Signature Restriction for Polymorphic Algebraic Effects

Taro Sekiyama
National Institute of Informatics

Takeshi Tsukada
University of Tokyo

Atsushi Igarashi
Kyoto University

@ ICFP 2020

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar ’09; ’13]

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar '09; '13]

- Enable users to define their own effects
- Structure effectful programs
- Can define various effects
  - E.g. exception, backtracking, state, etc.

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar '09; '13]

- Enable users to define their own effects
- Structure effectful programs
- Can define various effects
  - E.g. exception, backtracking, state, etc.

- Type-based approach to program reuse
- Often appears implicitly (e.g., as let-polymorphism)
- Effects as well as terms can be polymorphic

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar ’09; ’13]

E.g. random choice

Three constructs for effects

1. Declaration
2. Operation call
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
   #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar '09; '13]

E.g. random choice

Three constructs
for effects

1. **Declaration**
2. Operation call
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar ’09; ’13]

E.g. random choice

Three constructs for effects

1. Declaration
2. Operation call
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
   #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar ’09; ’13]

E.g. random choice

Three constructs for effects

1. Declaration
2. **Operation call**
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar '09; '13]

E.g. random choice

Three constructs for effects

1. Declaration
2. **Operation call**
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar '09; '13]

E.g. random choice

Three constructs for effects

1. Declaration
2. Operation call
3. <u>**Definition**</u>

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
   #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

This talk

A new type-safe approach to combining

Algebraic effect handlers and **Polymorphism**

[Plotkin & Pretnar ’09; ’13]

E.g. random choice

Three constructs for effects

1. Declaration
2. Operation call
3. Definition

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

Problem

The unrestricted use of

Algebraic Effect Handlers **+** **Implicit Polymorphism**

is ***unsafe***

Due to the ability to manipulate delimited continuations

[Harper and Lillibridge '93; Sekiyama and Igarashi '19]

Prior approaches

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Prior approaches

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Prior approache

```
effect choose : ∀α. α × α ⇒ α
let g () =
 let f : ∀β. β × β → β =
  #choose(λ(x,y).x, λ(x,y).y)
 in (f (0,1), f (true, false))
handle g () with choose(x,y) → …
```

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Prior approaches

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
   #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))
handle g () with choose(x,y) → ...
```

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Prior approaches

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))
handle g () with choose(x,y) → ...
```

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Prior approaches

```
effect choose : ∀α. α × α ⇒ α
let g () =
  let f : ∀β. β × β → β =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))
handle g () with choose(x,y) → ...
```

Approach 1

Restricts operation calls in polymorphic expressions

👍 Able to address any effect

👎 Any operation call is restricted even if it doesn't need restriction

Existing approaches

- Value restriction [Tofte '90, Garrigue '04]
- Weak polymorphism [Appel+ '91]
- Closure typing [Leroy&Weis '91], etc.

Approach 2

Restricts effect handlers (definitions)

👍 Restricts only operation calls of possibly unsafe effects

👎 Unclear to mix safe and possibly unsafe effects

Existing approaches

- Handler restriction [Sekiyama & Igarashi '19]

Our approach

- Restricts ***the types of effect operations***
- We can determine if any use of effects is safe only by examining the operation type

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β  =
    #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

Our approach

- Restricts ***the types of effect operations***
- We can determine if any use of effects is safe only by examining the operation type

```
effect choose : ∀α. α × α ⇒ α

let g () =
  let f : ∀β. β × β → β =
   #choose(λ(x,y).x, λ(x,y).y)
  in (f (0,1), f (true, false))

handle g () with choose(x,y) → ...
```

Ensures **choose** is safe no matter how it is used

This work

- ***Signature restriction (SR)*** to ensure safety of effects with polymorphism
  - The SR accepts effects that can be safely used anywhere without other restriction
  - The SR is
    - 👍 *Simple*: it only examines the ***typed signatures (interfaces)*** of effect operations
    - 👍 *Permissive*: it is satisfied by many practical effects (such as exception, nondeterminism, input streaming)
    - 👍 *Scalable*: it can easily support basic constructs (such as products, sums, and lists)
- A sound type system assuming all effects satisfy the SR

This work

- An effect system allowing the use of both effects satisfying and not satisfying the SR
  - Effects satisfying the SR can be used ***anywhere without restriction***
  - Effects not satisfying the SR can be used ***only in monomorphic expressions***

- An artifact that implements a tiny ML-like language enforcing all effects to satisfy the SR

https://github.com/skymountain/MLSR

Signature restriction

- Determines safety of effects with the signature

$$\mathbf{op} : \forall \alpha.\ \tau_1 \Rightarrow \tau_2$$

  only by examining polarities of α in τ_1 and τ_2

- **op** satisfies the SR if and only if
  - α occurs ***only negatively*** or ***strictly positively*** in τ_1
  - α occurs ***only positively*** in τ_2

Implementation: https://github.com/skymountain/MLSR

Signature restriction

- Determines safety of effects with the signature

$$\mathbf{op} : \forall \alpha.\ \tau_1 \Rightarrow \tau_2$$

  only by examining polarities of α in τ_1 and τ_2

- **op** satisfies the SR if and only if
  - α occurs ***only negatively*** or ***strictly positively*** in τ_1
  - α occurs ***only positively*** in τ_2

Ex. $(\alpha_1 \to \alpha_2) \to \alpha_3$

α_1 : non-strictly positive

α_2 : negative

α_3 : strictly positive

Implementation: https://github.com/skymountain/MLSR

Examples

$\mathtt{op} : \forall\alpha.\ \tau_1 \Rightarrow \tau_2$ satisfies the SR iff

- α occurs only negatively or strictly positively in τ_1
- α occurs only positively in τ_2

Operations satisfying the signature restriction

- $\mathtt{choose} : \forall\alpha.\alpha \times \alpha \Rightarrow \alpha$
  - Usage: random choice and nondeterminism
- $\mathtt{fail} : \forall\alpha.\mathtt{unit} \Rightarrow \alpha$
  - Usage: exception raising
- $\mathtt{satisfy} : \forall\alpha.(\mathtt{str} \to \mathtt{unit} + (\mathtt{str} \times \alpha)) \Rightarrow \alpha$
  - Usage: input streaming and parser combinators

Implementation: https://github.com/skymountain/MLSR

Future work

- Support for features in full-fledge languages
  - Type inference, particularly for the effect system
  - General algebraic datatypes
- CPS-based foundation
  - Is it possible to achieve type-preserving CPS transformation for the SR?
- Applying the SR to other mechanisms to address user-defined effects (e.g., monads)

Implementation: https://github.com/skymountain/MLSR

Conclusion

- Naive introduction of effects into a polymorphic language is unsafe
- We propose signature restriction to determine safety of effects with polymorphism
- Signature restriction is
  - Simple: it only examines the types of effects
  - Permissive: it accepts many useful effects
  - Scalable: it can easily support other constructs
- Implementation available at:

  https://github.com/skymountain/MLSR