# Effects, Linearity, and Modalities

Wenhao Tang

The University of Edinburgh

Programming Language Seminar, Peking University, 30th Aug 2024

# Effects and Handlers

# Computational Effects

Pure programs do not interact with their environment.

# Computational Effects

Pure programs do not interact with their environment.

Effectful programs interact with their environment.

# Computational Effects

Pure programs do not interact with their environment.

Effectful programs interact with their environment.

Effects are the way programs interact with their environment.

# Computational Effects

Pure programs do not interact with their environment.

Effectful programs interact with their environment.

Effects are the way programs interact with their environment.

Effects are pervasive, including input/output, concurrency, exceptions, nondeterminism, probability, etc.

# Computational Effects

Pure programs do not interact with their environment.

Effectful programs interact with their environment.

Effects are the way programs interact with their environment.

Effects are pervasive, including input/output, concurrency, exceptions, nondeterminism, probability, etc.

Typically ad hoc and hard-wired in programming languages.

# Algebraic Effects and Handlers

(One of) the most popular approaches to modelling effects in programming languages.

# Algebraic Effects and Handlers

(One of) the most popular approaches to modelling effects in programming languages.

- Plotkin and Power, 2002, *Notions of Computation determine Monads*, FoSSaCS
- Plotkin and Power, 2003, *Algebraic Operations and Generic Effects*, Applied Categorical Structures (journal version)
- Plotkin and Pretnar, 2009, *Handlers of Algebraic Effects*, ESOP
- Plotkin and Pretnar, 2013, *Handling Algebraic Effects*, LMCS (journal version)

# Algebraic Effects and Handlers

(One of) the most popular approaches to modelling effects in programming languages.

- Plotkin and Power, 2002, *Notions of Computation determine Monads*, FoSSaCS
- Plotkin and Power, 2003, *Algebraic Operations and Generic Effects*, Applied Categorical Structures (journal version)
- Plotkin and Pretnar, 2009, *Handlers of Algebraic Effects*, ESOP
- Plotkin and Pretnar, 2013, *Handling Algebraic Effects*, LMCS (journal version)

For a detailed introduction to the history of computational effects, see the first part of Nicolas Wu's keynote *Modular Higher-Order Effects* at PADL'24.

# Both Academic and Industrial Interest

Papers:

`https://github.com/yallop/effects-bibliography`

# Both Academic and Industrial Interest

Papers:

`https://github.com/yallop/effects-bibliography`

Research languages:

- Eff
- Frank
- Effekt
- Helium
- Koka
- Links
- Flix

# Both Academic and Industrial Interest

Papers:
`https://github.com/yallop/effects-bibliography`

Research languages:

- Eff
- Frank
- Effekt
- Helium
- Koka
- Links
- Flix

In Products:

- semantic (GitHub)
- React (Facebook)
- Pyro (Uber)

# Both Academic and Industrial Interest

Papers:

`https://github.com/yallop/effects-bibliography`

Research languages:

- Eff
- Frank
- Effekt
- Helium
- Koka
- Links
- Flix

In Products:

- semantic (GitHub)
- React (Facebook)
- Pyro (Uber)

Libraries in almost all mainstream languages even including C and C++.

# Both Academic and Industrial Interest

Papers:

`https://github.com/yallop/effects-bibliography`

Research languages:

- Eff
- Frank
- Effekt
- Helium
- Koka
- Links
- Flix

In Products:

- semantic (GitHub)
- React (Facebook)
- Pyro (Uber)

Libraries in almost all mainstream languages even including C and C++.

Primitive supports in industrial languages (for both user-defined effects and low-level features):

- OCaml
- Unison
- WebAssembly (ongoing)
- Cangjie (ongoing)

# Algebraic Effects Specify Syntax

The key idea is to separate the syntax of effects from their semantics.

# Algebraic Effects Specify Syntax

The key idea is to separate the syntax of effects from their semantics.

```
effect choose : 1 ⇒ Bool
```

# Algebraic Effects Specify Syntax

The key idea is to separate the syntax of effects from their semantics.

```
effect choose : 1 ⇒ Bool

pick (x, y) = if do choose () then x else y
```

# Algebraic Effects Specify Syntax

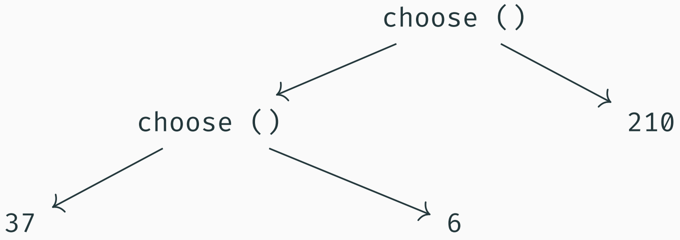
The key idea is to separate the syntax of effects from their semantics.

```
effect choose : 1 ⇒ Bool

pick (x, y) = if do choose () then x else y

prog _ = pick (pick (37, 6), 210)
```

# Algebraic Effects Specify Syntax

The key idea is to separate the syntax of effects from their semantics.

```
effect choose : 1 ⇒ Bool

pick (x, y) = if do choose () then x else y

prog _ = pick (pick (37, 6), 210)
```

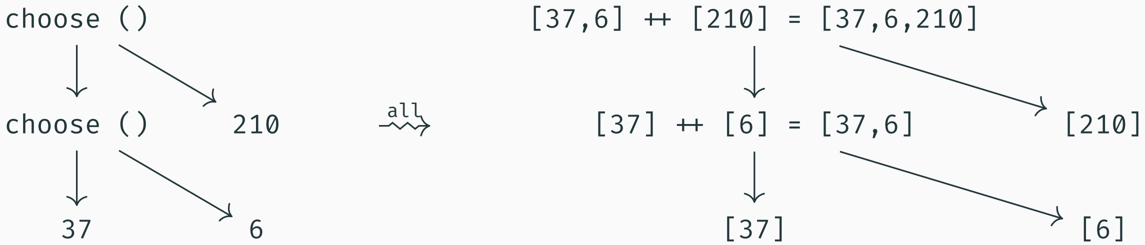


# Effect Handlers Provide Semantics

```
all m = handle m () with
  return x   ⇒ [x]
  choose _ r ⇒ r true ++ r false
```

# Effect Handlers Provide Semantics

```
all m = handle m () with
  return x   ⇒ [x]
  choose _ r ⇒ r true ++ r false
```



# Different Semantics without Changing Syntax

```
first m = handle m () with
  return x   ⇒ x
  choose _ r ⇒ r true
```

```
# first prog ⇝ 37
```

# Different Semantics without Changing Syntax

```
first m = handle m () with                    # first prog ⇝ 37
  return x   ⇒ x
  choose _ r ⇒ r true

last m = handle m () with                     # last prog ⇝ 210
  return x   ⇒ x
  choose _ r ⇒ r false
```

# Different Semantics without Changing Syntax

```
first m = handle m () with                  # first prog ⇝ 37
  return x   ⇒ x
  choose _ r ⇒ r true

last m = handle m () with                   # last prog ⇝ 210
  return x   ⇒ x
  choose _ r ⇒ r false

minimum m = handle m () with                # minimum prog ⇝ 6
  return x   ⇒ x
  choose _ r ⇒ min (r true) (r false)
```

# Different Semantics without Changing Syntax

```
first m = handle m () with                  # first prog ⇝ 37
  return x   ⇒ x
  choose _ r ⇒ r true

last m = handle m () with                   # last prog ⇝ 210
  return x   ⇒ x
  choose _ r ⇒ r false

minimum m = handle m () with                # minimum prog ⇝ 6
  return x   ⇒ x
  choose _ r ⇒ min (r true) (r false)

maximum m = handle m () with                # maximum prog ⇝ 210
  return x   ⇒ x
  choose _ r ⇒ max (r true) (r false)
```

# Composable Syntax and Semantics

```
effect ask : 1 ⇒ Int
```

# Composable Syntax and Semantics

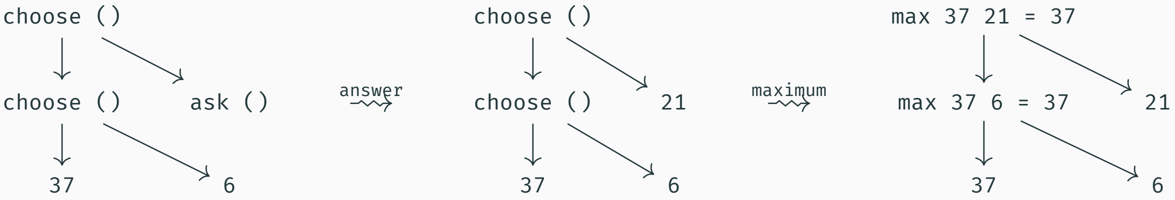
```
effect ask : 1 ⇒ Int

prog' _ = pick (pick (37, 6), do ask ())   # composable syntax
```

# Composable Syntax and Semantics

```
effect ask : 1 ⇒ Int

prog' _ = pick (pick (37, 6), do ask ())   # composable syntax

answer m = handle m () with
  ask _ r ⇒ r 21
```

# Composable Syntax and Semantics

```
effect ask : 1 ⇒ Int

prog' _ = pick (pick (37, 6), do ask ())   # composable syntax

answer m = handle m () with
  ask _ r ⇒ r 21
```




# composable semantics

## Different Points of View to Algebraic Effects and Handlers

- Modelling effects (ad hoc built-in effects, monads, monad transformers): *composable* and *customisable* effects interpretation in *direct style*.

# Different Points of View to Algebraic Effects and Handlers

- Modelling effects (ad hoc built-in effects, monads, monad transformers):
  *composable* and *customisable* effects interpretation in *direct style*.
- Control idioms (goto, if-then-else, call/cc, shift/reset):
  a structured approach to programming with delimited *continuations*.

# Different Points of View to Algebraic Effects and Handlers

- Modelling effects (ad hoc built-in effects, monads, monad transformers):
  *composable* and *customisable* effects interpretation in *direct style*.
- Control idioms (goto, if-then-else, call/cc, shift/reset):
  a structured approach to programming with delimited *continuations*.
- Programmers familiar with exceptions and try-catch:
  *restorable* exception / try-catch with *continuations*.

# Different Points of View to Algebraic Effects and Handlers

- Modelling effects (ad hoc built-in effects, monads, monad transformers):
  *composable* and *customisable* effects interpretation in *direct style*.
- Control idioms (goto, if-then-else, call/cc, shift/reset):
  a structured approach to programming with delimited *continuations*.
- Programmers familiar with exceptions and try-catch:
  *restorable* exception / try-catch with *continuations*.
- OOP programmers:
  (augmented) interfaces and implementations of objects

# Different Points of View to Algebraic Effects and Handlers

- Modelling effects (ad hoc built-in effects, monads, monad transformers):
  *composable* and *customisable* effects interpretation in *direct style*.
- Control idioms (goto, if-then-else, call/cc, shift/reset):
  a structured approach to programming with delimited *continuations*.
- Programmers familiar with exceptions and try-catch:
  *restorable* exception / try-catch with *continuations*.
- OOP programmers:
  (augmented) interfaces and implementations of objects
- Haskell programmers:
  freemonads and their algebras / folds / catamorphisms
- Coq programmers:
  (roughly) interaction trees

# Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

# Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

Traditional effect systems attach *effect types* to function arrows.

# Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

Traditional effect systems attach *effect types* to function arrows.

$$\texttt{pick} : \forall\, \texttt{a} \,.\, (\texttt{a}, \texttt{a}) \xrightarrow{\texttt{choose}} \texttt{a}$$

```
pick (x, y) = if do choose () then x else y
```

# Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

Traditional effect systems attach *effect types* to function arrows.

$$\texttt{pick} : \forall\, \texttt{a} \,.\, (\texttt{a}, \texttt{a}) \xrightarrow{\texttt{choose}} \texttt{a}$$

```
pick (x, y) = if do choose () then x else y
```

$$\texttt{prog} : 1 \xrightarrow{\texttt{choose, ask}} \texttt{Int}$$

```
prog _ = pick (pick (37, 6), do ask ())
```

# Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

Traditional effect systems attach *effect types* to function arrows.

```
pick : ∀ a . (a, a) --choose--> a
pick (x, y) = if do choose () then x else y

prog : 1 --choose, ask--> Int
prog _ = pick (pick (37, 6), do ask ())

all    : ∀ a . (1 --choose--> a) → List a  # handles choose
```

## Effect Systems

The type system is usually extended with an *effect system* which statically tracks the effects that a program may use when running.

Traditional effect systems attach *effect types* to function arrows.

$$\texttt{pick} : \forall\, a\,.\,(a, a) \xrightarrow{\texttt{choose}} a$$

```
pick (x, y) = if do choose () then x else y
```

$$\texttt{prog} : 1 \xrightarrow{\texttt{choose, ask}} \texttt{Int}$$

```
prog _ = pick (pick (37, 6), do ask ())
```

$$\texttt{all} : \forall\, a\,.\,(1 \xrightarrow{\texttt{choose}} a) \to \texttt{List}\ a \quad \texttt{\# handles choose}$$

$$\texttt{answer} : \forall\, a\,.\,(1 \xrightarrow{\texttt{ask}} a) \to a \quad \texttt{\# handles ask}$$

# Effect Polymorphism

The types for `all` and `answer` are not composable!

$$\texttt{both : } \forall\ \texttt{a . (1} \xrightarrow{\texttt{ask, choose}} \texttt{a)} \to \texttt{a}$$

```
both m = all {answer m}    # {...} is short for (fun () → ...)
```

# Effect Polymorphism

The types for `all` and `answer` are not composable!

$$\texttt{both} : \forall\, a\, .\, (1 \xrightarrow{\texttt{ask, choose}} a) \to a$$

```
both m = all {answer m}    # { ... } is short for (fun () → ... )
```

❎ Type error: answer expects a function of type $1 \xrightarrow{\texttt{ask}} a$, while m has type $1 \xrightarrow{\texttt{ask, choose}} a$.

# Effect Polymorphism

The types for `all` and `answer` are not composable!

```
both : ∀ a . (1 ─ask, choose→ a) → a
both m = all {answer m}     # {...} is short for (fun () → ...)
```

❎ Type error: answer expects a function of type $1 \xrightarrow{\text{ask}} a$, while m has type $1 \xrightarrow{\text{ask, choose}} a$.

The conventional solution is *effect polymorphism*, which introduces effect variables to quantify over other potential effects.

```
answer : ∀ a e . (1 ─ask, e→ a) ─e→ a
```

# Effect Polymorphism

The types for `all` and `answer` are not composable!

$$\texttt{both} : \forall\, \texttt{a} .\ (\texttt{1} \xrightarrow{\texttt{ask, choose}} \texttt{a}) \to \texttt{a}$$

```
both m = all {answer m}    # { ... } is short for (fun () → ... )
```

❎ Type error: answer expects a function of type $\texttt{1} \xrightarrow{\texttt{ask}} \texttt{a}$, while m has type $\texttt{1} \xrightarrow{\texttt{ask, choose}} \texttt{a}$.

The conventional solution is *effect polymorphism*, which introduces effect variables to quantify over other potential effects.

$$\texttt{answer} : \forall\, \texttt{a e} .\ (\texttt{1} \xrightarrow{\texttt{ask, e}} \texttt{a}) \xrightarrow{\texttt{e}} \texttt{a}$$

Instantiating `e` with `choose` gives a compatible type

$$(\texttt{1} \xrightarrow{\texttt{ask, choose}} \texttt{a}) \xrightarrow{\texttt{choose}} \texttt{a}$$

# Effect Polymorphism

We also need to make other types effect polymorphic.

$$
\begin{array}{llll}
\texttt{pick} & : \forall\ \texttt{a}\ \texttt{e} & . & (\texttt{a},\ \texttt{a}) \xrightarrow{\texttt{choose},\ \texttt{e}} \texttt{a} \\
\texttt{prog} & : \forall\ \texttt{e} & . & 1 \xrightarrow{\texttt{choose},\ \texttt{ask},\ \texttt{e}} \texttt{Int} \\
\texttt{all} & : \forall\ \texttt{a}\ \texttt{e} & . & (1 \xrightarrow{\texttt{choose},\ \texttt{e}} \texttt{a}) \xrightarrow{\texttt{e}} \texttt{List a} \\
\texttt{answer} & : \forall\ \texttt{a}\ \texttt{e} & . & (1 \xrightarrow{\texttt{ask},\ \texttt{e}} \texttt{a}) \xrightarrow{\texttt{e}} \texttt{a} \\
\texttt{both} & : \forall\ \texttt{a}\ \texttt{e} & . & (1 \xrightarrow{\texttt{ask},\ \texttt{choose},\ \texttt{e}} \texttt{a}) \xrightarrow{\texttt{e}} \texttt{a}
\end{array}
$$

# Effect Polymorphism

We also need to make other types effect polymorphic.

$$
\begin{array}{llcl}
\texttt{pick} & : \forall\ \texttt{a e .}\ \texttt{(a, a)} \xrightarrow{\texttt{choose, e}} \texttt{a} \\
\texttt{prog} & : \forall\ \texttt{e .}\ \texttt{1} \xrightarrow{\texttt{choose, ask, e}} \texttt{Int} \\
\texttt{all} & : \forall\ \texttt{a e .}\ \texttt{(1} \xrightarrow{\texttt{choose, e}} \texttt{a)} \xrightarrow{\texttt{e}} \texttt{List a} \\
\texttt{answer} & : \forall\ \texttt{a e .}\ \texttt{(1} \xrightarrow{\texttt{ask, e}} \texttt{a)} \xrightarrow{\texttt{e}} \texttt{a} \\
\texttt{both} & : \forall\ \texttt{a e .}\ \texttt{(1} \xrightarrow{\texttt{ask, choose, e}} \texttt{a)} \xrightarrow{\texttt{e}} \texttt{a}
\end{array}
$$

Including existing “pure” functions like

$$\texttt{map} : \forall\ \texttt{a b e .}\ \texttt{(a} \xrightarrow{\texttt{e}} \texttt{b, List a)} \xrightarrow{\texttt{e}} \texttt{List b}$$

Works well with ML-style type inference via row polymorphism as in Koka and Links.

# More Examples: Generators

```
effect yield : Int ⇒ 1

                  yield, e        e
asList : ∀ e . (1 ────────→ 1) ──→ List Int
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())

                        yield, e
gen : ∀ e . List Int ────────→ 1
gen xs = map (fun x → do yield x) xs; ()

> asList (gen [3,1,4,1,5,9])
[3,1,4,1,5,9]
```

# States

```
effect get : 1   ⇒ Int
effect put : Int ⇒ 1

state  : ∀ a . (1 --get, put, e--> a) -e-> Int -e-> (a, Int)
state m = handle m () with
  return x     ⇒ fun s → (x, s)
  get    () r  ⇒ fun s → r s s
  put    s' r  ⇒ fun s → r () s'
```

# States

```
effect get : 1   ⇒ Int
effect put : Int ⇒ 1

                      get, put, e        e       e
state  : ∀ a . (1 ─────────────→ a) ─→ Int ─→ (a, Int)
state m = handle m () with
  return x     ⇒ fun s → (x, s)
  get     () r ⇒ fun s → r s s
  put     s' r ⇒ fun s → r () s'

                                  get, put, yield, e
prefixSum : ∀ e . List Int ─────────────────────→ 1
prefixSum xs = map (fun x → do put (do get + x); do yield (do get)) xs; ()
```

# States

```
effect get : 1   ⇒ Int
effect put : Int ⇒ 1

                  get, put, e        e       e
state  : ∀ a . (1 ──────────→ a) ─→ Int ─→ (a, Int)
state m = handle m () with
  return x      ⇒ fun s → (x, s)
  get     () r  ⇒ fun s → r s s
  put     s' r  ⇒ fun s → r () s'

                           get, put, yield, e
prefixSum : ∀ e . List Int ───────────────→ 1
prefixSum xs = map (fun x → do put (do get + x); do yield (do get)) xs; ()

> asList (fun () →
    state (fun () → prefixSum [3,1,4,1,5,9]) 0; ())
[3,4,8,9,14,23]
```

# Linearity and Control-Flow Linearity

# Linear Resources

*Some of the best things in life are free; and some are not.* (*Philip Wadler, A taste of linear logic, 1993*)

# Linear Resources

> *Some of the best things in life are free; and some are not. (Philip Wadler, A taste of linear logic, 1993)*

Linear resources must be used exactly once.

- File handles.
- (Session-typed) communication channels.
- Network connections.
- Memory management (affine).

# Linear Types

Linear types, derived from Girard's linear logic via Curry-Howard correspondence, ensure the linearity of certain resources statically.

# Linear Types

Linear types, derived from Girard's linear logic via Curry-Howard correspondence, ensure the linearity of certain resources statically.

```
writer : File → String ⊸ 1
writer f s = let f' = write f s in close f'
```

# Linear Types

Linear types, derived from Girard's linear logic via Curry-Howard correspondence, ensure the linearity of certain resources statically.

```
writer : File → String ⊸ 1
writer f s = let f' = write f s in close f'

writer' : File → String ⊸ 1
writer' f s = let f' = write f s in close f'; write f s
❎ Type error: f has a linear type File but is used twice.
```

# Linear Types

Linear types, derived from Girard's linear logic via Curry-Howard correspondence, ensure the linearity of certain resources statically.

```
writer : File → String ⊸ 1
writer f s = let f' = write f s in close f'

writer' : File → String ⊸ 1
writer' f s = let f' = write f s in close f'; write f s
❎ Type error: f has a linear type File but is used twice.

writer'' : File → String → 1
writer'' f s = let f' = write f s in close f'
❎ Type error: f has a linear type File but is captured in
   a non-linear function of type String → 1.
```

# (Multi-Shot) Effect Handlers Break Linear Types

$$\texttt{dubiousWriter} : \texttt{File} \xrightarrow{\texttt{Choose}} \texttt{1}$$

```
dubiousWriter f = let f' = write f (pick "6" "37") in close f'
```

# (Multi-Shot) Effect Handlers Break Linear Types

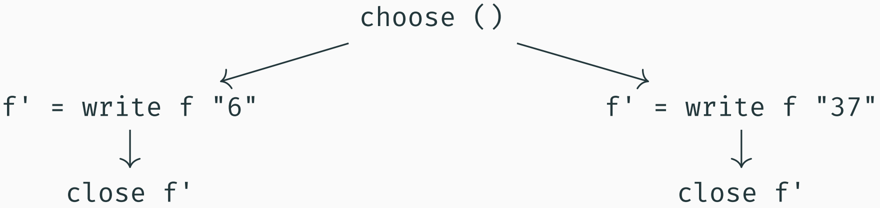
$$\texttt{dubiousWriter} : \texttt{File} \xrightarrow{\texttt{Choose}} 1$$

```
dubiousWriter f = let f' = write f (pick "6" "37") in close f'

> all (fun () → dubiousWriter (open "file.txt"))
❎ Runtime error: write to a non-existing file handle.
```

# (Multi-Shot) Effect Handlers Break Linear Types

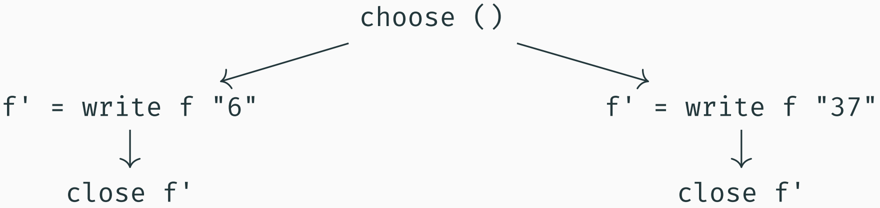
```
dubiousWriter : File —Choose→ 1
dubiousWriter f = let f' = write f (pick "6" "37") in close f'

> all (fun () → dubiousWriter (open "file.txt"))
```

❎ Runtime error: write to a non-existing file handle.



# (Multi-Shot) Effect Handlers Break Linear Types

```
dubiousWriter : File ─Choose→ 1
dubiousWriter f = let f' = write f (pick "6" "37") in close f'

> all (fun () → dubiousWriter (open "file.txt"))
❎ Runtime error: write to a non-existing file handle.
```




Conventional linear type systems only track *value linearity*; they assume continuations are used linearly. However, effect handlers enable more flexible uses of continuations.

# Tracking Control-Flow Linearity

Classify operations into two categories:

- *Control-flow-linear* operation — its continuation must be resumed exactly once.
- *Control-flow-unlimited* operation — its continuation can be resumed any times.

# Tracking Control-Flow Linearity

Classify operations into two categories:

- *Control-flow-linear* operation — its continuation must be resumed exactly once.
- *Control-flow-unlimited* operation — its continuation can be resumed any times.

$$\texttt{dubiousWriter : File} \xrightarrow{\texttt{choose:}\circ} \texttt{1}$$

```
dubiousWriter f = let f' = write f (pick "6" "37") in close f'
```

# Tracking Control-Flow Linearity

Classify operations into two categories:

- *Control-flow-linear* operation — its continuation must be resumed exactly once.
- *Control-flow-unlimited* operation — its continuation can be resumed any times.

```
dubiousWriter : File -choose:∘-> 1
dubiousWriter f = let f' = write f (pick "6" "37") in close f'

> all (fun () → dubiousWriter (open "file.txt"))
❎ Type error: choose is control-flow linear but resumed twice in all.
```

# Tracking Control-Flow Linearity

Classify operations into two categories:

- *Control-flow-linear* operation — its continuation must be resumed exactly once.
- *Control-flow-unlimited* operation — its continuation can be resumed any times.

$$\texttt{dubiousWriter} : \texttt{File} \xrightarrow{\texttt{choose}:\circ} \texttt{1}$$

```
dubiousWriter f = let f' = write f (pick "6" "37") in close f'

> all (fun () → dubiousWriter (open "file.txt"))
❎ Type error: choose is control-flow linear but resumed twice in all.

> first (fun () → dubiousWriter (open "file.txt"))
[()]  # "6" is written
```

# More in the Paper

I have omitted all details.

Two calculi which track control-flow linearity in the paper:

- $F^{\circ}_{\text{eff}}$ : A system F-style core calculus with subkinding-based linear types and row-based effect types. Requires syntactic overheads.
- $Q^{\circ}_{\text{eff}}$ : An ML-style calculus with linear and effect types both based on qualified types. Infers principal types with no extra annotation.

paper

blog post

# Modal Effect Types

# Verbosity of Conventional Effect Types

Effect polymorphism requires annotating almost all function arrows with effect variables.

$$\texttt{gen} \quad : \forall\ \texttt{e}\ .\ \texttt{List Int} \xrightarrow{\texttt{yield, e}} \texttt{1}$$

$$\texttt{asList} : \forall\ \texttt{e}\ .\ (\texttt{1} \xrightarrow{\texttt{yield, e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{List Int}$$

# Verbosity of Conventional Effect Types

Effect polymorphism requires annotating almost all function arrows with effect variables.

$$\texttt{gen} \quad : \forall\, e\, .\ \texttt{List Int} \xrightarrow{\texttt{yield},\ e} 1$$

$$\texttt{asList} : \forall\, e\, .\ (1 \xrightarrow{\texttt{yield},\ e} 1) \xrightarrow{e} \texttt{List Int}$$

Even for innocent higher-order functions which do not use or handle effects at all.

$$\texttt{map} \quad : \forall\, a\ b\ e\, .\ (a \xrightarrow{e} b,\ \texttt{List}\ a) \xrightarrow{e} \texttt{List}\ b$$

# Verbosity of Conventional Effect Types

Effect polymorphism requires annotating almost all function arrows with effect variables.

$$\texttt{gen} \quad : \forall\ e\ .\ \texttt{List Int} \xrightarrow{\texttt{yield},\ e} 1$$

$$\texttt{asList} : \forall\ e\ .\ (1 \xrightarrow{\texttt{yield},\ e} 1) \xrightarrow{e} \texttt{List Int}$$

Even for innocent higher-order functions which do not use or handle effects at all.

$$\texttt{map} \quad : \forall\ a\ b\ e\ .\ (a \xrightarrow{e} b,\ \texttt{List}\ a) \xrightarrow{e} \texttt{List}\ b$$

This verbosity severely hinders the adoption of effect systems in industrial languages: signatures of much existing library code must be rewritten no matter whether they use effects or not.

## Invisible Effect Polymorphism

Key observation of the Frank language: for higher-order functions the effect variables almost always match up because we typically use the function arguments.

# Invisible Effect Polymorphism

Key observation of the Frank language: for higher-order functions the effect variables almost always match up because we typically use the function arguments.

➡️ omit effect variables when they are the same one.

$$
\begin{aligned}
&\texttt{gen} &&: \texttt{List Int} \xrightarrow{\texttt{yield}} 1 \\
&\texttt{asList} &&: (1 \xrightarrow{\texttt{yield}} 1) \rightarrow \texttt{List Int} \\
&\texttt{map} &&: \forall\ a\ b\ e\ .\ (a \rightarrow b,\ \texttt{List}\ a) \rightarrow \texttt{List}\ b
\end{aligned}
$$

are syntactic sugar for

$$
\begin{aligned}
&\texttt{gen} &&: \forall\ e\ .\ \texttt{List Int} \xrightarrow{\texttt{yield},\ e} 1 \\
&\texttt{asList} &&: \forall\ e\ .\ (1 \xrightarrow{\texttt{yield},\ e} 1) \xrightarrow{e} \texttt{List Int} \\
&\texttt{map} &&: \forall\ a\ b\ e\ .\ (a \xrightarrow{e} b,\ \texttt{List}\ a) \xrightarrow{e} \texttt{List}\ b
\end{aligned}
$$

# Drawbacks of Using Syntactic Sugar

- Non-trivial mental programming model — programmers still need to understand effect polymorphism and reason about code by mentally desugaring.

# Drawbacks of Using Syntactic Sugar

- Non-trivial mental programming model — programmers still need to understand effect polymorphism and reason about code by mentally desugaring.
- Broken syntactic abstraction — explicit effect variables may still appear in error messages and intermediate information provided by language server protocols.

# Drawbacks of Using Syntactic Sugar

- Non-trivial mental programming model — programmers still need to understand effect polymorphism and reason about code by mentally desugaring.
- Broken syntactic abstraction — explicit effect variables may still appear in error messages and intermediate information provided by language server protocols.

A syntactical abstraction is neither satisfying from a theoretical point of view — is there a more fundamental system that captures the intuition of invisible effect polymorphism?

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

| | | |
|---|---|---|
| Variables in the context | ↔️ | Operations in the *effect context* |
| Programs can use any variables from the context | ↔️ | Programs can use any operations from the effect context |

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

| | | |
|---|---|---|
| Variables in the context | ↔️ | Operations in the *effect context* |
| Programs can use any variables from the context | ↔️ | Programs can use any operations from the effect context |

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

| | | |
|---|---|---|
| Variables in the context | ↔️ | Operations in the *effect context* |
| Programs can use any variables from the context | ↔️ | Programs can use any operations from the effect context |

```
map : ∀ a b . (a → b, List a) → List b
```

This `map` can be applied to any effectful functions. Both the parameter and result functions can use any effects from the context.

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

| | | |
|---|---|---|
| Variables in the context | ↔️ | Operations in the *effect context* |
| Programs can use any variables from the context | ↔️ | Programs can use any operations from the effect context |

```
map : ∀ a b . (a → b, List a) → List b
```

This `map` can be applied to any effectful functions. Both the parameter and result functions can use any effects from the context.

Everything still works even after currying `map`.

```
map : ∀ a b . (a → b) → List a → List b
```

# Effect Contexts

*Managing effectful computations is about managing open terms. (Leo White)*

| | | |
|---|---|---|
| Variables in the context | ↔️ | Operations in the *effect context* |
| Programs can use any variables from the context | ↔️ | Programs can use any operations from the effect context |

```
map : ∀ a b . (a → b, List a) → List b
```

This `map` can be applied to any effectful functions. Both the parameter and result functions can use any effects from the context.

Everything still works even after currying `map`.

```
map : ∀ a b . (a → b) → List a → List b
```

HOAS: use bindings of the meta-lang to encode both variable and effect bindings.

# Being in Scope ❎➡️ Being Used

Effect safety requires purity as a precondition. That is, if the effect system claims the global program does not use any effects, then there is no unhandled effect when running.

# Being in Scope ❎➡️ Being Used

Effect safety requires purity as a precondition. That is, if the effect system claims the global program does not use any effects, then there is no unhandled effect when running.

Pure programs may still introduce effects but not use them.

# Being in Scope ❎➡️ Being Used

Effect safety requires purity as a precondition. That is, if the effect system claims the global program does not use any effects, then there is no unhandled effect when running.

Pure programs may still introduce effects but not use them.

```
effect yield : Int ⇒ 1

main : 1 → Int
main () = 6 + 37
```

# Being in Scope ❎➡️ Being Used

Effect safety requires purity as a precondition. That is, if the effect system claims the global program does not use any effects, then there is no unhandled effect when running.

Pure programs may still introduce effects but not use them.

```
effect yield : Int ⇒ 1

main : 1 → Int
main () = 6 + 37
```

Typing judgements have form

$$\Gamma \vdash M : A \mathbin{@} E$$

As usual, contexts $\Gamma$ and $E$ are not visible to programmes.

# Being in Scope ❎➡️ Being Used

Effect safety requires purity as a precondition. That is, if the effect system claims the global program does not use any effects, then there is no unhandled effect when running.

Pure programs may still introduce effects but not use them.

```
effect yield : Int ⇒ 1

main : 1 → Int
main () = 6 + 37
```

Typing judgements have form

$$\Gamma \vdash M : A \mathbin{@} E$$

As usual, contexts $\Gamma$ and $E$ are not visible to programmes.

For the typing judgement of `main`, $E$ is empty.

# Absolute Modalities

How do programmers specify that `main` is pure?

# Absolute Modalities

How do programmers specify that `main` is pure?

```
main : 1 → Int
```

is shorthand for

```
main : [](1 → Int)
```

# Absolute Modalities

How do programmers specify that `main` is pure?

```
main : 1 → Int
```

is shorthand for

```
main : [](1 → Int)
```

An ***absolute modality*** specifies an effect context.

# Absolute Modalities

How do programmers specify that `main` is pure?

```
main : 1 → Int
```

is shorthand for

```
main : [](1 → Int)
```

An ***absolute modality*** specifies an effect context.

All global programs are implicitly boxed with `[]`.

```
map : ∀ a b . []((a → b, List a) → List b)

gen : [yield](List Int → 1)
gen xs = map (fun x → do yield x) xs; ()
```

# Absolute Modalities

How do programmers specify that `main` is pure?

```
main : 1 → Int
```

is shorthand for

```
main : [](1 → Int)
```

An ***absolute modality*** specifies an effect context.

All global programs are implicitly boxed with `[]`.

```
map : ∀ a b . []((a → b, List a) → List b)

gen : [yield](List Int → 1)
gen xs = map (fun x → do yield x) xs; ()
```

Similar to contextual modal types.

# Typing Handlers

Our approach so far is (more or less) a reminiscent of HOAS + contextual modal types.

# Typing Handlers

Our approach so far is (more or less) a reminiscent of HOAS + contextual modal types.

How to give types to handlers?

$$\texttt{asList} : \forall\, \texttt{e} \,.\, (\texttt{1} \xrightarrow{\texttt{yield},\ \texttt{e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{List Int}$$

```
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

# Typing Handlers

Our approach so far is (more or less) a reminiscent of HOAS + contextual modal types.

How to give types to handlers?

$$\texttt{asList : } \forall\, \texttt{e . (1} \xrightarrow{\texttt{yield, e}} \texttt{1)} \xrightarrow{\texttt{e}} \texttt{List Int}$$

```
asList m = handle m () with
  return ()   ⇒ nil
  yield  x r  ⇒ cons x (r ())
```

Effect handlers modify effect contexts! The parameter of `asList` is in a different effect context from the top-level one (extended with `yield`)!

# Typing Handlers

Our approach so far is (more or less) a reminiscent of HOAS + contextual modal types.

How to give types to handlers?

$$\texttt{asList : ∀ e . (1} \xrightarrow{\texttt{yield, e}} \texttt{1)} \xrightarrow{\texttt{e}} \texttt{List Int}$$

```
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

Effect handlers modify effect contexts! The parameter of `asList` is in a different effect context from the top-level one (extended with `yield`)!

Using absolute modalities to specify their differences would be too verbose:

```
asList : ∀ e . [e]([yield, e](1 → 1) → List Int)
```

# Relative Modalities

A ***relative modality*** specifies a local change to an effect context.

# Relative Modalities

A ***relative modality*** specifies a local change to an effect context.

```
asList : <yield>(1 → 1) → List Int
```

The *extension modality* `<yield>` extends the effect context with the `yield` operation.

# Relative Modalities

A ***relative modality*** specifies a local change to an effect context.

```
asList : <yield>(1 → 1) → List Int
```

The *extension modality* `<yield>` extends the effect context with the `yield` operation.

The parameter `<yield>(1 → 1)` can still use any other effects from the top-level context.

# Relative Modalities

A ***relative modality*** specifies a local change to an effect context.

```
asList : <yield>(1 → 1) → List Int
```

The *extension modality* `<yield>` extends the effect context with the `yield` operation.

The parameter `<yield>(1 → 1)` can still use any other effects from the top-level context.

We also have *mask modalities* `<L>` which remove effects `L` from the effect context.

## Relative Modalities

A ***relative modality*** specifies a local change to an effect context.

```
asList : <yield>(1 → 1) → List Int
```

The *extension modality* `<yield>` extends the effect context with the `yield` operation.

The parameter `<yield>(1 → 1)` can still use any other effects from the top-level context.

We also have *mask modalities* `<L>` which remove effects `L` from the effect context.

Relative modalities have the general form `<L|D>` where `L` and `D` are effects.

# Escaping Handlers

What type should we give to `answer`?

```
answer m = handle m () with
  ask () r ⇒ r 21
```

# Escaping Handlers

What type should we give to `answer`?

```
answer m = handle m () with
  ask () r ⇒ r 21
```

We cannot give type `∀ a . <ask>(1 → a) → a` to it!

```
foo : 1 → Int
foo = answer (fun _ → do ask)  # ask escapes from handler scope
> foo ()
❎ Runtime error: ask is used but not handled
```

# Escaping Handlers

What type should we give to `answer`?

```
answer m = handle m () with
  ask () r ⇒ r 21
```

We cannot give type `∀ a . <ask>(1 → a) → a` to it!

```
foo : 1 → Int
foo = answer (fun _ → do ask)  # ask escapes from handler scope
> foo ()
❎ Runtime error: ask is used but not handled
```

Instead, the typing rule always wraps the return type of handlers with extension modalities of the operations they handle.

```
answer : ∀ a . <ask>(1 → a) → <ask>a
```

# Absolute Kinds

Recall that we directly have

```
asList : <yield>(1 → 1) → List Int
```

instead of

```
asList : <yield>(1 → 1) → <yield>(List Int)
```

This is sound because the unit type cannot carry any escaped operation.

# Absolute Kinds

Recall that we directly have

```
asList : <yield>(1 → 1) → List Int
```

instead of

```
asList : <yield>(1 → 1) → <yield>(List Int)
```

This is sound because the unit type cannot carry any escaped operation.

In general we introduce a kind system with subkinding `Abs ⩽ Any` where

- values of type `A : Abs` do not rely on the ambient effect context, and
- values of type `A : Any` may use / capture effects from the ambient effect context.

# Absolute Kinds

Recall that we directly have

```
asList : <yield>(1 → 1) → List Int
```

instead of

```
asList : <yield>(1 → 1) → <yield>(List Int)
```

This is sound because the unit type cannot carry any escaped operation.

In general we introduce a kind system with subkinding `Abs ⩽ Any` where

- values of type `A : Abs` do not rely on the ambient effect context, and
- values of type `A : Any` may use / capture effects from the ambient effect context.

```
1 : Abs      Int : Abs      [yield](1 → 1) : Abs      (Bool, String) : Abs
```

# Absolute Kinds

Recall that we directly have

```
asList : <yield>(1 → 1) → List Int
```

instead of

```
asList : <yield>(1 → 1) → <yield>(List Int)
```

This is sound because the unit type cannot carry any escaped operation.

In general we introduce a kind system with subkinding `Abs ⩽ Any` where

- values of type `A : Abs` do not rely on the ambient effect context, and
- values of type `A : Any` may use / capture effects from the ambient effect context.

```
1 : Abs      Int : Abs      [yield](1 → 1) : Abs      (Bool, String) : Abs
```

Generalise to polymorphic types naturally

```
answer : ∀ [a] . <ask>(1 → a) → a  # short for ∀ a : Abs
```

# Modular Effectful Programming with Modal Effect Types

```
> asList <yield>(fun () → gen [3,1,4,1,5,9])
# [3,1,4,1,5,9] : List Int

> asList <yield>(fun () →
    state <get,put>(fun () → gen' [3,1,4,1,5,9]) 0; ())
# [3,4,8,9,14,23] : List Int
```

Unfortunately, my type inference cannot infer all modality introduction 🥲.

# Comparing with the Syntactic Sugar

For most common types they give similar results.

$$1 \xrightarrow{\texttt{choose, ask}} \texttt{Int}$$

$$\forall\ a\ .\ (1 \xrightarrow{\texttt{choose}} a) \to \texttt{List}\ a$$

$$\forall\ a\ .\ (1 \xrightarrow{\texttt{ask}} a) \to a$$

$$\forall\ a\ b\ .\ (a \to b,\ \texttt{List}\ a) \to \texttt{List}\ b$$

```
[choose, ask](1 → Int)
∀ [a] . <choose>(1 → a) → List a
∀ [a] . <ask>(1 → a) → a
∀ a b . (a → b, List a) → List b
```

# Comparing with the Syntactic Sugar

For most common types they give similar results.

| | |
|---|---|
| $1 \xrightarrow{\texttt{choose, ask}} \texttt{Int}$ | $\texttt{[choose, ask](1} \to \texttt{Int)}$ |
| $\forall\ a\ .\ (1 \xrightarrow{\texttt{choose}} a) \to \texttt{List}\ a$ | $\forall\ [a]\ .\ \langle\texttt{choose}\rangle(1 \to a) \to \texttt{List}\ a$ |
| $\forall\ a\ .\ (1 \xrightarrow{\texttt{ask}} a) \to a$ | $\forall\ [a]\ .\ \langle\texttt{ask}\rangle(1 \to a) \to a$ |
| $\forall\ a\ b\ .\ (a \to b,\ \texttt{List}\ a) \to \texttt{List}\ b$ | $\forall\ a\ b\ .\ (a \to b,\ \texttt{List}\ a) \to \texttt{List}\ b$ |

The contextual reading could give better types in some cases.

| | |
|---|---|
| $1 \xrightarrow{\texttt{choose, ask}} 1 \xrightarrow{\texttt{choose, ask}} 1$ | $\texttt{[choose, ask](1} \to 1 \to 1)$ |
| $1 \xrightarrow{\texttt{choose}} 1 \xrightarrow{\texttt{choose, ask}} 1$ | $\texttt{[choose](1} \to \langle\texttt{ask}\rangle(1 \to 1))$ |
| $\forall\ f\ .\ (\forall\ e\ .\ (1 \xrightarrow{\texttt{ask, e}} 1) \xrightarrow{e} 1) \xrightarrow{f} 1$ | $[\,](\langle\texttt{ask}\rangle(1 \to 1) \to 1) \to 1$ |

Check out our preprint for a formal compositional encoding from left to right.

# One More Example: Re-generating

Process all generated numbers with a function.

```
regen : [yield]((Int → Int) → <yield>(1 → 1) → 1)
regen f m = handle m () with
  return ()  ⇒ ()
  yield  s r ⇒ do yield (f s); r ()
```

# One More Example: Re-generating

Process all generated numbers with a function.

```
regen : [yield]((Int → Int) → <yield>(1 → 1) → 1)
regen f m = handle m () with
  return ()  ⇒ ()
  yield  s r ⇒ do yield (f s); r ()
```

In contrast, conventional effect systems (e.g., the one used in Koka) usually give

$$\texttt{regen} : \forall\ e\ .\ (\texttt{Int} \xrightarrow{\texttt{yield},\ e} \texttt{Int}) \xrightarrow{e} (\texttt{1} \xrightarrow{\texttt{yield},\ \texttt{yield},\ e} \texttt{1}) \xrightarrow{\texttt{yield},\ e} \texttt{1}$$

# Cooperative Concurrency

```
data Proc = proc (List Proc → ())

push : ∀ a . a → List a → List a
push x xs = xs ++ cons x nil
```

```
next : List Proc → ()
next q = case q of
  nil              → ()
  cons (proc p) ps → p ps
```

```
schedule : <ufork, suspend>(1 → 1) → List Proc → 1
schedule m = handle m () with
  return  ()   ⇒ fun q → next q
  suspend () r ⇒ fun q → next (push (proc (r ())) q)
  ufork   () r ⇒ fun q → r true (push (proc (r false)) q)
```

# Theoretical Foundation: (Simply) Multimodal Type Theory

Modal effect types have a solid theoretical foundation based on (the simply-typed fragment) of multimodal type theory, a dependent type theory parameterised by a mode theory, which specifies the structure of modes, modalities, and their transformations.

# More in the Paper

**MET:** A core calculus following simple *multimodal type theory*.

*Encoding* a fragment of conventional effect types into MET

**METE:** Extension with *effect variables*.

**METEL:** A surface language with *sound and complete type inference*.