Refinement Types and Higher-Order Model Checking for Algebraic Effects and Handlers

Hiroshi Unno

Tohoku University, Japan

This talk is based on the following papers:

1. Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. *Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers*, POPL 2024.
2. Taro Sekiyama and Hiroshi Unno. *Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification*, OOPSLA 2024

Algebraic Effects and Handlers

- A mechanism to structure programs with effects in a modular way
- Can represent many kinds of effects via **delimited continuations**
  - exception, state, I/O, nondeterminism, concurrency, etc.

```
handle
  Set 1; let a1 = Get () in
  Set 2; let a2 = Get () in
  a1 + a2
with
  | return x  -> λs. x
  | Set  v, k -> λs. k () v
  | Get (), k -> λs. k  s s
```

```
handle
  if Decide () then 1 else 2
with
  | return x      -> x
  | Decide (), k -> k true + k false
```

operation

delimited continuation

handler

- Have been adopted in OCaml 5, etc.

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()


with h handle (Tick (); Tock ())
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

delimited continuation

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

delimited continuation

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())  ⟶  1 :: with h handle ((); Tock ())
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())      ⟶  1 :: with h handle ((); Tock ())

                                      ⟶  1 :: with h handle (Tock ())
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

delimited continuation

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())      ⟶  1 :: with h handle ((); Tock ())

                                      ⟶  1 :: with h handle (Tock ())
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

delimited continuation

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())      ⟶  1 :: with h handle ((); Tock ())

                                      ⟶  1 :: with h handle (Tock ())

                                      ⟶  1 :: 0 :: with h handle ()
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())      ⟶  1 :: with h handle ((); Tock ())

                                      ⟶  1 :: with h handle (Tock ())

                                      ⟶  1 :: 0 :: with h handle ()
```

Algebraic Effects and Handlers

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

This talk focuses on deep handlers

```
with h handle (Tick (); Tock ())      ⟶  1 :: with h handle ((); Tock ())
                                      ⟶  1 :: with h handle (Tock ())
                                      ⟶  1 :: 0 :: with h handle ()
                                      ⟶  1 :: 0 :: []
                                      =  [1; 0]
```

Verification of Programs with Effect Handlers

- Delimited continuations enhance expressivity but introduce complex control flows, posing challenges for program verification
- This talk presents two complementary approaches that addresses the challenges by applying type systems [Danvy+90; Materzok+11] with **Answer Type Modification (ATM)** in different way
  1. Refinement types [Kawamata+ POPL'24]
     - Support functional correctness verification (ongoing extensions to temporal properties)
     - Support infinite data domains (e.g., integers, algebraic data types)
     - Undecidable but automated via Constrained Horn Clause (CHC) solving
  2. Higher-order model checking [Sekiyama and Unno OOPSLA'24]
     - Support modal mu-calculus model checking of effect trees
     - Restricted to finite data domains (e.g., booleans, enum types)
     - Decidable fragment expressible enough to support various effects

Verification of Programs with Effect Handlers

- Delimited continuations enhance expressivity but introduce complex control flows, posing challenges for program verification
- This talk presents two complementary approaches that addresses the challenges by applying type systems [Danvy+90; Materzok+11] with **Answer Type Modification (ATM)** in different way
  1. **Refinement types [Kawamata+ POPL'24]**
     - Support functional correctness verification (ongoing extensions to temporal properties)
     - Support infinite data domains (e.g., integers, algebraic data types)
     - Undecidable but automated via Constrained Horn Clause (CHC) solving
  2. Higher-order model checking [Sekiyama and Unno OOPSLA'24]
     - Support modal mu-calculus model checking of effect trees
     - Restricted to finite data domains (e.g., booleans, enum types)
     - Decidable fragment expressible enough to support various effects

Refinement Types

$$\{x : B \mid \phi\}$$

Formula

Base Type (int, bool etc.)

- Types equipped with predicates
  - E.g. $1 + 2 : \{x : \text{int} \mid x = 3\}$
    $1 + 2 : \{x : \text{int} \mid x > 0\}$
- Often combined with dependent function types $(x : T) \to C$
  - E.g. $\lambda x.\, x + 1 : (x : \text{int}) \to \{y : \text{int} \mid y = x + 1\}$
- Can represent more specific properties
  - A tool of program verification

Contribution

Refinement type system for algebraic effect handlers

$$\vdash \texttt{with h handle (Tick (); Tock ())} : \{z : \text{int list} \mid z = [1; 0]\}$$

- Enables precise verification of algebraic effect handlers
- Can be built on top of existing languages
  - Our implementation for OCaml 5 programs

Challenge

```
let h = handler
| return x -> x
| Tick (), k -> 1 :: k ()
| Tock (), k -> 0 :: k ()
```

The precise types depend on **what effects** occur in **what order**

```
with h handle
  (Tick (); Tock ())
(* ==> [1; 0] *)
```

$: \{z: \text{int list} \mid z = [1; 0]\}$

```
with h handle
  (Tock (); Tick ())
(* ==> [0; 1] *)
```

$: \{z: \text{int list} \mid z = [0; 1]\}$

Challenge

```
let h = handler
| return x -> x
| Tick (), k -> 1 :: k ()
| Tock (), k -> 0 :: k ()
```

$\{z\colon \mathbf{int\ list} \mid z = [0]\}$

$\{z\colon \mathbf{int\ list} \mid z = [\,]\}$

The precise types depend on **what effects** occur in **what order**

```
with h handle
  (Tick (); Tock ())
(* ==> [1; 0] *)
```

$: \{z\colon \mathrm{int\ list} \mid z = [1; 0]\}$

```
with h handle
  (Tock (); Tick ())
(* ==> [0; 1] *)
```

$: \{z\colon \mathrm{int\ list} \mid z = [0; 1]\}$

Challenge

```
let h = handler
| return x -> x
| Tick (), k -> 1 :: k ()
| Tock (), k -> 0 :: k ()
```

$\{z\colon \mathbf{int\ list} \mid z = [\,]\}$

$\{z\colon \mathbf{int\ list} \mid z = [\mathbf{1}]\}$

The precise types depend on **what effects** occur in **what order**

```
with h handle
  (Tick (); Tock ())
(* ==> [1; 0] *)
```

$: \{z\colon \mathrm{int\ list} \mid z = [1; 0]\}$

```
with h handle
  (Tock (); Tick ())
(* ==> [0; 1] *)
```

$: \{z\colon \mathrm{int\ list} \mid z = [0; 1]\}$

Challenge

```
let h = handler
| return x -> x
| Decide (), k -> k true - k false
```

```
with h handle
  (if Decide () then n1 else n2)
(* ==> n1 - n2 *)
```

$: \{z: \text{int} \mid z = n_1 - n_2\}$

The precise type depends on **which branch** returns **what value**

Challenge

```
let h = handler
| return x -> x
| Decide (), k -> k true - k false

with h handle
  (if Decide () then n1 else n2)
(* ==> n1 - n2 *)
```

$\{z: \text{int} \mid z = n_2\}$

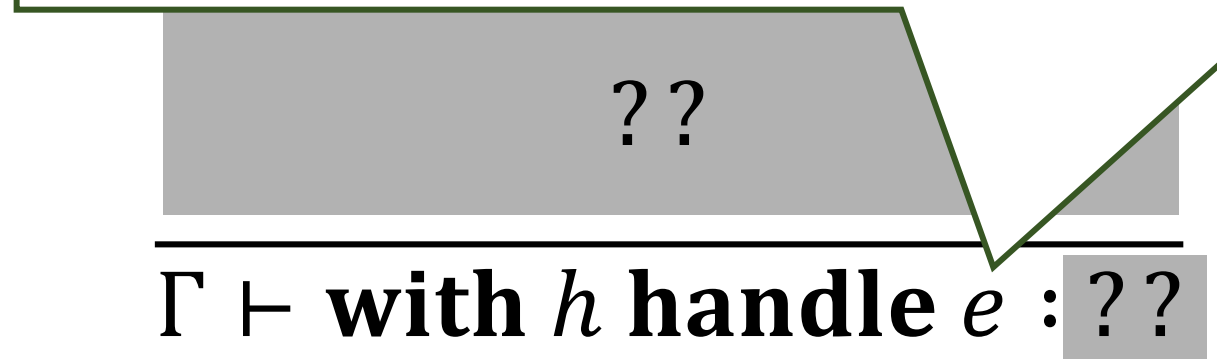
$\{z: \text{int} \mid z = n_1\}$

$: \{z: \text{int} \mid z = n_1 - n_2\}$

The precise type depends on **which branch** returns **what value**

Challenge

Need to track precise types of the contexts (i.e., answer types) of each operation call in e



We adopt

Answer Type Modification

[Danvy+90; Materzok+11]

Answer Types

- Types of **delimited contexts**
  - = return types of **delimited continuations**
  - = types of **the closest outer handling constructs (delimiters)**

$$\mathcal{E}[\ \textbf{with}\ h\ \textbf{handle}\ \mathcal{F}[e]\ ]$$

evaluation context

handler-free evaluation context

Answer type of e = type of $\textbf{with}\ h\ \textbf{handle}\ \mathcal{F}[\ ]$

$$\mathcal{F} ::= [\ ] \mid \textbf{let}\ x = \mathcal{F}\ \textbf{in}\ e$$
$$\mathcal{E} ::= [\ ] \mid \textbf{let}\ x = \mathcal{E}\ \textbf{in}\ e \mid \textbf{with}\ h\ \textbf{handle}\ \mathcal{E}$$

Answer Types in Ordinary Type Systems

value types $T ::= B \mid T \to C$

computation types $C ::= \Sigma \rhd T$

signatures $\Sigma ::= \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi}\}_i$

answer type

$$\frac{\Gamma \vdash e : \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi}\}_i \rhd T_0 \qquad \Gamma, x : T_0 \vdash e_r : \boldsymbol{C} \qquad (\Gamma, x : T_{ai}, k : T_{bi} \to \boldsymbol{C} \vdash e_i : \boldsymbol{C})_i}{\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : \boldsymbol{C}}$$

$$\frac{\Sigma \ni \mathrm{Op} : T_a \twoheadrightarrow T_b \qquad \Gamma \vdash v : T_a}{\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd T_b}$$

$\Gamma \vdash e : C$

Answer Types in Ordinary Type Systems

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

int list

int list

int list

unit → int list

```
⊢ with h handle (Tick();Tock()) : int list
```

Answer Type Modification

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

```
with h handle (Tick (); Tock ())
```

Answer Type Modification

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

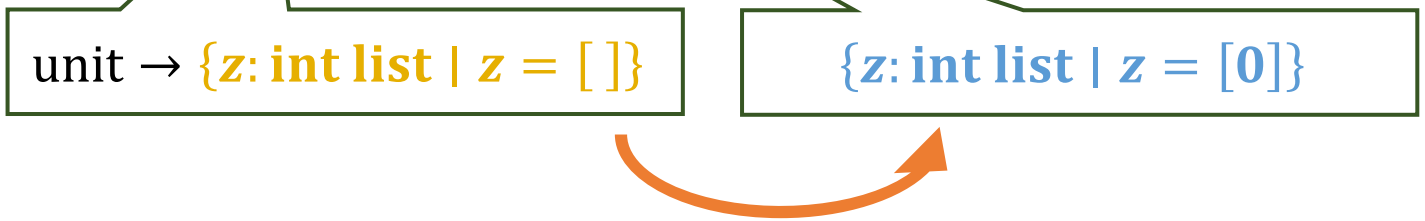
unit → $\{z: \textbf{int list} \mid z = [\,]\}$

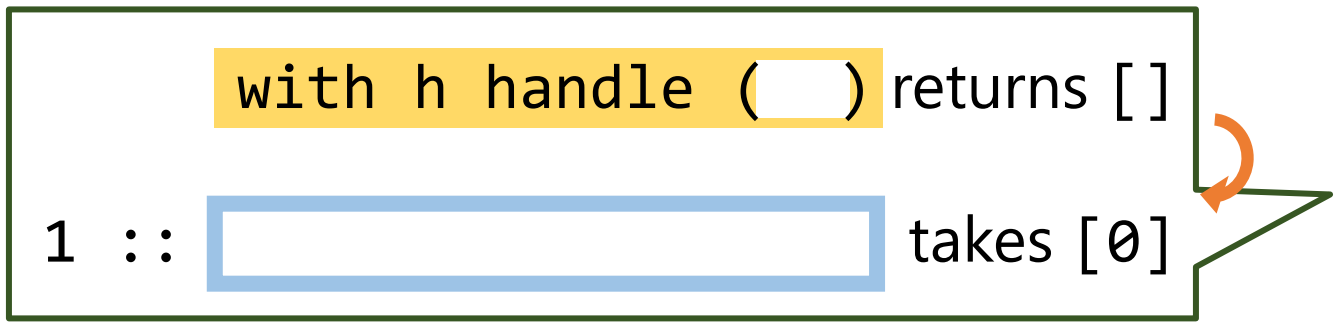
`with h handle ( )` returns `[]`

```
with h handle (Tick (); Tock ())
→ 1 :: with h handle ((); Tock ())
→ 1 :: with h handle (Tock ())
→ 1 :: 0 :: with h handle ()
→ 1 :: 0 :: []
```

Answer Type Modification

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

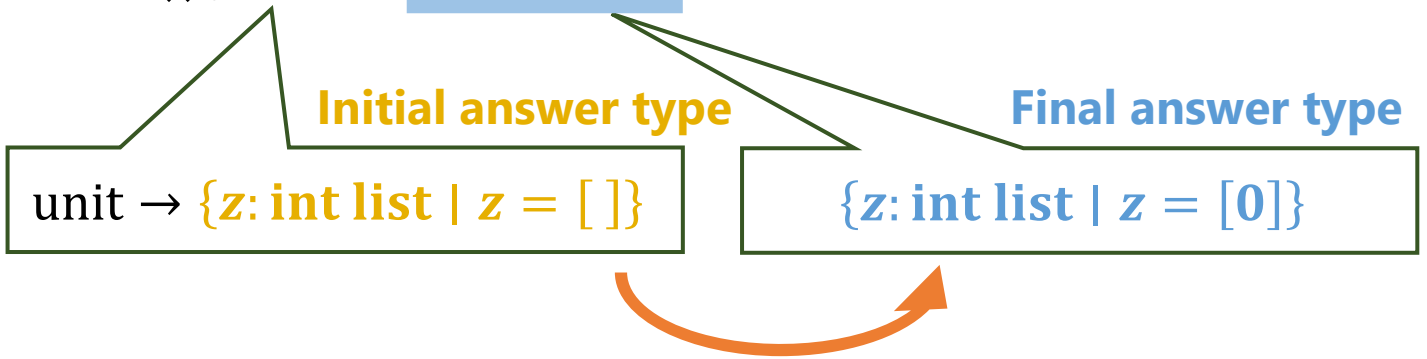





```
with h handle (Tick (); Tock ())
→ 1 :: with h handle ((); Tock ())
→ 1 :: with h handle (Tock ())
→ 1 :: 0 :: with h handle ()
→ 1 :: 0 :: []
```

Answer Type Modification

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```



Answer type modification (ATM)

Answer refinement modification (ARM)

```
with h handle (Tick (); Tock ())
→ 1 :: with h handle ((); Tock ())
→ 1 :: with h handle (Tock ())
→ 1 :: 0 :: with h handle ()
→ 1 :: 0 :: []
```

Answer Type Modification

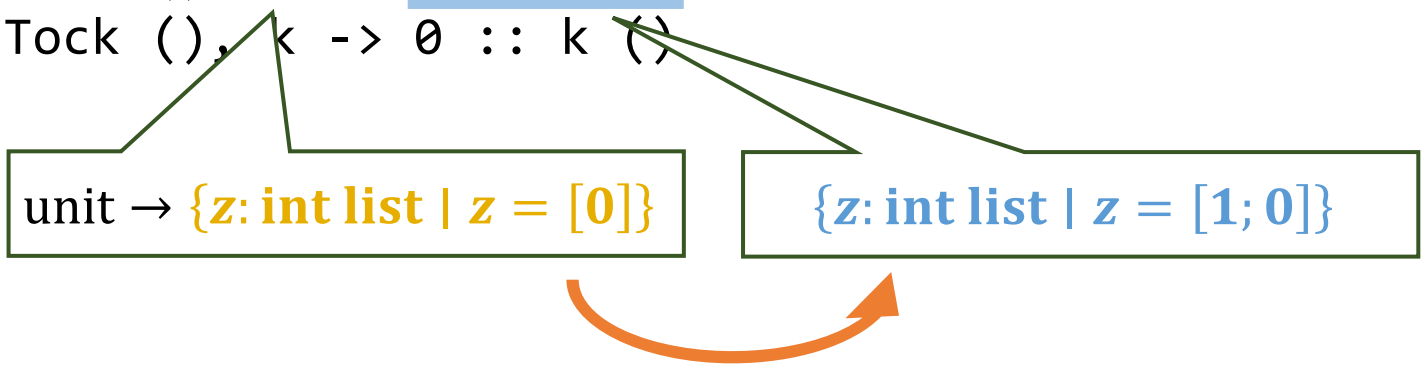
```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

$\text{unit} \to \{z\colon \textbf{int list} \mid z = [0]\}$

```
with h handle (Tick (); Tock ())
→ 1 :: with h handle ((); Tock ())
→ 1 :: with h handle (Tock ())
→ 1 :: 0 :: with h handle ()
→ 1 :: 0 :: []
```

Answer Type Modification

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```




```
with h handle (Tick (); Tock ())
→ 1 :: with h handle ((); Tock ())
→ 1 :: with h handle (Tock ())
→ 1 :: 0 :: with h handle ()
→ 1 :: 0 :: []
```

Type Syntax

value types $T ::= \{x : B \mid \phi\} \mid T \to C$

computation types $C ::= \Sigma \rhd T \,/\, \boldsymbol{S}$

signatures $\Sigma ::= \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \,/\, \boldsymbol{C_{1i}} \Rightarrow \boldsymbol{C_{2i}}\}_i$

control effects $\boldsymbol{S} ::= \square \mid \boldsymbol{C_1} \Rightarrow \boldsymbol{C_2}$

The answer type does not change

The initial answer type $\boldsymbol{C_1}$ is modified to the final answer type $\boldsymbol{C_2}$

Typing Rules

- **Example**

$$\Sigma = \{\ \text{Tick}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [0]\} \Rightarrow \{z: \text{int list} \mid z = [1;0]\},$$
$$\text{Tock}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [\,]\} \Rightarrow \{z: \text{int list} \mid z = [0]\}\ \}$$

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()

with h handle (

  Tick ();  : Σ ▷ unit / {z: int list | z = [0]} ⇒ {z: int list | z = [1;0]}

  Tock ()   : Σ ▷ unit / {z: int list | z = [ ]} ⇒ {z: int list | z = [0]}

)
```

$$\frac{\Gamma \vdash e_1 : \Sigma \rhd \text{unit} \,/\, \boldsymbol{C} \Rightarrow \boldsymbol{C_F} \qquad \Gamma \vdash e_2 : \Sigma \rhd T \,/\, \boldsymbol{C_I} \Rightarrow \boldsymbol{C}}{\Gamma \vdash e_1;\ e_2 : \Sigma \rhd T \,/\, \boldsymbol{C_I} \Rightarrow \boldsymbol{C_F}}$$

Typing Rules

- **Example**

$$\Sigma = \{\ \text{Tick}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [0]\} \Rightarrow \{z: \text{int list} \mid z = [1;0]\},$$
$$\text{Tock}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [\,]\} \Rightarrow \{z: \text{int list} \mid z = [0]\}\ \}$$

```
let h = handler
  | return x -> [] : {z: int list | z = []}
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()

with h handle (
  Tick ();
  Tock ()
)
```

$$\frac{\Gamma \vdash e_1 : \Sigma \rhd \text{unit} \,/\, \boldsymbol{C} \Rightarrow \boldsymbol{C_F} \qquad \Gamma \vdash e_2 : \Sigma \rhd T \,/\, \boldsymbol{C_I} \Rightarrow \boldsymbol{C}}{\Gamma \vdash e_1;\ e_2 : \Sigma \rhd T \,/\, \boldsymbol{C_I} \Rightarrow \boldsymbol{C_F}}$$

$$: \Sigma \rhd \text{unit} \,/\, \{z: \text{int list} \mid z = [\,]\} \Rightarrow \{z: \text{int list} \mid z = [1;0]\}$$

Typing Rules

- **Example**

$$\Sigma = \{\, \text{Tick}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [0]\} \Rightarrow \{z: \text{int list} \mid z = [1;0]\},$$
$$\text{Tock}: \text{unit} \twoheadrightarrow \text{unit} \,/\, \{z: \text{int list} \mid z = [\,]\} \Rightarrow \{z: \text{int list} \mid z = [0]\} \,\}$$

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()
```

```
with h handle (

  Tick ();

  Tock ()

)
```

$: \{z: \text{int list} \mid z = [1;0]\}$

Typing Rules

$$\frac{\Gamma \vdash e : \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \,/\, C_{1i} \Rightarrow C_{2i}\}_i \rhd T_0 \,/\, C_1 \Rightarrow C_2 \qquad \Gamma, x : T_0 \vdash e_r : C_1 \quad (\Gamma, x : T_{ai}, k : T_{bi} \to C_{1i} \vdash e_i : C_{2i})_i}{\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : C_2}$$

$$\frac{\Sigma \ni \mathrm{Op} : T_a \twoheadrightarrow T_b \,/\, C_1 \Rightarrow C_2 \qquad \Gamma \vdash v : T_a}{\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd T_b \,/\, C_1 \Rightarrow C_2}$$

$\boxed{\Gamma \vdash e : C}$

Typing Rules

$$\frac{\Gamma \vdash e : \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \;/\; C_{1i} \Rightarrow C_{2i}\}_i \rhd T_0 \;/\; C_1 \Rightarrow C_2 \qquad \Gamma, x : T_0 \vdash e_r : C_1 \qquad (\Gamma, x : T_{ai}, k : T_{bi} \to C_{1i} \vdash e_i : C_{2i})_i}{\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : C_2}$$

$$\frac{\Sigma \ni \mathrm{Op} : T_a \twoheadrightarrow T_b \;/\; C_1 \Rightarrow C_2 \qquad \Gamma \vdash v : T_a}{\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd T_b \;/\; C_1 \Rightarrow C_2}$$

$\Gamma \vdash e : C$

Typing Rules

$$\frac{\Gamma \vdash e : \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \;/\; C_{1i} \Rightarrow C_{2i}\}_i \rhd T_0 \;/\; C_1 \Rightarrow C_2 \qquad \Gamma, x : T_0 \vdash e_r : C_1 \qquad (\Gamma, x : T_{ai}, k : T_{bi} \to C_{1i} \vdash e_i : C_{2i})_i}{\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : C_2}$$

$$\frac{\Sigma \ni \mathrm{Op} : T_a \twoheadrightarrow T_b \;/\; C_1 \Rightarrow C_2 \qquad \Gamma \vdash v : T_a}{\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd T_b \;/\; C_1 \Rightarrow C_2}$$

$\Gamma \vdash e : C$

Extension 1: Predicate Polymorphism

value types $T ::= B \mid T \to C$

computation types $C ::= \Sigma \rhd T \,/\, S$

signatures $\Sigma ::= \{\mathrm{Op}_i : \forall X : \bar{B}.\, T_{ai} \twoheadrightarrow T_{bi} \,/\, C_{1i} \Rightarrow C_{2i}\}_i$

control effects $S ::= \square \mid C_1 \Rightarrow C_2$

$$\Gamma \vdash e : \{\mathrm{Op}_i : \forall X : \bar{B}.\, T_{ai} \twoheadrightarrow T_{bi} \,/\, C_{1i} \Rightarrow C_{2i}\}_i \rhd T_0 \,/\, C_1 \Rightarrow C_2 \qquad \Gamma, x : T_0 \vdash e_r : C_1 \quad (\Gamma, X : \bar{B}, x : T_{ai}, k : T_{bi} \to C_{1i} \vdash e_i : C_{2i})_i$$

$$\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : C_2$$

$$\Sigma \ni \mathrm{Op} : \forall X : \bar{B}.\, T_a \twoheadrightarrow T_b \,/\, C_1 \Rightarrow C_2 \qquad \Gamma \vdash v : T_a \qquad \Gamma \vdash Pred : \bar{B}$$

$$\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd (T_b \,/\, C_1 \Rightarrow C_2)[Pred/X]$$

$\Gamma \vdash e : C$

Extension 1: Predicate Polymorphism

- **Example**

```
let h = handler
  | return x -> []
  | Tick (), k -> 1 :: k ()
  | Tock (), k -> 0 :: k ()

with h handle (
  Tick ();
  Tock ();
  Tick ();
  Tock ()
)
```

$$\Sigma = \{\ \mathrm{Tick} : \forall X : (\mathrm{int\ list}).\,\mathrm{unit} \twoheadrightarrow \mathrm{unit} \,/\, \{z\colon \mathrm{int\ list} \mid X(z)\} \Rightarrow \{z\colon \mathrm{int\ list} \mid \forall l.\, X(l) \Rightarrow z = 1 :: l\},\ \mathrm{Tock} : \forall Y : (\mathrm{int\ list}).\,\mathrm{unit} \twoheadrightarrow \mathrm{unit} \,/\, \{z\colon \mathrm{int\ list} \mid Y(z)\} \Rightarrow \{z\colon \mathrm{int\ list} \mid \forall l.\, Y(l) \Rightarrow z = 0 :: l\}\ \}$$

$\{z\colon \mathrm{int\ list} \mid z = [1;0;1;0]\}$

$\{z\colon \mathrm{int\ list} \mid z = [0;1;0]\}$

$\{z\colon \mathrm{int\ list} \mid z = [1;0]\}$

$\{z\colon \mathrm{int\ list} \mid z = [0]\}$

$\{z\colon \mathrm{int\ list} \mid z = [\,]\}$

$X \mapsto \lambda z.\, z = [0;1;0]$

$Y \mapsto \lambda z.\, z = [1;0]$

$X \mapsto \lambda z.\, z = [0]$

$Y \mapsto \lambda z.\, z = [\,]$

Extension 2: Dependent Types for Continuations

value types $T ::= B \mid T \to C$

computation types $C ::= \Sigma \rhd T \,/\, S$

signatures $\Sigma ::= \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \,/\, y.\, C_{1i} \Rightarrow C_{2i}\}_i$

control effects $S ::= \square \mid y.\, C_1 \Rightarrow C_2$

$$\frac{\Gamma \vdash e : \{\mathrm{Op}_i : T_{ai} \twoheadrightarrow T_{bi} \,/\, y.\, C_{1i} \Rightarrow C_{2i}\}_i \rhd T_0 \,/\, C_1 \Rightarrow C_2 \qquad \Gamma, x : T_0 \vdash e_r : C_1 \qquad (\Gamma, x : T_{ai}, k : (y : T_{bi}) \to C_{1i} \vdash e_i : C_{2i})_i}{\Gamma \vdash \mathbf{with}\ \{x \mapsto e_r, (\mathrm{Op}_i\ x, k \mapsto e_i)_i\}\ \mathbf{handle}\ e : C_2}$$

$$\frac{\Sigma \ni \mathrm{Op} : T_a \twoheadrightarrow T_b \,/\, y.\, C_1 \Rightarrow C_2 \qquad \Gamma \vdash v : T_a}{\Gamma \vdash \mathrm{Op}\ v : \Sigma \rhd (T_b \,/\, y.\, C_1 \Rightarrow C_2)}$$

$\Gamma \vdash e : C$

Extension 2: Dependent Types for Continuations

- **Example**

```
let h = handler
| return x -> x
| Decide (), k -> k true - k false
```

$(b\colon \text{bool}) \to \{z\colon \text{int} \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\}$

$\Sigma = \{\text{Decide}\colon \text{unit} \twoheadrightarrow \text{bool} \,/\, b.\{z\colon \text{int} \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\}$
$\Rightarrow \{z\colon \text{int} \mid z = n_1 - n_2\}\}$

```
with h handle (
```

$\{z\colon \text{int} \mid z = n_1 - n_2\}$

```
  let b = Decide () in
```

$\{z\colon \text{int} \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\}$

```
  if b then n1 else n2
)
```

Automatic Type Inference via CHC Solving

1. Obtain an ML-typed AST using OCaml's compiler library
2. Infer refinement-free operation signatures and control effects
3. Generate refinement constraints for the program and its specification as Constrained Horn Clauses (CHCs) [Unno and Kobayashi '09]
4. Solve CHCs to check if the program satisfies the specification

The POPL'24 Paper also Includes:

- A proof of the type soundness
- Implementation in **RCaml**
  - On top of OCaml 5.0
  - Experiments on 50+ examples

https://github.com/hiroshi-unno/coar

- Another way of verification via CPS transformation
  - Removal of handlers + verification with existing refinement type systems
  - Bidirectionally type-preserving CPS transformation
    - Needs type annotations on source programs

Verification of Programs with Effect Handlers

- Delimited continuations enhance expressivity but introduce complex control flows, posing challenges for program verification
- This talk presents two complementary approaches that addresses the challenges by applying type systems [Danvy+90; Materzok+11] with **Answer Type Modification (ATM)** in different way
  1. Refinement types [Kawamata+ POPL'24]
     - Support functional correctness verification (ongoing extensions to temporal properties)
     - Support infinite data domains (e.g., integers, algebraic data types)
     - Undecidable but automated via Constrained Horn Clause (CHC) solving
  2. **Higher-order model checking [Sekiyama and Unno OOPSLA'24]**
     - Support modal mu-calculus model checking of effect trees
     - Restricted to finite data domains (e.g., booleans, enum types)
     - Decidable fragment expressible enough to support various effects

Model Checking

System

- Programs
- Graphs
- State transition systems

Property

- Temporal logic formulas
- Automata

$$M \stackrel{?}{\models} \phi$$

Model Checking Problem

Whether M satisfies ϕ?

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

System

HO programs yielding trees

- HO recursion schemes (tree grammars with HO funcs)
- PCF terms with finite ground types (generating Böhm trees)

$$M \stackrel{?}{\models} \phi$$

Property

Predicates over trees

- MSO formulas
- Modal μ-calculus formulas
- Alternating parity tree automata

HOMC Problem

Whether the tree generated by M satisfies ϕ?

- E.g. assertion checking, (non-)termination verification, and general branching-time temporal safety and liveness verification problems

Higher-Order Model Checking (HOMC)

[Ong, LICS'06; Kobayashi, JACM'13]

System

HO programs yielding trees

- HO recursion schemes (tree grammars with HO funcs)
- PCF terms with finite ground types (generating Böhm trees)

$$M \stackrel{?}{\models} \phi$$

Property

Predicates over trees

- MSO formulas
- Modal μ-calculus formulas
- Alternating parity tree automata

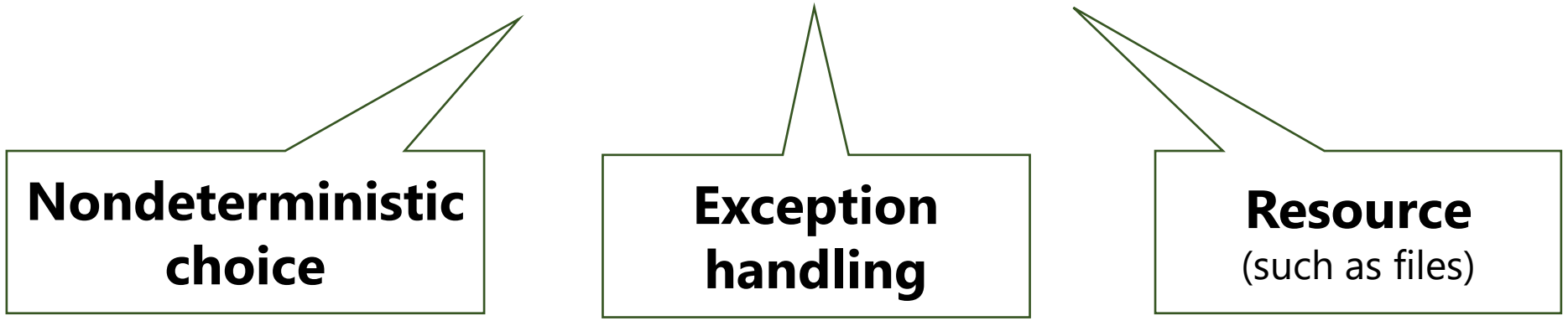
☛ **Data domains need to be finite**

HOMC Problem is Decidable 

Whether the tree generated by M satisfies ϕ?

- E.g. assertion checking, (non-)termination verification, and general branching-time temporal safety and liveness verification problems

HOMC for Effects



[Kobayashi, POPL'09] [Sato+, PEPM'13] [Kobayashi, POPL'09]

What about other effects?

E.g., mutable store, I/O, backtracking, coroutines, etc.

Can express global store, I/O,
nondeterministic choice, etc.

HOMC
+
Algebraic Effects & Effect Handlers

Can express exceptions,
local store, backtracking, etc.
by using **delimited continuations**

[Dal Lago and Ghyselen, POPL'24]

Can express global store, I/O, nondeterministic choice, etc.

Can express exceptions, local store, backtracking, etc. by using **delimited continuations**

HOMC + Algebraic Effects & Effect Handlers is Undecidable

[Dal Lago and Ghyselen, POPL'24]

Idea: encoding natural numbers through delimited continuations

Is there a useful fragment where HOMC is decidable?

Our Contributions

GitHub page of **EffCaml**

- A new class of HO programs with effect handlers where
  - **HOMC is decidable**
  - **Effect handlers are expressive enough to implement various effects**

Key idea : restrict the use of delimited continuations through an Answer-Type Modification (ATM) type system [Danvy+90; Materzok+11]

- A CPS-transformation to obtain terms in a ***decidable*** variant of PCF with finite ground types and effect operations ***without handlers***
  - Crucial both theoretically and practically
- Implementation of a model checker **EffCaml** for the new class

ct Handlers [Dal Lago and Ghyselen, POPL'24]

Higher-order functions
+ General recursion
+ Finite data domains (like Bool)

- Target langua: HEPCF
  - HEPCF = A variant of PCF + **Effect Operations + Handlers**

$$\text{Terms } M ::= \cdots \mid \mathrm{op}(V, x.M) \mid \text{with } H \text{ handle } M$$
$$\text{Handlers } H ::= \{\, \text{return } x \to M \} \uplus \{\, \mathrm{op}_i(x_i, k_i) \to M_i \}_i$$

$$\text{let } x = \mathrm{op}(V, y.M) \text{ in } M_2 \longrightarrow \mathrm{op}(V, y.\text{let } x = M \text{ in } M_2)$$
$$\text{with } H \text{ handle } \mathrm{op}(V, y.M) \longrightarrow M'[x \mapsto V][k \mapsto \lambda y.\text{with } H \text{ handle } M] \quad (\text{if } op(x,k) \to M' \in H)$$

M' can access to the delimited continuation via k

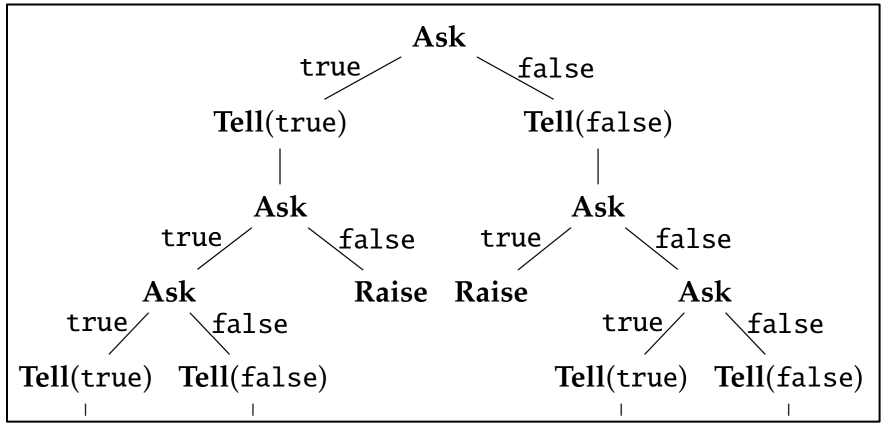
HOMC for Effect Handlers [Dal Lago and Ghyselen, POPL'24]

- Target language: HEPCF
  - HEPCF = A variant of PCF + **Effect Operations + Handlers**
  - Equipped with effect tree semantics
    - The generated trees comprise **unhandled** operations as well as arguments and returns values of the operations

HEPCF term

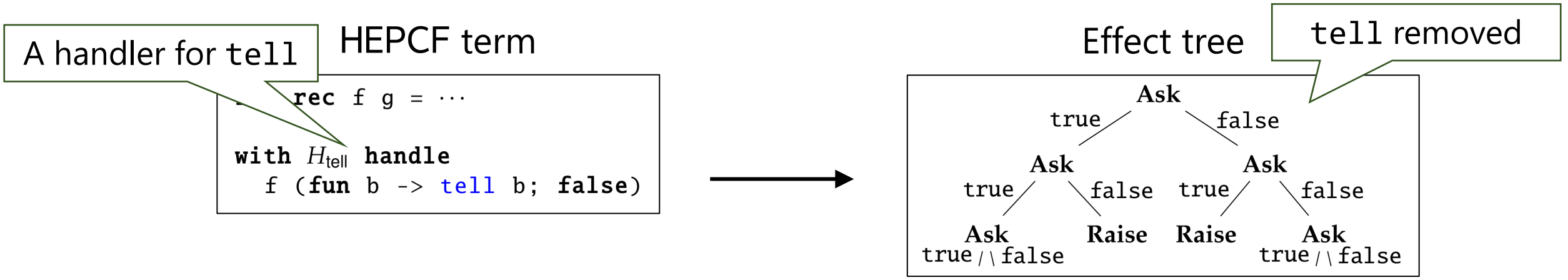
```
let rec f g =
  let b1 = ask x in
  if g b1 then ()
  else
    let b2 = ask x in
    if b1 = b2 then f g
    else raise ()

f (fun b -> tell b; false)
```

Effect tree



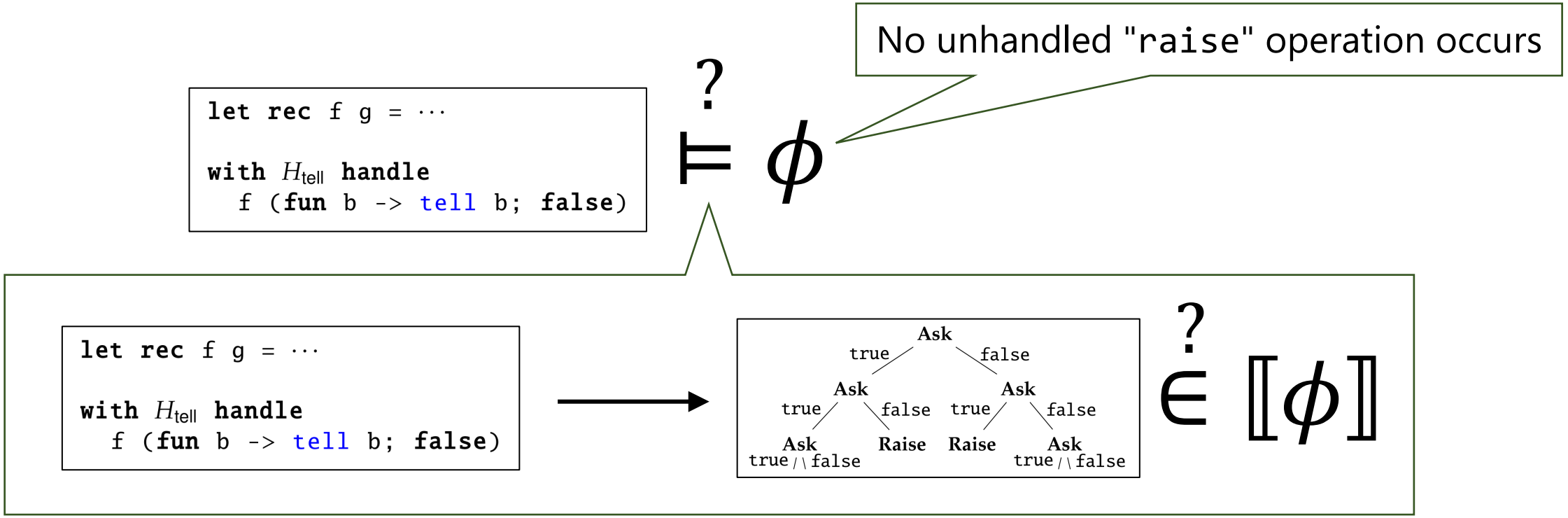
HOMC for Effect Handlers [Dal Lago and Ghyselen, POPL'24]

- Target language: HEPCF
  - HEPCF = A variant of PCF + **Effect Operations + Handlers**
  - Equipped with effect tree semantics
    - The generated trees comprise **unhandled** operations as well as arguments and returns values of the operations
    - Handlers "fold" over effect trees



HOMC for Effect Handlers [Dal Lago and Ghyselen, POPL'24]

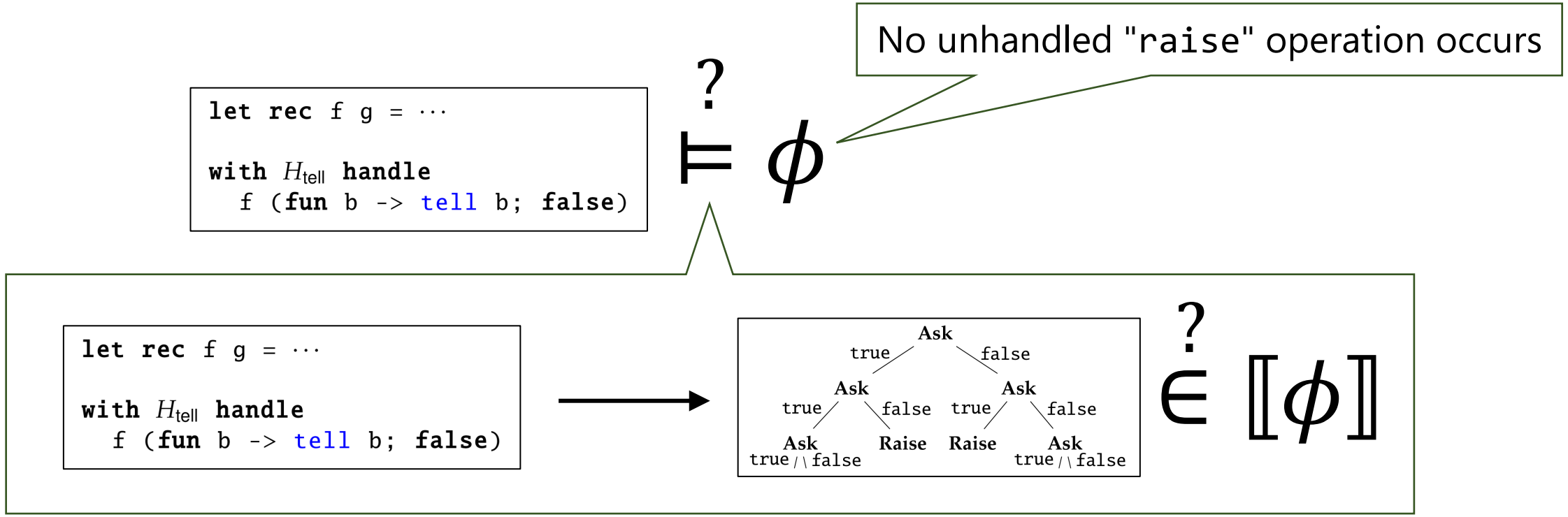
- Model checking



HOMC for Effect Handlers

Reminder: infinite data domains makes the HOMC undecidable

- Model checking is **undecidable** 😕 for MSO formulas ϕ because **effect handlers can encode natural numbers**



Encoding of Natural Numbers [Dal Lago and Ghyselen, POPL'24]

n calls to operation succ

$$[\![n]\!] = \lambda x.\, \mathrm{succ}(); \cdots; \mathrm{succ}(); x$$

$$[\![\mathrm{case}(V; 0 \mapsto M_0; \mathrm{succ}(x) \mapsto M_1]\!] = \text{with } H \text{ handle } [\![V]\!]\ ()$$
$$\text{where H} = \{\mathrm{return}\ \_ \to [\![M_0]\!], \mathrm{succ}(\_, x) \to [\![M_1]\!]\}$$

(Here shallow effect handlers are assumed, but
it is possible to adapt the encoding to deep handlers)
[Hillerström and Lindley, APLAS'18]

- Encoding of basic arithmetic operations

$$[\![V - n]\!] = \text{with } H \text{ handle } (\cdots(\text{with } H \text{ handle } [\![V]\!]\ ())\cdots)$$
$$\text{where } H = \{\mathrm{return}\ \_ \to [\![0]\!], \mathrm{succ}(\_, x) \to x()\ \}$$

Key Idea to Prevent the Nat Encoding

Bounding # of simultaneously active effect handlers

Formal Statement

For any term e to be model checked,

$$\exists n.\, \forall e', H_1, \ldots, H_n.\, e \not\longrightarrow \text{with } H_1 \text{ handle } (\ldots (\text{with } H_n \text{ handle } e') \ldots)$$

Why does it prevent the encoding?

$[\![V - n]\!] = \text{with } H \text{ handle } (\cdots (\text{with } H \text{ handle } [\![V]\!]\ ()) \cdots)$

Predecessor is implemented by effect handlers

➔ Bounding # of simultaneously active effect handlers results in bounding # of applications of the predecessor

➔ The restriction bounds the range of accessible natural numbers

Approach to Implementing the Restriction

Types of delimited continuations

- Use a type system with ***answer types*** [Danvy+90; Materzok+11]
  - Essentially the same as the ATM type system in [Kawamata+ POPL'24] without polymorphism, dependency, and refinements
  - Answer types reveal # of delimited continuations necessary to evaluate terms
  - **# of delimited continuations = # of effect handlers that can be active**
  - Bounding answer types enables bounding # of simul. active effect handlers

Types

$$
\begin{array}{rrcl}
\textbf{Value types} & T & ::= & B \mid T \to C \\
\textbf{Computation types} & C & ::= & \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} \\
\textbf{Operation signatures} & \Sigma & ::= & \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, A_i^{\mathrm{ini}} \Rightarrow A_i^{\mathrm{fin}}\}^{1 \le i \le n} \\
\textbf{Answer types} & A & ::= & T \mid C
\end{array}
$$

Initial answer type: the return type of the delimited cont. enclosing terms

Final answer type: the argument type of the meta-continuation

$$
\begin{array}{rrcl}
\textbf{Value types} & T & ::= & \cdots \mid T \to C \\
\textbf{Computation types} & C & ::= & \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} \\
\textbf{Operation signatures} & \Sigma & ::= & \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, A_i^{\mathrm{ini}} \Rightarrow A_i^{\mathrm{fin}}\}^{1 \le i \le n} \\
\textbf{Answer types} & A & ::= & T \mid C
\end{array}
$$

- CPS interpretation of computation types (except for Σ)

$$[\![T \,/\, A_1 \Rightarrow A_2]\!]_{\mathrm{CPS}} = ([\![T]\!]_{\mathrm{CPS}} \to [\![A_1]\!]_{\mathrm{CPS}}) \to [\![A_2]\!]_{\mathrm{CPS}}$$

- For a computation type $C = T_1 \,/\, A_1 \Rightarrow (T_2 \,/\, A_2 \Rightarrow (\ldots \Rightarrow (T_n \,/\, A_n \Rightarrow T) \ldots))$

$$
\begin{aligned}
[\![C]\!]_{CPS} = {} & ([\![T_1]\!]_{\mathrm{CPS}} \to [\![A_1]\!]_{\mathrm{CPS}}) \to \\
& ([\![T_2]\!]_{\mathrm{CPS}} \to [\![A_2]\!]_{\mathrm{CPS}}) \to \cdots \to \\
& ([\![T_n]\!]_{\mathrm{CPS}} \to [\![A_n]\!]_{\mathrm{CPS}}) \to [\![T]\!]_{\mathrm{CPS}}
\end{aligned}
$$

Results

The model checking of HEPCF terms well-typed in our type system is **decidable**

Proof strategy: Reducing the HOMC for effect handlers to
the (decidable) HOMC for algebraic effects
via a semantics-preserving CPS transformation

Our type system accepts effect handlers for exceptions, local store, backtracking, etc.

Because it restricts the use of effect handlers, not themselves

Implementation: EffCaml

GitHub page of **EffCaml**

- A model checker for a subset of OCaml 5 with effect handlers
  1. Inference of ML types, operation signatures, and control effects
  2. CPS-transformation to eliminate effect handlers
  3. Apply HOMC tool HorSat2 [Kobayashi 2016] to the result of 2.

- Preliminary experiment results

File name	Lines of code	Safe/Unsafe	Result correct?	Time (sec.)
`file.ml`	53	Safe	Yes	0.005
`file_unsafe.ml`	52	Unsafe	Yes	0.004
`immutable_false.ml`	42	Safe	Yes	0.004
`immutable_mutable_set_not_b_false.ml`	64	Safe	Yes	0.004
`immutable_set_not_b_false.ml`	44	Safe	Yes	0.004
`immutable_true.ml`	42	Safe	Yes	0.004
`mutable_false.ml`	43	Safe	Yes	0.004
`mutable_immutable_set_not_b_false.ml`	63	Unsafe	Yes	0.004
`mutable_set_not_b_false.ml`	45	Unsafe	Yes	0.005
`mutable_true.ml`	43	Safe	Yes	0.003

Conclusions

Automated precise verification of effect-handling programs

◊ By an ATM refinement type system that precisely tracks answer refinement types

Decidable HOMC of effect-handling programs

◊ By an ATM type system that bounds # of simultaneously active effect handlers

Takeaway: Answer types are effective in reasoning about effect-handling programs

- Other application: type-based temporal verification [Sekiyama & Unno, POPL'23]

Ongoing and Future Work

- Support other variants of effect handlers
  - E.g., shallow and lexical handlers
- Extend ATM type systems with varying levels of polymorphism to balance modularity and precision of verification
  - Effect polymorphism
    - specifying a part of an operation signature as a parameter
  - Control effect polymorphism
  - Computation and value type polymorphism
- Enhance **RCaml** and **EffCaml** to be more scalable and to support more language features and specifications
  - Combination with other effects such as parallelism and temporal effects