

GETTING A HANDLE ON (SCOPED) EFFECTS

FP DAG 2022

ROGER BOSMAN*

BIRTHE VAN DEN BERG*

WENHAO TANG*

TOM SCHRIJVERS

The logo of KU Leuven, featuring the text "KU LEUVEN" in white capital letters on a dark blue rectangular background.

KU LEUVEN

ROGER.BOSMAN@KULEUVEN.BE

22-04-2022

GOAL: EXTEND AN EXISTING
CALCULUS THAT SUPPORTS
ALGEBRAIC EFFECT & HANDLERS
TO ONE THAT SUPPORTS
SCOPED EFFECTS & HANDLERS

- What are algebraic effects & handlers

- What are algebraic effects & handlers
- A calculus with algebraic effects & handlers

- What are algebraic effects & handlers
- A calculus with algebraic effects & handlers
- Not all effects are algebraic, some are **scoped**

- What are algebraic effects & handlers
- A calculus with algebraic effects & handlers
- Not all effects are algebraic, some are **scoped**
- A calculus with **scoped effects & handlers**

THE IDEA: SEPARATE
SYNTAX FROM SEMANTICS
BY SPLITTING INTO
OPERATIONS AND HANDLERS

OPERATIONS

Operations represent the **invocation** of some effect, and are of shape

op ℓ v ($y . c$)

OPERATIONS

Operations represent the invocation of some effect, and are of shape

op ℓ v ($y.c$)

Example: for **nondeterministic choice** we have the choose operation:

```
cND = op choose () (b.if b then return "heads"  
                    else return "tails")
```

OPERATIONS

Operations represent the invocation of some effect, and are of shape

$\underbrace{\text{op}}_{\text{keyword}} \underbrace{\ell}_{\text{effect label}} \underbrace{v}_{\text{argument}} (\underbrace{y}_{\text{result}} . \underbrace{c}_{\text{continuation}})$

Example: for nondeterministic choice we have the choose operation:

$c_{ND} = \text{op choose } () (b . \text{if } b \text{ then return "heads"}$
 $\text{else return "tails"})$

OPERATIONS

Operations represent the invocation of some effect, and are of shape

$\underbrace{\text{op}}_{\text{keyword}} \quad \underbrace{\ell}_{\text{effect label}} \quad \underbrace{v}_{\text{argument}} \quad \underbrace{(y)}_{\text{result}} \cdot \underbrace{c}_{\text{continuation}} \quad)$

Example: for nondeterministic choice we have the choose operation:

$c_{ND} = \text{op choose } () (b . \text{if } b \text{ then return "heads"}$
 $\qquad \qquad \qquad \text{else return "tails"})$

choose has **signature** $() \rightarrow \text{Bool}$

HANDLERS

Handlers give **meaning** to **operations** by giving **clauses**

- A **return** clause that handles values
- A number of **op** clauses that handle operations

HANDLERS

Handlers give **meaning** to **operations** by giving **clauses**

- A **return** clause that handles values
- A number of **op** clauses that handle operations

For example, h_{ND} interprets `choose` by collecting all branches in a list:

$$h_{ND} = \text{handler } \{ \text{return } x \quad \mapsto \text{return } [x] \\ , \text{op choose } _ k \mapsto \text{do } xs \leftarrow k \text{ true} ; \\ \text{do } ys \leftarrow k \text{ false} ; \\ xs ++ ys \}$$

HANDLERS

Handlers give **meaning** to **operations** by giving **clauses**

- A **return** clause that handles values
- A number of **op** clauses that handle operations

For example, h_{ND} interprets `choose` by collecting all branches in a list:

$$h_{ND} = \text{handler } \{ \text{return } x \quad \mapsto \text{return } [x] \\ , \text{op choose } _ k \mapsto \text{do } xs \leftarrow k \text{ true} ; \\ \text{do } ys \leftarrow k \text{ false} ; \\ xs ++ ys \}$$

Apply with \star

```
ghci> hND  $\star$  cND
["heads", "tails"]
```

BENEFIT 1: OVERLOADING

Benefit 1: **Overloading**

We can give a **different** semantic to the **same** syntax by changing the handler:

```
 $c_{ND} = \text{op choose } () (b. \text{if } b \text{ then return "heads"}$   
 $\qquad \qquad \qquad \text{else return "tails"})$ 
```

```
 $h_{1st} = \text{handler } \{ \text{return } x \quad \mapsto \text{return } x$   
 $\qquad \qquad \qquad , \text{op choose } \_ k \mapsto k \text{ true} \}$ 
```

```
ghci>  $h_{1st} \star c_{ND}$   
"heads"
```

MODULAR COMPOSITION

Benefit 2: **Modular composition**

We can write handlers that handle a subset of effects, and compose them.

Consider effect `get : () → String` in combination with `choose`:

```
 $c_{c,g} = \text{op choose } () \text{ (} b. \text{if } b \text{ then return "heads"}$   
 $\qquad\qquad\qquad \text{else op get } () \text{ (} x. \text{return } x \text{))}$ 
```

```
 $h_{\text{get}} = \text{handler } \{ \text{return } x \qquad \mapsto \text{return } x$   
 $\qquad\qquad\qquad , \text{op get } \_ k \qquad \mapsto k \text{ "surprise"} \}$ 
```

```
ghci>  $h_{ND} \star (h_{\text{get}} \star c_{c,g})$   
["heads", "surprise"]
```


SEQUENCING

Final component: sequencing. How do we sequence computations? **Do statements!**

```
do  $x \leftarrow$  op choose () ( $b$ .if ... "heads" ...); return  $x \oplus x$ 
```

SEQUENCING

Final component: sequencing. How do we sequence computations? Do statements!

do $x \leftarrow$ **op** choose () (b . **if** ... "heads" ...); **return** $x \uplus x$

Operations **commute with sequencing** as witnessed by the **algebraicity property**:

Algebraicity

do $x \leftarrow$ **op** $\ell \vee (y . c_1) ; c_2 \equiv$ **op** $\ell \vee (y .$ **do** $x \leftarrow c_1 ; c_2)$

SEQUENCING

Final component: sequencing. How do we sequence computations? Do statements!

$$\begin{aligned} & \mathbf{do} \ x \leftarrow \mathbf{op} \ \text{choose} \ () \ (b.\mathbf{if} \ \dots \text{"heads"} \ \dots) ; \mathbf{return} \ x \ ++ \ x \\ \equiv & \ \mathbf{op} \ \text{choose} \ () \ (b.\mathbf{do} \ x \leftarrow \mathbf{if} \ \dots \text{"heads"} \ \dots ; \mathbf{return} \ x \ ++ \ x) \end{aligned}$$

Operations commute with sequencing as witnessed by the **algebraicity property**:

Algebraicity

$$\mathbf{do} \ x \leftarrow \mathbf{op} \ \ell \ v \ (y.c_1) ; c_2 \equiv \mathbf{op} \ \ell \ v \ (y.\mathbf{do} \ x \leftarrow c_1 ; c_2)$$

SEQUENCING

Final component: sequencing. How do we sequence computations? Do statements!

$$\begin{aligned} & \mathbf{do} \ x \leftarrow \mathbf{op} \ \text{choose} \ () \ (b.\mathbf{if} \ \dots \text{"heads"} \ \dots) ; \mathbf{return} \ x \ ++ \ x \\ \equiv & \ \mathbf{op} \ \text{choose} \ () \ (b.\mathbf{do} \ x \leftarrow \mathbf{if} \ \dots \text{"heads"} \ \dots ; \mathbf{return} \ x \ ++ \ x) \end{aligned}$$

Operations commute with sequencing as witnessed by the algebraicity property:

Algebraicity

$$\mathbf{do} \ x \leftarrow \mathbf{op} \ \ell \ v \ (y.\ c_1) ; c_2 \equiv \mathbf{op} \ \ell \ v \ (y.\ \mathbf{do} \ x \leftarrow c_1 ; c_2)$$

We will **revisit** this when discussing scoped effects

You could make a calculus out of this!

Version here based on Eff by Bauer and Pretnar [1, 2].

Many others: Frank [6], Effekt [3], Links [4], Koka [5], Multicore OCaml [8].

You could make a calculus out of this!

Version here based on Eff by Bauer and Pretnar [1, 2].

Many others: Frank [6], Effekt [3], Links [4], Koka [5], Multicore OCaml [8].

Note: I will be ignoring types for the most part here

$$\begin{array}{ll}
 \text{values } v & ::= () \mid x \mid \lambda x. c \mid h \\
 \text{handlers } h & ::= \mathbf{handler} \{ \mathbf{return} \ x \mapsto c \\
 & \qquad \qquad \qquad , ops \} \\
 ops & ::= . \\
 & \mid \mathbf{op} \ \ell \ x \ k \mapsto c, ops \\
 \text{computations } c & ::= \mathbf{return} \ v \\
 & \mid \mathbf{op} \ \ell \ v \ (y. c) \\
 & \mid v \star c \\
 & \mid \mathbf{do} \ X \leftarrow c_1 ; c_2 \\
 & \mid v_1 \ v_2
 \end{array}$$

DONE?

So, great! Are we done?

DONE?

So, great! Are we done?

As already mentioned in the introduction, **not all effects are algebraic**

DONE?

So, great! Are we done?

As already mentioned in the introduction, not all effects are algebraic

Class of effects considered today: **scoped effects**

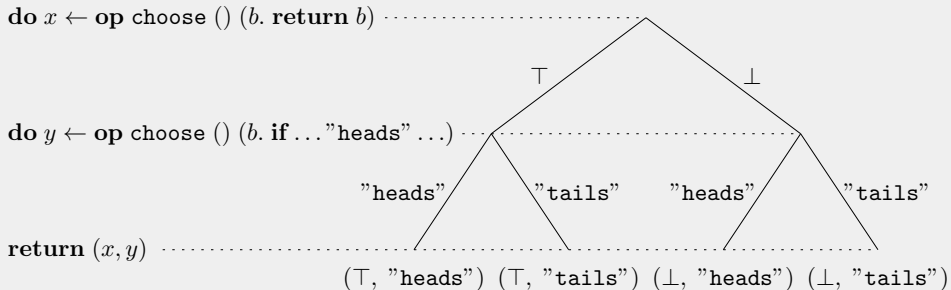
EFFECT once I

C_{once} features 2 occurrences of choose:

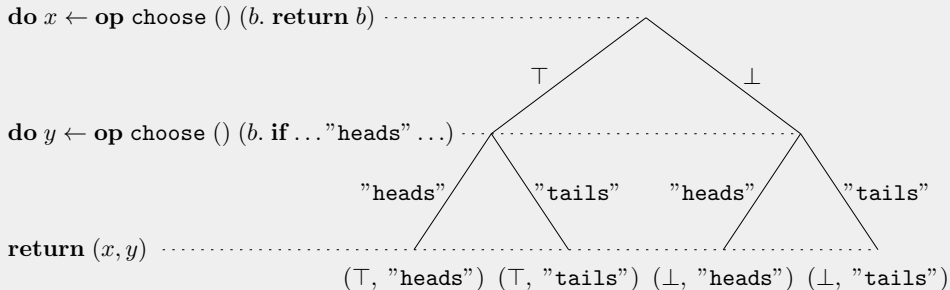
```
 $C_{\text{once}} = \text{do } x \leftarrow \text{op choose } () (b.\text{return } b);$   
           $y \leftarrow \text{op choose } () (b.\text{if } b \dots \text{"heads"} \dots);$   
           $\text{return } (x, y)$ 
```

```
ghci>  $h_{ND} \star C_{\text{once}}$   
[(true, "heads"), (true, "tails"),  
 (false, "heads"), (false, "tails")]
```

EFFECT once II

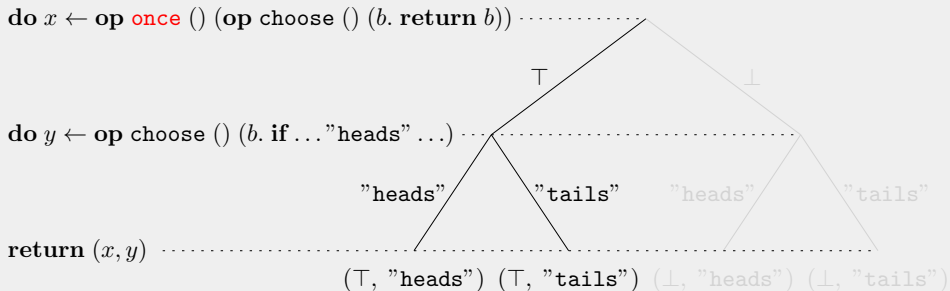


EFFECT once II



Idea: limit **the first choose** to only return the **first result** by operation **once** : $() \rightarrow ()$

EFFECT once II



Idea: limit **the first choose** to only return the **first result** by operation **once** : $() \rightarrow ()$

EFFECT once III

once : () \rightarrow ()

```
conce = do x  $\leftarrow$  op once () (_. op choose () (b. return b));  
      do y  $\leftarrow$  op choose () (b. if b ... "heads" ...);  
      return (x, y)
```

```
honce = handler { return x            $\mapsto$  ...  
                  , op choose _ k  $\mapsto$  ...  
                  , op once      _ k  $\mapsto$  do ts  $\leftarrow$  k (); head ts }
```

EFFECT once III

once : () \rightarrow ()

```
Conce = do x  $\leftarrow$  op once () (_.op choose () (b.return b));  
      do y  $\leftarrow$  op choose () (b.if b... "heads" ...);  
      return (x, y)
```

```
honce = handler { return x            $\mapsto$  ...  
                  , op choose k  $\mapsto$  ...  
                  , op once    k  $\mapsto$  do ts  $\leftarrow$  k (); head ts }
```

We would **expect**:

```
ghci> honce  $\star$  Conce  
[(true, "heads"), (true, "tails")]
```


EFFECT once III

once : () \rightarrow ()

```
Conce = do x  $\leftarrow$  op once () (_.op choose () (b.return b));  
      do y  $\leftarrow$  op choose () (b.if b... "heads" ...);  
      return (x, y)
```

```
honce = handler { return x            $\mapsto$  ...  
                  , op choose _ k  $\mapsto$  ...  
                  , op once      _ k  $\mapsto$  do ts  $\leftarrow$  k (); head ts }
```

We would expect:

```
ghci> honce  $\star$  Conce  
[(true, "heads"), (true, "tails")]
```

However, we get:

```
[(true, "heads")]
```

EFFECT once III

once : () → ()

```
Conce = do x ← op once () (_. op choose () (b. return b));  
      do y ← op choose () (b. if b ... "heads" ...);  
      return (x, y)
```

```
honce = handler { return x           ↦ ...  
                  , op choose k ↦ ...  
                  , op once    k ↦ do ts ← k (); head ts }
```

We would expect:

```
ghci> honce ★ Conce  
[(true, "heads"), (true, "tails")]
```

However, we get:

```
[(true, "heads")]
```

How did this happen?

once **delimits a scope**:

```
 $c_{once} = \text{do } x \leftarrow \text{op once } () (\text{op choose } () (b.\text{return } b));$   
           $\text{do } y \leftarrow \text{op choose } () (b.\text{if } b \dots \text{"heads"} \dots);$   
           $\text{return } (x, y)$ 
```

- The first choose is **in scope**: we want to take the first result of this sub-computation **only**
- The rest is **out of scope**: it should be **unaffected**

The problem lies in the **algebraicity property**: it does not hold for once:

$$\mathbf{do} \ x \leftarrow \mathbf{op} \ \text{once} \ () \ (y . c_1); c_2 \not\equiv \mathbf{op} \ \text{once} \ () \ (y . \mathbf{do} \ x \leftarrow c_1; c_2)$$

Computation c_2 is supposed to be **out of scope**, but algebraicity brings it into scope!

The problem lies in the **algebraicity property**: it does not hold for once:

$$\mathbf{do} \ x \leftarrow \mathbf{op} \ \text{once} \ () \ (y . c_1) ; c_2 \not\equiv \mathbf{op} \ \text{once} \ () \ (y . \mathbf{do} \ x \leftarrow c_1 ; c_2)$$

Computation c_2 is supposed to be **out of scope**, but algebraicity brings it into scope!

Effects that delimit a scope are a class separate from algebraic effects, called **scoped effects** [9]

TWO COMPUTATIONS

So, how do we model scoped effects?

We take algebraic operations and add a second computation:

- A **scoped computation** (this one is new)
- A **continuation** (same as the algebraic)

op ℓ v $(z . c)$

sc ℓ v $(y . c_1) (z . c_2)$

TWO COMPUTATIONS

So, how do we model scoped effects?

We take algebraic operations and add a second computation:

- A **scoped computation** (this one is new)
- A **continuation** (same as the algebraic)

op ℓ v $(z.c)$

sc ℓ v $(y.c_1) (z.c_2)$

Algebraicity (generalized)

$$\begin{aligned} & \text{do } x \leftarrow \text{sc } \ell \ v \ (y.c_1) \ (z.c_2) ; c_3 \\ \equiv & \text{sc } \ell \ v \ (y.c_1) \ (z. \text{do } x \leftarrow c_2 ; c_3) \end{aligned}$$

EFFECT once AS **sc**

Same computation, with **sc**:

```
sc once () (op choose () (b. return b))  
    (x. do y ← op choose () (b. if b ... "heads" ...);  
    return (x, y)  
    )
```

Handler is extended with additional argument p : the **scoped computation**:

```
 $h_{\text{once}} = \text{handler } \{ \text{return } x \quad \mapsto \dots$   
    , op choose –  $k \mapsto \dots$   
    , sc once –  $p \ k \mapsto \text{do } ts \leftarrow p ();$   
     $\text{do } t \leftarrow \text{head } ts;$   
     $k \ t$   
    }
```


ADD TO OUR CALCULUS

$$\begin{array}{ll} \text{values } v & ::= () \mid x \mid \lambda x. c \mid h \\ \text{handlers } h & ::= \text{handler } \{ \text{return } x \mapsto c_r, \text{opr}s \} \end{array} \quad \begin{array}{l} \text{return} \\ \text{effect clauses} \end{array}$$

```

opr. clauses oprs ::=  $\cdot$ 
                    | op  $\ell \ x \ k \mapsto c, oprs$            algebraic eff. clause
                    | sc  $\ell \ x \ p \ k \mapsto c, oprs$        scoped eff. clause

```

computations c	$::=$	return v	return value
		op ℓ v $(y.c)$	algebraic opr.
		sc ℓ v $(y.c_1) (z.c_2)$	scoped opr.
		$v \star c$	handle
		do $x \leftarrow c_1 ; c_2$	do-statement
		$v_1 v_2$	application

FORWARDING I

We need to talk about forwarding.

Forwarding occurs when a handler applied to some effect ℓ does not contain a clause for ℓ .

$$h_{\text{once}} = \mathbf{handler} \{ \mathbf{return} \dots, \mathbf{op} \text{ choose } \dots, \mathbf{sc} \text{ once } \dots \}$$

h_{once} handles choose and once, but not algebraic effect

rand : $() \rightarrow \text{Int}$.

FORWARDING II

h_{once} handles choose and once, but not algebraic effect
rand : $() \rightarrow \text{Int}$.

How do we evaluate the following?

$h_{once} \star \mathbf{op} \text{ rand } () (i.c)$

c may contain effects that h_{once} handles, so we **cannot do nothing**

FORWARDING II

h_{once} handles choose and once, but not algebraic effect
rand : $() \rightarrow \text{Int}$.

How do we evaluate the following?

$$h_{\text{once}} \star \mathbf{op} \text{ rand } () (i.c)$$

c may contain effects that h_{once} handles, so we cannot do nothing

We **forward** rand:

- handle inner computation
- leave effect to be handled by another handler

$$\begin{aligned} & h_{\text{once}} \star \mathbf{op} \text{ rand } () (i.c) \\ \rightsquigarrow & \mathbf{op} \text{ rand } () (i.h_{\text{once}} \star c) \end{aligned}$$

FORWARDING III

Forwarding scoped effects **cannot be done generically**. Consider scoped operation **catch** : $\text{String} \rightarrow \text{Bool}$

$$h_{\text{once}} \star \text{sc catch "oops" } \overbrace{(b.\text{if } b \text{ then return } 5 \text{ else } \dots)}^{\text{Bool} \rightarrow \text{Int}} \underbrace{(x.\text{return } x)}_{\text{Int} \rightarrow \text{Int}}$$

FORWARDING III

Forwarding scoped effects cannot be done generically. Consider scoped operation **catch** : $\text{String} \rightarrow \text{Bool}$

$$h_{\text{once}} \star \text{sc catch "oops"} \left(\overbrace{(b.\text{if } b \text{ then return } 5 \text{ else } \dots)}^{\text{Bool} \rightarrow \text{Int}} \right) \left(\overbrace{(x.\text{return } x)}^{\text{Int} \rightarrow \text{Int}} \right)$$

Handling changes types: h_{once} applies **List**:

$$\text{sc catch "oops"} \left(\overbrace{(b.h_{\text{once}} \star \text{if } b \text{ then return } 5 \text{ else } \dots)}^{\text{Bool} \rightarrow \text{List Int}} \right) \left(\overbrace{(x.h_{\text{once}} \star \text{return } x)}^{\text{Int} \rightarrow \text{List Int}} \right)$$

FORWARDING III

Forwarding scoped effects cannot be done generically. Consider scoped operation **catch** : $\text{String} \rightarrow \text{Bool}$

$$h_{\text{once}} \star \text{sc catch "oops"} \left(\overbrace{(b.\text{if } b \text{ then return } 5 \text{ else } \dots)}^{\text{Bool} \rightarrow \text{Int}} \right) \left(\overbrace{(x.\text{return } x)}^{\text{Int} \rightarrow \text{Int}} \right)$$

Handling changes types: h_{once} applies **List**:

$$\text{sc catch "oops"} \left(\overbrace{(b.h_{\text{once}} \star \text{if } b \text{ then return } 5 \text{ else } \dots)}^{\text{Bool} \rightarrow \text{List Int}} \right) \left(\overbrace{(x.h_{\text{once}} \star \text{return } x)}^{\text{Int} \rightarrow \text{List Int}} \right)$$

Solution: $\text{concatMap} : \text{List } \alpha \rightarrow (\alpha \rightarrow \text{List } \beta) \rightarrow \text{List } \beta.$

FORWARDING IV

Given a handler we can easily **addresses the type mismatch**

```

$$h_{once} = \text{handler } \{ \text{return } x \mapsto \dots$$

$$, \text{op choose } \_ \quad k \mapsto \dots$$

$$, \text{sc once } \_ \quad p \ k \mapsto \dots$$

$$, \text{fwd } \ell \ v \ p \ k \mapsto \dots$$

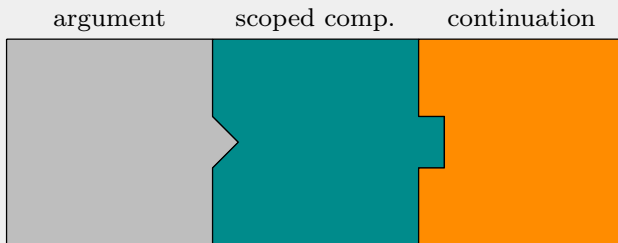
$$\}$$

```


FORWARDING IV

Given a handler we can easily **addresses the type mismatch**

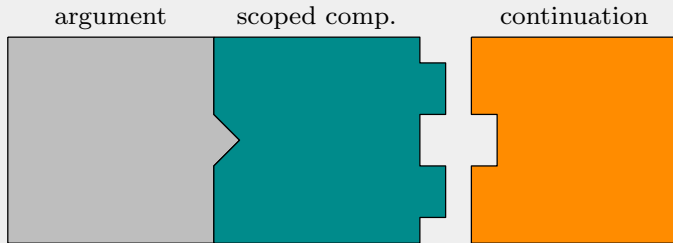
```
 $h_{\text{once}} = \text{handler } \{$   
   $\text{return } x \mapsto \dots$   
   $, \text{op choose } \_ \text{ } k \mapsto \dots$   
   $, \text{sc once } \_ \text{ } p \text{ } k \mapsto \dots$   
   $, \text{fwd } \ell \text{ } v \text{ } p \text{ } k \mapsto \dots$   
   $\}$ 
```



FORWARDING IV

Given a handler we can easily **addresses the type mismatch**

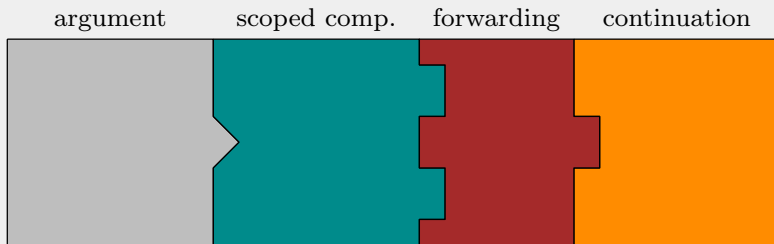
```
 $h_{once} = \text{handler } \{$   
  return       $x \mapsto \dots$   
  , op choose  $\_ \quad k \mapsto \dots$   
  , sc once    $\_ \quad p \quad k \mapsto \dots$   
  , fwd        $\ell \ v \ p \quad k \mapsto \dots$   
   $\}$ 
```



FORWARDING IV

Given a handler we can easily **addresses the type mismatch**

```
 $h_{\text{once}} = \text{handler} \{ \text{return } x \mapsto \dots$   
  , op choose  $\_ \quad k \mapsto \dots$   
  , sc once  $\_ \quad p \quad k \mapsto \dots$   
  , fwd  $\ell \vee p \quad k \mapsto \dots$   
  }
```



Paper in review, featuring Type- and effect system

1

A Calculus for Scoped Effects & Handlers

ANONYMOUS AUTHOR(S)

Algebraic effects & handlers have become a standard approach for working with side-effects in functional programming languages. Their modular composition with other effects and clean separation of syntax and semantics make them attractive to a wide audience of programmers. However, not all effects can be classified as algebraic; some need a more sophisticated handling. In particular, effects that have or create a delimited scope need special care, as their continuation consists of two parts—in and out of the scope—and their modular composition with other effects brings in additional complexity. This class of effects is called *scoped effects* and has gained attention by their growing applicability and adoption in popular libraries. Although calculi have been designed with algebraic effects & handlers as first-class operations, effectively easing programming with those features, a calculus to support scoped effects & handlers in a similar matter is missing from the literature. In this work we fill this gap: we present a novel calculus, named λ_{sc} , in which both algebraic and scoped effects & handlers are first-class. This involves the need for polymorphic handlers and explicit forwarding clauses to forward unknown scoped operations to other handlers. Our calculus is based on *Eff*, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics and (type-safe) type-and-effect system. We demonstrate the usability of our work by discussing a range of scoped effect examples. In summary, our novel calculus allows modular composition of different algebraic and scoped effects & handlers and eases programming with these features.

1 INTRODUCTION

Whereas monads [Moggi 1989, 1991; Wadler 1995] have long been the standard approach for modelling effects, algebraic effects & handlers [Plotkin and Pretnar 2009; Plotkin and Power 2003] are gaining steadily more traction. They offer a more structured and *modular* approach to composing effects that is based on an algebraic model. Algebraic effects & handlers consist of two parts: effects denote the syntax of operations and handlers interpret them by means of structural recursion. This clean separation is a coveted property for programming with effects. Each handler is appointed to interpret its part of the syntax and forwards the unknown parts to other handlers. This allows a programmer to write dedicated handlers for each effect that occurs in the program (e.g., state, interactive input/output, nondeterminism). Different effect handlers can give a different interpretation to the same effectful operation. By means of composing the handlers in the desired order, one can modularly build an interpretation for the entire program. The idea of algebraic effects & handlers and their modular composition has been adopted by many libraries in functional programming, e.g., *fused-effects* [Rix et al. 2018], *extensible-effects* [Kiselyov et al. 2019] and *Eff* in OCaml [Kiselyov and Sivaramakrishnan 2018]. Additionally, various languages support programming with algebraic effects & handlers, e.g., Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Effekt [Brachthäuser et al. 2020].

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not applicable to all kinds of effects. In particular, those effects that have or introduce a *delimited scope* (e.g., *exceptions*, *concurrency*, *local state*) are not algebraic. In this scope

Paper in review, featuring

Type- and effect system

Examples!

- Nondeterminism with Once
- Nondeterminism with Cut
- Exceptions
- Local state
- Depth-bounded search
- Parsers

1

A Calculus for Scoped Effects & Handlers

ANONYMOUS AUTHOR(S)

Algebraic effects & handlers have become a standard approach for working with side-effects in functional programming languages. Their modular composition with other effects and clean separation of syntax and semantics make them attractive to a wide audience of programmers. However, not all effects can be classified as algebraic; some need a more sophisticated handling. In particular, effects that have or create a delimited scope need special care, as their continuation consists of two parts—in and out of the scope—and their modular composition with other effects brings in additional complexity. This class of effects is called *scoped effects* and has gained attention by their growing applicability and adoption in popular libraries. Although calculi have been designed with algebraic effects & handlers as first-class operations, effectively easing programming with those features, a calculus to support scoped effects & handlers in a similar matter is missing from the literature. In this work we fill this gap: we present a novel calculus, named λ_{sc} , in which both algebraic and scoped effects & handlers are first-class. This involves the need for polymorphic handlers and explicit forwarding clauses to forward unknown scoped operations to other handlers. Our calculus is based on *Eff*, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics and (type-safe) type-and-effect system. We demonstrate the usability of our work by discussing a range of scoped effect examples. In summary, our novel calculus allows modular composition of different algebraic and scoped effects & handlers and eases programming with these features.

1 INTRODUCTION

Whereas monads [Moggi 1989, 1991; Wadler 1995] have long been the standard approach for modelling effects, algebraic effects & handlers [Plotkin and Pretnar 2009; Plotkin and Power 2003] are gaining steadily more traction. They offer a more structured and *modular* approach to composing effects that is based on an algebraic model. Algebraic effects & handlers consist of two parts: effects denote the syntax of operations and handlers interpret them by means of structural recursion. This clean separation is a coveted property for programming with effects. Each handler is appointed to interpret its part of the syntax and forwards the unknown parts to other handlers. This allows a programmer to write dedicated handlers for each effect that occurs in the program (e.g., state, interactive input/output, nondeterminism). Different effect handlers can give a different interpretation to the same effectful operation. By means of composing the handlers in the desired order, one can modularly build an interpretation for the entire program. The idea of algebraic effects & handlers and their modular composition has been adopted by many libraries in functional programming, e.g., fused-effects [Rix et al. 2018], extensible-effects [Kiselyov et al. 2019] and *Eff* in OCaml [Kiselyov and Sivaramakrishnan 2018]. Additionally, various languages support programming with algebraic effects & handlers, e.g., Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Effekt [Brachthäuser et al. 2020].

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not applicable to all kinds of effects. In particular, those effects that have or introduce a delimited scope (e.g., exceptions, concurrency, local state) are not algebraic. In this scope,

Paper in review, featuring

Type- and effect system

Examples!

- Nondeterminism with Once
- Nondeterminism with Cut
- Exceptions
- Local state
- Depth-bounded search
- Parsers

Email me for a copy:
roger.bosman@kuleuven.be

1

A Calculus for Scoped Effects & Handlers

ANONYMOUS AUTHOR(S)

Algebraic effects & handlers have become a standard approach for working with side-effects in functional programming languages. Their modular composition with other effects and clean separation of syntax and semantics make them attractive to a wide audience of programmers. However, not all effects can be classified as algebraic; some need a more sophisticated handling. In particular, effects that have or create a delimited scope need special care, as their continuation consists of two parts—in and out of the scope—and their modular composition with other effects brings in additional complexity. This class of effects is called *scoped effects* and has gained attention by their growing applicability and adoption in popular libraries. Although calculi have been designed with algebraic effects & handlers as first-class operations, effectively easing programming with those features, a calculus to support scoped effects & handlers in a similar matter is missing from the literature. In this work we fill this gap: we present a novel calculus, named λ_{sc} , in which both algebraic and scoped effects & handlers are first-class. This involves the need for polymorphic handlers and explicit forwarding clauses to forward unknown scoped operations to other handlers. Our calculus is based on Eff, an existing calculus for algebraic effects, extended with Koka-style row polymorphism, and consists of a formal grammar, operational semantics and (type-safe) type-and-effect system. We demonstrate the usability of our work by discussing a range of scoped effect examples. In summary, our novel calculus allows modular composition of different algebraic and scoped effects & handlers and eases programming with these features.

1 INTRODUCTION

Whereas monads [Moggi 1989, 1991; Wadler 1995] have long been the standard approach for modeling effects, algebraic effects & handlers [Plotkin and Pretnar 2009; Plotkin and Power 2003] are gaining steadily more traction. They offer a more structured and modular approach to composing effects that is based on an algebraic model. Algebraic effects & handlers consist of two parts: effects denote the syntax of operations and handlers interpret them by means of structural recursion. This clean separation is a coveted property for programming with effects. Each handler is appointed to interpret its part of the syntax and forwards the unknown parts to other handlers. This allows a programmer to write dedicated handlers for each effect that occurs in the program (e.g., state, interactive input/output, nondeterminism). Different effect handlers can give a different interpretation to the same effectful operation. By means of composing the handlers in the desired order, one can modularly build an interpretation for the entire program. The idea of algebraic effects & handlers and their modular composition has been adopted by many libraries in functional programming, e.g., fused-effects [Rix et al. 2018], extensible-effects [Kiselyov et al. 2019] and Eff in OCaml [Kiselyov and Sivaramakrishnan 2018]. Additionally, various languages support programming with algebraic effects & handlers, e.g., Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Effekt [Brachthäuser et al. 2020].

Although the modular approach of algebraic effects & handlers is desirable for every effectful program, it is not applicable to all kinds of effects. In particular, those effects that have or introduce a delimited scope (e.g., exceptions, concurrency, local state) are not algebraic. In this scope

Improve the **usability** with:

1. Backwards compatibility

Allow handlers without forwarding clauses to handle computations without scoped effects (i.e. Eff programs)

Improve the **usability** with:

1. Backwards compatibility

Allow handlers without forwarding clauses to handle computations without scoped effects (i.e. Eff programs)


2. Type inference

Eff has type inference [7], our calculus not (yet). Types are nice!



THANKS!

REFERENCES I



 Andrej Bauer and Matija Pretnar. “An Effect System for Algebraic Effects and Handlers”. In: *Algebra and Coalgebra in Computer Science*. Ed. by Reiko Heckel and Stefan Milius. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–16. ISBN: 978-3-642-40206-7.

 Andrej Bauer and Matija Pretnar. “Programming with algebraic effects and handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208. DOI: <https://doi.org/10.1016/j.jlamp.2014.02.001>. URL: <https://www.sciencedirect.com/science/article/pii/S2352220814000194>.




REFERENCES II

-  Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. “Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428194. URL: <https://doi.org/10.1145/3428194>.
-  Daniel Hillerström and Sam Lindley. “Liberating Effects with Rows and Handlers”. In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 15–27. ISBN: 9781450344357. DOI: 10.1145/2976022.2976033. URL: <https://doi.org/10.1145/2976022.2976033>.

REFERENCES III

-  Daan Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. ISBN: 9781450346603. DOI: 10.1145/3009837.3009872. URL: <https://doi.org/10.1145/3009837.3009872>.
-  Sam Lindley, Conor McBride, and Craig McLaughlin. “Do Be Do Be Do”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 500–514. ISBN: 9781450346603. DOI: 10.1145/3009837.3009897. URL: <https://doi.org/10.1145/3009837.3009897>.

REFERENCES IV

-  Matija Pretnar. “Inferring algebraic effects”. In: *Logical methods in computer science* 10 (2014).
-  KC Sivaramakrishnan et al. “Retrofitting Effect Handlers onto OCaml”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221. ISBN: 9781450383912. DOI: 10.1145/3453483.3454039. URL: <https://doi.org/10.1145/3453483.3454039>.
-  Nicolas Wu, Tom Schrijvers, and Ralf Hinze. “Effect Handlers in Scope”. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell ’14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 1–12. ISBN: 9781450330411. DOI: 10.1145/2633357.2633358. URL: <https://doi.org/10.1145/2633357.2633358>.