Handling polymorphic algebraic effects

Taro Sekiyama
National Institute of Informatics

Atsushi Igarashi
Kyoto University

Accepted at ESOP'19
Full version: https://arxiv.org/abs/1811.07332

The problem of interest

Polymorphic effects
(ML references, continuations, etc.)

+

let-polymorphism
[Milner 1978]

Type safety is broken

```
let x : ∀α.(α list) ref = ref []
in
x := [1];
if (head !x) ...
```

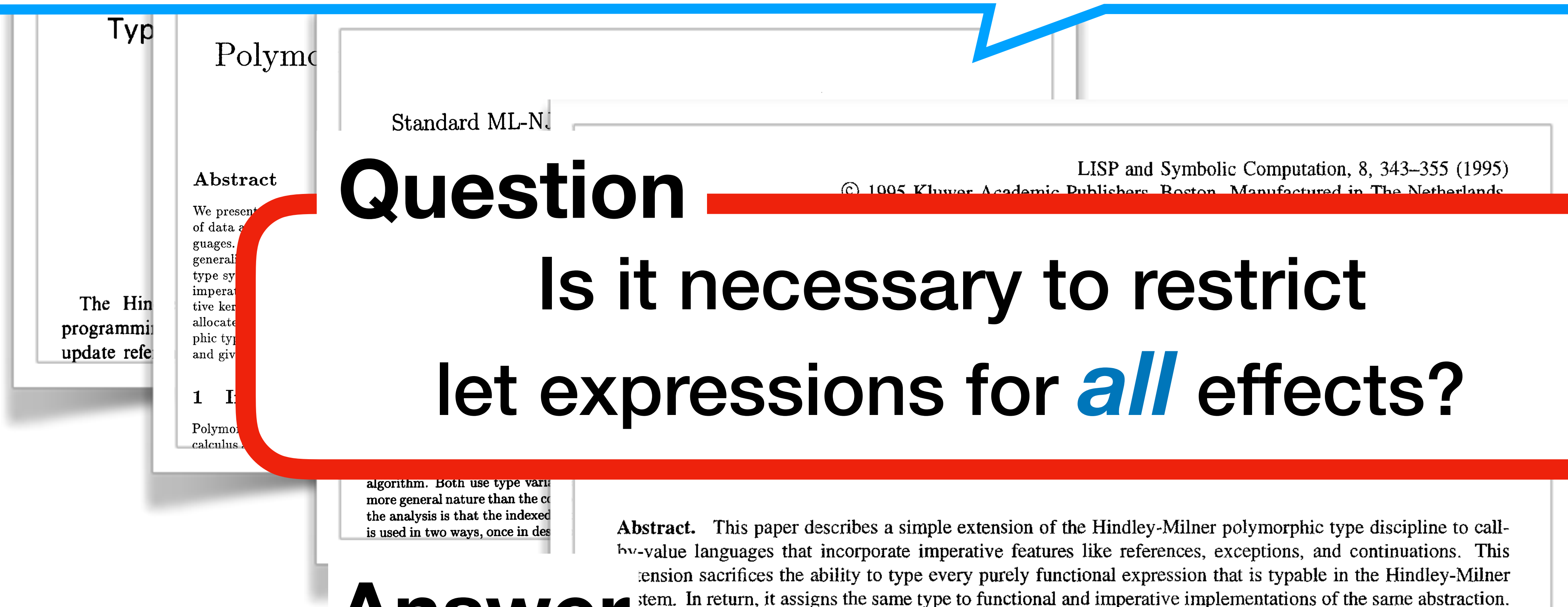
α := int

α := bool

Integer 1 will be used as Boolean

Solutions in the literature

Common key idea: Restrict *let expressions*


Typ
Polymo
Standard ML-NJ
Abstract
We present
of data a
guages.
generali
type sy
imperat
tive ker
allocate
phic typ
and giv
1 I
Polymo
calculus
The Hin
programmi
update refe
algorithm. Both use type varia
more general nature than the co
the analysis is that the indexed
is used in two ways, once in des
LISP and Symbolic Computation, 8, 343–355 (1995)
© 1995 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.
Abstract. This paper describes a simple extension of the Hindley-Milner polymorphic type discipline to call-by-value languages that incorporate imperative features like references, exceptions, and continuations. This ension sacrifices the ability to type every purely functional expression that is typable in the Hindley-Milner stem. In return, it assigns the same type to functional and imperative implementations of the same abstraction.

Question

Is it necessary to restrict let expressions for ***all*** effects?

Answer

No! There are effects that can occur safely in nonrestricted let expressions

Our approach

To restrict ***definitions of polymorphic effects*** used in let expressions

- Effects with properly restricted definitions can occur safely in unrestricted let expressions
- Complementary to the known approaches that restrict let expressions

This work

- Design of a λ-calculus where:
  - Polymorphic effects are given by ***algebraic effects & handlers***
  - The type system restricts handlers so that **effect defs don’t interfere with each other**
- Proof of type soundness of the calculus

Outline

1. Introduction
2. **Background: algebraic effects & handlers**
   - **Resumption**
   - **Extended with polymorphism**
3. A lesson from a counterexample
4. Our work, formally

Algebraic effects & handlers

[Plotkin & Pretnar ’09, ’13]

- Abstract mechanism to define control effects (a.k.a. to use continuations in a “well-structured” manner)
- Separate ***interfaces*** and ***implementations*** of effects
  - Invoked via ***operations***
  - Interpreted by ***handlers***
- Handlers give the ability to call ***continuations***
- Easily extendable to polymorphic effects together with, e.g., value restriction

Example

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail “div0”; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Example

Declare fail operation

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail “div0”; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Example

Declare fail operation

Invoke fail

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail “div0”; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Example

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail “div0”; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Declare fail operation

Invoke fail

Inject interpretation into effects invoked in “div 42 x”

Example

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail "div0"; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Declare fail operation

Invoke fail

Inject interpretation into effects invoked in "`div 42 x`"

Interpretation of fail operation

`f 0 ⟶ Left "div0"`

Example

```
effect fail : str → unit

let div (x:int) (y:int) =
  if y = 0 then (#fail “div0”; -1)
  else x / y

let f (x:int) =
  handle (div 42 x) with
    return (y:int) → Right y
    fail   (y:str) → Left y
```

Declare fail operation

Invoke fail

Inject interpretation into effects invoked in “`div 42 x`”

Evaluated with the value of “`div 42 x`”

Interpretation of fail operation

`f 0 ⟶ Left “div0”`   `f 7 ⟶ Right 6`

Resumption

Handlers support resumption of the computation from the point of the effect invocation

- Reminiscent of delimited continuation

```
effect choose : int × int → int

handle #choose(1,2) +
       #choose(10,20) with
 return (x:int)         → x
 choose (x:int,y:int) → resume x
```

Resumption

Handlers support resumption of the computation from the point of the effect invocation

- Reminiscent of delimited continuation

```
effect choose : int × i

handle #choose(1,2) +
       #choose(10,20) with
 return (x:int)         → x
 choose (x:int,y:int) → resume x
```

x := 1

Return x as
the result of #choose

Resumption

Handlers support resumption of the computation from the point of the effect invocation

- Reminiscent of delimited continuation

```
effect choose : int × i        t

handle         1             +
       #choose(10,20) with
 return (x:int)        → x
 choose (x:int,y:int) → resume x
```

x := 1

Return x as the result of #choose

Resumption

Handlers support resumption of the computation from the point of the effect invocation

- Reminiscent of delimited continuation

```
effect choose : int × int → int

handle        1          +
       #choose(10,20) with
  return (x:int)          → x
  choose (x:int,y:int) → resume x
```

Return x as the result of #choose

Resumption

Handlers support resumption of the computation from the point of the effect invocation

- Reminiscent of delimited continuation

```
effect choose : int × int → int

handle      1         +
           10           with
  return (x:int)        → x
  choose (x:int,y:int) → resume x
```

Return x as the result of #choose

Resumption, formally

Replace "`resume e`" with
"`let y = e in handle E[y] with h`"

`handle E[#op v] with h`

$\longrightarrow$ `e[v/x]`<u>`[Eʰ/resume]`</u>

(if `op(x)`→`e` ∈ `h` and `E` doesn't handle `#op`)

- "`resume e`" calls the delimited continuation `E` from the point of the effect invocation up to the handle—with expression

Resumption example, formally

```
effect choose : int × int → int

handle #choose(1,2) +
       #choose(10,20) with
 return (x:int)         → x
 choose (x:int,y:int)   → resume x
```

```
E ≡ [] + #choose(10,20)

h ≡ return (x)    → x
```

```
= handle E[#choose(1,2)] with h

⟶ (resume x)[1/x,2/y][Eʰ/resume]
```

Resumption example, formally

```
effect choose : int × int → int

handle #choose(1,2) +
       #choose(10,20) with
 return (x:int)         → x
 choose (x:int,y:int) → resume x
```

E ≡ [] + #choose(10,20)

h ≡ return (x) → x

= handle E[#choose(1,2)] with h

⟶ (resume 1) [E^h/resume]

Replace "resume v" with "handle E[v] with h"

= handle E[1] with h

= handle 1 + #choose(10,20) with h

⟶ (resume x)[10/x,20/y][$(1+[])^h$/resume]

= handle 1 + 10 with h ⟶ 11

Polymorphic effects

```
effect choose : ∀α. α × α → α

handle if #choose(true,false)
       then #choose(1,2)
       else #choose(10,20) with
    return (x:int)    → x
 Λα.choose (x:α,y:α) → resume x
```

Polymorphic effects

Polymorphic signature

```
effect choose : ∀α. α × α → α

handle if #choose(true,false)
       then #choose(1,2)
       else #choose(10,20) with
    return (x:int)    → x
 Λα.choose (x:α,y:α) → resume x
```

Polymorphic effects

Polymorphic signature

α := bool

```
effect choose : ∀α. α × α → α

handle if #choose(true,false)
       then #choose(1,2)
       else #choose(10,20) with
    return (x:int)    → x
 Λα.choose (x:α,y:α) → resume x
```

Polymorphic effects

Polymorphic signature

α := bool

```
effect choose : ∀α. α × α → α

handle if #choose(true,false)
       then #choose(1,2)
       else #choose(10,20) with
    return (x:int)    → x
 Λα.choose (x:α,y:α) → resume x
```

α := int

Polymorphic effects

Polymorphic signature

α := bool

```
effect choose : ∀α. α × α → α

handle if #choose(true,false)
       then #choose(1,2)
       else #choose(10,20) with
    return (x:int)    → x
  Λα.choose (x:α,y:α) → resume x
```

α := int

Abstracted over types

Outline

1. Introduction
2. Background: algebraic effects & handlers
3. **A lesson from a counterexample**
4. Our work, formally

Our observation

☠

Type safety is broken if multiple resumptions share type information via type variables

Counterexample to type safety

```
effect get_id : ∀α. unit → (α → α)

handle
  let id : ∀α. α → α =
    #get_id ()
  in
  if (id true)
  then (id 1) else 2
with
     return (x:int)  → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexample to type safety

```
effect get_id : ∀α. unit → (α → α)

handle
   let id : ∀α. α → α =
     #get_id ()
   in
   if (id true)
   then (id 1) else 2
with
      return (x:int) → x
   Λα.get_id (x:unit) →
      resume (λy:α. resume (λz:α.y); y)
```

α → α

α → α

Counterexample to type safety

```
effect get_id : ∀α. unit → (α → α)

handle
  let id : ∀α. α → α =
    #get_id ()
  in
  if (id true)
  then (id 1) else 2
with
     return (x:int)  → x
  Λα.get_id (x:unit) →
     resume (λv:α. resume (λz:α.v); v)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)      → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id : ∀

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[Eʰ/resume]; y
  in
  if (id true)
  then (id 1) else 2
with
     return (x:int)   → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)         → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id :

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[Eʰ/resume]; y
  in
  if (id true)
  then (id 1) else 2
with
     return (x:int)  → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)      → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id : ∀

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[E^h/resume]; y
  in
  if (id true)
  then (id 1) else 2
with
     return (x:int)   → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)        → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id : ∀

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[E^h/resume]; y
  in
  if (resume (λz.true))[E^h/resume]; true
  then (id 1) else 2
with
     return (x:int)   → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)        → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id :

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[E^h/resume]; y
  in
  if (resume (λz.true))[E^h/resume]; true
  then (id 1) else 2
with
     return (x:int)   → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

Counterexa

```
effect get_id : ∀

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[E^h/resume]; y
  in
  if (resume (λz.true))[E^h/resume]; true
  then (id 1) else 2
with
     return (x:int)
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)       → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

Replaces "`resume λz.true`" with "`handle E[λz.true] with h`"

Counterexa

```
effect get_id : ∀

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[E^h/resume]; y
  in
  if handle E[λz.true] with h;       true
  then (id 1) else 2
with
     return (x:int)   → x
  Λα.get_id (x:unit) →
     resume (λy:α. resume (λz:α.y); y)
```

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)      → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

Counterexa

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)       → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
effect get_id :

handle
  let id : ∀α. α → α =
    λy.(resume (λz.y))[Eʰ/resume]; y
  in
  if handle E[λz.true] with h;        true
  then (id 1) el
with
    return (x:in
  Λα.get_id (x:unit) →
    resume (λy:α. resume (λz:α.y); y)
```

```
= let id : ∀α.α→α = λz.true in
  if (id true) then (id 1) else 2
```

Counterexa

```
effect get_id : ∀

handle
   let id : ∀α. α → α =
     λy.(resume (λz.y))[Eʰ/resume]; y
   in
   if handle E[λz.true] with h;        true
   then (id 1) el   2
with
      return (x:in
   Λα.get_id (x:unit) →
      resume (λy:α. resume (λz:α.y); y)
```

```
E ≡ let id : ∀α.α→α = [] in
    if (id true) then (id 1) else 2

h ≡ return (x:int)       → x
    Λα.get_id (x:unit) →
     resume (λy. resume (λz.y); y)
```

```
= let id : ∀α.α→α = λz.true in
  if (id true) then (id 1) else 2
```

Our observation

Type safety is broken if multiple resumptions share type information via type variables

- For clause “**`resume (λy:α. resume (λz:α.y); y)`**”, function **`(λz:α.y)`** is injected into a polymorphic context after replacing **`α`** with **`bool`** and **`y`** with **`true`**

Type safety is achieved if resumptions do not share type variables

- This ensures **resumptions do not interfere with each other**

Our idea
prohibition of sharing type variables

```
effect get_id : ∀α. unit → (α → α)

handle
  let id : ∀α. α → α =
    #get_id ()
  in
  if (id true)
  then (id 1) else
with
    return (x:int)
  Λα.get_id (x:unit) →
    resume (λy   . resume (λz   .y); y)
```

The argument of a resumption must have a type obtained by renaming **α** to a fresh type variable

Our idea
prohibition of sharing type variables

```
effect get_id : ∀α. unit → (α → α)

handle
  let id : ∀α. α → α =
    #get_id ()
  in
  if (id true)
  then (id 1) else
with
     return (x:int) → x
   Λα.get_id (x:unit) →
     resume (λy:β. resume (λz:γ.y); y)
```

The argument of a resumption must have a type obtained by renaming **α** to a fresh type variable

REJECTED

Check: its type is β→β

Check: its type is γ→γ

Our idea
prohibition of sharing type variables

```
effect get_id : ∀α. unit → (α → α)

handle
  let id : ∀α. α → α =
    #get_id ()
  in
  if (id true
  then (id 1)
with
      return (x:int)  → x
  Λα.get_id (x:unit) →
      resume (λy:β. resume (λz:γ.z); y)
```

Acceptable polymorphic effects:
random choice, failure exception, etc.

ACCEPTED

Typed at γ→γ

Outline

1. Introduction
2. Background: algebraic effects & handlers
3. A lesson from a counterexample
4. **Our work, formally**

Summary

Support for
let-polymorphism

- We define a statically typed λ-calculus where:
  - The body of a type abstractions is evaluated
  - Algebraic effects & handlers are polymorphic
  - Resumption arguments are typechecked with assignment of fresh type variables
- We prove type safety of the calculus

Syntax

A, B (types) ::= α | A $\rightarrow_\varepsilon$ B | `int` | `bool` | …

ε (effects) ::= { op_i }$_i$

A$_1$ … A$_2$

e (terms) ::= x **A** | c | λx.e | e$_1$ e$_2$

Λα$_1$ … Λα$_2$

| `let x =` Λ**α**`.e` |

| `#op(`**A**`,e)`

| `handle e with h`

| `resume` Λ**α**`.e`

≈ Allocate fresh type variables

h (handlers) ::= `return x→e` | `h;` Λ**α**`.op(x)→e`

Semantics

$$\boxed{e_1 \longrightarrow e_2}$$

Evaluation rule

$$\boxed{\frac{e_1 \longrightarrow e_2 \qquad E \neq [\,]}{E[e_1] \longrightarrow E[e_2]}}$$

E (evaluation contexts) ::= $[\,] \mid E\ e_2 \mid v_1\ E \mid \ldots$

$\mid$ `let x = Λ`$\underline{\boldsymbol{\alpha}}$`.E`

Allows evaluation under type abstractions

Reduction of effect handling

$$\mathsf{handle}\ \#\mathsf{op}(\forall\, \boldsymbol{\beta}^J . \boldsymbol{A}^I, \Lambda \boldsymbol{\beta}^J . v, E^{\boldsymbol{\beta}^J})\ \mathsf{with}\ h \rightsquigarrow$$
$$e[\mathsf{handle}\ E^{\boldsymbol{\beta}^J}\ \mathsf{with}\ h / \mathsf{resume}]^{\forall\, \boldsymbol{\beta}^J . \boldsymbol{A}^I}_{\Lambda \boldsymbol{\beta}^J . v} [\boldsymbol{A}^I[\bot / \boldsymbol{\beta}^J] / \boldsymbol{\alpha}^I][v[\bot / \boldsymbol{\beta}^J] / x]$$
$$(\text{where } h^{\mathsf{op}} = \Lambda \boldsymbol{\alpha}^I . \mathsf{op}(x) \rightarrow e)$$

- The rule is designed with care about type variables bound in evaluation context E
- See the paper for detail

e sys

Resumption type

Effects that may occur in evaluation of e

$$\Gamma; R \vdash e : A \mid \varepsilon$$

R (resumption types) ::= none | $(\underline{\alpha}, A, B \rightarrow_{\varepsilon} C)$

Type variables bound in an operation clause

Argument type of an effect signature

Function type of continuation

$$\frac{e_0 \text{ and } h \text{ are well typed} \qquad ty(op) = \forall\underline{\alpha}.\, A \rightarrow B \qquad \Gamma, x{:}A; (\underline{\alpha}, A, B \rightarrow_{\varepsilon} C) \vdash e : C \mid \varepsilon}{\Gamma; R_0 \vdash \mathtt{handle}\ e_0\ \mathtt{with}\ h;\ \Lambda\underline{\alpha}.\mathtt{op}(x) \rightarrow e : C \mid \varepsilon}$$

Typing rule for handle—with expressions

e sys

Resumption type

Effects that may occur in evaluation of e

$$\Gamma; R \quad\quad e : A \mid \boldsymbol{\varepsilon}$$

R (resumption types) ::= none | $(\underline{\boldsymbol{\alpha}}, A, B \rightarrow^{\varepsilon} C)$

Type variables bound in an operation clause

Argument type of an effect signature

Function type of continuation

$$\frac{\varepsilon \subseteq \varepsilon' \qquad \Gamma, x{:}A[\underline{\boldsymbol{\beta/\alpha}}];\ \ (\underline{\boldsymbol{\alpha}}, A, B \rightarrow^{\varepsilon} C) \vdash e : B[\underline{\boldsymbol{\beta/\alpha}}] \ \mid \varepsilon'}{\Gamma, x{:}D; (\underline{\boldsymbol{\alpha}}, A, B \rightarrow^{\varepsilon} C) \vdash \mathtt{resume}\ \Lambda\underline{\boldsymbol{\beta}}.e : C \mid \varepsilon'}$$

Typing rule for resumptions

Type safety

If $\emptyset$; none $\quad$ `e : A` $|\,\emptyset$ **,**

then `e` **does not get stuck**

Conclusion

- Type safety is broken in a polymorphic setting if neither effects nor let expressions are restricted
- We take an approach to restricting effects
  - Observation: there are no problem if effects don’t interfere with each other
  - In effect handlers, prohibition of sharing type variables among resumptions ensures the non-interference