

Answer-Refinement Modification

**A refinement type system for
algebraic effect handlers**

Fuga Kawamata
Waseda University

Hiroshi Unno
University of Tsukuba

Taro Sekiyama
National Institute
of Informatics

Tachio Terauchi
Waseda University

Background: Refinement types

- ◆ Types endowed with first-order predicates

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

E.g. $\{x : \mathbf{int} \mid x \neq 0\}$ for non-zero numbers

- ◆ Extended to dependent function types $x : T_1 \rightarrow T_2$

E.g. $x : \mathbf{int} \rightarrow \{y : \mathbf{int} \mid y > x\}$ for
functions returning numbers larger than arguments

- ◆ Implementations on top of the existing languages
 - ◊ **RCaml** for OCaml
 - ◊ **LiquidHaskell** for Haskell

Contribution:

Refinement type system for algebraic effect handlers

decide : unit → bool

handle

 if decide() then n₁ else n₂

with

 return x → x

decide () k → (k true) + (k false)

Contribution: Refinement type system for algebraic effect handlers

decide : unit → bool

handle

 if decide() then n₁ else n₂

with

 return x → x

decide () k → (k true) + (k false)

: { z : int | z = n₁ + n₂ }

Contribution:

Refinement type system for algebraic effect handlers

set : unit → int

get : int → unit

handle

```
set n1;  
let x = get() in  
set n2;  
let y = get() in  
x + y
```

with

```
return x → λ_. x  
set x k → λ_. k () x  
get () k → λs. k s s
```

Contribution: Refinement type system for algebraic effect handlers

set : unit → int

get : int → unit

handle

```
set n1;  
let x = get() in  
set n2;  
let y = get() in  
x + y
```

with

```
return x → λ_. x  
set x k → λ_. k () x  
get () k → λs. k s s
```

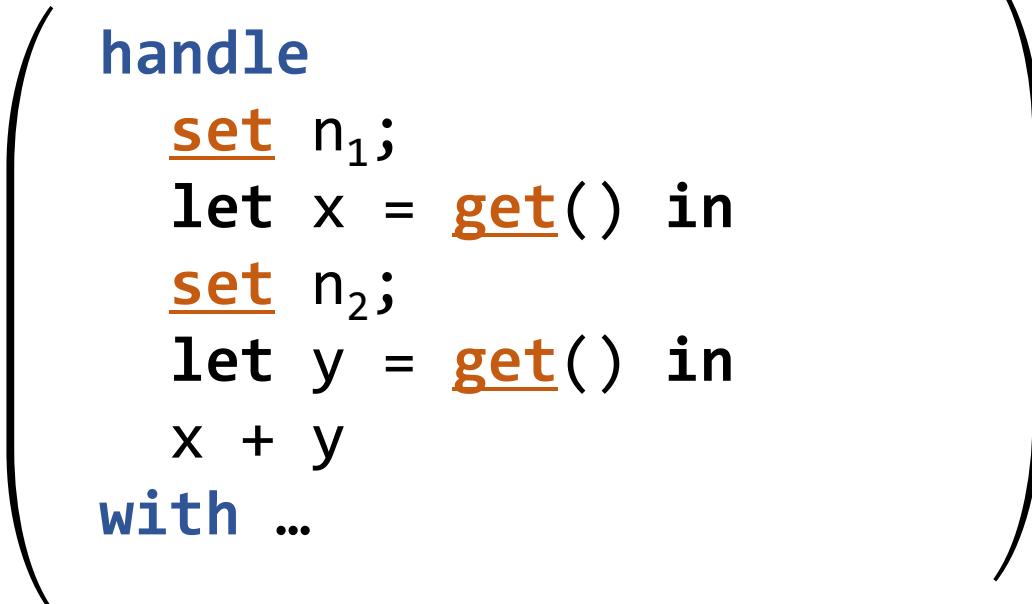
: int → { z : int | z = n₁ + n₂ }

What is a challenge?

Naïve addition of refinement types doesn't work well

```
set : {z : int |  $\phi_{\text{set}}$ }  $\rightarrow$  unit  
get : unit  $\rightarrow$  {z : int |  $\phi_{\text{get}}$ }
```

```
handle  
  set n1;  
  let x = get() in  
  set n2;  
  let y = get() in  
  x + y  
with ...
```



: int \rightarrow {z : int | z = n₁ + n₂}

What is a challenge?

Naïve addition of refinement types doesn't work well

```
set : {z : int |  $\phi_{\text{set}}$ }  $\rightarrow$  unit  
get : unit  $\rightarrow$  {z : int |  $\phi_{\text{get}}$ }
```

```
handle  
  set n1;  
  let x = get() in  
  set n2;  
  let y = get() in  
  x + y  
with ...
```

: int \rightarrow {z : int | z = n₁ + n₂}

OK if x : {z : int | z = n₁}, y : {z : int | z = n₂}

What is a challenge?

Naïve addition of refinement types doesn't work well

set : $\{z : \text{int} \mid \phi_{\text{set}}\} \rightarrow \text{unit}$

get : $\text{unit} \rightarrow \{z : \text{int} \mid \phi_{\text{get}}\}$

handle

set n_1 ;

let $x = \text{get}()$ $\vdash \phi_{\text{get}} \Rightarrow z = n_1$

set n_2 ;

let $y = \text{get}()$ $\vdash \phi_{\text{get}} \Rightarrow z = n_2$

$x + y$

with ...

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

OK if $x : \{z : \text{int} \mid z = n_1\}, y : \{z : \text{int} \mid z = n_1\}$

What is a challenge?

Naïve addition of refinement types doesn't work well

```
set : {z : int |  $\phi_{set}$ }  $\rightarrow$  unit  
get : unit  $\rightarrow$  {z : int |  $\phi_{get}$ }
```

handle

```
set n1;
```

```
let x = get()
```

```
set n2;
```

```
let y = get()
```

```
x + y
```

with ...

: int \rightarrow {z : int | z = n₁ + n₂}

$\phi_{get} \Rightarrow z = n_1$

$\phi_{get} \Rightarrow z = n_2$



No ϕ_{get} satisfying both!

OK if x : {z : int | z = n₁}, y : {z : int | z = n₁}

What is a challenge?

Naïve addition of refinement types doesn't work well

```
set : {z : int |  $\phi_{set}$ } → unit  
get : unit → {z : int |  $\phi_{get}$ }
```

```
handle
```

```
  set n1;
```

```
  let x = get() in
```

```
  set n2;
```

```
  let y = get() in
```

```
  x + y
```

```
with ...
```

Invoked in context

(**handle** let x = [] in ... with h) n₁

Invoked in context

(**handle** let y = [] in ... with h) n₂

No way to reason about
contexts of operation calls

What is a challenge?

Naïve addition of refinement types doesn't work well

decide : int $\rightarrow \{ z : \text{bool} \mid \phi \}$

handle

 if decide() then n_1 else n_2

with

 return $x \rightarrow x$

decide () $k \rightarrow (k \text{ true}) + (k \text{ false})$

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

What is a challenge?

Naïve addition of refinement types doesn't work well

decide : int $\rightarrow \{ z : \text{bool} \mid \phi \}$

handle

 if decide() then n_1 else n_2

with

 return $x \rightarrow x$

decide () $k \rightarrow (k \text{ true}) + (k \text{ false})$

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



Cannot be of $\{ z : \text{int} \mid z = n_1 + n_2 \}$
because it comes from the if-expression

What is a challenge?

Naïve addition of refinement types doesn't work well

decide : int $\rightarrow \{ z : \text{bool} \mid \phi \}$

handle

 if decide() then n_1 else n_2

with

 return $x \rightarrow x$

decide () $k \rightarrow (k \text{ true}) + (k \text{ false})$

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



Cannot be of $\{ z : \text{int} \mid z = n_1 + n_2 \}$
because it comes from the if-expression

No way to reflect the execution order
of clauses in static reasoning

Our approach

- ◆ Adapting **Answer-Type Modification (ATM)** to algebraic effect handlers
 - ◊ Able to express pre- and post-states about contexts at operation calls
 - ◊ Able to allow different clauses to have different (answer) types

```
handle
  //  int → {z | z = n1 + n2}
  set n1;
  // x:int → {z | z = x + n2}
  let x = get() in
    //  int → {z | z = x + n2}
    set n2;
    // y:int → {z | z = x + y}
    let y = get() in
      //  int → {z | z = x + y}
      x + y
with ...
```

```
handle
  if decide() then n1 else n2
  with
    return x → x : int
    decide ()
      (k : b : bool → {z | b ⇒ z = n1 ∧ ¬b ⇒ z = n2})
       → (k true) + (k false)
         : {z | z = n1 + n2}
```

Language

Syntax

Values $v ::= c \mid x \mid \lambda x. e$

Expressions $e ::= v \mid v_1 v_2 \mid \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \mid \text{op } v \mid \mathbf{handle} \ e \ \mathbf{with} \ h$

Handlers $h ::= \mathbf{return} \ x \mapsto e \mid h \uplus \text{op } x \ k \mapsto e$

Pure evaluation contexts

$$\mathcal{E} ::= [] \mid \mathbf{let} \ x = \mathcal{E} \ \mathbf{in} \ e$$

Dynamic semantics (excerpt)

handle v **with** h $\rightarrow e[x \mapsto v]$ (**return** $x \mapsto e \in h$)

handle $\mathcal{E}[\text{op } v]$ **with** h $\rightarrow e[x \mapsto v, k \mapsto \lambda x. \mathbf{handle} \ \mathcal{E}[x] \ \mathbf{with} \ h]$
 $(\text{op } x \ k \mapsto e \in h)$

Answer types = Types for contexts

$\mathcal{C}[\text{ handle } \mathcal{E}[e] \text{ with } h]$

Pure evaluation contexts

$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } e$

Evaluation contexts

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

Answer types = Types for contexts

$\mathcal{C}[\text{ handle } \mathcal{E}[e] \text{ with } h]$

Pure evaluation contexts

$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } e$

Evaluation contexts

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

Answer types = Types for contexts

$\mathcal{C}[\text{ handle } \mathcal{E}[e] \text{ with } h]$

Pure evaluation contexts

$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } e$

Evaluation contexts

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

For expression e ,

- ◆ **Initial answer type** = return type of cont. $\text{handle } \mathcal{E}[] \text{ with } h$
= **assumption on cont.**
- ◆ **Final answer type** = argument type of meta-cont. $\mathcal{C}[]$
= **guarantee for meta-cont.**

Answer types in ordinary type systems

Value types $T, S ::= \mathbf{B} \mid T \rightarrow C$

Computation types $C ::= \Sigma \triangleright T$

Signatures $\Sigma ::= \{ \text{op}_i : T_i \twoheadrightarrow S_i \}_i$

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

$$\frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle} \ e \ \mathbf{with} \ h : C_2}$$

$$\frac{\Gamma \vdash v : \Sigma \triangleright T \quad \text{op} : T_i \twoheadrightarrow S_i \in \Sigma}{\Gamma \vdash \text{op } v : \Sigma \triangleright S}$$

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2}$$

$$\Gamma, x_r : T_r \vdash e_r : C$$

$$\forall i. \Gamma, x : T_i, k : S_i \rightarrow C \vdash e : C$$

$$\Gamma \vdash \mathbf{return} \ x_r \mapsto e_r \ ; \ \cup \ \{ \text{op}_i \ x \ k \mapsto e_i \}_i : \{ \text{op}_i : T_i \rightarrow S_i \}_i \triangleright T_r \rightsquigarrow C$$

Answer types in ordinary type systems

Value types $T, S ::= \mathbf{B} \mid T \rightarrow C$

Computation types $C ::= \Sigma \triangleright T$

Signatures $\Sigma ::= \{ \text{op}_i : T_i \twoheadrightarrow S_i \}_i$

$$\boxed{\Gamma \vdash e : C} \quad \frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle}\ e \ \mathbf{with}\ h : C_2} \quad \frac{\Gamma \vdash v : \Sigma \triangleright T \quad \text{op} : T_i \twoheadrightarrow S_i \in \Sigma}{\Gamma \vdash \text{op } v : \Sigma \triangleright S}$$

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2} \quad \frac{\Gamma, x_r : T_r \vdash e_r : C \quad \forall i. \Gamma, x : T_i, k : S_i \rightarrow C \vdash e : C}{\Gamma \vdash \mathbf{return}\ x_r \mapsto e_r \ \mathbf{of}\ \{\text{op}_i : T_i \rightarrow S_i\}_i : \{\text{op}_i : T_i \rightarrow S_i\}_i \triangleright T_r \rightsquigarrow C}$$

Final answer type

$$\Gamma \vdash \mathbf{return}\ x_r \mapsto e_r \ \mathbf{of}\ \{\text{op}_i : T_i \rightarrow S_i\}_i : \{\text{op}_i : T_i \rightarrow S_i\}_i \triangleright T_r \rightsquigarrow C$$

Answer types in ordinary type systems

Value types

$$T, S ::= \mathbf{B} \mid T \rightarrow C$$

Computation types

$$C ::= \Sigma \triangleright T$$

Signatures

$$\Sigma ::= \{ \mathbf{op}_i : T_i \twoheadrightarrow S_i \}_i$$

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

$$\frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle}\ e\ \mathbf{with}\ h : C_2}$$

$$\frac{\Gamma \vdash v : \Sigma \triangleright T \quad \mathbf{op}_i : T_i \rightarrow S_i \in \Sigma}{\Gamma \vdash \mathbf{op}\ v : T_i \rightarrow S_i \in \Sigma}$$

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2}$$

$$\frac{\Gamma, x_r : T_r \vdash e_r : \mathbf{C}}{\forall i. \Gamma, x : T_i, k : S_i \rightarrow C \vdash e : \mathbf{C}}$$

Final answer type
for pure comp.

Final answer type
for \mathbf{op}_i call

Final answer type

$$\Gamma \vdash \mathbf{return}\ x_r \mapsto e_r \cup \{ \mathbf{op}_i\ x\ k \mapsto e_i \}_i : \{ \mathbf{op}_i : T_i \rightarrow S_i \}_i \triangleright T_r \rightsquigarrow \mathbf{C}$$

Answer types in ordinary type systems

Value types

$$T, S ::= \mathbf{B} \mid T \rightarrow C$$

Computation types

$$C ::= \Sigma \triangleright T$$

Signatures

$$\Sigma ::= \{ \mathbf{op}_i : T_i \twoheadrightarrow S_i \}_i$$

Initial answer type
for pure comp.

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

$$\frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle}\ e\ \mathbf{with}\ h : C_2}$$

$$\frac{\Gamma \vdash v : \Sigma \triangleright T}{\Gamma \vdash \mathbf{op}\ v}$$

Initial answer type
for \mathbf{op}_i call

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2}$$

$$\frac{\Gamma, x_r : T_r \vdash e_r : \textcolor{blue}{C} \quad \forall i. \Gamma, x : T_i, k : S_i \rightarrow \textcolor{blue}{C} \vdash e : C}{\Gamma \vdash \mathbf{return}\ x_r \mapsto e_r \cup \{ \mathbf{op}_i\ x\ k \mapsto e_i \}_i : \{ \mathbf{op}_i : T_i \rightarrow S_i \}_i \triangleright T_r \rightsquigarrow C}$$

$$\Gamma \vdash \mathbf{return}\ x_r \mapsto e_r \cup \{ \mathbf{op}_i\ x\ k \mapsto e_i \}_i : \{ \mathbf{op}_i : T_i \rightarrow S_i \}_i \triangleright T_r \rightsquigarrow C$$

Example with different Initial and final answer types

op : unit → unit

handle

op()

with

return () → “foo”

op () _ → 1

Example with different Initial and final answer types

op : unit → unit

```
C[  
  handle  
    op()  
  with  
    return () → “foo”  
    op () _ → 1]
```

- ◆ **Initial answer type** is **string** because

```
handle [] with  
return () → “foo”  
op () _ → 1
```

 returns “foo”
- ◆ **Final answer type** is **int** because $\mathcal{C}[]$ takes 1

Ours: endowed with answer-type modification (ATM)

[Danvy&Fillinski'90] for shift/reset, [Materzok&Biernacki'11] for shift0/reset0

- ◆ **Allowing initial and final answer types to be different**
for precise tracking of value flow

$$\Gamma \vdash e : \Sigma \triangleright T / \textcolor{blue}{C_I} \Rightarrow \textcolor{blue}{C_F}$$

- ◊ Evaluating e may perform effect operations in Σ and return a value of T
- ◊ If expression e is evaluated under $\mathcal{C}[\text{handle } \mathcal{E}[\] \text{ with } h]$ such that

$$(\lambda x. \text{handle } \mathcal{E}[x] \text{ with } h) : T \rightarrow \textcolor{blue}{C_I},$$

then the delimiter will evaluate to an expression e' of $\textcolor{blue}{C_F}$

$$\mathcal{C}[\text{handle } \mathcal{E}[e] \text{ with } h] \longrightarrow^* \mathcal{C}[e']$$

Changes in typing

Computation types $C ::= \Sigma \triangleright T / A$ **Control effects** $A ::= \square | \textcolor{blue}{C_I} \Rightarrow \textcolor{blue}{C_F}$

Signatures $\Sigma ::= \{ \text{op}_i : T_i \rightarrow S_i / \textcolor{blue}{C_{I,i}} \Rightarrow \textcolor{blue}{C_{F,i}} \}_i$

$$\boxed{\Gamma \vdash e : C} \quad \frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle}\ e \ \mathbf{with}\ h : C_2} \quad \frac{\Gamma \vdash v : \Sigma \triangleright T \quad T \rightarrow\!\!\! \rightarrow S / \textcolor{blue}{C_I} \Rightarrow \textcolor{blue}{C_F} \in \Sigma}{\Gamma \vdash \text{op } v : \Sigma \triangleright S / \textcolor{blue}{C_I} \Rightarrow \textcolor{blue}{C_F}}$$

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2} \quad \frac{\Gamma, x_r : T_r \vdash e_r : \textcolor{blue}{C_I} \quad \dots}{\Gamma \vdash \begin{array}{c} \mathbf{return}\ x_r \mapsto e_r \\ \cup \{ \text{op}_i\ x\ k \mapsto e_i \}_i \end{array} : \Sigma \triangleright T_r / \textcolor{blue}{C_I} \Rightarrow \textcolor{blue}{C_F} \rightsquigarrow \textcolor{blue}{C_F}}$$

Changes in typing

Computation types $C ::= \Sigma \triangleright T / A$ **Control effects** $A ::= \square | \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F$

Signatures $\Sigma ::= \{ \text{op}_i : T_i \rightarrow S_i / \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$

$$\boxed{\Gamma \vdash e : C} \quad \frac{\Gamma \vdash e : C_1 \quad \Gamma \vdash h : C_1 \rightsquigarrow C_2}{\Gamma \vdash \mathbf{handle}\ e \ \mathbf{with}\ h : C_2} \quad \frac{\Gamma \vdash v : \Sigma \triangleright T \quad T \rightarrow S / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F \in \Sigma}{\Gamma \vdash \text{op } v : \Sigma \triangleright S / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F}$$

$$\boxed{\Gamma \vdash h : C_1 \rightsquigarrow C_2} \quad \Sigma = \{ \text{op}_i : T_i \rightarrow S_i / \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$$

$$\Gamma, x_r : T_r \vdash e_r : \textcolor{blue}{C}_I \quad \forall i. \Gamma, x : T_i, k : S_i \rightarrow \textcolor{blue}{C}_{I,i} \vdash e_i : \textcolor{blue}{C}_{F,i}$$

$$\Gamma \vdash \frac{\mathbf{return}\ x_r \mapsto e_r : \Sigma \triangleright T_r / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F \quad \cup \{ \text{op}_i x k \mapsto e_i \}_i \rightsquigarrow \textcolor{blue}{C}_F}{\mathbf{return}\ x k \mapsto e_i : \Sigma \triangleright T_r / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F}$$

Changes in typing

Computation types $C ::= \Sigma \triangleright T / A$ **Control effects** $A ::= \square | \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F$

Signatures $\Sigma ::= \{ \text{op}_i : T_i \rightarrow S_i / \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$

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

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash v : \Sigma \triangleright T / \textcolor{blue}{C} \Rightarrow \textcolor{blue}{C}}$$

$$\frac{\Gamma \vdash e_1 : \Sigma \triangleright S / \textcolor{blue}{C} \Rightarrow \textcolor{blue}{C}_F \quad \Gamma, x:S \vdash e_2 : \Sigma \triangleright T / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \Sigma \triangleright T / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F} \quad x \notin \text{fv}(T) \cup \text{fv}(\textcolor{blue}{C}_I)$$

1. Final answer type $\textcolor{blue}{C}_F$ comes from pre-computation e_1
2. Initial answer type $\textcolor{blue}{C}_I$ comes from post-computation e_2
3. Initial answer type of e_1 and final one of e_2 need to agree

Changes in typing

Computation types $C ::= \Sigma \triangleright T / A$ **Control effects** $A ::= \square | \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F$

Signatures

$$\Sigma ::= \{ \text{op}_i : T_i \rightarrow S_i / \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$$

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

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash v : \Sigma \triangleright T / \textcolor{blue}{C} \Rightarrow \textcolor{blue}{C}}$$

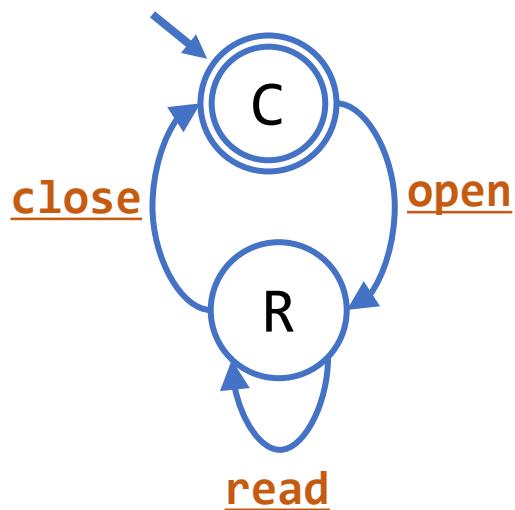
$$\frac{\Gamma \vdash e_1 : \Sigma \triangleright S / \textcolor{blue}{C} \Rightarrow \textcolor{blue}{C}_F \quad \Gamma, x:S \vdash e_2 : \Sigma \triangleright T / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \Sigma \triangleright T / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F} \quad x \notin \text{fv}(T) \cup \text{fv}(\textcolor{blue}{C}_I)$$

// $\textcolor{blue}{C}_F$
let $x = e_1$ **in**
// C
 e_2
// $\textcolor{blue}{C}_I$

1. Final answer type $\textcolor{blue}{C}_F$ comes from pre-computation e_1
2. Initial answer type $\textcolor{blue}{C}_I$ comes from post-computation e_2
3. Initial answer type of e_1 and final one of e_2 need to agree

Example: Reasoning about file operations usage

Aim: To verify that the program uses the operations according the following DFA



open : unit → unit
read : unit → int
close : unit → unit

type state = C | R

$$A_{s_1 \leftrightarrow s_2} \equiv \{z : \text{state} \mid z = s_2\} \rightarrow \text{int} \Rightarrow \{z : \text{state} \mid z = s_1\} \rightarrow \text{int}$$

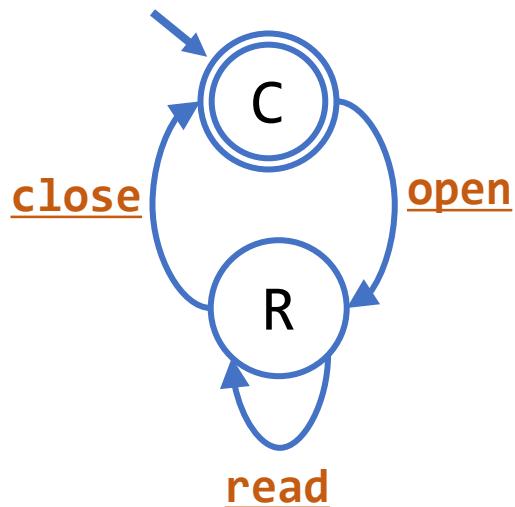
$$\Sigma \equiv \begin{aligned} &\{ \text{open} : \text{unit} \rightarrow \text{unit} / A_{C \leftrightarrow R}, \\ &\text{read} : \text{unit} \rightarrow \text{int} / A_{R \leftrightarrow R}, \\ &\text{close} : \text{unit} \rightarrow \text{unit} / A_{R \leftrightarrow C} \} \end{aligned}$$

handle

```
// {z : state | z = C} → int
let _ = open() in
// {z : state | z = R} → int
let x = read() in
// {z : state | z = R} → int
let _ = close() in
// {z : state | z = C} → int
x
// {z : state | z = C} → int
with
return x → λ_. x :
    {z : state | z = C} → int
open () k → λs. k () R
read () k → λs. k 42 R
close () k → λs. k () C
```

Example: Reasoning about file operations usage

Aim: To verify that the program uses the operations according the following DFA



open : unit → unit
read : unit → int
close : unit → unit

type state = C | R

$$A_{s_1 \leftrightarrow s_2} \equiv \{z : \text{state} \mid z = s_2\} \rightarrow \text{int} \Rightarrow \{z : \text{state} \mid z = s_1\} \rightarrow \text{int}$$

$$\Sigma \equiv \begin{aligned} &\{\text{open} : \text{unit} \rightarrow \text{unit} / A_{C \leftrightarrow R}, \\ &\text{read} : \text{unit} \rightarrow \text{int} / A_{R \leftrightarrow R}, \\ &\text{close} : \text{unit} \rightarrow \text{unit} / A_{R \leftrightarrow C}\} \end{aligned}$$

handle

```
// {z : state | z = C} → int
let _ = open() in
// {z : state | z = R} → int
let x = read() in
// {z : state | z = R} → int
x
// {z : state | z = R} → int
with
return x → λ_. x :
{z : state | z = C} → int
```

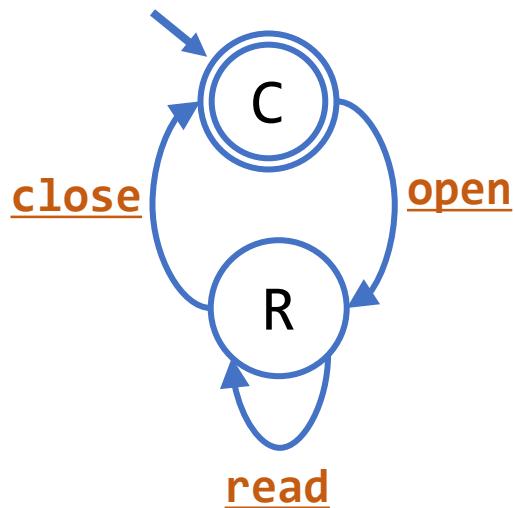
Finishes at the state R

\neq

open () k → λs. k () R
read () k → λs. k 42 R
close () k → λs. k () C

Example: Reasoning about file operations usage

Aim: To verify that the program uses the operations according the following DFA



open : unit → unit
read : unit → int
close : unit → unit

type state = C | R

$$A_{s_1 \leftrightarrow s_2} \equiv \{z : \text{state} \mid z = s_2\} \rightarrow \text{int} \Rightarrow \{z : \text{state} \mid z = s_1\} \rightarrow \text{int}$$

$$\Sigma \equiv \begin{aligned} &\{ \text{open} : \text{unit} \rightarrow \text{unit} / A_{C \leftrightarrow R}, \\ &\text{read} : \text{unit} \rightarrow \text{int} / A_{R \leftrightarrow R}, \\ &\text{close} : \text{unit} \rightarrow \text{unit} / A_{R \leftrightarrow C} \} \end{aligned}$$

handle

The initial state is R

```
// {z : state | z = R} → int
let x = read() in
// {z : state | z = R} → int
let _ = close() in
// {z : state | z = C} → int
x
// {z : state | z = C} → int
with
return x → λ_. x :
{z : state | z = C} → int
open () k → λs. k () R
read () k → λs. k 42 R
close () k → λs. k () C
```

Extension 1: Argument-dependent continuation type

$$\Gamma \vdash e : \Sigma \triangleright T / \textcolor{brown}{x}.\, \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F$$

- ◊ Expression e is assumed to be evaluated under $\mathcal{C}[\text{ handle } \mathcal{E}[\] \text{ with } h]$ such that $(\lambda x. \text{handle } \mathcal{E}[x] \text{ with } h) : (\textcolor{brown}{x} : T \rightarrow \textcolor{blue}{C}_I)$

Signatures

$$\Sigma ::= \{ \text{op}_i : T_i \twoheadrightarrow S_i / \textcolor{brown}{x}.\, \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$$

$$\Sigma = \{ \text{op}_i : T_i \twoheadrightarrow S_i / \textcolor{brown}{x}.\, \textcolor{blue}{C}_{I,i} \Rightarrow \textcolor{blue}{C}_{F,i} \}_i$$

$$\Gamma, x_r : T_r \vdash e_r : \textcolor{blue}{C}_I \quad \forall i. \Gamma, x : T_i, k : (\textcolor{brown}{x} : S_i) \rightarrow \textcolor{blue}{C}_{I,i} \vdash e_i : \textcolor{blue}{C}_{F,i}$$

$$\Gamma \vdash \frac{\text{return } x_r \mapsto e_r}{\cup \{ \text{op}_i x k \mapsto e_i \}_i} : \Sigma \triangleright T_r / \textcolor{blue}{C}_I \Rightarrow \textcolor{blue}{C}_F \rightsquigarrow \textcolor{blue}{C}_F$$

Example: Decide

```
 $\Sigma = \{ \text{decide} : \text{unit} \rightarrow \text{bool} /$ 
 $b. \{z \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\} \Rightarrow$ 
 $\{z \mid z = n_1 + n_2\}$ 
 $\}$ 
```

handle

```
// {z | z = n1 + n2}
let b = decide() in
// {z | b  $\Rightarrow$  z = n1  $\wedge$   $\neg$ b  $\Rightarrow$  z = n2}
let x = if b then n1 else n2 in
// {z | z = x}
x
```

with

```
return x → x : {z | z = x}
decide () k  $\rightarrow$  (k true) + (k false)
```

Example: Decide

```
 $\Sigma = \{ \text{decide} : \text{unit} \rightarrow \text{bool} /$ 
 $b. \{z \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\} \Rightarrow$ 
 $\{z \mid z = n_1 + n_2\}$ 
 $\}$ 
```

handle

```
// {z | z = n1 + n2}
let b = decide() in
// {z | b  $\Rightarrow$  z = n1  $\wedge$   $\neg b \Rightarrow z = n_2\}$ 
let x = if b then n1 else n2 in
// {z | z = x}
x
```

$b : \text{bool} \rightarrow \{z \mid b \Rightarrow z = n_1 \wedge \neg b \Rightarrow z = n_2\}$

with

```
return x
decide () k  $\rightarrow$  (k true) + (k false)
```

Extension 2: Predicate polymorphism

Signatures $\Sigma ::= \{ \text{op}_i : \forall X: \bar{\mathbf{B}}. T_i \Rightarrow S_i / x. C_{I,i} \Rightarrow C_{F,i} \}_i$

$$\frac{\Gamma \vdash v : \Sigma \triangleright T \quad \forall X: \bar{\mathbf{B}}. T \Rightarrow S / x. C_I \Rightarrow C_F \in \Sigma \quad \Gamma \vdash P : \bar{\mathbf{B}}}{\Gamma \vdash \text{op } v : \Sigma \triangleright (S / x. C_I \Rightarrow C_F)[X \mapsto P]}$$

$$\frac{\Sigma = \{ \text{op}_i : \forall X: \bar{\mathbf{B}}. T_i \Rightarrow S_i / x. C_{I,i} \Rightarrow C_{F,i} \}_i \quad \Gamma, x_r : T_r \vdash e_r : C_I \quad \forall i. \Gamma, X: \bar{\mathbf{B}}, x : T_i, k : (x : S_i) \rightarrow C_{I,i} \vdash e_i : C_{F,i}}{\Gamma \vdash \text{return } x_r \mapsto e_r : \cup \{ \text{op}_i x k \mapsto e_i \}_i : \Sigma \triangleright T_r / C_I \Rightarrow C_F \rightsquigarrow C_F}$$

Example: State

$\Sigma = \{ \text{set} : \forall X: (\text{int}, \text{int}). x : \text{int} \rightarrow \text{unit} / _ \cdot s: \text{int} \rightarrow \{ z: \text{int} \mid X(z, s) \} \Rightarrow s': \text{int} \rightarrow \{ z: \text{int} \mid X(z, x) \},$

$\text{get} : \forall Y: (\text{int}, \text{int}, \text{int}). \text{unit} \rightarrow \text{int} / y. s: \text{int} \rightarrow \{ z: \text{int} \mid Y(z, s, y) \} \Rightarrow s': \text{int} \rightarrow \{ z: \text{int} \mid X(z, s', s') \}$

handle

```
// int → {z | z = n1 + n2}  
set n1; (* X ↦ λz, s. z = s + n2 *)  
// s: int → {z | z = s + n2}  
let x = get() in (* Y ↦ λz, s, y. z = y + n2 *)  
// int → {z | z = x + n2}  
set n2; (* X ↦ λz, s. z = x + s *)  
// s: int → {z | z = x + s}  
let y = get() in (* Y ↦ λz, s, y. z = x + y *)  
// int → {z | z = x + y}  
x + y  
with ...
```

Open Questions

- ◆ Modularity
 - ◊ Adding effect polymorphism is challenging (due to ATM)
- ◆ Abstraction versus Preciseness
- ◆ Lexical handlers
- ◆ Scalability for real programs
 - ◊ Preliminary experiments for OCaml 5
- ◆ More dependency?

Implementation →

