Effect Handlers in Scope

Tom Schrijvers



with Nicolas Wu and Ralf Hinze

Modular Monads

Monad Transformers

Traditional Approach (Liang et al. 1995)

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?

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

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)
```

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 :: (M                                    m)
      => m
decr = do
```




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

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

Monad Transformers

Algebraic Effect Handlers



Effect Interaction



Effect Interaction?

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

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)
```

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 :: (S
      => Pr
decr = do
```




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

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

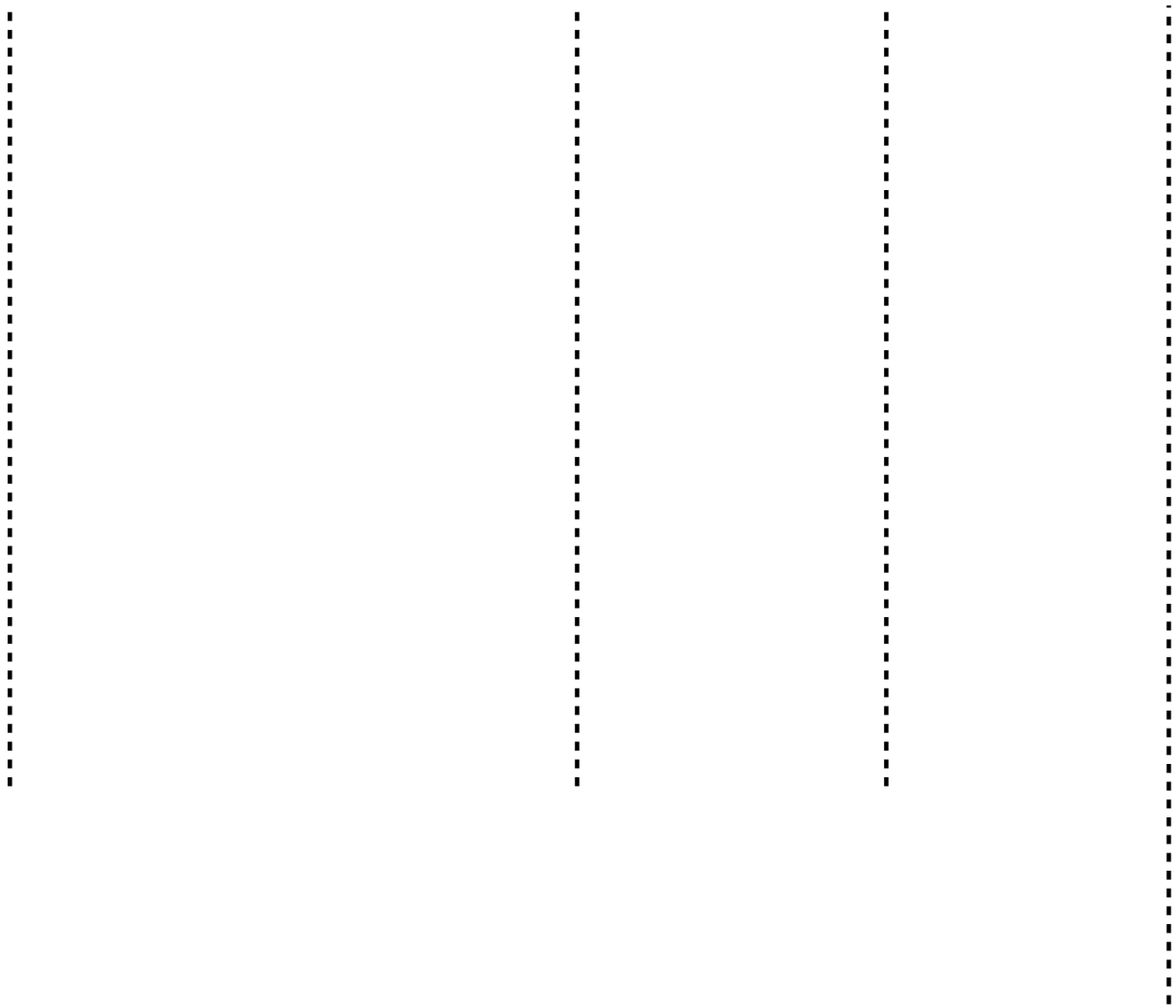
Monad Transformers	Algebraic Effect Handlers
	

Effect Interaction

Effect Interaction with Scoping Constructs

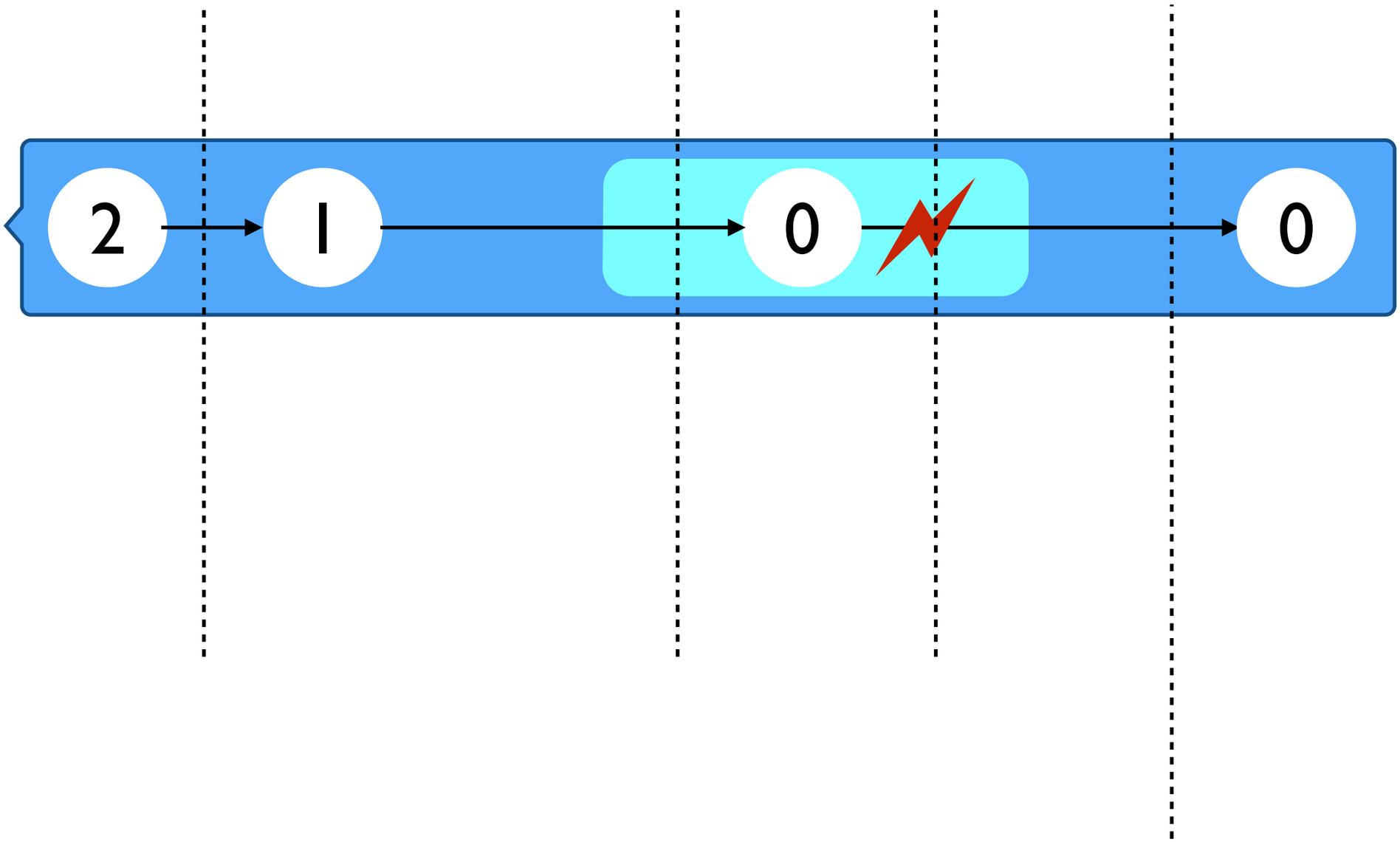
```
tripleDecr = decr >> catch (decr >> decr) return
```

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



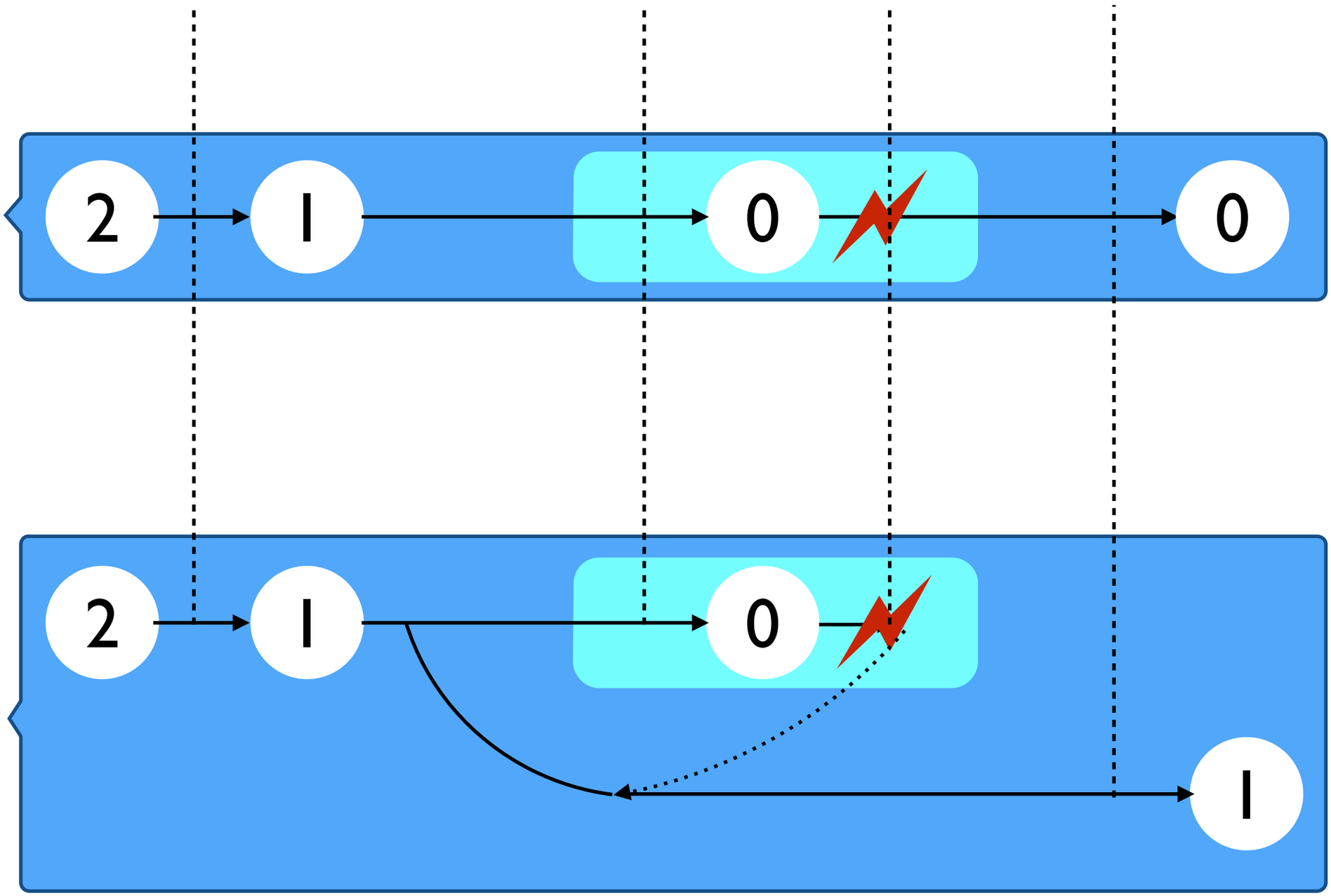
```
tripleDecr = decr >> catch (decr >> decr) return
```

global update semantics



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

global update semantics



local update semantics

Global Updates?
Local Updates?

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

Global Updates?
Local Updates?

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

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

Global Updates?

Local Updates?

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

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

global update

Global Updates? ✅

Local Updates?

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

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

global update

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

Global Updates? ✅

Local Updates? ✅

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

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

global update

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

local update

Global Updates? ✓

Local Updates? ✓

```
tripleDe                                    xcept () m)
tripleDe                                    r) return
```




```
ghci> (runId . runStateT 0 . 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
```

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 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

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 ((), 0)
```

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 ((), 0)
```

global update

Global Updates?

Local Updates?

```
tripleDe                                    <: sig)
tripleDe                                    r) return
```




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

global update

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

global update

Monad Transformers

Algebraic Effect Handlers



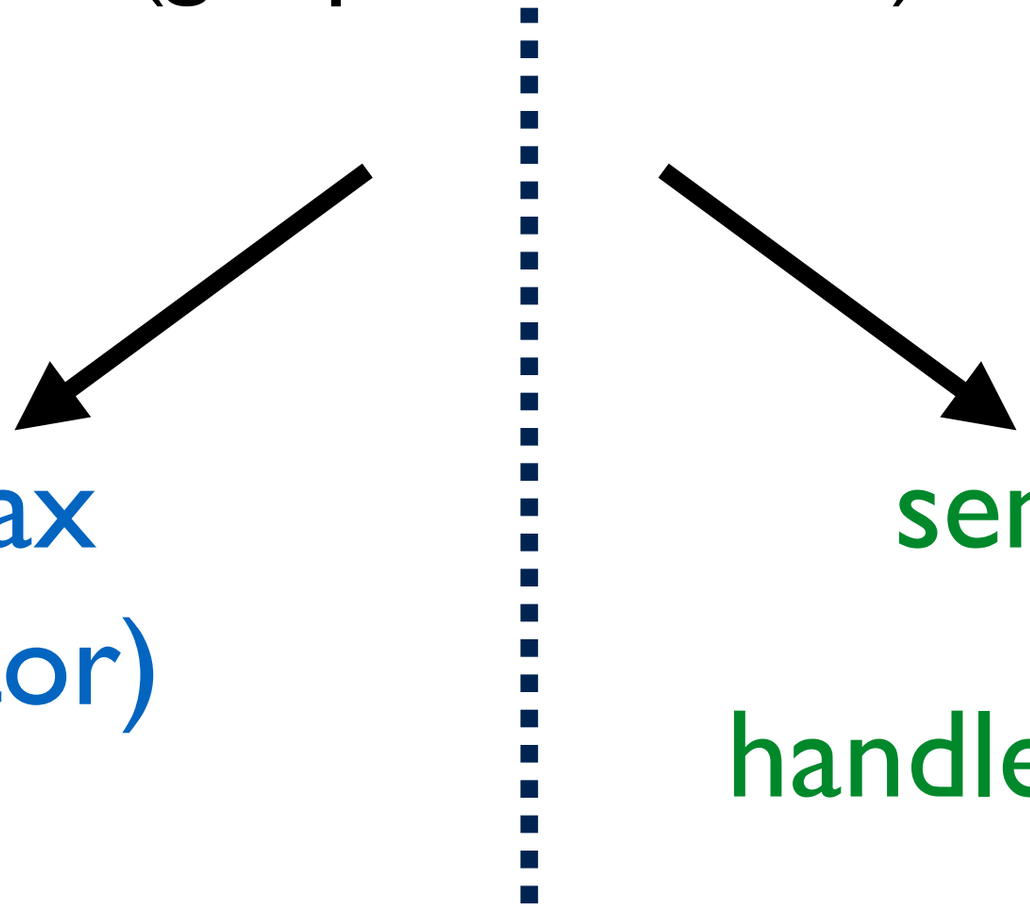
Effect Interaction



Why is Catch Different?

Effect Handlers

atomic operations
(get/put/throw/…)

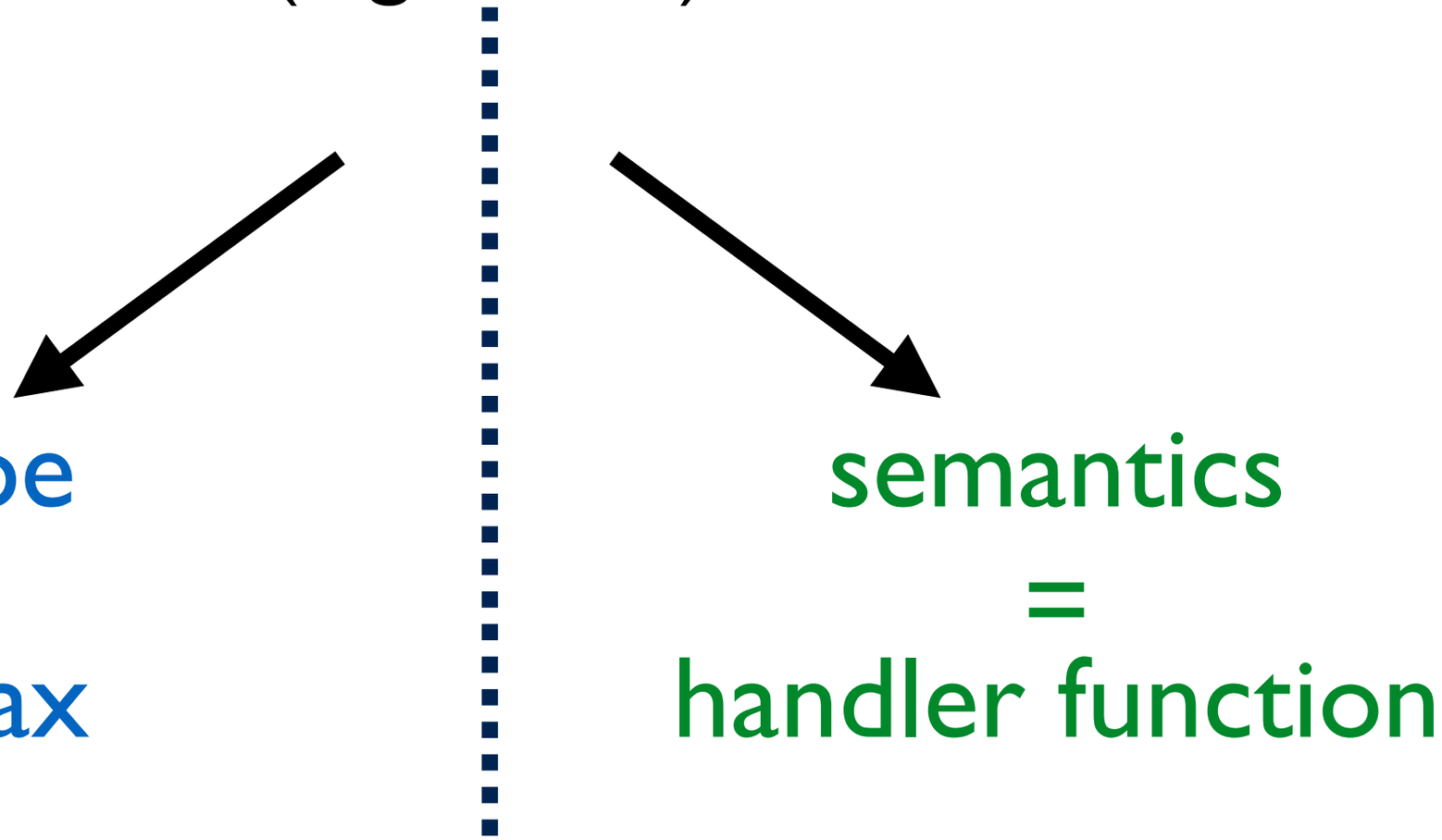


syntax
(functor)

semantics
=
handler function

Effect Handlers

scoping operations
(e.g., catch)

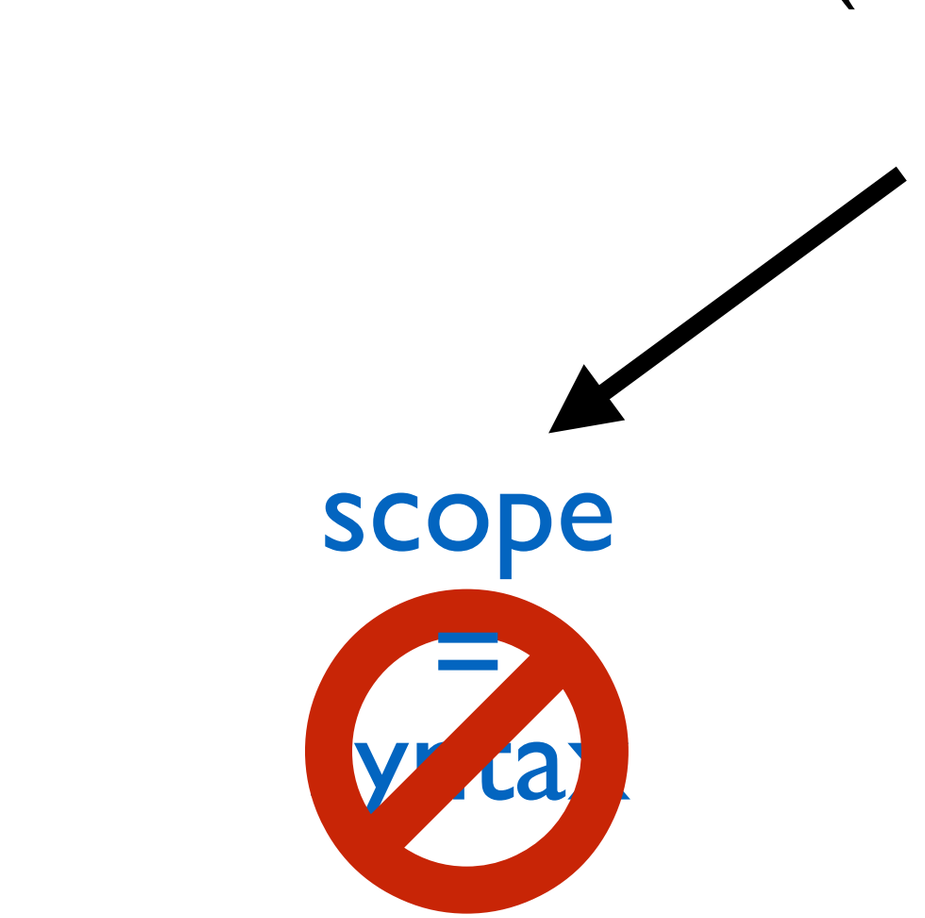


scope
=
syntax

semantics
=
handler function

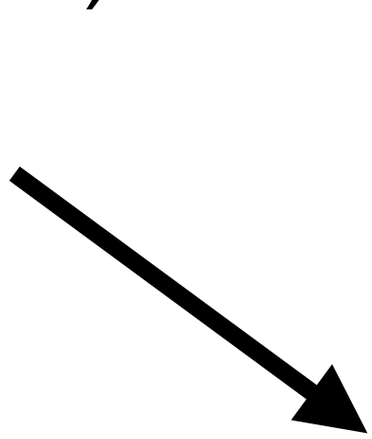
Effect Handlers

scoping operations
(e.g., catch)



scope
=
syntax

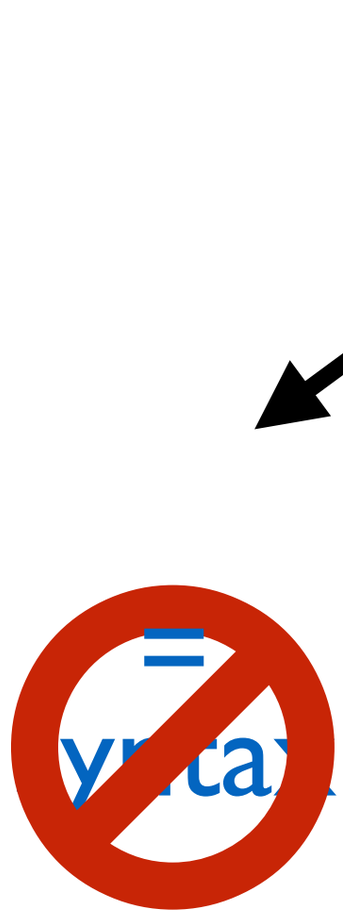
shape does not fit



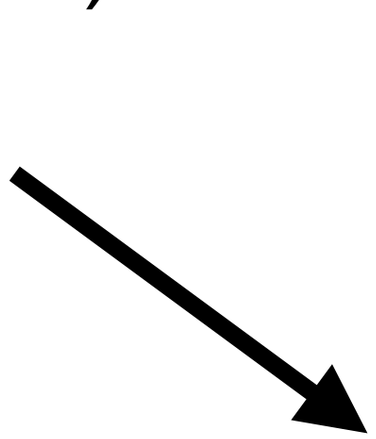
semantics
=
handler function

Effect Handlers

scoping operations
(e.g., catch)



shape does not fit

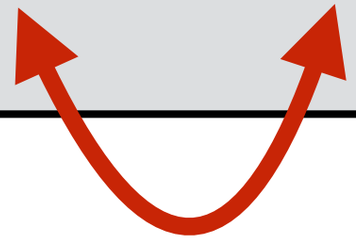


semantics & scope
=
handler function

Reordering the Right Handlers

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

```
ghci> (run . runExc . runState 2) tripleDecr
```



Reordering the Right Handlers

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

```
ghci> (run . runExc . runState 2) tripleDecr
```



Reordering the Right Handlers

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

actual semantic interaction!

```
. runState 2) tripleDecr
```

Reordering the Right Handlers

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

```
ghci> (run . runExc . runState 2) tripleDecr
```

Change order to get:

- ✅ local update semantics

Reordering the Right Handlers

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

```
ghci> (run . runExc . runState 2) tripleDecr
```

Change order to get:

- ✅ local update semantics
- ❌ different scope

Reordering the Right Handlers

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

Catch-22

```
. runState 2) tripleDecr
```

Change order to get:

- ✓ local update semantics
- ✗ different scope

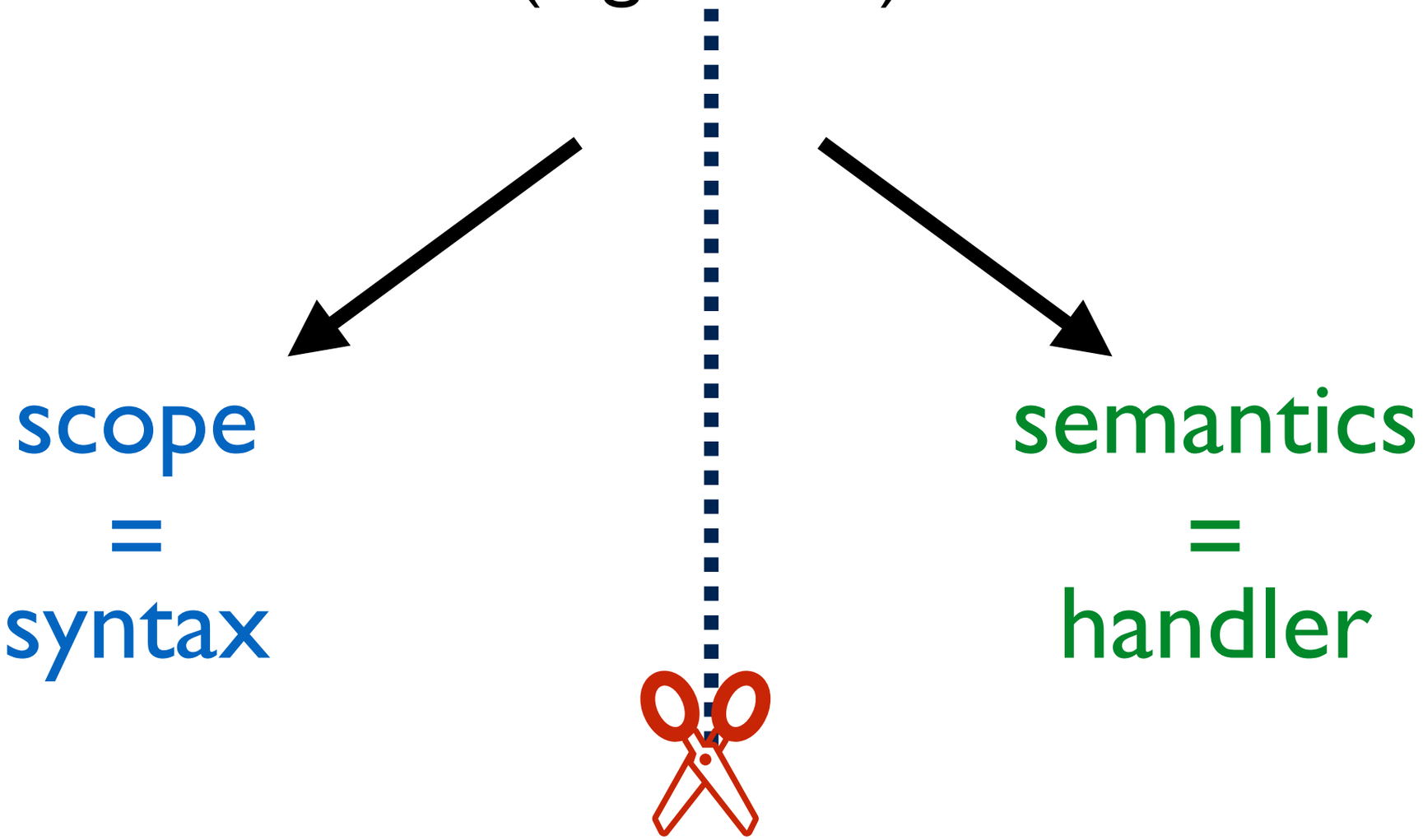
This Paper

scoping construct
(e.g., catch)



This Paper

scoping construct
(e.g., catch)



scope
=
syntax

semantics
=
handler

Solution:

Higher-Order Syntax

First-Order Syntax

Program Syntax

`throw` exc `>>=` k

Operation Syntax

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

Operation Syntax

```
data Exc
=
```

```
Throw
```

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

x

Operation Syntax

```
data Exc
=
```

```
Throw
```

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

x

Operation Syntax

```
data Exc x
=
```

```
Throw x
```

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

`exc`: `x`

`k`: `a -> Prog sig b`

Operation Syntax

```
data Exc x
=
```

`Throw x` [redacted]

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

`exc`: `x`

`k`: `a -> Prog sig b`

Operation Syntax

```
data Exc x
= forall a. Throw x (a -> r)
```

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

result type

```
x
a -> Prog sig b
```

Operation Syntax

```
data Exc x
= forall a. Throw x (a -> r)
```

First-Order Syntax

Program Syntax

```
throw exc >>= k
```

result type

```
x
a -> Prog sig b
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
```

abstract result type

First-Order Syntax

Program Syntax

```
catch p h >>= k
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
```

First-Order Syntax

Program Syntax

```
catch p h >>= k
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
|
    Catch
```

First-Order Syntax

Program Syntax

```
a -> Prog sig b
```

```
catch p h >>= k
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
|
    Catch
```

First-Order Syntax

Program Syntax

```
a -> Prog sig b
```

```
catch p h >>= k
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
| forall a.
    Catch
```

```
(a -> r)
```

First-Order Syntax

Program Syntax

```
catch p h >>= k
```

`a -> Prog sig b`

`Prog sig a`

`x -> Prog sig a`

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
| forall a.
    Catch
```

`(a -> r)`

First-Order Syntax

Program Syntax

```
catch p h >>= k
```

`a -> Prog sig b`

`Prog sig a`

`x -> Prog sig a`

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
| forall a.
    Catch (Prog sig a)
          (x -> Prog sig a) (a -> r)
```

First-Order Syntax

Program Syntax

```
a -> Prog sig b

catch p h >>= k

Prog sig a   x -> Prog sig a
```

Operation Syntax

```
data Exc x r
= forall a. Throw x (a -> r)
| forall a.
    Catch (Prog sig a)
          (x -> Prog sig a) (a -> r)
```

INVALID:
Not expressed in terms of r

First-Order Syntax

```
Prog sig b ↦ r
```

Higher-Order Syntax

```
Prog sig b ↦ m b
```

Higher-Order Syntax

```
Prog sig b ↦ m b
```

Operation Syntax

```
data Exc x m b
= forall a. Throw x (a -> m b)
| forall a.
    Catch (m a)
          (x -> m a) (a -> m b)
```

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
```

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

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

Wrapping Up

Executive Summary

Handlers should not determine scope!

Executive Summary

Handlers should not determine scope!

Syntax should.

Executive Summary

Handlers should not determine scope!

Syntax should.

Higher-order syntax does.

More in the Paper

- Introduction to effect handlers
  (Reviewer 2: ``*interesting and educative in itself*'')
- More examples
  parsing with cut, multi-threading
- Partial first-order solution
  based on begin/end markers
- The full code



Effect Handlers in Scope

Nicolas Wu
University of Oxford
nicolas.wu@cs.ox.ac.uk

Tom Schrijvers
Ghent University
tom.schrijvers@ugent.be

Ralf Hinze
University of Oxford
ralf.hinze@cs.ox.ac.uk

Abstract

Algebraic effect handlers are a powerful means for describing effectful computations. They provide a lightweight and orthogonal technique to define and compose the syntax and semantics of different effects. The semantics is captured by handlers, which are functions that transform syntax trees.

Unfortunately, the approach does not support syntax for scoping constructs, which arise in a number of scenarios. While handlers can be used to provide a limited form of scope, we demonstrate that this approach constrains the possible interactions of effects and rules out some desired semantics.

This paper presents two different ways to capture scoped constructs in syntax, and shows how to achieve different semantics by reordering handlers. The first approach expresses scopes using the existing algebraic handlers framework, but has some limitations. The problem is fully solved in the second approach where we introduce higher-order syntax.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Languages

Keywords Haskell, effect handlers, modularity, monads, syntax, semantics

1. Introduction

Effect handlers [13] have established themselves as a lightweight and compositional means of describing effectful computations. At the heart of the solution is the idea that a program is composed out of fragments of syntax that are often orthogonal to one another. Those fragments can then in turn be given a semantics by handlers that systematically deal with different effects.

One aspect of handlers that has not received much attention are scoping constructs. Examples of this abound: we see it in constructions for control flow, such as while loops and conditionals, but we also see it in pruning nondeterministic computations, exception handling, and multi-threading. The current work on effect handlers considers scoping to be in the province of handlers, that not only provide semantics but also delimit the scope of their effects.

However, as this paper illustrates, using handlers for scoping has an important limitation. The reason is that the semantics of handlers are not entirely orthogonal: applying handlers in different orders may give rise to different interactions between effects—perhaps the best known example is that of the two possible interactions between state and non-determinism. The flexibility of ordering handlers is of course crucial: we need control over the interaction of effects to obtain the right semantics for a particular application. However, if handlers double as scoping constructs, the two roles may be at odds: one order of the handlers provides the right scopes and the other order provides the right semantics. Unfortunately, we cannot have it both ways.

This paper solves the dilemma by shifting the responsibility of creating scopes from handlers to syntax. This way we can safely reorder handlers to control the interaction semantics while scoping is unaffected. Of course, handlers are still responsible for assigning a semantics to syntax that create scopes.

The specific contributions of this paper are:

1. We present a gentle and accessible introduction to the effect handlers approach and its infrastructure.
2. We provide several examples that demonstrate the problem of scoping through handlers: pruning nondeterministic choices, exception handling, and multi-threading.
3. We develop two different approaches for handling scoping through syntax.
   (a) First, we use syntax within the existing effect handlers framework to delimit scopes, and show how to write a handler that works with this syntax. This solution is conceptually lightweight, since it makes use of syntax and is nothing other than another handler. However, it is not general enough to capture syntax that truly requires programs as arguments.
   (b) As a second solution we provide higher-order syntax that truly allows to embed programs within scoping constructs. This solution is more general, but requires a substantial adaptation of the effect handlers approach.
4. We illustrate both syntax scoping approaches on the examples and show how they effectively solve the problem.

The remainder of this paper is structured as follows. The first part provides background on the effect handlers approach through a number of examples and sets up the necessary infrastructure. We start with a gentle introduction to handling backtracking computations in Section 2. In Section 3 we prepare the ground for more modular syntax by using the datatypes à la carte approach. We demonstrate this modularity in Section 4, where we show how state can be added to nondeterministic computation. We then show how handlers can span different syntax signatures in Section 5.

The second part of this paper focuses on scoped effects. Section 6 builds grammars to parse input, and shows how using handlers to create local scopes imposes undesired semantics. In Section 7 we fix this problem by using syntax to delimit scope. Section 8 demonstrates exception handling as another example that requires

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell'14, September 04-05 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-3041-1/14/09…$15.00.
http://dx.doi.org/10.1145/2633357.2633358

Ongoing & Future Work

- Optimize free monad
  (cfr. van der Ploeg & Kiselyov, Haskell '14)
- Higher-order syntax in strict languages
  (cfr. Kammar et al., ICFP '13)
- Expressing compositional handlers as folds
  (cfr. Gibbons & Wu, ICFP '14)
- Lifting first-order to higher-order handlers

The End