Algebraic Subtyping for Algebraic Effects and Handlers

Axel Faes
KU Leuven

What are algebraic effects and handlers?

Algebraic effects and handlers

Model impure behaviour such as mutable state or I/O [1]

Exceptions on steroids

[1] Pretnar, M., 2015. An introduction to algebraic effects and handlers. Electr. Notes Theor. Comput. Sci, 319, pp.19-35.

Algebraic effects and handlers

Model impure behaviour such as mutable state or I/O [1]

Exceptions on steroids

```
effect Op : unit -> int

let someFun b =
        handle (
                if (b == 0) then
                        let a = #Op () in a
                else
                        b
        ) with
                | val x -> x + 1
                | #Op () cont -> cont 1
```

[1] Pretnar, M., 2015. An introduction to algebraic effects and handlers. Electr. Notes Theor. Comput. Sci, 319, pp.19-35.

Algebraic effects and handlers

```
effect Op : unit -> int

let someFun b =
        handle (
                if (b == 0) then
                        let a = #Op () in a
                else
                        b
        ) with
                | val x -> x + 1
                | #Op () cont -> cont 1
```

```
someFun 2
        => 3

someFun 0
        => 2
```

N-queens

```
effect Decide : unit -> bool
effect Fail : unit -> empty

let rec choose = function
    | [] -> (match (#Fail ()) with)
    | x :: xs -> if #Decide () then x else choose xs
```

n by n chessboard
n queens

no queen can attack
another queen.

N-queens

```
effect Decide : unit -> bool
effect Fail : unit -> empty

let rec choose = function
    | [] -> (match (#Fail ()) with)
    | x :: xs -> if #Decide () then x else choose xs

let choose_all = handler
    | val x -> [x]
    | #Decide _ k -> k true @ k false
    | #Fail _ _ -> []
```

N-queens

```
effect Decide : unit -> bool
effect Fail : unit -> empty

let rec choose = function
    | [] -> (match (#Fail ()) with)
    | x :: xs -> if #Decide () then x else choose xs

let optionalize = handler
    | val y -> (Some y)
    | #Decide _ k -> (match k true with Some x -> Some x | None -> k false)
    | #Fail _ _ -> None
```

Eff programming language [1]

```
effect Op : unit -> int

let someFun b =
        handle (
                if (b == 0) then
                        let a = #Op () in a
                else
                        b
        ) with
                | val x -> x + 1
                | #Op () cont -> cont 1
```

Functional language

Algebraic effects and handlers

ML style language

[1] http://www.eff-lang.org/

Eff programming language [1]

```
effect Op : unit -> int

let someFun b =
        handle (
                if (b == 0) then
                        let a = #Op () in a
                else
                        b
        ) with
                | val x -> x + 1
                | #Op () cont -> cont 1
```

Functional language

Algebraic effects and handlers

ML style language

- Type inference

[1] http://www.eff-lang.org/

Type inference

```
let a = 5
```

Functional language

Algebraic effects and handlers

ML style language

    Type inference

Type inference

```
let a = 5
```

‘a’ has type ‘int’

Functional language

Algebraic effects and handlers

ML style language

    Type inference

Type inference

```
let a = 5

let comp b = b < 1.2
```

Functional language

Algebraic effects and handlers

ML style language

- Type inference

Type inference

```
let a = 5

let comp b = b < 1.2
```

‘comp’ has type ‘float -> bool’

Functional language

Algebraic effects and handlers

ML style language

    Type inference

Type inference

```
let a = 5

let comp b = b < 1.2
```

‘comp’ has type ‘float -> bool’

int < float

Functional language

Algebraic effects and handlers

ML style language

Type inference

Eff programming language [1]

Functional language

Algebraic effects and handlers

ML style language

  Type inference

  Subtyping

Problems

[1] http://www.eff-lang.org/

Eff programming language [1]

Functional language

Algebraic effects and handlers

ML style language

- Type inference
- Subtyping

→

Problems

- Slow to compile
- Hard to debug

[1] http://www.eff-lang.org/

Example: Twice

```
let twice f x =
    f (f x)
```

What does this function do?

Example: Twice

```
let twice f x =
    f (f x)
```

What does this function do?

f is a function

Example: Twice

```
let twice f x =
    f (f x)
```

What does this function do?

f is a function
    accepts x and (f x)

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \to \gamma$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \text{ -> } \gamma$

$\alpha \leq \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \to \gamma$

$\alpha \leq \beta, \gamma \leq \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \to \gamma$

$twice : (\beta \to \gamma) \to \alpha \to \gamma \mid \alpha \leq \beta, \gamma \leq \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

x : α

f : β -> γ

twice : (β -> γ) -> α -> γ | α ≤ β, γ ≤ β

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \to \gamma$

$\text{twice} : (\beta \to \gamma) \to \alpha \to \gamma \mid \alpha \leq \beta, \gamma \leq \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Subtyping

$x : \alpha$

$f : \beta \rightarrow \gamma$

$\text{twice} : (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \mid \alpha \leq \beta, \gamma \leq \beta$

If constraints solved:

Get actual type

https://cdn-images-1.medium.com/max/888/1*XJ7YrEAwzoFSql6HFT4d4g.jpeg

Algebraic Subtyping for Algebraic Effects and Handlers

Dolan, Stephen, and Alan Mycroft. "Polymorphism, subtyping, and type inference in MLsub." In ACM SIGPLAN Notices, vol. 52, no. 1, pp. 60-72. ACM, 2017.

Algebraic Subtyping for Algebraic Effects and Handlers

Stephen Dolan

Dolan, Stephen, and Alan Mycroft. "Polymorphism, subtyping, and type inference in MLsub." In ACM SIGPLAN Notices, vol. 52, no. 1, pp. 60-72. ACM, 2017.

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

Result of f

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

Result of f
    Output of twice

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

Result of f
    Output of twice
    Input of f

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

- $x : \alpha$
- $\text{return} : \beta$
- $f : \alpha \text{ -> } ?$

$\text{twice} : (\alpha \text{ -> } ?) \text{ -> } \alpha \text{ -> } \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

$x : \alpha$

$\text{return} : \beta$

$f : \alpha \rightarrow \alpha \sqcap \beta$

=> accept both α AND β

$\text{twice} : (\alpha \rightarrow \alpha \sqcap \beta) \rightarrow \alpha \rightarrow \beta$

Example: Twice

```
let twice f x =
    f (f x)
```

Algebraic Subtyping

twice : $(\alpha \to \alpha \sqcap \beta) \to \alpha \to \beta$

Subtyping:

twice : $(\beta \to \gamma) \to \alpha \to \gamma \mid \alpha \leq \beta, \gamma \leq \beta$

Effects?

Algebraic effects and handlers

Algebraic subtyping

???

Algebraic subtyping + effects

(pure) type A, B	$::=$	$\mathtt{bool}$	bool type
	$\mid$	$A \rightarrow \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	**handler type**
	$\mid$	α	type variable
	$\mid$	$\mu\alpha.A$	recursive type
	$\mid$	$\top$	top
	$\mid$	$\bot$	bottom
	$\mid$	$A \sqcap B$	intersection
	$\mid$	$A \sqcup B$	union
dirty type $\underline{C}, \underline{D}$	$::=$	$A \,!\, \Delta$	

Algebraic subtyping + effects

dirt Δ	$::=$	Op	operation
	$\mid$	δ	dirt variable
	$\mid$	$\emptyset$	empty dirt
	$\mid$	$\Delta_1 \sqcap \Delta_2$	intersection
	$\mid$	$\Delta_1 \sqcup \Delta_2$	union

What are the semantics?

Algebraic subtyping + effects

What are the semantics?

Set operations?

$(Op \sqcup Op2) \sqcap (Op \sqcup Op3)$

Algebraic subtyping + effects

What are the semantics?

Set operations?

$(Op \sqcup Op2) \sqcap (Op \sqcup Op3)$

$\Rightarrow Op$

Algebraic subtyping + effects

What are the semantics?

Set operations?

$(Op \sqcup Op2) \sqcap (Op \sqcup Op3)$

$\Rightarrow Op$

$\alpha \sqcap \beta$

accept both α AND β

Algebraic subtyping + effects

What are the semantics?

Set operations?

$(Op \sqcup Op2) \sqcap (Op \sqcup Op3)$

=> Op

$\alpha \sqcap \beta$

accept both α AND β

Input vs Output

=> Inputs cause $\sqcap$

=> Outputs cause $\sqcup$

Example

```
effect Op : unit -> int
effect Op2 : int -> unit

let select p v d =
    if (p v) then
        (v, (fun () -> #Op () ))
    else
        (d, (fun () -> #Op2 () ))
```

p : (α -> bool ! □)

v : α

d : β

Example

```
effect Op : unit -> int
effect Op2 : int -> unit

let select p v d =
    if (p v) then
        (v, (fun () -> #Op () ))
    else
        (d, (fun () -> #Op2 () ))
```

p : (α -> bool ! □)

v : α

d : β

select: (α ⊔ β) x (unit -> (unit ⊔ int) ! (Op ⊔ Op2))

Example

```
effect Op : unit -> int
effect Op2 : int -> unit
effect Op3 : int -> unit

let h = handler
    | val x -> x + 1
    | #Op () k -> #Op3 1
    | #Op2 n k -> k ()
```

h: int ! (Op ⊓ Op2) => unit ! Op3

Design of a type-&-effect system

Formulation of typing rules

Define relationship to subtyping

Biunification algorithm (input <-> output)

Type inference algorithm

Keep design close to Dolan’s design

Type-&-effect simplification

```
effect Get : unit -> int
effect Put : int -> unit

let rec loop n =
    if (b == 0) then ()
    else (#Put ((#Get ()) + 1); loop (n - 1))
```

Inferred types can be huge

Type-&-effect simplification

Inferred types can be huge

```
effect Get : unit -> int
effect Put : int -> unit

let rec loop n =
    if (b == 0) then ()
    else (#Put ((#Get ()) + 1); loop (n - 1))
```

$$(\alpha_1 \sqcap \alpha_2 \sqcap \alpha_3 \sqcap \alpha_4 \sqcap \alpha_5 \sqcap \alpha_6 \sqcap \alpha_7 \sqcap int)$$

$$\rightarrow \quad \alpha_8 \sqcup unit \ ! \ (\delta_1 \sqcup \delta_2 \sqcup \delta_3 \sqcup Get \sqcup Put)$$

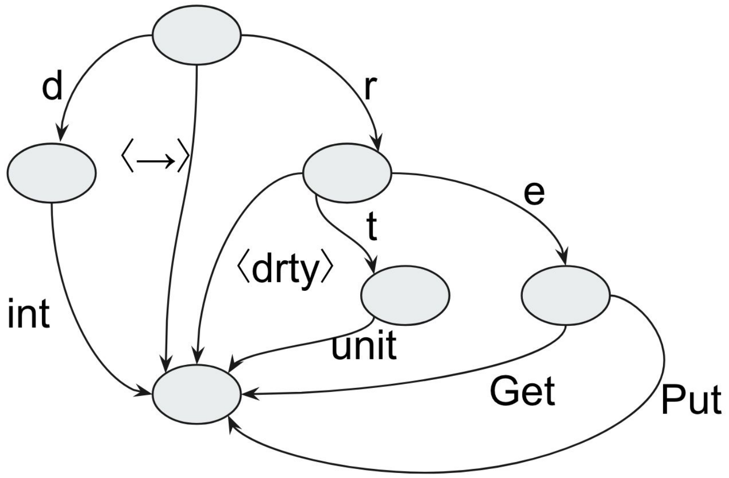
Type-&-effect simplification

```
effect Get : unit -> int
effect Put : int -> unit

let rec loop n =
    if (b == 0) then ()
    else (#Put ((#Get ()) + 1); loop (n - 1))
```

$$(\alpha_1 \sqcap \alpha_2 \sqcap \alpha_3 \sqcap \alpha_4 \sqcap \alpha_5 \sqcap \alpha_6 \sqcap \alpha_7 \sqcap int)$$
$$\to \quad \alpha_8 \sqcup unit \ ! \ (\delta_1 \sqcup \delta_2 \sqcup \delta_3 \sqcup Get \sqcup Put)$$

Inferred types can be huge

These can be simplified using automata



Type-&-effect simplification

```
effect Get : unit -> int
effect Put : int -> unit

let rec loop n =
    if (b == 0) then ()
    else (#Put ((#Get ()) + 1); loop (n - 1))
```

Inferred types can be huge

These can be simplified using automata

$$int \rightarrow unit \; ! \; (Get \sqcup Put)$$

Proofs

Instantiation
Weakening
Substitution
Soundness

Equivalence of typing rules
Correctness of biunification
Principality of types
Correctness of simplification

Reason to stay close to Dolan

Implementation

Instead of using a ‘toy’ language

Use Eff programming language

Replace the type inference engine

~2700 loc ⇔ ~6000

Simplification is not yet implemented

Implementation

Instead of using a ‘toy’ language

Use Eff programming language

Replace the type inference engine

~2700 loc ⇔ ~6000

Simplification is not yet implemented

Testing against other systems

Subtyping

Type inference performance benchmark

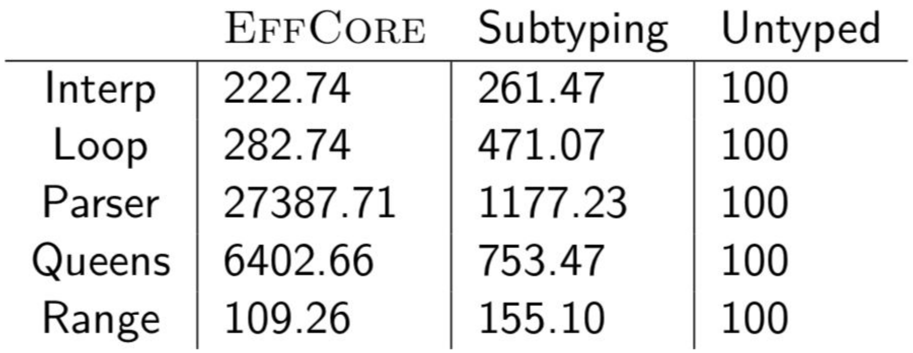
Type comparison

Empirical evaluation

3 systems

5 programs

10000 iterations



	EffCore	Subtyping	Untyped
Interp	222.74	261.47	100
Loop	282.74	471.07	100
Parser	27387.71	1177.23	100
Queens	6402.66	753.47	100
Range	109.26	155.10	100

Result

Extension of Algebraic Subtyping

For algebraic effects and handlers

Type System

Algorithmic

Proofs

Implementation

Future Work

Simplification algorithm implementation

Biunification with Type Automata

Optimization of algebraic effects and handlers

Simplify constraint generation for types AND EFFECTS by using extended algebraic subtyping

Questions?

Type-&-effect simplification

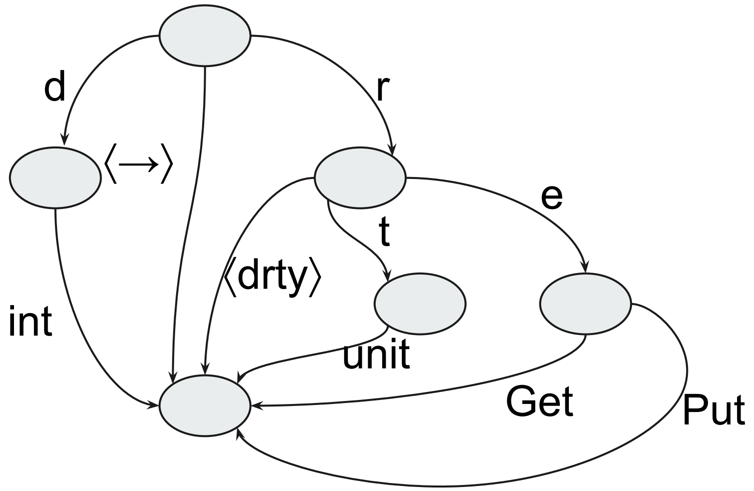
```
effect Get : unit -> int
effect Put : int -> unit

let rec loop n =
    if (b == 0) then ()
    else (#Put ((#Get ()) + 1); loop (n - 1))
```

$$(\alpha_1 \sqcap \alpha_2 \sqcap \alpha_3 \sqcap \alpha_4 \sqcap \alpha_5 \sqcap \alpha_6 \sqcap \alpha_7 \sqcap int)$$

$$\to \quad \alpha_8 \sqcup unit \ ! \ (\delta_1 \sqcup \delta_2 \sqcup \delta_3 \sqcup Get \sqcup Put)$$

Inferred types can be huge

These can be simplified using automata


d
r
⟨→⟩
e
t
⟨drty⟩
int
unit
Get
Put