Effect Handlers in Scope

N. Wu[1] T. Schrijvers[2] R. Hinze[1]

[1]University of Oxford

[2]Ghent University

ICFP 2014

Monad Transformers

Traditional Approach
(Liang et al. 1995)

Algebraic Effect Handlers

Recent Developments
(Plotkin&Power 2002,
Kiselyov et al. 2013,
Kammar et al. 2013,
Brady 2013)

Monad Transformers		Algebraic Effect Handlers
✗	**Methodology**	✓
✗	**Composition**	✓

Monad
Transformers

Algebraic
Effect Handlers

? **Effect Interaction** ?

Effect Interaction?

```haskell
decr :: (MonadState Int m, MonadExcept () m)
     => m ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (runId . runStateT 0 . runExceptT) decr
(Left (), 0)
```

```
ghci> (runId . runExceptT . runStateT 0) decr
Left ()
```

Effect Interaction?

```
decr :: (MonadState Int m, MonadExcept () m)
     => m ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (runId . runStateT 0 . runExceptT) decr
(Left (), 0)
```

```
ghci> (runId . runExceptT . runStateT 0) decr
Left ()
```

Monad Transformers

Effect Interaction?

```
decr :: (MonadState Int m, MonadExcept () m)
     => m ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (runId . runStateT 0 . runExceptT) decr
(Left (), 0)
```

```
ghci> (runId . runExceptT . runStateT 0) decr
Left ()
```

Effect Interaction?

```
decr :: (Mona
     => m ()
decr = do x <
          if
```

👌 Effect interaction à la carte!

```
ghci> (runId . runStateT 0 . runExceptT) decr
(Left (), 0)
```

```
ghci> (runId . runExceptT . runStateT 0) decr
Left ()
```

Monad Transformers		Algebraic Effect Handlers
✓	**Effect Interaction**	?

Effect Interaction?

```haskell
decr :: (State Int <: sig, Exc () <: sig)
     => Prog sig ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (run . runState 0 . runErr) decr
(Left (), 0)
```

```
ghci> (run . runErr . runState 0) decr
Left ()
```

Effect Interaction?

```
decr :: (State Int <: sig, Exc () <: sig)
     => Prog sig ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (run . runState 0 . runErr) decr
(Left (), 0)
```

```
ghci> (run . runErr . runState 0) decr
Left ()
```

Effect Interaction?

```
decr :: (State Int <: sig, Exc () <: sig)
     => Prog sig ()
decr = do x <- get
          if x > 0 then put (pred x)
                   else throw ()
```

```
ghci> (run . runState 0 . runErr) decr
(Left (), 0)
```

```
ghci> (run . runErr . runState 0) decr
Left ()
```

Effect Interaction?

```
decr :: (Stat
     => Prog
decr = do x <
          if
```

👌 Effect interaction à la carte!

```
ghci> (run . runState 0 . runErr) decr
(Left (), 0)
```

```
ghci> (run . runErr . runState 0) decr
Left ()
```

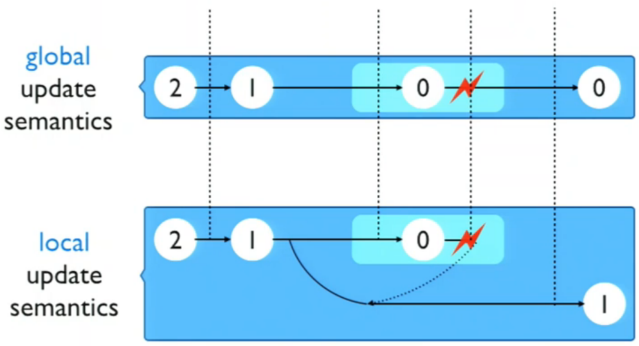
Monad Transformers		Algebraic Effect Handlers
✓	**Effect Interaction**	✓

Effect Interaction with Scoping Constructs

```
tripleDecr = decr >> catch (decr >> decr) return
```

Effect Interaction with Scoping Constructs

```
tripleDecr = decr >> catch (decr >> decr) return
```



Monad Transformers

Global Updates?
Local Updates?

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

Monad Transformers

Global Updates?
Local Updates?

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

Monad Transformers

Global Updates?
Local Updates?

```haskell
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

global update

Monad Transformers

Global Updates? ✓
Local Updates?

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

global update

Monad Transformers

Global Updates? ✓
Local Updates?

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

global update

```
ghci> (runId . runExceptT . runStateT 2) tripleDecr
Right ((), 1)
```

Monad Transformers

Global Updates? ✓
Local Updates?

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

global update

```
ghci> (runId . runExceptT . runStateT 2) tripleDecr
Right ((), 1)
```

local update

Monad Transformers

Global Updates? ✓
Local Updates? ✓

```
tripleDecr :: (MonadState Int m, MonadExcept () m)
           => m ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (runId . runStateT 2 . runExceptT) tripleDecr
(Right (), 0)
```

global update

```
ghci> (runId . runExceptT . runStateT 2) tripleDecr
Right ((), 1)
```

local update

Monad Transformers		Algebraic Effect Handlers
✓✓	**Effect Interaction**	✓?

Global Updates?
Local Updates?

```
tripleDecr :: (State Int <: sig, Exc () <: sig)
           => Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (run . runState 2 . runExc) tripleDecr
(Right (), 0)
```

Algebraic Effect Handlers

Global Updates? ✓
Local Updates?

```haskell
tripleDecr :: (State Int <: sig, Exc () <: sig)
           => Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (run . runState 2 . runExc) tripleDecr
(Right (), 0)
```

global update

Global Updates? ✓
Local Updates?

```haskell
tripleDecr :: (State Int <: sig, Exc () <: sig)
           => Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (run . runState 2 . runExc) tripleDecr
(Right (), 0)
```

global update

```
ghci> (run . runExc . runState 2) tripleDecr
Right ((), 0)
```

Algebraic Effect Handlers

Global Updates? ✓
Local Updates? ✗

```haskell
tripleDecr :: (State Int <: sig, Exc () <: sig)
           => Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (run . runState 2 . runExc) tripleDecr
(Right (), 0)
```

global update

```
ghci> (run . runExc . runState 2) tripleDecr
Right ((), 0)
```

global update

Why is Catch Different?

Effect Handlers

atomic operations
(get/put/throw/. . .)

syntax
(functor)

semantics
=
handler
function

Effect Handlers

scoping operations
(e.g., catch)

scope
=
syntax

semantics
=
handler
function

Effect Handlers

scoping operations
(e.g., catch)

scope
=
syntax

shape does not fit

semantics
=
handler
function

Effect Handlers

scoping operations
(e.g., catch)

semantics&scope
=
handler
function

This paper

scoping construct
(e.g., catch)

scope
=
syntax

semantics
=
handler

Solution:
Higher-Order Syntax

First-Order Syntax

```
data Exc e cnt
    = Throw' e
  deriving (Functor)

data Catch e cnt
    = BCatch' cnt (e -> cnt)
    | ECatch' cnt
  deriving (Functor)
```

Higher-Order Syntax

```
data HExc e m a
  = Throw' e
  | forall x. Catch' (m x) (e -> m x) (x -> m a)
```

Implications of Higher-Order Syntax

- Syntax:
  higher-order functors
- Free monad infrastructure:
  for higher-order functors
- Handlers:
  compositional semantics satisfying a distributive property

Implications of Higher-Order Syntax

- Syntax:
  higher-order functors
- Free monad infrastructure:
  for higher-order functors
- Handlers:
  compositional semantics satisfying a distributive property

See the paper

Global Updates? ✓
Local Updates? ✓

```
tripleDecr :: (State Int <: sig, Exc () <: sig)
           => Prog sig ()
tripleDecr = decr >> catch (decr >> decr) return
```

```
ghci> (run . runState 2 . runExc) tripleDecr
(Right (), 0)
```

global update

```
ghci> (run . runExc . runState 2) tripleDecr
Right ((), 1)
```

local update