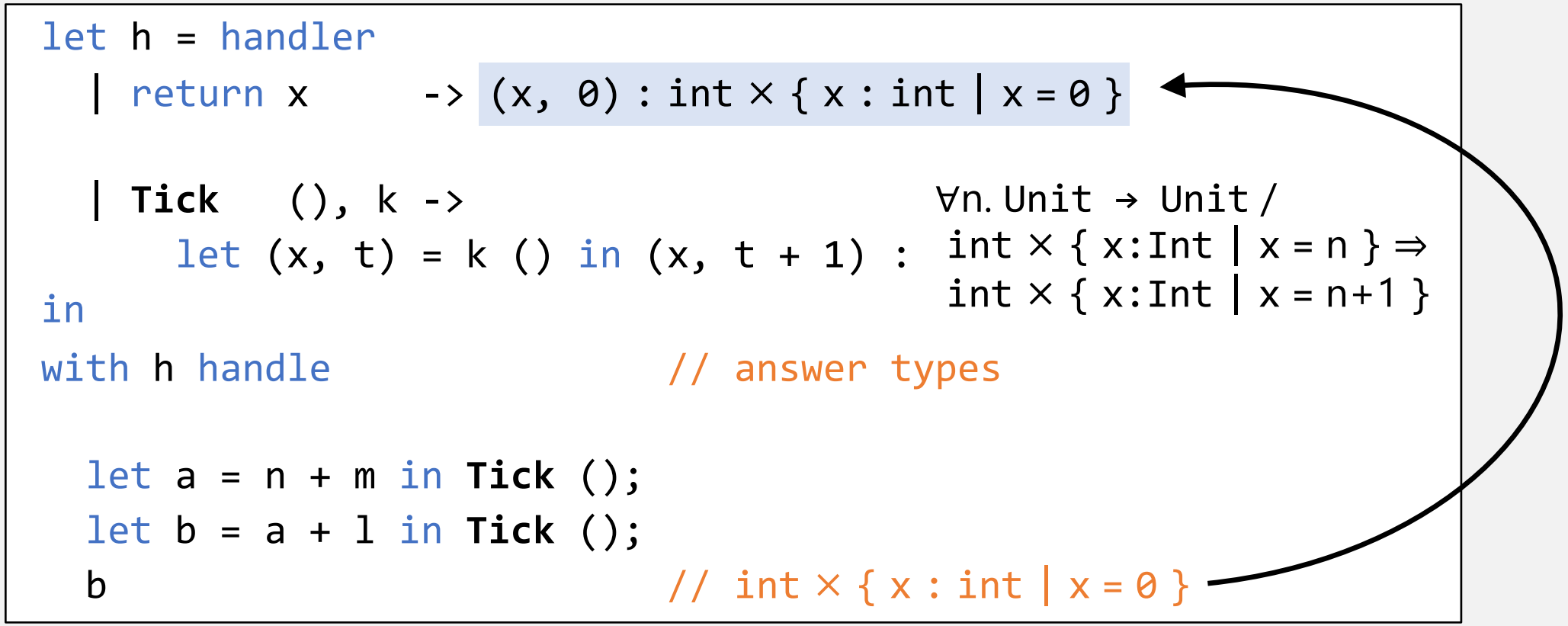
- Allows two kinds of answer types to be different
- CPS view: $[\![e]\!] : ([\![T]\!] \to \alpha) \to \beta$



# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->                              ∀n. Unit → Unit /
      let (x, t) = k () in (x, t + 1) :  int × { x:Int | x = n } ⇒
                                         int × { x:Int | x = n+1 }
in
with h handle                  // answer types

  let a = n + m in Tick ();
  let b = a + 1 in Tick ();
  b
```

# Reasoning by ATM

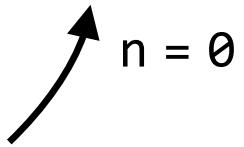


# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->                              ∀n. Unit → Unit /
      let (x, t) = k () in (x, t + 1) :           int × { x:Int | x = n } ⇒
                                                  int × { x:Int | x = n+1 }
in
with h handle                    // answer types

  let a = n + m in Tick ();
  let b = a + 1 in Tick (); // int × { x : int | x = 0 }
  b                         // int × { x : int | x = 0 }
```

# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->                                    ∀n. Unit → Unit /
      let (x, t) = k () in (x, t + 1) :                 int × { x:Int | x = n } ⇒
in                                                      int × { x:Int | x = n+1 }
with h handle                  // answer types

  let a = n + m in Tick (); // int × { x : int | x = 1 }
  let b = a + 1 in Tick (); // int × { x : int | x = 0 }
  b                         // int × { x : int | x = 0 }
```



# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1) :
in
with h handle                    // answer types
                                 // int × { x : int | x = 2 }
  let a = n + m in Tick ();      // int × { x : int | x = 1 }
  let b = a + 1 in Tick ();      // int × { x : int | x = 0 }
  b                              // int × { x : int | x = 0 }
```

∀n. Unit → Unit /
int × { x:Int | x = n } ⇒
int × { x:Int | x = n+1 }

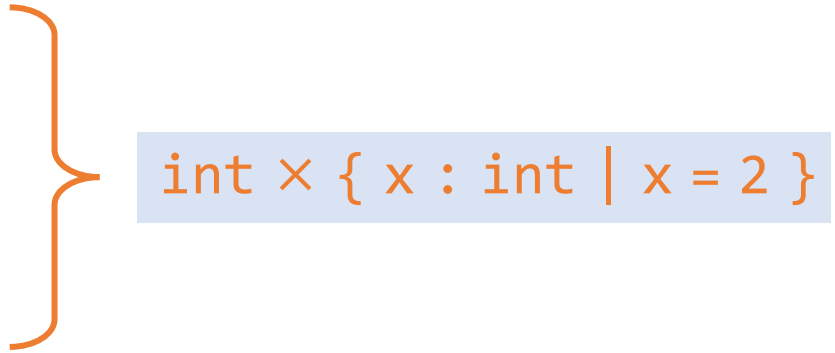
n = 1

# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1) :
in
with h handle                    // answer types
                                 // int × { x : int | x = 2 }
  let a = n + m in Tick (); // int × { x : int | x = 1 }
  let b = a + 1 in Tick (); // int × { x : int | x = 0 }
  b                          // int × { x : int | x = 0 }
```

# Reasoning by ATM

```
let h = handler
  | return x      -> (x, 0) : int × { x : int | x = 0 }

  | Tick   (), k ->
      let (x, t) = k () in (x, t + 1) :
in
with h handle

  let a = n + m in Tick ();
  let b = a + l in Tick ();
  b
```



# Example: Typestate

```
let h = handler
  | return x     -> … : CLOSE
  | Open   x, k -> … : … / OPEN  ⇒ CLOSE
  | Read   x, k -> … : … / OPEN  ⇒ OPEN
  | Write  x, k -> … : … / OPEN  ⇒ OPEN
  | Close  x, k -> … : … / CLOSE ⇒ OPEN
in
with h handle
  Open (); Write (); Close ();
  Read ();
```

can be rejected as expected

# Other applications

- Temporal verification [Sekiyama & Unno '23]
  - Verifying trace properties for both safety and liveness
  - Equipped with effect systems
- Higher-order model checking (HOMC) [Sekiyama+ '24, '25]
  - HOMC is an extension of model checking to HO programs [Ong '09]
  - Effect handlers make HOMC undecidable in general
  - ATM can make HOMC decidable by restricting the usage of continuations
  - Will be presented on Oct 18 (Sat) at SPLASH

# Open questions

- ATM for other control operators
  - Multi-prompt (tagged) control operators
    - cupto [Gunter+ '95]
    - Lexical effect handlers [Biernacki+ '19,'20; Zhang & Meyers '19; Brachthäuser+ '20]
  - Shallow / bidirectional / scoped / selection effect handlers, and others
- Relationship to predicate transformer
  - Known that CPS semantics is closely relevant to predicate transformer [Polak '81; Ahman+ '17; Kura '23]
  - ATM is based on CPS transformation
  - What's a relationship between ATM and predicate transformer?

# Takeaway

- Answer-type modification is an effective tool to reason about higher-order programs with control effects
- It can lift type-based reasoning for languages *without* control effects to those *with* them

**A Functional Abstraction of Typed Contexts**

Olivier Danvy & Andrzej Filinski