Handlers for Non-Monadic Computations

Ruben Pieters [1] Tom Schrijvers [1] Exequiel Rivas [2]

1 KU LEUVEN

2 CONICET CIFASIS

September 8, 2018

Agenda

- Introduction to Effect Handlers
- Monadic Effect Handlers
- Motivation Non-Monadic Effect Handlers
- Wrap-up

What are (effect) handlers?

Represent occurrence of side-effects with calls to **operations**.

The **meaning** of these operations are given by **handlers**.

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

What are (effect) handlers?

Represent occurrence of side-effects with calls to **operations**.
The **meaning** of these operations are given by **handlers**.
Various implementations

- as language (Eff[1], Frank, Koka, ...)
- as library (e.g. in Haskell)

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

Throw

Operation

```
throw :  String -> a
```

Computation

```
let div x y =
 if y != 0
  then x / y
  else throw "division by zero!"
```

Handling Throw

Evaluation

```
div 10 0
```

Handling Throw

Evaluation

```
div 10 0
= if 0 != 0
    then 10 / 0
    else throw "division by zero!"
```

Handling Throw

Evaluation

```
div 10 0
= if false
    then 10 / 0
    else throw "division by zero!"
```

Handling Throw

Evaluation

```
div 10 0
= if false
   then 10 / 0
   else throw "division by zero!"
= throw "division by zero!"
```

Handling Throw

Evaluation

```
div 10 0
= if false
    then 10 / 0
    else throw "division by zero!"
= throw "division by zero!"
> uncaught operation:  throw
```

Handling Throw

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
```

Handling Throw

Handler

```
let h = handler
  val x       -> x
  throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
```

Handling Throw

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
```

Handling Throw

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
= handle
   if 0 != 0
   then 10 / 0
   else throw "division by zero!"
  with h
```

Handling Throw

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
= handle
   if false
   then 10 / 0
   else throw "division by zero!"
  with h
```

Handling Throw

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
= handle
   throw "division by zero!"
  with h
```

Handling Throw

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 0 with h
= handle
   throw "division by zero!"
  with h
> 0 :  int
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
= handle
   if 5 != 0
   then 10 / 5
   else throw "division by zero!"
  with h
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
= handle
    if true
    then 10 / 5
    else throw "division by zero!"
  with h
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
= handle
    10 / 5
  with h
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
= handle
    2
  with h
```

Handling Throw - Value Case

Handler

```
let h = handler
 val x       -> x
 throw s k -> 0
```

Evaluation

```
handle div 10 5 with h
= handle
    2
  with h
> 2 :  int
```

Handling Throw

Handler

```
let h = handler
 val x      -> Some x
 throw s k -> None
```

Handling Throw

Handler

```
let h = handler
 val x      -> Some x
 throw s k -> None
```

Evaluation

```
handle div 10 0 with h
> None :  int option
```

Handling Throw

Handler

```
let h = handler
 val x     -> Some x
 throw s k -> None
```

Evaluation

```
handle div 10 0 with h
> None :  int option
handle div 10 5 with h
> Some 2 :  int option
```

Special K

Handler

```
let h = handler
 val x      -> x
 throw s k -> 0
```

The k variable gives access to the continuation.

What is a Continuation?

Small Example

$\diamond + 1$

What is a Continuation?

Small Example

$\diamond + 1$

```
:  int -> int
```

What is a Continuation?

Small Example

$\diamond + 1$

```
:  int -> int
```

Operations introduce $\diamond$ and we access the continuation with k

Handling with Continuation

Without k

```
handle throw "" + 1 with
 throw s k -> 42
```

With k

```
handle throw "" + 1 with
 throw s k -> k 42
```

Handling with Continuation

Without k

```
handle throw "" + 1 with
 throw s k -> 42
= 42 where
   k ← ◇ + 1
   s ← ""
```

With k

```
handle throw "" + 1 with
 throw s k -> k 42
```

Handling with Continuation

Without k

```
handle throw "" + 1 with
  throw s k -> 42
= 42 where
    k ← ◇ + 1
    s ← ""
> 42
```

With k

```
handle throw "" + 1 with
  throw s k -> k 42
```

Handling with Continuation

Without k

```
handle throw "" + 1 with
 throw s k -> 42
= 42 where
   k ← ◇ + 1
   s ← ""
> 42
```

With k

```
handle throw "" + 1 with
 throw s k -> k 42
= k 42 where
   k ← ◇ + 1
   s ← ""
```

Handling with Continuation

Without k

```
handle throw "" + 1 with
 throw s k -> 42
= 42 where
   k ← ◇ + 1
   s ← ""
> 42
```

With k

```
handle throw "" + 1 with
 throw s k -> k 42
= k 42 where
   k ← ◇ + 1
   s ← ""
> 43
```

Handling Rules

Value Rule

handle v **with** **val** x -> c_v
 = c_v **where**
 x ← v

Handling Rules

Value Rule

```
handle v with val x -> c_v
 = c_v where
    x ← v
```

Operation Rule

```
handle K(op_i p') with op_i p k -> c_i
 = c_i where
    p ← p'
    k ← \n -> handle (K n)
```

State

Operation

```
get  :  () -> s
put  :  s -> ()
```

State

Computation

```
let comp =
 let s = get () in
 set s ;
 s
```

State

Different Interpretations

State Passing Functions (State Monad):

```
handle comp with h
:  s -> (a, s)
```

State

Different Interpretations

State Passing Functions (State Monad):

```
handle comp with h
:  s -> (a, s)
```

Connect to DB with IO:

```
handle comp with h
:  IO a
```

State

Different Interpretations

State Passing Functions (State Monad):

```
handle comp with h
:  s -> (a, s)
```

Connect to DB with IO:

```
handle comp with h
:  IO a
```

...

Counting Operations?

Computation

```
let comp =
 let s = get () in
 set s ;
 s
```

Counting Operations?

Handler

```
let h = handler
```

Counting Operations?

Handler

```
let h = handler
 val x    -> 0
```

Counting Operations?

Handler

```
let h = handler
  val x    -> 0
  put p k -> (k ()) + 1
```

Counting Operations?

Handler

```
let h = handler
 val x   -> 0
 put p k -> (k ()) + 1
 get p k -> (k ?) + 1
```

Counting Operations?

Handler

```
let h = handler
 val x   -> 0
 put p k -> (k ()) + 1
 get p k -> (k X) + 1
```

Monadic Handlers

Computation

```
let comp' =
 let s = get () in
 for i in s:  set i ;
 s
```

Monadic Handlers

Computation

```
let comp' =
 let s = get () in
 for i in s:  set i ;
 s
```

Monadic handlers must be able to handle **all** monadic computations.

This condition **restricts** the possibilities of these handlers.

Inspiration

Algebraic Effects and Effect Handlers for Idioms and Arrows

Sam Lindley

Introduces calculus λ_{flow} which defines handler constructs for applicative, arrow and monad effects

By **restricting the computations**, we **increase the possibilities** for the corresponding handlers.

Category Theoretical Approach

Based on:

Notions of Computations as Monoids

Exequiel Rivas and Mauro Jaskelliof

Monads, Applicatives and Arrows are monoids in monoidal categories

Category Theoretical Approach

Based on:

Notions of Computations as Monoids

Exequiel Rivas and Mauro Jaskelliof
Monads, Applicatives and Arrows are monoids in monoidal categories

All these notions are **monoids** in a monoidal category,
can we derive a notion of handler from this?

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
Handling			
Handling Rules			

`val x ->` c_v

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
	$b : \Sigma B \to B$		
Handling			
Handling Rules			

`op`$_i$ `p k ->` `c`$_i$[2]

[2] $\Sigma = P_0 \times -^{N_0} + \ldots + P_n \times -^{N_n}$

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
	$b : \Sigma B \to B$		
Handling	$h : \Sigma^* A \to B$		
Handling Rules			

```
handle comp with handler
```

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
	$b : \Sigma B \to B$		
Handling	$h : \Sigma^* A \to B$		
Handling Rules	$h \circ v = f$		

```
handle v with (val x -> c_v) = c_v
where
      x ← v
```

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
	$b : \Sigma B \to B$		
Handling	$h : \Sigma^* A \to B$		
Handling Rules	$h \circ v = f$		
	$h \circ op = b \circ \Sigma h$		

```
handle K(op_i p') with (op_i p k -> c_i) = c_i
where
      p ← p'
      k ← \n -> handle (K n)
```

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$		
	$b : \Sigma B \to B$		
Handling	$h : \Sigma^* A \to B$		
Handling Rules	$h \circ v = f$		
	$h \circ op = b \circ \Sigma h$		

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	
	$b : \Sigma B \to B$	(M, e_M, m_M)	
Handling	$h : \Sigma^* A \to B$		
Handling Rules	$h \circ v = f$		
	$h \circ op = b \circ \Sigma h$		

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	
	$b : \Sigma B \to B$	(M, e_M, m_M)	
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	
Handling Rules	$h \circ v = f$		
	$h \circ op = b \circ \Sigma h$		

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	
	$b : \Sigma B \to B$	(M, e_M, m_M)	
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	
Handling Rules	$h \circ v = f$	$h \circ ins = f$	
	$h \circ op = b \circ \Sigma h$		

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	
	$b : \Sigma B \to B$	(M, e_M, m_M)	
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	
Handling Rules	$h \circ v = f$	$h \circ ins = f$	
	$h \circ op = b \circ \Sigma h$	$h \circ e_{\Sigma^*} = e_M$	
		$h \circ m_{\Sigma^*} = m_M \circ (h \otimes h)$	

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	$\phi_1 : I \to F$
	$b : \Sigma B \to B$	(M, e_M, m_M)	$\phi_2 : \Sigma \otimes F \to F$
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	
Handling Rules	$h \circ v = f$	$h \circ ins = f$	
	$h \circ op = b \circ \Sigma h$	$h \circ e_{\Sigma^*} = e_M$	
		$h \circ m_{\Sigma^*} =$ $m_M \circ (h \otimes h)$	

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	$\phi_1 : I \to F$
	$b : \Sigma B \to B$	(M, e_M, m_M)	$\phi_2 : \Sigma \otimes F \to F$
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	$h : \Sigma^* \to F$
Handling Rules	$h \circ v = f$	$h \circ ins = f$	
	$h \circ op = b \circ \Sigma h$	$h \circ e_{\Sigma^*} = e_M$	
		$h \circ m_{\Sigma^*} = m_M \circ (h \otimes h)$	

The Table

	Free Algebra	Free Monoid	Free Monoid $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	$\phi_1 : I \to F$
	$b : \Sigma B \to B$	(M, e_M, m_M)	$\phi_2 : \Sigma \otimes F \to F$
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	$h : \Sigma^* \to F$
Handling Rules	$h \circ v = f$	$h \circ ins = f$	$h \circ in_1 = \phi_1$
	$h \circ op = b \circ \Sigma h$	$h \circ e_{\Sigma^*} = e_M$	$h \circ in_2 = \phi_2 \circ (\Sigma \otimes h)$
		$h \circ m_{\Sigma^*} = m_M \circ (h \otimes h)$	

The Table

	Free Algebra	Free Monoid	Free Monad $(\!|-|\!)$
Handler	$f : A \to B$	$f : \Sigma \to M$	$\phi_1 : A \to FA$
	$b : \Sigma B \to B$	(M, e_M, m_M)	$\phi_2 : \Sigma(FA) \to FA$
Handling	$h : \Sigma^* A \to B$	$h : \Sigma^* \to M$	$h : \Sigma^* A \to FA$
Handling Rules	$h \circ v = f$	$h \circ ins = f$	$h \circ in_1 = \phi_1$
	$h \circ op = b \circ \Sigma h$	$h \circ e_{\Sigma^*} = e_M$	$h \circ in_2 = \phi_2 \circ \Sigma h$
		$h \circ m_{\Sigma^*} = m_M \circ (h \otimes h)$	

Counting Operations

Handler

```
let h = handler
 val x -> ΔN 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$I \to F$ (ϕ_1)

Counting Operations

Handler

```
let h = handler
 val x -> ΔN 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$Id(A) \rightarrow FA$

Counting Operations

Handler

```
let h = handler
 val x -> Δℕ 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$A \to FA$

Counting Operations

Handler

```
let h = handler
 val x -> ΔN 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$$\Sigma \otimes F \to F \qquad (\phi_2)$$

Counting Operations

Handler

```
let h = handler
 val x -> Δℕ 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$\Sigma \star F \to F$

Counting Operations

Handler

```
let h = handler
 val x -> Δℕ 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$$(P_i \times -^{N_i}) \star F \to FA$$

Counting Operations

Handler

```
let h = handler
 val x -> Δℕ 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$$\int^{Y,Z}(P_i \times Y^{N_i} \times FZ \times (Y \times Z \to -)) \to F$$

Counting Operations

Handler

```
let h = handler
 val x -> Δ_ℕ 0
 put p y k f -> k + 1
 get p y k f -> k + 1
```

$$\int^{Y,Z}(P_i \times Y^{N_i} \times FZ \times (Y \times Z \to A)) \to FA$$

Key Point

The standard handlers can be derived from Free Algebras, while our non-monadic handlers are derived from Free Monoids.

Paper

- Much more in-depth theory
- Using less expressive handlers on more expressive computations (e.g. monadic handler on applicative computation, by utilizing lax monoidal functors)

Ongoing Work

- Simplify signatures
- Investigate use of Continuation Monad to obtain interface in the base category $\mathscr{C}$

Thanks for your attention!

ruben.pieters@cs.kuleuven.be