Efficient Compilation of Algebraic Effect Handlers

Ningning Xie
Can you implement a function, which takes an integer \( i \), and returns the result of \( 42 \) divided by \( i \)?
div42 :: Int -> Int
div42 i = 42 / i
\texttt{\texttt{div42} :: Int -> Int}

\begin{verbatim}
\texttt{div42 } \texttt{i }= \texttt{
if } \texttt{i == 0 }\texttt{ then error "divided by Zero" else 42 / i}
\end{verbatim}
divn :: Int -> Int
divn i =
    n <- getUserInput ()
    if i == 0
    then error "divided by Zero"
    else n / i
divn :: Int -> Int
divn i =
  n <- getUserInput ()
  if i == 0
    then error "divided by Zero"
  else writeLog "success"
    n / i
divn :: Int -> Int
divn i =
    n <- getUserInput ()
    if i == 0
        then error "divided by Zero"
        else writeLog "success"
            count += 1
            n / i
calculator
calculator

0 / 0

Coq proof assistant
Coq proof assistant
calculator

0 / 0

1 / 0

Coq proof assistant
calculator

0 / 0

1 / 0

Coq proof assistant
calculator

0 / 0

1 / 0

1 + (1 / 0)

Coq proof assistant
calculator

0 / 0

0

1 / 0

0

1 + (1 / 0)

1

Coq proof assistant
calculator

\[
\begin{array}{c}
0 \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

0 / 0

\[
\begin{array}{c}
\text{Error} \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

1 / 0

\[
\begin{array}{c}
\text{Error} \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

1 + (1 / 0)

\[
\begin{array}{c}
\text{Error} \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

0

\[
\begin{array}{c}
\text{Error} \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

0

\[
\begin{array}{c}
\text{Infinity} \\
7 8 9 \div + \text{AC} \\
4 5 6 \times \text{AC} \\
1 2 3 - \text{AC} \\
0 \cdot = +
\end{array}
\]

1

Coq proof assistant
calculator

Coq proof assistant
Coq proof assistant

calculator

0 / 0

1 / 0

1 + (1 / 0)
1. How to compose computational effects?
2. How to handle effects according to applications?
Algebraic effects and handlers

Composable and modular computational effects
Algebraic effects and handlers

Composable and modular computational effects
Algebraic effects and handlers

Composable and modular computational effects

**algebraic effects**

define a family of operations
Algebraic effects and handlers

Composable and modular computational effects

algebraic effects

define a family of operations

effect handlers

give semantics to operations
Algebraic effects and handlers

Composable and modular computational effects

**algebraic effects**

define a family of operations

**effect handlers**

give semantics to operations

Algebraic Operations and Generic Effects

Applied Categorical Structures 2003

Gordon Plotkin and John Power

Division of Informatics, University of Edinburgh, King’s Buildings,
Edinburgh EH9 3JZ, Scotland
Algebraic effects and handlers

Composable and modular computational effects

**algebraic effects**

define a family of operations

**effect handlers**
give semantics to operations

---

**Algebraic Operations and Generic Effects**

Applied Categorical Structures 2003

Gordon Plotkin and John Power

Division of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland

**Handlers of Algebraic Effects**

ESOP 2019

Gordon Plotkin * and Matija Pretnar **

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Scotland

**HANDLING ALGEBRAIC EFFECTS**

Logical Methods in Computer Science 2013

GORDON D. PLOTKIN* AND MATIJA PRETNAR**
Eff

Programming with algebraic effects and handlers
Journal of Logical and Algebraic Methods in Programming 2015
Andrej Bauer, Matija Pretnar

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
Eff

Programming with algebraic effects and handlers
Journal of Logical and Algebraic Methods in Programming 2015
Andrej Bauer, Matija Pretnar

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types
Mathematically Structured Functional Programming 2014

Daan Leijen
Microsoft Research
daan@microsoft.com
Eff

Programming with algebraic effects and handlers
Journal of Logical and Algebraic Methods in Programming 2015
Andréj Bauer, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types
Mathematically Structured Functional Programming 2014
Daan Leijen
Microsoft Research
daan@microsoft.com

Links

Row-based Effect Types for Database Integration
TLDI 2012
Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk
James Cheney
The University of Edinburgh
j Cheney@inf.ed.ac.uk
Eff

Programming with algebraic effects and handlers
Journal of Logical and Algebraic Methods in Programming 2015
Andréj Bauer, Matija Pretnar

Facility of Mathematics and Physics, University of Ljubljana, Slovenia

Koka: Programming with Row-polymorphic Effect Types
Mathematically Structured Functional Programming 2014
Daan Leijen
Microsoft Research
daan@microsoft.com

Links

Row-based Effect Types for Database Integration
TLDI 2012
Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk
James Cheney
The University of Edinburgh
jcheney@inf.ed.ac.uk

Frank

Do Be Do Be Do
POPL 2017

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk
Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk
Craig McLaughlin
The University of Edinburgh, UK
 craigmclaughlin@ed.ac.uk
Eff

Programming with algebraic effects and handlers
Journal of Logical and Algebraic Methods in Programming 2015
Andréj Bauer, Matija Pretnar*
Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Frank

Do Be Do Be Do
POPL 2017

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk

Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk

Craig McLaughlin
The University of Edinburgh, UK
craig.mclaughlin@ed.ac.uk

Koka: Programming with Row-polymorphic Effect Types
Mathematically Structured Functional Programming 2014
Daan Leijen
Microsoft Research
daan@microsoft.com

Effekt

Effekt: Lightweight Effect Polymorphism for Handlers
OOPSLA 2020

Jonathan Immanuel Brachthäuser, EPFL, Switzerland
Philipp Schuster, University of Tübingen, Germany
Klaus Ostermann, University of Tübingen, Germany

Links

Row-based Effect Types for Database Integration
TLDI 2012
Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

James Cheney
The University of Edinburgh
jचेन्य@inf.ed.ac.uk
Retrofitting Effect Handlers onto OCaml

KC Sivaramakrishnan  
IIT Madras  
Chennai, India  
kcsef@cse.iitm.ac.in

Stephen Dolan  
OCaml Labs  
Cambridge, UK  
estephen.dolan@ocaml.org

Leo White  
Jane Street  
London, UK  
leo@jpz25.net

T. Kelly  
OCaml Labs  
Cambridge, UK  
tom.kelly@ocaml.org

Sadiq Jaffar  
Opam and OCaml Labs  
Cambridge, UK  
sadiq@isaol.com

Anil Madhavapeddy  
University of Cambridge and OCaml Labs  
Cambridge, UK  
avm2@cl.cam.ac.uk
Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcs18@ee.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@ocaml.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@ocamtib.net

Sadig Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadig@tsoa.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm@edocam.ac.uk

Multicore OCaml: September 2021, effect handlers will be in OCaml 5.0!

avsm 19d

Welcome to the September 2021 Multicore OCaml monthly report! This month’s update along with the previous updates have been compiled by me, @cth21, @kaycesrk and @shakthiamaan. The team has been working over the past few months to finish the last few features necessary to reach feature parity with stock OCaml. We also worked closely with the core OCaml team to develop the timeline for upstreaming Multicore OCaml to stock OCaml, and have now agreed that:

OCaml 5.0 will support shared-memory parallelism through domains and direct-style concurrency through effect handlers (without syntactic support).
Retrofitting Effect Handlers onto OCaml

PLDI 2021

KC Sivaramakrishnan
IIT Madras
Chennai, India
kscsi@iitb.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@ocaml.org

Leo White
Jane Street
London, UK
leow@janestreet.com

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@ocaml.org

Sadeq Jafar
Open and OCaml Labs
Cambridge, UK
sadeq@osao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm@idi.cam.ac.uk

https://github.com/WebAssembly/design/issues/1359

https://discuss.ocaml.org/t/multicore-ocaml-september-2021-effect-handlers-will-be-in-ocaml-5-0/8554

Multicore OCaml: September 2021, effect handlers will be in OCaml 5.0!

Community multicore, multicore-monthly

avsm Maintainer

Welcome to the September 2021 Multicore OCaml monthly report! This month’s update along with the previous updates  have been compiled by me, @ctk21, @samyserk, and @shakhthmaan. The team has been working over the past few months to finish the last few features necessary to reach feature parity with stock OCaml. We also worked closely with the core OCaml team to develop the timeline for upstreaming Multicore OCaml to stock OCaml, and have now agreed that:

OCaml 5.0 will support shared-memory parallelism through domains and direct-style concurrency through effect handlers (without syntactic support).
Retrofitting Effect Handlers onto OCaml

One specific way of typing continuations and the values communicated back and forth is by following the approach taken by so-called effect handlers, one modern way of representing delimited continuations, …
React: A JavaScript library for building user interfaces

PYRO: Deep Universal Probabilistic Programming Language

Algebraic Effects for React Developers

https://reesew.io/posts/react-algebraic-effects/

Article — JavaScript

Poutine (Effect handlers)

Beneath the built-in inference algorithms, Pyro has a library of composable effect handlers for creating new inference algorithms and working with probabilistic programs. Pyro’s inference algorithms are all built by applying these handlers to stochastic functions. In order to get a general understanding what effect handlers are and what problem they solve, read An Introduction to Algebraic Effects and Handlers by Matija Pretnar.

https://docs.pyro.ai/en/dev/poutine.html
Algebraic Effect Handlers go Mainstream

Organizers

Sivaramakrishnan Krishnamoorthy Chandrasekaran (University of Cambridge, GB)
Daan Leijen (Microsoft Research – Redmond, US)
Matija Pretnar (University of Ljubljana, SI)
Tom Schrijvers (KU Leuven, BE)
Agenda

• Algebraic effects 101
• Examples, and more examples
• Efficient compilation of algebraic effects
• Koka: algebraic effects via evidence-passing semantics
Algebraic effects 101
Algebraic effects 101

```haskell

effect read { 
    ask : () -> int 
}

handler { 
    ask x k -> k 1 
}

(\_. 
    perform ask () + perform ask ()
)
Algebraic effects 101

```
effect read {
  ask : () -> int
}

handler {
  ask x k -> k 1
}
\_.
  perform ask () + perform ask ()
```
Algebraic effects 101

```effect read {
  ask : () -> int
}
```

```handler {
  ask x k -> k 1
}
```

```(_. perform ask () + perform ask ()
)```
Algebraic effects 101

```
effect read {  
    ask : () -> int  
}

handler {  
    ask x k -> k 1  
}

(\_.  
    perform ask () + perform ask ()  
)
Algebraic effects 101

effect signature

```
effect read {
  ask : () -> int
}
```

```
handler {
  ask x k -> k 1
}
```
Algebraic effects 101

**effect signature**

```plaintext
effect read {  
  ask : () -> int  
}
```

**operation**

```plaintext
(handler {  
  ask x k -> k 1  
})  
(_.  
  perform ask () + perform ask ()  )
```
Algebraic effects 101

**effect signature**

```effect
read { 
  ask : () -> int
}
```

**operation**

**effect handler**

```handler
{
  ask x k -> k 1
}
```

```
(_.
  perform ask () + perform ask ()
)
```
Algebraic effects 101

**Effect signature**

```java
effect read {
    ask : () -> int
}
```

**Operation**

**Effect handler**

```java
handler {
    ask x k -> k 1
}
```

**Implementation**

```java
(\_.
    perform ask () + perform ask ()
)
```
Algebraic effects 101

**Effect Signature**

```effect
read {
  ask : () -> int
}
```

**Operation**

```
```

**Effect Handler**

```handler
{
  ask x k -> k 1
}
```

**Implementation**

```
(_.
  perform ask () + perform ask ()
)
```
Algebraic effects 101

**effect signature**

```effect
effect read {
  ask : () -> int
}
```

**operation**

**effect handler**

```handler
handler {
  ask x k -> k 1
}
```

**implementation**

```(_.
  perform ask () + perform ask ()
)
```
Algebraic effects 101

**effect signature**

```scala
effect read {
  ask : () -> int
}
```

**operation**

```java
handler {
  ask x k -> k 1
}
```

**implementation**

```java
(\_.
  perform ask () + perform ask ()
)
```
Algebraic effects 101

**Effect Signature**

```effect
read {
  ask : () -> int
}
```

**Effect Handler**

```handler
{
  ask x k -> k 1
}
```

**Computation**

```(
_
  perform ask () + perform ask ()
)
```
Algebraic effects 101

**effect signature**

```
effect read {  
  ask : () -> int  
}
```

**operation**

**effect handler**

```
handler {  
  ask x k -> k 1  
}
```

**implementation**

**computation**

```
(_._.  
  perform ask () + perform ask ()  
) // 2
```

**perform an effect**
Exception

effect exn {
    throw : () -> a
}
effect exn { 
  throw : () -> a 
}

div m n = if n == 0 
  then perform throw () 
  else m / n
Exception

**effect** exn {  
  *throw*: () → a
}

div m n  
  = if n == 0  
    then perform throw ()  
    else m / n

**handler** {  
  *throw* x k → Nothing
}
  (\_.  
  Just (div 42 2))
)  // Just 21

**handler** {  
  *throw* x k → Nothing
}
  (\_.  
  Just (div 42 0))
)  // Nothing
effect exn { 
  throw : () -> a 
}

div m n
  = if n == 0
    then perform throw ()
    else m / n

handler {
  throw x k -> Nothing
  return v   -> Just v
}

(handler { 
  throw x k -> Nothing
  return v   -> Just v
}) (\_.
  div 42 2
) // Just 21

(handler {
  throw x k -> Nothing
  return v   -> Just v
}) (\_.
  div 42 0
) // Nothing
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
      then perform throw ()
      else m / n

handler {
  throw x k -> []
  return v -> [v]
}
(\_.
  div 42 2
) // Just 21

handler {
  throw x k -> []
  return v -> [v]
}
(\_.
  div 42 0
) // Nothing
\[ 2 \times (1 + 20) \]
\[ 2 \times (1 + 20) \]
\[ 2 \times (1 + 20) \quad 2 \times 21 \]
\[2 \times (1 + 20)\]  
\[2 \times 21\]  
\[42\]  

\[2 \times \]

\[1 + \]

\[20\]  

\[2 \times \]

\[21\]  

\[42\]
return x -> e1
\texttt{return x \rightarrow e1}
\texttt{op x k \rightarrow e2}
\textit{return} \ x \rightarrow \ e_1

\textit{op} \ x \ k \rightarrow e_2
every computation either calls an operation or returns a value
every computation either calls an operation or returns a value

\[ \text{return } x \rightarrow e_1 \]

\[ \text{op } x \; k \rightarrow e_2 \]
every computation either calls an operation or returns a value
every computation either calls an operation or returns a value

return x -> e1

op x k -> e2
every computation either calls an operation or returns a value

\[
\begin{align*}
\text{return } x & \rightarrow e_1 \\
\text{op } x \ k & \rightarrow e_2
\end{align*}
\]

\[
\begin{array}{c}
\text{handle} \\
\text{handle} \\
\text{handle}
\end{array}
\]

\[
\begin{array}{c}
ev \\
\end{array}
\]
return $x \rightarrow e_1$

$op \ x \ k \rightarrow e_2$

every computation either calls an operation or returns a value

handle

handle

handle

$v$

$e_1 [x:=v]$

e2 [x:=v, k:= ]$

perform $op \ v$

\textcolor{orange}{return \ x \rightarrow \ e_1}

\textcolor{orange}{op \ x \ k \rightarrow \ e_2}

\underline{every \ computation \ either \ calls \ an \ operation \ or \ returns \ a \ value}

\textcolor{green}{\text{handle}}

\textcolor{green}{\text{handle}}

\textcolor{green}{\text{handle}}

\textcolor{green}{e_1 \ [x:=v]}

\textcolor{green}{v}

\underline{\text{handle}}

\underline{\text{handle}}

\underline{\text{handle}}

\underline{\text{e_2 \ [x:=v, \ k:=\bullet]}}

\underline{\text{perform \ op \ v}}
every computation either calls an operation or returns a value
return \( x \rightarrow e_1 \)

\[\text{op} \ x \ k \rightarrow e_2\]

every computation either calls an operation or returns a value
handler { 
    throw x k -> Nothing
    return v -> Just v
} (\_. div 42 2)

every computation either calls an operation or returns a value

handle
handle
e1 \[x:=v\]

handle
handle

v
e2 \[x:=v, k:=\]

handle
perform op v
```plaintext
handler {  
  throw x k -> Nothing  
  return v -> Just v  
} (\_._.  
div 42 2)
```

```
  handle

  e1 [x:=v]
  
  v

  handle

  handle

  every computation either calls an operation or returns a value

  handle

  handle

  perform op v

  e2 [x:=v, k:=]

  handle
```
```haskell
handler { throw x k -> Nothing return v -> Just v } (_ . div 42 2)
```

every computation either calls an operation or returns a value

```haskell
handler
```

```haskell
div 42 2
```

```haskell
v
```

```haskell
perform op v
```

```haskell
e2 [x:=v, k:= ]
```

```haskell
e1 [x:=v]
```
\[
\text{handler} \{ \\
\text{throw } x \text{ } k \rightarrow \text{Nothing} \\
\text{return } v \rightarrow \text{Just } v \\
\} \ (\_\_. \ \\
\text{div } 42 \text{ } 2) \\
\]

\[
\begin{align*}
\text{handle} & \rightarrow \\
\text{div } 42 & \rightarrow 21 \\
2 & \\
\text{perform op } v & \\
\text{e1 [x:=v]} & \\
\end{align*}
\]

---

every computation either calls an operation or returns a value

\[
\begin{align*}
\text{e2 [x:=v, k:=\_]} & \\
\text{handler} & \\
\text{handle} & \\
\end{align*}
\]
handler { 
  throw x k -> Nothing
  return v -> Just v
} (_.

div 42 2)

handle
div 42
2

Just 21
every computation either calls an operation or returns a value

div 42
21

handle
e2 [x:=v, k:= ]
every computation either calls an operation or returns a value
handler {
    throw x k -> Nothing
    return v -> Just v
} (\_.
    div 42 0)

every computation either calls an operation or returns a value
handler { 
    throw x k -> Nothing
    return v -> Just v
} (_, 
    div 42 0)

Every computation either calls an operation or returns a value.
```
handler {  
  throw x k -> Nothing  
  return v -> Just v  
} (\_.  
  div 42 0)
```

```

every computation either calls an operation or returns a value
```

```
perform throw ()
```
handler {  
  throw x k -> Nothing  
  return v -> Just v  
} (\_.  
  div 42 0)

every computation either calls an operation or returns a value

perform throw ()

handle

Nothing[k:=•]

21

div 42

handle
Every computation either calls an operation or returns a value.

```haskell
handler {  
  throw x k  ->  k 0  
  return v  ->  Just v  
}  
\_._.  
  div 42 0)
```
every computation either calls an operation or returns a value

```
handler {
  throw x k -> k 0
  return v -> Just v
}
\_.

div 42 0
```
Nothing

```
handler {
  throw x k -> k 0
  return v  -> Just v
}
```

Just 42

```
(k 0)[k:=] 0
```

```
(handler (
  throw x k -> k 0
  return v  -> Just v
}
\_.

(div 42 0)
```

```java
perform throw ()
```
handler {
  throw x k -> k 0
  return v -> Just v
}
(
  \_.
  div 42 0)

Every computation either calls an operation or returns a value.
effect divByZero {
    divByZero : Int -> Int
}

div m n
    = if n == 0
      then perform divByZero m
      else m / n
effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
ios_div m n =
  handle {
    _
  } (div m n)

google_div m n =
  handle {
    _
  } (div m n)

coeq_div m n =
  handle {
    _
  } (div m n)

effect divByZero {
  divByZero : Int -> Int
}

div m n
  = if n == 0
    then perform divByZero m
    else m / n
ios_div m n =
handle {
  divByZero x k -> Error
} (div m n)

google_div m n =
handle {
  if x == 0 then Error
  else Infinit
} (div m n)

calc_div m n =
handle {
  k
} (div m n)

effect divByZero {
  divByZero : Int -> Int
}
div m n
  = if n == 0 then perform divByZero m
    else m / n
ios_div m n = 
    handle { 
        divByZero x k -> Error 
    } (div m n)

go const m n = 
    handle { 
        divByZero x k -> 
            if x == 0 then Error 
            else Infinity 
    } (div m n)

calc_div m n = 
    handle { 
    } (div m n)

effect divByZero { 
    divByZero : Int -> Int 
}

div m n 
    = if n == 0 
        then perform divByZero m 
        else m / n
ios_div m n =  
    handle {  
        divByZero x k -> Error  
    } (div m n)

google_div m n =  
    handle {  
        divByZero x k ->  
            if x == 0 then Error  
            else Infinity  
    } (div m n)

coaq_div m n =  
    handle {  
        divByZero x k ->  
    } (div m n)

effect divByZero {  
    divByZero : Int -> Int
}  

div m n  
    = if n == 0  
    then perform divByZero m  
    else m / n
ios_div m n =
  handle {
    divByZero x k -> Error
  } (div m n)

google_div m n =
  handle {
    divByZero x k ->
    if x == 0 then Error
    else Infinity
  } (div m n)

coop_div m n =
  handle {
    divByZero x k -> k 0
  } (div m n)

effect divByZero {
  divByZero : Int -> Int
}
div m n
  = if n == 0
    then perform divByZero m
    else m / n
ios_div m n = 
handle {
  divByZero x k -> Error
} (div m n)

google_div m n = 
handle {
  divByZero x k ->
    if x == 0 then Error
    else Infinity
} (div m n)

crq_div m n = 
handle {
  divByZero x k -> k 0
} (div m n)

effect divByZero {
  divByZero : Int -> Int
}
div m n
  = if n == 0
    then perform divByZero m
    else m / n
State

effect st<a> {
  get : () -> a
  set : a -> ()
}

State

```language=ocaml
effect st<a> {  
  get : () -> a  
  set : a -> ()
}

(handler {  
  get x k -> (\y. k y y)  
  set x k -> (\y. k () x)  
  return x -> (\_. x)
} (\_.
  perform set 21; w <- perform get (); w + w))
0
```
effect st<a> {  
  get : () -> a  
  set : a -> ()  
}

(handler {  
  get x k -> (\y. k y y)  
  set x k -> (\y. k () x)  
  return x -> (\_. x)  
} (\_.  
  perform set 21; w <- perform get (); w + w))
0
State

effect st<a> { 
  get : () -> a 
  set : a -> () 
}

(handler { 
  get x k -> (\y. k y y) 
  set x k -> (\y. k () x) 
  return x -> (\_. x) 
}) (\_.
  perform set 21; w <- perform get (); w + w)
0
// 42
Choice
Choice

effect choice {
    flip : () -> bool
}

Choice

```
exteffect choice {
    flip : () -> bool
}
```

```
x <- perform flip ()
y <- perform flip ()
x && y
```
Choice

```haskell

effect choice {
  flip : () -> bool
}

handler {
  flip x k -> k True ++ k False
  return x -> [x]
} (
    x <- perform flip ()
    y <- perform flip ()
    x && y
  )
```
Choice

```
null

**effect** choice {
    *flip* : () -> bool
}

**handler** {
    *flip*  x k -> k **True** ++ k **False**
    return x  -> [x]
} (\_.
    x <- perform *flip* ()
    y <- perform *flip* ()
    x && y
)

x  **True**
```
Choice

effect choice {
    $flip : () \rightarrow \text{bool}$
}

handler {
    $flip \quad x \ k \rightarrow k \text{ True} \oplus k \text{ False}$
    return $x \rightarrow [x]$
} (\_.
    $x \leftarrow \text{perform} \ flip ()$
    $y \leftarrow \text{perform} \ flip ()$
    $x \&\& y$
)

$x \quad \text{True}$
$y \quad \text{True}$
**Choice**

```haskell
**effect** choice {
    **flip**: () -> bool
}

**handler** {
    **flip**  x k -> k **True** ++ k **False**
    return x   -> [x]
} (\_.
    x <- **perform** **flip** ()
    y <- **perform** **flip** ()
    x && y
)

// [True
    x  **True**
    y  **True**
```
Choice

```haskell
effect choice {
  flip : () -> bool
}

handler {
  flip  x k -> k True ++ k False
  return x   -> [x]
} (\_.
  x <- perform flip ()
  y <- perform flip ()
  x && y
)

// [True
x  True  True
y  True
```
**Choice**

```plaintext
effect choice {
    flip : () -> bool
}

handler {
    flip x k -> k True ++ k False
    return x -> [x]
} (\_.
    x <- perform flip ()
    y <- perform flip ()
    x && y
)

// [True
x  True  True
y  True  False
```
Choice

define choice {
    flip : () -> bool
}

handler {
    flip x k -> k True ++ k False
    return x -> [x]
} (\_.
    x <- perform flip ()
    y <- perform flip ()
    x && y
)

// [True, False
 x   True   True
 y   True   False
**Choice**

```haskell

**effect** choice {
  _flip : () -> bool
}

**handler** {
  _flip x k -> k True ++ k False
  return x   -> [x]
} (\_.
  x <- perform flip ()
  y <- perform flip ()
  x && y
)

// [True, False
x  True  True  False
y  True  False
```
Choice

```haskell
effect choice {
  flip : () -> bool
}

handler {
  flip x k -> k True ++ k False
  return x -> [x]
} (\_.
  x <- perform flip ()
  y <- perform flip ()
  x && y
)

// [True, False
  x  True  True  False  False
  y  True  False
```
Choice

```haskell

effect choice {
    flip : () -> bool
}

handler {
    flip  x k -> k True ++ k False
    return x   -> [x]
} (\_.
    x <- perform flip ()
    y <- perform flip ()
    x && y
)

// [True, False
x   True True False False
y   True False True
```
Choice

```
effect choice {
  flip : () -> bool
}

handler {
  flip  x k -> k True ++ k False
  return x   -> [x]
} (\_.
  x <- perform flip ()
  y <- perform flip ()
  x && y
)

// [True, False, False
  x   True  True  False  False
  y   True  False  True
```
Choice

**effect** choice {
  flip : () -> bool
}

**handler** {
  flip x k -> k True ++ k False
  return x -> [x]
} (\_.
  x <- perform flip ()
  y <- perform flip ()
  x && y
)

// [True, False, False
  x True True False False
  y True False True False
Choice

```haskell

effect choice {
    flip : () -> bool
}

handler {
    flip x k -> k True ++ k False
    return x -> [x]
} (_).
    x <- perform flip ()
    y <- perform flip ()
    x && y

// [True, False, False, False]

x True True False False
y True False True False
```
Choice and Exception
Choice and Exception

```
effect choice {
    flip : () -> bool
}

effect exn {
    throw : () -> a
}
```
Choice and Exception

```
handler {
  \_.
}

effect choice {
  flip : () -> bool
}

effect exn {
  throw : () -> a
}
```
Choice and Exception

```haskell
handler {
  flip x k -> k True ++ k False
  return x x -> [x]
}

(handler {
  throw x k -> Nothing
  return x x -> Just x
}

(handler {
  x <- perform flip ()
  if x then
    perform flip ()
  else
    perform throw ()
)}
```
## Choice and Exception

```haskell
effect choice {  
    flip : () -> bool
}
effect exn {  
    throw : () -> a
}
```

```haskell
handler {  
    flip  x k -> k True ++ k False  
    return x  -> [x]
} (\_.
handler {  
    throw  x k -> Nothing  
    return x  -> Just x
} (\_.
    x <- perform flip ()
    if x then  
        perform flip ()
    else  
        perform throw ()  
)}
// [Just True
Choice and Exception

```
handler {
    flip  x k -> k True ++ k False
    return x -> [x]
} (\_.
handler {
    throw x k -> Nothing
    return x -> Just x
} (\_.
  x <- perform flip ()
  if x then
    perform flip ()
  else
    perform throw ()
)
// [Just True, Just False
```
Choice and Exception

**effect** choice {
    flip : () -> bool
}

**effect** exn {
    throw : () -> a
}

```latex
\begin{align*}
\text{handler} & \{ \\
    \text{flip} & \quad x \ k \rightarrow k \ True \ (+) \ k \ False \\
    \text{return} & \quad x \rightarrow [x] \\
\} \ (\_._.) \\
\text{handler} & \{ \\
    \text{throw} & \quad x \ k \rightarrow \text{Nothing} \\
    \text{return} & \quad x \rightarrow \text{Just} \ x \\
\} \ (\_._.) \\
x & \leftarrow \text{perform} \ \text{flip} \ () \\
\text{if} & \ x \ \text{then} \\
    & \quad \text{perform} \ \text{flip} \ () \\
\text{else} \\
    & \quad \text{perform} \ \text{throw} \ () \\
\) \\
// \ [\text{Just True, Just False, Nothing}]
\end{align*}
```
Choice and Exception

```
handler {  
  throw x k -> Nothing  
  return x -> Just x
} () .  
handler {  
  flip x k -> k True ++ k False  
  return x -> [x]
} () .  
  x <- perform flip ()  
  if x then  
    perform flip ()  
  else  
    perform throw ()
}
```

```
effect choice {  
  flip : () -> bool
}

effect exn {  
  throw : () -> a
}
```
choice and exception

**effect** choice {
  flip : () -> bool
}

**effect** exn {
  throw : () -> a
}

```haskell
handler {
  throw x k -> Nothing
  return x   -> Just x
}
\_.

handler {
  flip x k -> k True ++ k False
  return x   -> [x]
}
\_.

x <- perform flip ()
if x then
  perform flip ()
else
  perform throw ()
// Nothing
```
Select
effect select<a> {  
  select : [a] -> a
}

failed = perform select []
Select

effect select<a> { 
    select : [a] -> a
}

failed = perform select []

x <- perform select [1..15]
y <- perform select [1..15]
z <- perform select [1..15]
if x * x + y * y == z * z
then (x,y,z)
else failed
select \langle a \rangle \ { 
  \textit{select} : [a] \rightarrow a 
}

failed = \text{perform} \ select \ [ ]

\textbf{handler} \ {
  \textit{select} \ xs \ k \rightarrow \ \text{concatMap} \ k \ xs \\
  \text{return} \ x \rightarrow [x] 
} \ (\_.

x \leftarrow \text{perform} \ select \ [1..15] \\
y \leftarrow \text{perform} \ select \ [1..15] \\
z \leftarrow \text{perform} \ select \ [1..15] \\
if \ x * x + y * y == z * z \\
then \ (x,y,z) \\
else \ \text{failed} 
)
**Select**

\[
\text{effect select}\langle a \rangle \{ \\
\quad \text{select} : [a] \rightarrow a \\
\}
\]

failed = **perform** select []

\[
\text{handler} \{ \\
\quad \text{select} \; xs \; k \rightarrow \text{concatMap} \; k \; xs \\
\quad \text{return} \; x \rightarrow [x] \\
\}
\]

\[
\begin{aligned}
&x \leftarrow \text{perform} \; \text{select} \; [1..15] \\
y \leftarrow \text{perform} \; \text{select} \; [1..15] \\
z \leftarrow \text{perform} \; \text{select} \; [1..15] \\
&\text{if} \; x \ast x + y \ast y \; == \; z \ast z \\
&\text{then} \; (x,y,z) \\
&\text{else} \; \text{failed}
\end{aligned}
\]

// [(3,4,5),(4,3,5),(5,12,13),(6,8,10) // ,(8,6,10),(9,12,15),(12,5,13),(12,9,15)]
\begin{center}
\textbf{Select}
\end{center}

\texttt{handler} \{ \\
\texttt{select} xs k -> \\
\texttt{let} f ys = \texttt{case} ys of \\
[] -> \texttt{Nothing} \\
\texttt{y':ys'} -> \texttt{case} k y' of \texttt{Nothing} -> f ys' \\
\texttt{r} \\
\texttt{in} f xs \\
\texttt{return} x -> \texttt{Just} x
\}

\texttt{x <- perform select [1..15]} \\
\texttt{y <- perform select [1..15]} \\
\texttt{z <- perform select [1..15]} \\
\texttt{if} x * x + y * y == z * z \\
\texttt{then} (x,y,z) \\
\texttt{else} failed
\}

\texttt{effect select\textless \texttt{a} \texttt{\rangle} \{} \\
\texttt{select : [a] -> a} \\
\}

\texttt{failed = perform select []}
select \{ 
  select : [a] -> a 
\}

failed = perform select []

handler { 
  select xs k ->
    let f ys = case ys of
      [] -> Nothing
      y':ys' -> case k y' of Nothing -> f ys'
        Just v -> Just v
    in f xs
    return x -> Just x
  )

  x <- perform select [1..15]
  y <- perform select [1..15]
  z <- perform select [1..15]
  if x * x + y * y == z * z
    then (x,y,z)
    else failed
  )
  // Just (3,4,5)
N-Queens

effect select<a> { 
  select : [a] -> bool 
}

failed = perform select []
N-Queens

```haskell
nQueens n = fold f [] [1..n] where
  f rows col = row <- perform select [1..n] if (safeAddition rows row 1) then (row : rows) else failed

// is it safe to add the new queen?
safeAddition rows r i =
  case rows of
    [] -> True
    (r:rows) ->
      row /= r &&
      row /= r &&
      abs (row - r) /= i &&
      safeAddition rows row (i + 1)
```

**effect select<a> {**

  ```haskell```
  select : [a] -> bool
  ```haskell```

**failed = perform select []**
Cooperative multi-threading
Cooperative multi-threading

```plaintext
effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}
effect coop {
  yield : () -> ()
  fork : (() -> ()) -> ()
}
```
Cooperative multi-threading

```haskell
import Control.Concurrent

effect queue {  
enqueue : (() -> ()) -> ()  
dequeue : () -> (() -> ())
}
effect coop {  
yield : () -> ()  
fork : (() -> ()) -> ()
}
scheduler f =

handler {  
yield _ k ->  
  perform enqueue k  
next <- perform dequeue ();  
next ()  
fork g k ->  
  perform enqueue k  
schedule g  
return _ ->  
  next <- perform dequeue ()  
next ()
}
f
```
Cooperative multi-threading

```
effect queue {
  enqueue : ((() -> ())) -> ()
  dequeue : () -> ((() -> ()))
}
effect coop {
  yield : () -> ()
  fork : (() -> ()) -> ()
}
scheduler f =
handler {
  yield _ k ->
    perform enqueue k
  next <- perform dequeue ();
  next ()
  fork g k ->
    perform enqueue k
    schedule g
  return _ ->
    next <- perform dequeue ()
    next ()
}
f
scheduler (_).
  print "A"; perform fork (_._. print "B"; perform yield (); print "E")
  print "C"; perform fork (_._. print "D"; perform yield (); print "G"); print "F"
```
**Cooperative multi-threading**

```plaintext

**Effect** queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}

**Effect** coop {
  yield : () -> ()
  fork : (() -> ()) -> ()
}

scheduler f =

```

```plaintext
handler {
  yield _ k ->
    perform enqueue k
  next <- perform dequeue ()
  next ()
  fork g k ->
    perform enqueue k
    schedule g
  return _ ->
    next <- perform dequeue ()
    next ()
}
```

scheduler (\_.

```plaintext
print "A"; perform fork (\_. print "B"; perform yield (); print "E"));
print "C"; perform fork (\_. print "D"; perform yield (); print "G")); print "F"
```
Cooperative multi-threading

```haskell

effect queue { 
    enqueue : (() -> ()) -> ()
    dequeue : () -> (() -> ())
}
effect coop { 
    yield : () -> () 
    fork : (() -> ()) -> ()
}

scheduler f = 
  handler { 
    yield _ k -> 
      perform enqueue k 
      next <- perform dequeue ();
      next ()
    fork g k -> 
      perform enqueue k 
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }
  f

scheduler (_._. 
  print "A"; perform fork (_._. print "B"; perform yield (); print "E")
  print "C"; perform fork (_._. print "D"; perform yield (); print "G")
) // A B
```
Cooperative multi-threading

```effect queue {  
    enqueue : (() -> ()) -> ()  
    dequeue : () -> (() -> ())  
}
effect coop {  
    yield : () -> ()  
    fork : (() -> ()) -> ()  
}
scheduler f =  
    handler {  
        yield _ k ->  
            perform enqueue k  
            next <- perform dequeue ();  
            next ()  
        fork g k ->  
            perform enqueue k  
            schedule g  
        return _ ->  
            next <- perform dequeue ()  
            next ()  
    }  
    f

scheduler (_,).
    print “A”; perform fork (_, print “B”; perform yield (); print “E”));
    print “C”; perform fork (_, print “D”; perform yield (); print “G”); print “F”)
    // A B C
Cooperative multi-threading

```plaintext

**effect** queue {  
  enqueue : (() -> ()) -> ()  
  dequeue : () -> (() -> ())  
}

**effect** coop {  
  yield : () -> ()  
  fork : (() -> ()) -> ()  
}

scheduler f =

  **handler** {  
    yield _ k ->  
      perform enqueue k  
      next <- perform dequeue ();  
      next ()  
    fork g k ->  
      perform enqueue k  
      schedule g  
    return _ ->  
      next <- perform dequeue ()  
      next ()  
  }

  f

scheduler (_._.
  print "A"; perform fork (_._. print "B"; perform yield (); print "E"));
  print "C"; perform fork (_._. print "D"; perform yield (); print "G")); print "F"
  // A B C D
```
Cooperative multi-threading

```javascript
/*
  Cooperative multi-threading
*/

// Effect queue

effect queue {
  enqueue : (() -> ()) -> ()
  dequeue : () -> (() -> ())
}

// Effect coop

effect coop {
  yield : () -> ()
  fork : (() -> ()) -> ()
}

// Scheduler f

scheduler f =

  handler {
    yield _ k ->
      perform enqueue k
      next <- perform dequeue ();
      next ()
    fork g k ->
      perform enqueue k
      schedule g
    return _ ->
      next <- perform dequeue ()
      next ()
  }

  f

// Scheduler (\_.

  print "A"; perform fork (\_. print "B"; perform yield (); print "E"));
  print "C"; perform fork (\_. print "D"; perform yield (); print "G")); print "F"
} // A B C D E
Cooperative multi-threading

```
effect queue {  
enqueue : (() -> ()) -> ()  
dequeue : () -> (() -> ())  
}
effect coop {  
yield : () -> ()  
fork : (() -> ()) -> ()  
}

scheduler f =  
handler {  
yield _ k ->  
    perform enqueue k  
    next <- perform dequeue ();  
    next ()  
    fork g k ->  
    perform enqueue k  
    schedule g  
    return _ ->  
    next <- perform dequeue ()  
    next ()  
  }  
f

scheduler (\_. print "A"; perform fork (\_. print "B"; perform yield (); print "E"));  
print "C"; perform fork (\_. print "D"; perform yield (); print "G")); print "F"  
// A B C D E F
```
Cooperative multi-threading

```effect
queue {  
enqueue : (()) -> (()) -> ()  
dequeue : () -> (()) -> ()  
}
effect coop {  
yield : () -> ()  
fork : (()) -> (()) -> ()  
}
```

```
scheduler f =
handler {  
yield _ k ->
  perform enqueue k
  next <- perform dequeue ();
  next ()
  fork g k ->
    perform enqueue k
    schedule g
  return _ ->
    next <- perform dequeue ()
    next ()
}
```

scheduler (\_.
  print "A"; perform fork (\_. print "B"; perform yield (); print "E")
  print "C"; perform fork (\_. print "D"; perform yield (); print "G")
) // A B C D E F G
Algebraic effects Summary

Composable and modular computational effects
Algebraic effects Summary

Composable and modular computational effects

Key ideas:
Algebraic effects Summary

Composable and modular computational effects

Key ideas:

1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value
Algebraic effects Summary
Composable and modular computational effects

Key ideas:

1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value

Examples:
Algebraic effects Summary
Composable and modular computational effects

Key ideas:

1. algebraic effects define a family of operations
2. effect handlers give semantics to operations
3. every computation either calls an operation or returns a value

Examples:

read, exn, state, choice, select, coop, …
Challenges
Challenges

handle
handle
handle
perform op v
Challenges
Challenges

handle
handle
handle
handle

perform op v
Challenges

- handle
- handle
- handle
- perform op v
Challenges

\( \text{op } x \ k \rightarrow e2 \)

handle

handle

handle

perform op v
Challenges

\[ op \ x \ k \rightarrow e2 \]

- handle
- handle
- handle
- perform \( op \ v \)
Challenges

\[ op \ x \ k \rightarrow e2 \]

\[ e2 \ [x:=v, k:=\_] \]

handle

handle

handle

perform \( op \ v \)
Challenges

1. **Searching**
   
a *linear* search through the current evaluation context
Challenges

1. **Searching**
   a *linear search* through the current evaluation context

2. **Capturing**
capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function
Challenges

1. **Searching**
   - a *linear* search through the current evaluation context

2. **Capturing**
   - capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

Can we implement algebraic effects efficiently?
**Continuation-passing style**

**Links**  Hillerström et al 2017, 2020
Leijen 2017
Schuster et al 2020
......

**Capability-passing style**

Effekt

Schuster et al 2020
Brachthäuser et al 2020
......

**Segmented Stacks**

Sivaramakrishnan et al 2021
......

**Rewriting**

**Eff**

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalios et al 2021
......
Continuation-passing style

Closure allocation cost

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020

……

Segmented Stacks

Sivaramakrishnan et al 2021

……

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021

……
handle

handle

handle

handle

perform op v
raise Unhandled ()

handle

handle

handle

perform op

Fiber
handle
handle
handle
Fiber
perform op v
handle

\text{op} x k \rightarrow e2

handle

perform op v

Fiber
handle
handle
handle
handle
perform op v

Fiber
\[ e2 \{ x := v, k := \} \]
handle

1 +

k 0

handle

handle
**Continuation-passing style**

Closure allocation cost

**Capability-passing style**

Effekt

Schuster et al 2020
Brachthäuser et al 2020

**Segmented Stacks**

Sivaramakrishnan et al 2021

**Rewriting**

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021

......
Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Effekt

Schuster et al 2020
Brachthäuser et al 2020

Rewriting

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021

......
effect exn {
  throw : () -> a
}

div m n
    = if n == 0
        then perform throw ()
        else m / n
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
      then perform throw ()
      else m / n

div m n throw
  = if n == 0
      then perform throw ()
      else m / n
effect exn {
  throw : () -> a
}

div m n
  = if n == 0
    then perform throw ()
    else m / n

handler {
  throw x k -> Nothing
} (_.
  div 42 0
) // Nothing
effect exn {
    throw : () -> a
}

div m n
    = if n == 0
        then perform throw ()
        else m / n

handler {
    throw x k -> Nothing
} (\_.
    div 42 0
) // Nothing

handle {
    throw x k -> Nothing
}
(div 42 0 throw)
handle
handle
handle
\_ perform op v
handle
handle
handle

_. perform op v
handle

handle

_. perform op v

handle

handle

_. perform op v

handle

handle

handle

perform op v
**Continuation-passing style**

Closure allocation cost

**Segmented Stacks**

Efficient one-shot resumption

**Capability-passing style**

Effekt

Schuster et al 2020
Brachthäuser et al 2020
......

**Rewriting**

Eff

Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al 2021
......
Continuation-passing style
Closure allocation cost

Capability-passing style
Efficient lexically scoped handlers

Segmented Stacks
Efficient one-shot resumption

Rewriting
Eff
Kiselyov and Sivaramakrishnan 2018
Saleh et al. 2018
Karachalias et al. 2021
......
Continuation-passing style

Closure allocation cost

Segmented Stacks

Efficient one-shot resumption

Capability-passing style

Efficient lexically scoped handlers

Rewriting

Source-to-source transformations
Algebraic effects and evidence-passing semantics in Koka

https://koka-lang.github.io/
Algebraic effects and evidence-passing semantics in Koka

Koka: Programming with Row Polymorphic Effect Types
Leijen, MSFP 2014

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, POPL 2017

Implementing Algebraic Effects in C
Leijen, APLAS 2017

https://koka-lang.github.io/
Algebraic effects and evidence-passing semantics in Koka

Koka: Programming with Row Polymorphic Effect Types
Leijen, MSFP 2014

Type DirectedCompilation of Row-Typed Algebraic Effects
Leijen, POPL 2017

Implementing Algebraic Effects in C
Leijen, APLAS 2017

https://koka-lang.github.io/
Algebraic effects and evidence-passing semantics in Koka

Koka: Programming with Row Polymorphic Effect Types
Leijen, MSFP 2014

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, POPL 2017

Implementing Algebraic Effects in C
Leijen, APLAS 2017

Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, ICFP 2020

Effect Handlers in Haskell, Evidently
Xie and Leijen, Haskell 2020

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, ICFP 2021

https://koka-lang.github.io/
Algebraic effects and evidence-passing semantics in Koka

Koka: Programming with Row Polymorphic Effect Types
Leijen, MSFP 2014

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, POPL 2017

Implementing Algebraic Effects in C
Leijen, APLAS 2017

Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, ICFP 2020

Effect Handlers in Haskell, Evidently
Xie and Leijen, Haskell 2020

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, ICFP 2021

Perceus: Garbage Free Reference Counting with Reuse
Reinking*, Xie*, de Moura and Leijen, PLDI 2021

https://koka-lang.github.io/
Algebraic effects and evidence-passing semantics in Koka

Koka: Programming with Row Polymorphic Effect Types
Leijen, MSFP 2014

Type Directed Compilation of Row-Typed Algebraic Effects
Leijen, POPL 2017

Implementing Algebraic Effects in C
Leijen, APLAS 2017

Effect Handlers, Evidently
Xie, Brachthäuser, Hillerström, Schuster and Leijen, ICFP 2020

Effect Handlers in Haskell, Evidently
Xie and Leijen, Haskell 2020

Generalized Evidence Passing for Effect Handlers (Efficient Compilation of Effect Handlers to C)
Xie and Leijen, ICFP 2021

Perceus: Garbage Free Reference Counting with Reuse
Reinking*, Xie*, de Moura and Leijen, PLDI 2021

First-class Handler Names
Xie, Cong and Leijen, HOPE 2021

https://koka-lang.github.io/
Evidence-passing semantics
Evidence-passing semantics

Algebraic effects

Efficient C (with no special runtime support)
Evidence-passing semantics

Algebraic effects

Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Efficient C (with no special runtime support)
Evidence-passing semantics

- Algebraic effects
  - Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
  - Evidence-passing semantics

Efficient C (with no special runtime support)
Evidence-passing semantics

- Algebraic effects
  - Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
  - Evidence-passing semantics
    - Bubbling Yields [Pretnar 2015]
  - Efficient C (with no special runtime support)
Evidence-passing semantics

- Algebraic effects
  - Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
  - Evidence-passing semantics
    - Bubbling Yields [Pretnar 2015]
      - Monadic translation
        - Efficient C (with no special runtime support)
Evidence-passing semantics

Algebraic effects

- Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

Evidence-passing semantics
- optimization of tail-resumptive operations
- insertion- versus canonical ordered evidence vector

- Bubbling Yields [Pretnar 2015]
  - short-cut resumption [Kiselyov and Ishii 2015]

- Monadic translation
  - bind-inlining and join-point sharing

Efficient C (with no special runtime support)
Evidence-passing semantics

- Algebraic effects
  - Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]
  - Evidence-passing semantics
    - optimization of tail-resumptive operations
    - insertion- versus canonical ordered evidence vector
  - Bubbling Yields [Pretnar 2015]
    - short-cut resumption [Kiselyov and Ishii 2015]
  - Monadic translation
    - bind-inlining and join-point sharing

Efficient C (with no special runtime support)
Evidence-passing semantics

- Algebraic effects
  - Multi-prompt delimited control [Forster et al. 2019; Gunter et al. 1995]

- Evidence-passing semantics
  - optimization of tail-resumptive operations
  - insertion- versus canonical ordered evidence vector

- Bubbling Yields [Pretnar 2015]
  - short-cut resumption [Kiselyov and Ishii 2015]

- Monadic translation
  - bind-inlining and join-point sharing

- Efficient C (with no special runtime support)

---

PLDI 2021: Perceus - Garbage Free Reference Counting with Reuse

Alex Reinking  
Microsoft Research  
Redmond, WA, USA  
alexr@seas.harvard.edu

Leonardo de Moura  
Microsoft Research  
Redmond, WA, USA  
leo@le.com

Ningning Xie  
University of Hong Kong  
Hong Kong, China  
nxie@cs.hku.hk

Daan Leijen  
Microsoft Research  
Redmond, WA, USA  
daan@microsoft.com
**Challenge**

1. **Searching**
   a *linear* search through the current evaluation context

2. **Capturing**
   capture the evaluation context (i.e., stacks and registers) up to the found handler, and create a resumption function

\[
\text{op } x \text{ k } \rightarrow e2
\]

- handle
- handle
- handle
- perform \( op \ v \)

\[
e2 \ [x:=v, \ k:=\bullet]
\]

- handle
- handle
- handle
Multi-prompt semantics

separating searching from capturing
Multi-prompt semantics
separating searching from capturing

handle
handle
handle
handle

perform op v
Multi-prompt semantics
separating searching from capturing
Multi-prompt semantics
separating searching from capturing
Multi-prompt semantics

separating searching from capturing

handle
handle
handle
perform op v

prompt m1
handle
handle
perform op v
Multi-prompt semantics
separating searching from capturing

m1: a unique marker identifying handlers
Multi-prompt semantics

separating searching from capturing

handle

prompt m1

m1: a unique marker identifying handlers

handle

prompt m2

handle

perform op v

perform op v
Multi-prompt semantics
separating searching from capturing

handle

prompt m1
m1: a unique marker identifying handlers

handle

prompt m2

handle

prompt m3

perform op v

perform op v
Multi-prompt semantics
separating searching from capturing

handle
handle
handle
perform op v

prompt m1
prompt m2
prompt m3

m1: a unique marker identifying handlers

yield m1 v
Multi-prompt semantics
separating searching from capturing

handle

prompt m1
m1: a unique marker identifying handlers

handle

prompt m2

handle

prompt m3

perform op v

yield m1 v

yielding to a handler identified by m1
Multi-prompt semantics

separating searching from capturing

handle

prompt m1

prompt m2

prompt m3

perform op v

yield m1 v

e2 \[x:=v, k:=\]

m1: a unique marker identifying handlers

yielding to a handler identified by m1

prompt m1

prompt m2

prompt m3
Multi-prompt semantics
separating searching from capturing

handle

prompt m1

prompt m2

prompt m3

m1: a unique marker identifying handlers

yield m1 v

yielding to a handler identified by m1

e2 \[x:=v, k:=\]

prompts m1, m2, and m3

Searching
Multi-prompt semantics
separating searching from capturing

handle

prompt m1

prompt m2

prompt m3

m1: a unique marker identifying handlers

perform op v

yield m1 v

e2 [x:=v, k:=e]

yielding to a handler identified by m1

Searching

capturing
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

handle

prompt m1

prompt m2

prompt m3

perform op v

yield m1 v

e2 \[x:=v, k:=\]

prompt m1

prompt m2

prompt m3
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

handle
handle
handle

prompt m1
prompt m2
prompt m3

perform op v
yield m1 v

e2 [x:=v, k:= ]

prompt m1
prompt m2
prompt m3
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

\[
\text{handle} \quad \text{prompt } m_1
\]

\[
\text{handle} \quad 
\begin{array}{c}
\ll<br>
\end{array}
\begin{array}{c}
\begin{array}{c}
\ll
\end{array}
\ll
\begin{array}{c}
\ll
\end{array}
\end{array}
\ll
\begin{array}{c}
\ll
\end{array}
\ll
\begin{array}{c}
\ll
\end{array}
\ll
\begin{array}{c}
\ll
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
\end{array}
\ll
\begin{array}{c}
\ll<br>
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

```
handle
<
handle
< e1:(m1, h1)
handle
< e1:(m1, h1),
< e2:(m2, h2)
handle
perform op v

prompt m1
<>
prompt m2
<>
prompt m3
```

\[ e2 [x:=v, k:=\textbullet] \]
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

\[ \ll e_1: (m_1, h_1) \rr \ll e_2: (m_2, h_2) \rr \frac{\cdot }{e_2 \left[ x:=v, k:=\_ \right]} \]

\[ \ll e_1: (m_1, h_1), e_2: (m_2, h_2) \rr \frac{\cdot }{\text{prompt } m_3} \]

\[ \ll e_1: (m_1, h_1), e_2: (m_2, h_2), e_3: (m_3, h_3) \rr \frac{\cdot }{\text{yield } m_1 v} \]
**Evidence-passing semantics**
make performs local: push down the current handlers as an evidence vector

```plaintext
| handle | << e1:(m1, h1) >> | prompt m1 |
| handle | << e1:(m1, h1), e2:(m2, h2) >> | prompt m2 |
| handle | << e1:(m1, h1), e2:(m2, h2), e3:(m3, h3) >> | yield m1 v |
|
| e2 [x:=v, k:=] |
```

```
/uni226A
/uni226B
/uni226A
/uni00A0
/uni226A
/uni226B
/uni226A
/uni00A0
/uni226A
/uni00A0
/uni226B
```

```
perform op v
```

```
prompt m1
prompt m2
prompt m3
```
Evidence-passing semantics
make performs local: push down the current handlers as an evidence vector

Constant-time Searching
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

\[
\begin{align*}
&\text{handle} \quad \text{prompt m1} \\
&\text{handle} \\
&\text{handle} \\
&\text{handle} \\
&\text{handle} \\
\text{perform op v} \quad \text{yield m1 v}
\end{align*}
\]
Optimization of tail-resumptive operations
avoid yields: evaluate tail-resumptive operations in-place

\[ \text{handle} \]

\[ \ll e_1 : (m_1, h_1) \gg \]

\[ \text{prompt m1} \]

\[ \ll e_2 : (m_2, h_2) \gg \]

\[ \text{prompt m2} \]

\[ \ll e_1 : (m_1, h_1), e_2 : (m_2, h_2) \gg \]

\[ \text{prompt m3} \]

\[ \text{perform op } v \]

\[ \text{yield m1 } v \]

\[ e_2 [x := v, k := \_] \]

\[ \text{prompt m1} \]

\[ \text{prompt m2} \]

\[ \text{prompt m3} \]

\[ \text{Optimization to tail-resumptive operations} \]

avoid yields: evaluate tail-resumptive operations in-place

\[ \text{perform op } v \]

\[ \text{yield m1 } v \]
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

\[
\text{op} \mapsto \lambda x. \lambda k. k e
\]
where \( k \notin \text{fv}(e) \)
Optimization of tail-resumptive operations
avoid yields: evaluate tail-resumptive operations in-place

\[ e \rightarrow \lambda x. \lambda k. k \ e \]

where \( k \notin \text{fv}(e) \)
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

\[
\text{prompt m1}
\]

\[
\text{prompt m2}
\]

\[
\text{prompt m3}
\]

\[
e_2 [x:=v, k:=\_]
\]

\[
\text{perform op v}
\]

\[
\text{handle}
\]

\[
\text{handle}
\]

\[
\text{handle}
\]

\[
\text{prompt m1}
\]

\[
\text{prompt m2}
\]

\[
\text{prompt m3}
\]

\[
\text{e}
\]

\[
\text{e_1 = (m_1, h_1)}
\]

\[
\text{e_2 = (m_2, h_2)}
\]

\[
\text{e_3 = (m_3, h_3)}
\]
Optimization of tail-resumptive operations

avoid yields: evaluate tail-resumptive operations in-place

\[ \text{handle} \]

\[ \langle \langle \langle e_1 (m_1, h_1), e_2 (m_2, h_2) \rangle \rangle \rangle \]

\[ \text{prompt m1} \]

\[ \text{prompt m2} \]

\[ \text{prompt m3} \]

\[ \text{perform op v} \]

\[ \langle \langle e_1 (m_1, h_1), e_2 (m_2, h_2), e_3 (m_3, h_3) \rangle \rangle \]

\[ \text{under } e_1 \]

\[ \langle \langle e \rangle \rangle \]

\[ \text{e} \]

\[ \text{e}_2 [x := v, k := \_] \]

\[ \text{op} \mapsto \lambda x. \lambda k. k e \]

where \( k \notin \text{fv}(e) \)

\[ \text{prompt m1} \]

\[ \text{prompt m2} \]

\[ \text{prompt m3} \]
handle \leftrightarrow [x:=v, k:=\bullet] handle

\[ e_1: (m_1, h_1), \]

\[ e_2: (m_2, h_2) \] \leftrightarrow

\[ e_1: (m_1, h_1), \]

\[ e_2: (m_2, h_2), \]

\[ e_3: (m_3, h_3) \] \Rightarrow

\[ \epsilon_1 = (m_1, h_1) \]

yield m_1 v

perform op v
handle

handle

handle

perform op v

c2 [x:=v, k:=v]

prompt m1

prompt m2

prompt m3

\[ e_2: (m_2, h_2) \]

\[ e_3: (m_3, h_3) \]

\[ e_1 = (m_1, h_1) \]

yield m1 v

prompt m1

prompt m2

prompt m3
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

handle

handle

handle

perform op v

\(<\>)

\(<e1:\langle m1, h1\rangle>\)

\(<e1:\langle m1, h1\rangle, e2:\langle m2, h2\rangle>\)

\(<e1:\langle m1, h1\rangle, e2:\langle m2, h2\rangle, e3:\langle m3, h3\rangle>\)

\(e1 = \langle m1, h1\rangle\)

prompt m1

prompt m2

prompt m3

\(\text{yield m1 v}\)

\(\text{e2 [x:=v, k:=\bullet]}\)

prompt m1

prompt m2

prompt m3
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

```
handle
handle
prompt m1
prompt m2
prompt m3
yield m1 v
e2 [x:=v, k:=•]

handle
handle
prompt m1
prompt m2
prompt m3

handle
handle
prompt m1
prompt m2
prompt m3
```

perform op v
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

handle
handle
handle
perform op v

prompt m1
prompt m2
prompt m3

yield m1 v

e2 [x:=v, k:=v]
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

handle

prompt m1

e2 [x:=v, k:=e]

prompt m2

prompt m3

handle

prompt m1

prompt m2

prompt m3

perform op v

yield m1 v
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

\[ e_2[x:=v, k:=\bullet] \]
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

*handle*

```
handle
handle
handle
perform op v
```

```
< e1:(m1, h1) >
prompt m1

< e1:(m1, h1), e2:(m2, h2) >
prompt m2

< e1:(m1, h1), e2:(m2, h2), e3:(m3, h3) >
yield m1 v

< e1 = (m1, h1) >
prompt m3
```

```
e2 [x:=v, k:=\_]
```

```
prompt m1
prompt m2
prompt m3
prompt m3
```
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

```
handle
```

```
handle
```

```
handle
```

```
perform op v
```

```
< e1: (m1, h1) >
```

```
prompt m1
```

```
yield m1 v
```

```
e2 [x:=v, k:=●]
```

```
< e1: (m1, h1), e2: (m2, h2) >
```

```
prompt m2
```

```
prompt m1
```

```
prompt m2
```

```
prompt m1
```

```
prompt m3
```

```
prompt m3
```

```
prompt m3
```
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

handle
handle
handle
perform op v

\[
\langle e_1 : (m_1, h_1), \quad e_2 : (m_2, h_2) \rangle
\]

\[
\langle e_1 : (m_1, h_1), \quad e_2 : (m_2, h_2), \quad e_3 : (m_3, h_3) \rangle
\]

\[
e_1 = (m_1, h_1)
\]

prompt m1
prompt m2
prompt m3

\[
e_2 [x := v, k := \cdot]
\]

\[
\text{e2 } [x := v, k := \cdot]
\]

prompt m1
prompt m2
prompt m3
Bubbling yields
make yields local: bubbling it up until it meets its corresponding prompt frame

\[
\begin{align*}
\text{handle} & \quad \langle \rangle \\
\text{handle} & \quad \langle e_1: (m_1, h_1) \rangle \\
\text{handle} & \quad \langle e_1: (m_1, h_1), e_2: (m_2, h_2) \rangle \\
\text{perform op v} & \quad \langle e_1: (m_1, h_1), e_2: (m_2, h_2), e_3: (m_3, h_3) \rangle \\
\end{align*}
\]
Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1
  (\_.
    perform ask () + perform ask ())
```

A evidence-passing multi-prompt delimited control monad

```
type Mon \mu \alpha = Evv \mu \to Ctl \mu \alpha
```

```
e \triangleright g = \lambda w. \text{case } e \text{ of } \text{Pure } x \to g \times w
  \text{Yield } m \times f \times k \to \text{Yield } m \times f \times (\lambda x. k \times x \triangleright g)
```
Monadic translation
all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

\[ \text{handler } h1 \]
\[ (\_ \cdot \text{perform ask }() + \text{perform ask }()) \]
\[ \rightarrow \]
\[ \text{handler } h1 \]
\[ (\_ \cdot \text{perform ask }() \triangleright (\_ \cdot \text{perform ask }()) \triangleright (\_ \cdot \text{perform ask }())) \]

A evidence-passing multi-prompt delimited control monad

evidence passing

type Mon \( \mu \alpha = [\text{Evv } \mu \rightarrow \text{Ctl } \mu \alpha] \)

e \triangleright g = \lambda w. \text{case } e w \text{ of Pure } x \rightarrow g x w

Yield \( m f k \rightarrow \text{Yield } m f (\lambda x. k x \triangleright g) \)
Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1
    (\_.
        perform ask () + perform ask ()
    )
```

A evidence-passing multi-prompt delimited control monad

```
(type Mon μ α = Evv μ → Ctl μ α control monad

e ⊢ g = λw. case e w of Pure x → g x w
                  Yield m f k → Yield m f (λx. k x ⊢ g)
```
Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```
handler h1
(\_.
  perform ask () + perform ask ()
)
```

```
handler h1
(\_.
  perform ask () >>= (\x.
    perform ask () >>= (\y.
      Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

type Mon μ α = Evv μ → Ctl μ α

control monad

```
e ⊢ g = λw. case e w of Pure x → g x w
                   Yield m f k → Yield m f (λx. k x ⊢ g)
```

pass the result and the current evidence
Monadic translation

all transitions are local: translate algebraic effects into a pure lambda calculus with a multi-prompt delimited control monad

```plaintext
handler h1
(\_.
 perform ask () + perform ask ())
```

```plaintext
handler h1
(\_.
 perform ask () >>= (\x.
 perform ask () >>= (\y.
 Pure (x + y))))
```

A evidence-passing multi-prompt delimited control monad

evidence passing

type Mon \(\mu \alpha\) = Evv \(\mu\) \(\rightarrow\) Ctl \(\mu \alpha\)

control monad

\[
e \triangleright g = \lambda w. \text{case } e \text{ of } \text{Pure } x \rightarrow g \ x \ w
\]

Yield \(mf k \rightarrow \text{Yield } mf \ (\lambda x. k \ x \triangleright g)\) bubbling

pass the result and the current evidence
Compiling to C

\[
\text{handler } \text{hl} (\_.) \\
\quad \text{perform } \text{ask } () + \text{perform } \text{ask } () \\
\leadsto \quad \text{handler } \text{hl} (\_.) \\
\quad \text{perform } \text{ask } () \triangleright (\_x. \text{perform } \text{ask } () \triangleright (\_y. \text{Pure } (x + y)))
\]

\[
\text{int } \text{expr}( \text{unit}_t \ u, \text{context}_t* \ ctx) \{ \\
\quad \text{int } x = \text{perform}_\text{ask}( \text{ctx}→w[0], \text{unit}, \text{ctx} ); \\
\quad \text{if } (\text{ctx}→\text{is}_\text{yielding}) \{ \text{yield}_\text{extend}(\&\text{join}_2,\text{ctx}); \text{return } 0; \} \\
\quad \text{int } y = \text{perform}_\text{ask}( \text{ctx}→w[0], \text{unit}, \text{ctx} ); \\
\quad \text{if } (\text{ctx}→\text{is}_\text{yielding}) \{ \text{yield}_\text{extend}(\text{alloc}_\text{closure}_\text{join}_1(x,\text{ctx}),\text{ctx}); \text{return } 0; \} \\
\quad \text{return } (x+y); \\
\}
\]
Compiling to C

\textbf{handler} h1
\[
\lambda . \quad \text{perform} \ ask() + \text{perform} \ ask() \\
\rightsquigarrow \text{handler} h1
\[
\lambda . \quad \text{perform} \ ask()\triangleright (\lambda x. \quad \text{perform} \ ask()\triangleright (\lambda y. \quad \text{Pure} \ (x + y))))
\]

evidence passing

\begin{verbatim}
int expr( unit_t u, context_t* ctx) {
    int x = perform_ask( ctx->w[0], unit, ctx );
    if (ctx->is_yielding) { yield_extend(&join2,ctx); return 0; }
    int y = perform_ask( ctx->w[0], unit, ctx );
    if (ctx->is_yielding) { yield_extend(aloc_closure_join1(x,ctx),ctx); return 0; }
    return (x+y); }
\end{verbatim}
Compiling to C

```
handler h1
(\_.
 perform ask () + perform ask ())

\rightarrow
handler h1
(\_.
 perform ask ()\triangleright (\x.
 perform ask ()\triangleright (\y.
 Pure (x + y))))
```

```
int expr( unit_t u, context_t* ctx) {
 int x = perform_ask( ctx→w[0], unit, ctx );
 if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; } 
 int y = perform_ask( ctx→w[0], unit, ctx );
 if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; } 
 return (x+y); }
```
Compiling to C

```c
int expr( unit_t u, context_t* ctx) {
    int x = perform_ask( ctx->w[0], unit, ctx );
    if (ctx->is_yielding) { yield_extend(&join2,ctx); return 0; }
    int y = perform_ask( ctx->w[0], unit, ctx );
    if (ctx->is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
    return (x+y); }
```

```
handler h1
(\_.
   perform ask () + perform ask ()
) ⇒
(handler h1
(\_.
   perform ask () ➝ (\x.
   perform ask () ➝ (\y.
   Pure (x + y))))

control monad
evidence passing
```

Integrating evidence passing with control mechanisms. The `expr` function takes an `unit_t` value and a context pointer, and performs a computation that involves two `ask` operations. The `perform ask` function is used to evaluate expressions and update the evidence vector. The `is_yielding` check is used to determine if the function is about to be resumed, and if so, the `yield_extend` function is called with a join point. The `Pure` function is used to return a monadic result.

### Compiling to C

1. **Compilation to C**: The algebraic evidence calculus is translated into a C-like expression. The `expr` function corresponds to the algebraic expression `expr(u, ctx) -> Pure(x + y)`.
2. **Evidence Passing**: The `perform ask` function is used to evaluate expressions and update the evidence vector. The `is_yielding` check is used to determine if the function is about to be resumed, and if so, the `yield_extend` function is called with a join point.
3. **Constant-Time Lookup**: The `w[0]` value is looked up in the evidence vector at runtime. This is done in constant time.
4. **Control Mechanism**: The `control monad` integrates evidence passing with control mechanisms, allowing for the manipulation of evidence and continuation values.
Compiling to C

\[ \text{handler } h1 \\
(\_ \cdot \quad \text{perform } ask () + \text{perform } ask ()) \]

\[ \xrightarrow{\text{evidence passing}} \text{handler } h1 \\
(\_ \cdot \quad \text{perform } ask () \triangleright (\_ \cdot \quad \text{perform } ask () \triangleright (\_ \cdot \quad \text{Pure } (x + y)))) \]

```
int expr( unit_t u, context_t* ctx) {
    int x = perform_ask( ctx→w[0], unit, ctx );
    if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; } // bubbling
    int y = perform_ask( ctx→w[0], unit, ctx );
    if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; } // bubbling
    return (x+y); }
```
Compiling to C

```
handler h1
(\_.
 perform ask () + perform ask ()
) →

handler h1
(\_.
 perform ask ()▹ (\x.
 perform ask ()▹ (\y.
 Pure (x + y))))
```

```
int expr( unit_t u, context_t* ctx) {
    int x = perform_ask( ctx→w[0], unit, ctx );
    if (ctx→is_yielding) { yield_extend(&join2,ctx); return 0; }
    int y = perform_ask( ctx→w[0], unit, ctx );
    if (ctx→is_yielding) { yield_extend(alloc_closure_join1(x,ctx),ctx); return 0; }
    return (x+y); }
```
Theorem 7. *(Semantics Preserving).* Given $\emptyset \vdash e : \text{int} \mid \langle \rangle \rightarrow e'$, if $e \rightarrow^* n$ in $F^\varepsilon$, then $e' \langle \rangle \rightarrow^*$ Pure $\langle \rangle \text{int } n$, in the polymorphic lambda calculus and if $e \uparrow$ in $F^\varepsilon$, then $e' \langle \rangle \uparrow$ in the polymorphic lambda calculus.

Theorem 5. *(Tail-resumptive Optimization is Sound).* If $\emptyset \vdash e : \sigma \mid \epsilon$, then $e \simeq_{ctx} e$. 

*ICFP 2021*
**Theorem 7. (Semantics Preserving).** Given $\emptyset \vdash e : int \mid \langle \rangle \rightsquigarrow e'$, if $e \rightarrow^* n$ in $F^e$, then $e' \langle \rangle \rightarrow^*\text{Pure } \langle \rangle \text{ int } n$, in the polymorphic lambda calculus and if $e \uparrow$ in $F^e$, then $e' \langle \rangle \uparrow$ in the polymorphic lambda calculus.

**Theorem 5. (Tail-resumptive Optimization is Sound).** If $\emptyset \vdash e : \sigma \mid \varepsilon$, then $e \equiv_{ctx} e$. 
Theorem 7. (*Semantics Preserving*). Given $\emptyset \vdash e : \text{int} \mid \langle \rangle \leadsto e'$, if $e \xrightarrow{*} n$ in $F^e$, then $e' \langle \rangle \xrightarrow{*}$ Pure $\langle \rangle \text{int} n$, in the polymorphic lambda calculus and if $e \xrightarrow{\uparrow}$ in $F^e$, then $e' \langle \rangle \xrightarrow{\uparrow}$ in the polymorphic lambda calculus.

Theorem 5. (*Tail-resumptive Optimization is Sound*). If $\emptyset \vdash e : \sigma \mid \epsilon$, then $e \simeq_{\text{ctx}} e$. 
**Theorem 7. (Semantics Preserving).** Given $\emptyset \vdash e : \mathit{int} \mid \langle \rangle \leadsto e'$, if $e \xrightarrow{\ast} n$ in $F^e$, then $e' \langle \rangle \xrightarrow{\ast}$ in the polymorphic lambda calculus and if $e \uparrow$ in $F^e$, then $e' \langle \rangle \uparrow$ in the polymorphic lambda calculus.

**Theorem 5. (Tail-resumptive Optimization is Sound).** If $\emptyset \vdash e : \sigma \mid \epsilon$, then $e \equiv_{\text{ctx}} e$. 


Benchmarks
ICFP 2021

Fig. 6. Execution time averaged over 10 runs

Our benchmarks are taken from [Kiselyov and Ishii 2015], and each is designed to probe specific aspects of effect handling with minimal other computation and allocation overheads:

- **counter** shows how the most common tail-resumptive effects are handled;
- **counter1** and **counter10** emphasize the impact of nested handlers;
- **mstate** demonstrates the use of first-class resumptions (captured under a lambda);
- **nqueens** and **triple** uses multi-shot resumptions.

Below we discuss the benchmark results.

- **counter**. This benchmark implements a state effect using a mutable reference such that both get and set operations are tail-resumptive. It then performs 200M get and set operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly and the execution times are very close. The libhandler C implementation is 1.5⇥ faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]). Moreover, Mp.E is about 4⇥ slower as Koka, but Ev.E is 4⇥ faster! This is because GHC is able to fully inline the handler and operations and optimizes almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02s which is more in line with the results seen in **counter1** and **counter10**. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings making it 10⇥ slower than the optimized version.

- **counter1**. This is the same as **counter** but with one (unused) reader effect handler in between. This time Koka is 1.5⇥ faster than OCaml: due to evidence passing, the execution times of

https://koka-lang.github.io/
## Benchmarks

### ICFP 2021

### Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>A state effect using a mutable reference with both get and set operations. It performs 200 million get and set operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly, and the execution times are very close. The libhandler C implementation is 1.5 times faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]). Moreover, Mp.E is about 4 times slower than Koka, but Ev.E is 4 times faster! This is because GHC is able to fully inline the handler and operations and optimize almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02 seconds, which is more in line with the results seen in counter1 and counter10. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings, making it 10 times slower than the optimized version.</td>
</tr>
<tr>
<td>counter1</td>
<td>The same as counter but with one (unused) reader effect handler in between. This time, Koka is 1.5 times faster than OCaml: due to evidence passing, the execution times of</td>
</tr>
<tr>
<td>counter10</td>
<td></td>
</tr>
<tr>
<td>mstate</td>
<td></td>
</tr>
<tr>
<td>nqueens</td>
<td></td>
</tr>
<tr>
<td>triple</td>
<td></td>
</tr>
</tbody>
</table>

### Explanation of Benchmarks

- **counter**: This benchmark implements a state effect using a mutable reference such that both `get` and `set` operations are tail-resumptive. It then performs 200 million `get` and `set` operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly, and the execution times are very close. The libhandler C implementation is 1.5 times faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]). Moreover, Mp.E is about 4 times slower than Koka, but Ev.E is 4 times faster! This is because GHC is able to fully inline the handler and operations and optimize almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02 seconds, which is more in line with the results seen in counter1 and counter10. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings, making it 10 times slower than the optimized version.

- **counter1**: This is the same as `counter` but with one (unused) reader effect handler in between. This time, Koka is 1.5 times faster than OCaml: due to evidence passing, the execution times of...
Benchmarking
ICFP 2021

Koka
multi-core OCaml
Mp.Eff (Haskell)
Ev.Eff (Haskell)
libhandler (C)
Koka, Insertion-ordered
Koka, No short-cut resumption
Koka, No bind-inlining
Koka, No tail-resumptive opt.

Fig. 6. Execution time averaged over 10 runs

such, the results are meant to establish if the effect handler compilation strategies described in this paper are viable and can be competitive, but should not be interpreted as a measure of absolute performance between systems and languages. Execution times are shown in Figure 6. The execution times are averaged over 10 runs, on an AMD 5950X at 3.4Ghz with 32GiB memory running Ubuntu 20.04, with Koka v2.1.2, multi-core OCaml 4.10, libhandler v0.5, and GHC 8.6.5.

Our benchmarks are taken from [Kiselyov and Ishii 2015], and each is designed to probe specific aspects of effect handling with minimal other computation and allocation overheads:

- **counter**: shows how the most common tail-resumptive effects are handled;
- **counter1** and **counter10**: emphasize the impact of nested handlers;
- **mstate**: demonstrates the use of full first-class resumptions (captured under a lambda);
- **nqueens** and **triple**: uses multi-shot resumptions.

Below we discuss the benchmark results.

- **counter**: This benchmark implements a state effect using a mutable reference such that both get and set operations are tail-resumptive. It then performs 200M get and set operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly and the execution times are very close. The libhandler C implementation is 1.5⇥ faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]). Moreover, Mp.E is about 4⇥ slower as Koka, but Ev.E is 4⇥ faster! This is because GHC is able to fully inline the handler and operations and optimizes almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02s which is more in line with the results seen in counter1 and counter10. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings making it 10⇥ slower than the optimized version.

- **counter1**: This is the same as **counter** but with one (unused) reader effect handler in between. This time Koka is 1.5⇥ faster than OCaml: due to evidence passing, the execution times of


ICFP	2021

Benchmarks
### Benchmarks

ICFP 2021

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>3.97s</td>
</tr>
<tr>
<td>counter1</td>
<td>3.96s</td>
</tr>
<tr>
<td>counter10</td>
<td>3.96s</td>
</tr>
<tr>
<td>mstate</td>
<td>1.14s</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.15s</td>
</tr>
<tr>
<td>triple</td>
<td>1.15s</td>
</tr>
</tbody>
</table>

**Koka**, **Insertion-ordered**
**Koka, No short-cut resumption**
**Koka, No bind-inlining**
**Koka, No tail-resumptive opt.**

#### Description

Our benchmarks are taken from [Kiselyov and Ishii 2015], and each is designed to probe specific aspects of effect handling with minimal other computation and allocation overheads:

- **counter** shows how the most common tail-resumptive effects are handled;
- **counter1** and **counter10** emphasize the impact of nested handlers;
- **mstate** demonstrates the use of first-class resumptions (captured under a lambda);
- **nqueens** and **triple** uses multi-shot resumptions.

Below we discuss the benchmark results.

- **counter**. This benchmark implements a state effect using a mutable reference such that both get and set operations are tail-resumptive. It then performs 200M get and set operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly and the execution times are very close. The libhandler C implementation is 1.5⇥ faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]).

Moreover, **Mp.E** is about 4⇥ slower as Koka, but **Ev.E** is 4⇥ faster! This is because GHC is able to fully inline the handler and operations and optimizes almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02s which is more in line with the results seen in **counter1** and **counter10**. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings making it 10⇥ slower than the optimized version.

- **counter1**. This is the same as **counter** but with one (unused) reader effect handler in between. This time Koka is 1.5⇥ faster than OCaml: due to evidence passing, the execution times of

Ninging Xie and Daan Leijen

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>1.14</td>
</tr>
<tr>
<td>counter1</td>
<td>1.15</td>
</tr>
<tr>
<td>counter10</td>
<td>1.15</td>
</tr>
<tr>
<td>mstate</td>
<td>1.83</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.06</td>
</tr>
<tr>
<td>triple</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Elapsed time (lower is better)

- Koka
- multi-core OCaml
- Mp.Eff (Haskell)
- Ex.Eff (Haskell)
- libhandler (C)

**Fig. 6.** Execution time averaged over 10 runs such, the results are meant to establish if the effect handler compilation strategies described in this paper are viable and can be competitive, but should not be interpreted as a measure of absolute performance between systems and languages. Execution times are shown in Figure 6. The execution times are averaged over 10 runs, on an AMD 5950X at 3.4Ghz with 32GiB memory running Ubuntu 20.04, with Koka v2.1.2, multi-core OCaml 4.10, libhandler v0.5, and GHC 8.6.5.

Our benchmarks are taken from [Kiselyov and Ishii 2015], and each is designed to probe specific aspects of effect handling with minimal other computation and allocation overheads:

- **counter** shows how the most common tail-resumptive effects are handled;
- **counter1** and **counter10** emphasize the impact of nested handlers;
- **mstate** demonstrates the use of full first-class resumptions (captured under a lambda);
- **nqueens** and **triple** uses multi-shot resumptions.

Below we discuss the benchmark results.

- **counter**. This benchmark implements a state effect using a mutable reference such that both get and set operations are tail-resumptive. It then performs 200M get and set operations in a tight loop. The tail-resumptive optimization in Koka and the fast stack switching in OCaml seem to perform similarly and the execution times are very close. The libhandler C implementation is 1.5× faster than Koka – we believe this is because it does no allocation at all. In contrast, both Koka and OCaml still allocate at each operation (for example, OCaml allocates a continuation object per resumption [Sivaramakrishnan et al. 2021]).

  Moreover, **Mp.Eff** is about 4× slower as Koka, but **Ex.Eff** is 4× faster! This is because GHC is able to fully inline the handler and operations and optimizes almost all effect handling code away. When we remove the inline pragma on the state handler definition, the benchmark takes about 2.02s which is more in line with the results seen in **counter1** and **counter10**. We also ran this benchmark with the tail-resumption optimization turned off; this causes Koka to always allocate a resumption and take the slow path through the monadic bindings making it 10× slower than the optimized version.

- **counter1**. This is the same as **counter** but with one (unused) reader effect handler in between.

  This time Koka is 1.5× faster than OCaml: due to evidence passing, the execution times of Proc. ACM Program. Lang., Vol. 5, No. ICFP, Article 71. Publication date: August 2021.
Take-aways
Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?
Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

Algebraic effects and handlers: composable and modular computational effects
Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

   Algebraic effects and handlers: composable and modular computational effects

3. Can we implement algebraic effects efficiently?
Take-aways

1. How to compose computational effects?
2. How to handle effects according to applications?

   **Algebraic effects and handlers: composable and modular computational effects**

3. Can we implement algebraic effects efficiently?

   **Evidence-passing semantics**
Take-aways

April 22 – 27, 2018, Dagstuhl Seminar 18172

Algebraic Effect Handlers go Mainstream

Organizers
Sivaramakrishnan Krishnamoorthy Chandrasekaran (University of Cambridge, GB)
Daan Leijen (Microsoft Research – Redmond, US)
Matija Pretnar (University of Ljubljana, SI)
Tom Schrijvers (KU Leuven, BE)
Efficient Compilation of Algebraic Effect Handlers

Ningning Xie