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
α := int
x := [1];
if (head !x)
```

\[\alpha := \text{bool}\]

Integer 1 will be used as Boolean
Solutions in the literature

Common key idea: Restrict *let expressions*

**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 $\lambda$-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

```ml
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

eaffect 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 \rightarrow 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) \rightarrow Right y
  fail (y:str) \rightarrow 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

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

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

```plaintext
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 × int → int
handle #choose(1,2) + #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 + #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
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 $E[#\text{op } v]$ with $h$

\[ \longrightarrow e[v/x] [E^h/\text{resume}] \]

(if $\text{op}(x) \rightarrow e \in h$ and $E$ doesn’t handle $\#\text{op}$)

- “resume e” calls the delimited continuation $E$ from the point of the effect invocation up to the handle—with expression
Resumption example, formally

\[
\text{effect choose : int} \times \text{int} \rightarrow \text{int}
\]

\[
\text{handle } \#\text{choose}(1,2) + \#\text{choose}(10,20) \text{ with } \\
\quad \text{return (x:int)} \rightarrow x \\
\quad \text{choose (x:int,y:int)} \rightarrow \text{resume } x
\]

\[
\equiv \text{handle } E[\#\text{choose}(1,2)] \text{ with } h \\
\Rightarrow (\text{resume } x)[1/x,2/y][E^h/\text{resume}]
\]

\[
E \equiv [\emptyset] + \#\text{choose}(10,20) \\
h \equiv \text{return (x)} \rightarrow x
\]
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)
E ≡ return (x) → x

≡ handle E[#choose(1,2)] with h
  (resume 1) [E^h/resume]
≡ 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
```

Replace “resume v” with “handle E[v] with h”
Polymorphic effects

effect choose : \( \forall \alpha. \, \alpha \times \alpha \to \alpha \)

handle if \#choose(true,false) then \#choose(1,2) else \#choose(10,20) with return (x:int) \to x 
\( \Lambda \alpha. \) choose (x: \alpha,y: \alpha) \to resume x
Polymorphic effects

**Polymorphic signature**

effect choose : \( \forall \alpha. \ \alpha \times \alpha \rightarrow \alpha \)

handle if #choose(true, false)
then #choose(1, 2)
else #choose(10, 20) with
return (x:int) \( \rightarrow x \)
\( \wedge \alpha. \) choose (x: \alpha, y: \alpha) \( \rightarrow \) resume x
Polymorphic effects

\[\alpha := \text{bool}\]

**Effect**

\[\text{effect choose : } \forall \alpha. \alpha \times \alpha \rightarrow \alpha\]

**Handling**

\[\text{handle if } \#\text{choose(true,false)}\]

\[\text{then } \#\text{choose(1,2)}\]

\[\text{else } \#\text{choose(10,20)} \text{ with}\]

\[\text{return (x:int) } \rightarrow x\]

\[\forall \alpha.\text{choose (x:}\alpha,y:\alpha) \rightarrow \text{resume x}\]
Polymorphic effects

\[ \text{effect choose : } \forall \alpha. \; \alpha \times \alpha \rightarrow \alpha \]

\[ \text{handle if } \#\text{choose(1,2)} \text{ then } \#\text{choose(1,2)} \]
\[ \text{else } \#\text{choose(10,20)} \text{ with with} \]
\[ \text{return } (x:\text{int}) \rightarrow x \]
\[ \land \alpha. \text{choose } (x:\alpha,y:\alpha) \rightarrow \text{resume } x \]
Polymorphic effects

\[ \alpha := \text{bool} \]

\[ \text{effect choose : } \forall \alpha. \; \alpha \times \alpha \rightarrow \alpha \]

\[ \text{handle if } \#\text{choose(true,false)} \]
\[ \text{then } \#\text{choose(1,2)} \]
\[ \text{else } \#\text{choose(10,20)} \text{ with} \]
\[ \text{return } (x:\text{int}) \rightarrow x \]

\[ \Lambda \alpha. \; \text{choose (}x:\alpha,y:\alpha) \rightarrow \text{resume } x \]

Polymorphic signature

\[ \alpha := \text{int} \]

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

```plaintext
effect get_id : \forall \alpha. \text{unit} \to (\alpha \to \alpha)

handle
  let id : \forall \alpha. \alpha \to \alpha =
    \#get_id ()
in
if (id true)
  then (id 1) else 2
with
  return (x:int) \to x
\lambda \alpha.\text{get_id} (x:\text{unit}) \to
  \text{resume} (\lambda y:\alpha. \text{resume} (\lambda z:\alpha.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
  ∀α.get_id (x:unit) →
  resume (λy:α. resume (λz:α.y); y)
Counterexample to type safety

effect get_id : \forall \alpha. \text{unit} \rightarrow (\alpha \rightarrow \alpha)

handle

let id : \forall \alpha. \alpha \rightarrow \alpha =

\text{#get_id ()}

in

if (id true)
then (id 1)
else 2

with

return (x:int) \rightarrow x

\lambda \alpha. \text{get_id (x:unit) } \rightarrow

\text{resume (\lambda y:\alpha. resume (\lambda z:\alpha.y): y)}
Countereexample
effect get_id : ∀α. α → α = []

handle

let id : ∀α. α → α = 

\( \lambda y.(\text{resume}(\lambda z.y))[E^h/\text{resume}]; y \)

in

if (id true)
then (id 1) else 2

with

return (x:int) → x

\( \Lambda \alpha. \text{get_id}(x:\text{unit}) \rightarrow \)

resume (\( \lambda v: \alpha. \text{resume}(\lambda z: \alpha.v): v \))
Counterexample to type safety

effect get_id : \forall \alpha. \alpha \rightarrow \alpha = [] in
    if (id true) then (id 1) else 2

handle
    let id : \forall \alpha. \alpha \rightarrow \alpha = 
        \lambda y. (resume (\lambda z.y))[E^h/resume]; y
    in
    if (id true)
    then (id 1)
    else 2

with
    return (x:int) \rightarrow x

\lambda \alpha. get_id (x:unit) \rightarrow
    resume (\lambda y. resume (\lambda z:\alpha. y); y)
Counterexample to type safety

effect get_id : ∀α. α → α

handle

let id : ∀α. α → α =

\[ \lambda y. (\text{resume } (\lambda z. y))[E^h/\text{resume}] ; y \]

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)
Counterexample to type safety

effect get_id : \forall \alpha. \alpha \rightarrow \alpha = []

handle

let id : \forall \alpha. \alpha \rightarrow \alpha =

\lambda y. (\text{resume (} \lambda z.y\text{)})[E^h/\text{resume}]; y

in

if (\text{resume (} \lambda z.\text{true}\text{)})[E^h/\text{resume}]; true

then (id 1) else 2

with

return (x:int) \rightarrow x

\Lambda \alpha.\text{get_id (}x:\text{unit}\text{)} \rightarrow

\text{resume (} \lambda y: \alpha. \text{resume (} \lambda z: \alpha. y\text{); y}\text{)}
Counterexample to type safety

effect get_id : ∀α.α→α = [] in
  if (id true) then (id 1) else 2

handle
  let id : ∀α. α → α = λy.(resume (λz.y))[Eh/resume]; y
  in
    if (resume (λz.true))[Eh/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)
Counterexample to type safety

effect get_id : ∀α. α → α = λy. (resume (λz.y))[Eh/resume]; y

handle

let id : ∀α. α → α = λy. (resume (λz.y))[Eh/resume]; y

in

if (resume (λz.true))[Eh/resume] = true

then (id 1) else 2

with

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”
Counterexample to type safety

effect get_id : \forall \alpha. \alpha \rightarrow \alpha = []

handle

let id : \forall \alpha. \alpha \rightarrow \alpha =

\lambda y. (resume (\lambda z. y))[E^h/resume]; y

in

if handle E[\lambda z. true] with h; true
then (id 1) else 2

with

return (x:int) \rightarrow x

\Lambda \alpha. get_id (x:unit) \rightarrow
resume (\lambda y: \alpha. resume (\lambda z: \alpha. y); y)
Counterexample to type safety

effect get_id : \forall \alpha. \alpha \rightarrow \alpha = [] in

\lambda y. (resume (\lambda z.y))[E^h/resume]; y

handle

let id : \forall \alpha. \alpha \rightarrow \alpha =

\lambda y. (resume (\lambda z.y))[E^h/resume]; y

in

if handle \ E[\lambda z.true] with h; true

then (id 1) else 2

with

return (x:int) \rightarrow x

\land \alpha. get_id (x:unit) \rightarrow

resume (\lambda y: \alpha. resume (\lambda z: \alpha.y); y)
Counterexample to type safety

effect get_id : ∀α. α → α = [] in 
  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 
  let id : ∀α. α → α = λz.true in 
  if (id true) then (id 1) else 2

Λα. get_id (x:unit) → 
  resume (λy:α. resume (λz:α.y); y)
Our observation

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

• For clause “resume (\(\lambda y:\alpha.\) resume (\(\lambda z:\alpha.y\)); y)”, function (\(\lambda z:\alpha.y\)) is injected into a polymorphic context after replacing \(\alpha\) with \texttt{bool} and \(y\) with \texttt{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) → x
    ∀α.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 : \forall \alpha. \text{unit} \rightarrow (\alpha \rightarrow \alpha)

handle
   let id : \forall \alpha. \alpha \rightarrow \alpha =
       \#get_id ()
in
       if (id \text{true})
       then (id 1) else
       with
       return (x:int) \rightarrow
       \Lambda \alpha. \text{get_id} (x:\text{unit}) \rightarrow
       \text{resume} \ (\lambda y:\beta. \text{resume} \ ((\lambda z:\gamma.y); y))

Check: its type is $\beta \rightarrow \beta$

Check: its type is $\gamma \rightarrow \gamma$

The argument of a resumption must have a type obtained by renaming $\alpha$ to a fresh type variable.
Our idea
prohibition of sharing type variables

effect get_id : \forall \alpha. \unit \to (\alpha \to \alpha)

handle
    let id : \forall \alpha. \alpha \to \alpha =
        #get_id ()
in
    if (id true)
        then (id 1)
    with
    return (x:int) \to x
\lambda \alpha. get_id (x:unit) \to
    resume (\lambda y:\beta. resume (\lambda z:\gamma. z); y)

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

Typed at \gamma \to \gamma
Outline

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

• 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

Support for let-polymorphism
Syntax

A, B (types) ::= α | A → B | int | bool | ...

ε (effects) ::= { op }_i A_1 ... A_2

e (terms) ::= x A | c | λx.e | e_1 e_2 | let x = Λa.e | #op(A,e) | handle e with h | resume Λa.e

h (handlers) ::= return x←e | h; Λa.op(x)←e

≈ Allocate fresh type variables
Semantics

\[ e_1 \rightarrow e_2 \]

Evaluation rule

\[
\begin{array}{c}
E \neq [] \\
\hline
E[e_1] \rightarrow E[e_2]
\end{array}
\]

E (evaluation contexts) ::= [ ] | E e_2 | v_1 E | ... | let x = \texttt{\Lambda} a. E

Allows evaluation under type abstractions
Reduction of effect handling

\[ \text{handle } \#\text{op}(\forall \beta^J . A^I, \Lambda\beta^J . v, E^\beta^J) \text{ with } h \leadsto e[\text{handle } E^\beta^J \text{ with } h/\text{resume}]^{\forall \beta^J . A^I}_{\Lambda\beta^J . v} [A^I [\bot/\beta^J]/\alpha^I][v[\bot/\beta^J]/x] \]

(\text{where } h^{\text{op}} = \Lambda\alpha^I . \text{op}(x) \to e)

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

Resumption type

\[ \Gamma; R \vdash e : A | \epsilon \]

R (resumption types) ::= none | (\(\alpha, A, B \rightarrow \epsilon\) C)

Type variables bound in an operation clause

Argument type of an effect signature

Function type of continuation

\( e_0 \) and \( h \) are well typed

\( \Gamma, x : A; (\alpha, A, B \rightarrow \epsilon\) C \vdash e : C | \epsilon \)

\( \Gamma; R_0 \vdash \text{handle } e_0 \text{ with } h; \ \land \alpha.\text{op}(x) \rightarrow e : C | \epsilon \)

Typing rule for handle—with expressions
Resumption type

\[ \Gamma; R \quad e : A | \varepsilon \]

R (resumption types) ::= none \mid (\alpha, A, B \rightarrow \varepsilon C)

Effects that may occur in evaluation of \( e \)

Type variables bound in an operation clause

Argument type of an effect signature

Function type of continuation

\[ \varepsilon \subseteq \varepsilon' \]

\[ \begin{array}{c}
\Gamma, x : A[\beta/\alpha]; \\
(\alpha, A, B \rightarrow \varepsilon C) \vdash e : B[\beta/\alpha] \mid \varepsilon'
\end{array} \]

\[ \begin{array}{c}
\Gamma, x : D; \\
(\alpha, A, B \rightarrow \varepsilon C) \vdash \text{resume } \lambda \beta. e : C \mid \varepsilon'
\end{array} \]

Typing rule for resumptions
Type safety

If $\emptyset; \text{none} \quad e : A \mid \emptyset$, 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.