Effect typing

Sam Lindley

The University of Edinburgh

FoPSS 2026

Effect polymorphism

Flexible composition of effectful programs is supported by **effect polymorphism**.

Effect polymorphism

Flexible composition of effectful programs is supported by **effect polymorphism**.

Example: choice and failure

$$\mathsf{Choose} = \mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool} \qquad \mathsf{Fail} = \mathsf{fail} : c.1 \twoheadrightarrow c$$

$$\begin{array}{l}
\mathsf{maybeFail} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Fail}\} \uplus e) \to \mathsf{Maybe}\ a \,!\, e \\
\mathsf{maybeFail}\ m = \\
\quad \textbf{handle}\ m\,()\ \textbf{with} \\
\qquad \textbf{return}\ x \ \mapsto\ \mathsf{Just}\ x \\
\qquad \langle \mathsf{fail}\,() \rangle \ \mapsto\ \mathsf{Nothing}
\end{array}$$

$$\begin{array}{l}
\mathsf{allChoices} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Choose}\} \uplus e) \to \mathsf{List}\ a \,!\, e \\
\mathsf{allChoices}\ m = \\
\quad \textbf{handle}\ m\,()\ \textbf{with} \\
\qquad \textbf{return}\ x \ \mapsto\ [x] \\
\qquad \langle \mathsf{choose}\,() \to r \rangle \ \mapsto\ r\,\mathsf{tt} +\!\!+ r\,\mathsf{ff}
\end{array}$$

Effect polymorphism

Flexible composition of effectful programs is supported by **effect polymorphism**.

Example: choice and failure

$$\mathsf{Choose} = \mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool} \qquad \mathsf{Fail} = \mathsf{fail} : c.1 \twoheadrightarrow c$$

$$\mathsf{maybeFail} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Fail}\} \uplus e) \to \mathsf{Maybe}\ a \,!\, e$$
$$\mathsf{allChoices} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Choose}\} \uplus e) \to \mathsf{List}\ a \,!\, e$$

$$\mathsf{Tosses} = \mathsf{List}\ \mathsf{Toss}$$
$$\mathsf{drunkTosses} : \mathsf{Nat} \to \mathsf{Tosses} \,!\, \{\mathsf{Choice}, \mathsf{Failure}\}$$

Effect polymorphism

Flexible composition of effectful programs is supported by **effect polymorphism**.

Example: choice and failure

$$\mathsf{Choose} = \mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool} \qquad \mathsf{Fail} = \mathsf{fail} : c.1 \twoheadrightarrow c$$

$$\begin{aligned}
&\mathsf{maybeFail} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Fail}\} \uplus e) \to \mathsf{Maybe}\ a \,!\, e \\
&\mathsf{allChoices} : \forall a\, e.(1 \to a \,!\, \{\mathsf{Choose}\} \uplus e) \to \mathsf{List}\ a \,!\, e
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Tosses} = \mathsf{List}\ \mathsf{Toss} \\
&\mathsf{drunkTosses} : \mathsf{Nat} \to \mathsf{Tosses} \,!\, \{\mathsf{Choice}, \mathsf{Failure}\}
\end{aligned}$$

With explicit type applications we may write:

$$\mathsf{allChoices}\,(\mathsf{Maybe}\ \mathsf{Tosses})\,\emptyset\,(\lambda().\mathsf{maybeFail}\ \mathsf{Tosses}\ \{\mathsf{Choose}\}\ (\lambda().\mathsf{drunkTosses}\ 2))$$

or

$$\mathsf{maybeFail}\,(\mathsf{List}\ \mathsf{Tosses})\,\emptyset\,(\lambda().\mathsf{allChoices}\ \mathsf{Tosses}\ \{\mathsf{Fail}\}\ (\lambda().\mathsf{drunkTosses}\ 2))$$

Effect polymorphism via row polymorphism

Intuitively, a row type is a type-level map from labels to types $\ell_1 : A_1, ..., \ell : A_n$

Effect polymorphism via row polymorphism

Intuitively, a row type is a type-level map from labels to types $\ell_1 : A_1, ..., \ell : A_n$

Row polymorphism supports abstracting over *the rest* of a row type:
$\ell_1 : A_1, ..., \ell_n : A_n; \rho$

There can be at most one row variable in a row type

Effect polymorphism via row polymorphism

Intuitively, a row type is a type-level map from labels to types $\ell_1 : A_1, ..., \ell : A_n$

Row polymorphism supports abstracting over *the rest* of a row type:
$\ell_1 : A_1, ..., \ell_n : A_n; \rho$

There can be at most one row variable in a row type

Originally designed for polymorphic record typing [Wand, LICS 1989]

Row polymorphism also works nicely for polymorphic variants and **effect polymorphism**

Effect polymorphism via row polymorphism

Intuitively, a row type is a type-level map from labels to types $\ell_1 : A_1, ..., \ell : A_n$

Row polymorphism supports abstracting over *the rest* of a row type:
$\ell_1 : A_1, ..., \ell_n : A_n; \rho$

There can be at most one row variable in a row type

Originally designed for polymorphic record typing [Wand, LICS 1989]

Row polymorphism also works nicely for polymorphic variants and **effect polymorphism**

For effect handlers labels are either operation names or effect names

Rémy-style row polymorphism

Rows as maps from labels to type-level maybes — each label is either present with type A (Pre(A), abbreviated as A) or absent (Abs)

Duplicate labels disallowed

Rémy-style row polymorphism

Rows as maps from labels to type-level maybes — each label is either present with type A (Pre(A), abbreviated as A) or absent (Abs)

Duplicate labels disallowed

Example:

$$\begin{aligned}
&\text{maybeFail} : \forall(a : \text{Type}), (e : \text{Row}_{\{\text{fail}\}}), (p : \text{Presence}).\\
&\qquad (1 \to a \,!\, \{\text{fail} : (c : \text{Type}).1 \twoheadrightarrow c; e\}) \to \text{Maybe}\ a \,!\, \{\text{fail} : p; e\}\\
&\text{allChoices} : \forall(a : \text{Type}), (e : \text{Row}_{\{\text{choose}\}}), (p : \text{Presence}).\\
&\qquad (1 \to a \,!\, \{\text{choose} : 1 \twoheadrightarrow \text{Bool}; e\}) \to \text{List}\ a \,!\, \{\text{choose} : p; e\}
\end{aligned}$$

Leijen-style row polymorphism

Rows as maps from labels to type-level lists — each label may be present multiple times at different types

Duplicate labels allowed; order of duplicates matters

Leijen-style row polymorphism

Rows as maps from labels to type-level lists — each label may be present multiple times at different types

Duplicate labels allowed; order of duplicates matters

Example:

$$\begin{aligned}\text{maybeFail} &: \forall(a : \text{Type}), (e : \text{Row}).\\ &\qquad (1 \to a \,!\, \{\text{Fail}; e\}) \to \text{Maybe}\ a \,!\, e\\ \text{allChoices} &: \forall(a : \text{Type}), (e : \text{Row}).\\ &\qquad (1 \to a \,!\, \{\text{Choose}; e\}) \to \text{List}\ a \,!\, e\end{aligned}$$

Handler composition with row polymorphism

Instantiating an effect variable supports handler composition

Handler composition with row polymorphism

Instantiating an effect variable supports handler composition

Rémy style (explicit instantiation):

$$\begin{array}{l}\mathsf{allChoices}\,(\mathsf{Maybe}\,\mathsf{Tosses})\,\emptyset\,\mathsf{Abs}\,(\lambda().\\ \quad \mathsf{maybeFail}\,\mathsf{Tosses}\,\{\mathsf{Choose}\}\,\mathsf{Abs}\,(\lambda().\\ \quad\quad \mathsf{drunkTosses}\,2))\end{array}$$

Handler composition with row polymorphism

Instantiating an effect variable supports handler composition

Rémy style (explicit instantiation):

$$\begin{array}{l}\mathsf{allChoices}\,(\mathsf{Maybe}\,\mathsf{Tosses})\,\emptyset\,\mathsf{Abs}\,(\lambda().\\ \quad \mathsf{maybeFail}\,\mathsf{Tosses}\,\{\mathsf{Choose}\}\,\mathsf{Abs}\,(\lambda().\\ \quad\quad \mathsf{drunkTosses}\,2))\end{array}$$

Leijen style (explicit instantiation):

$$\begin{array}{l}\mathsf{allChoices}\,(\mathsf{Maybe}\,\mathsf{Tosses})\,\emptyset\,(\lambda().\\ \quad \mathsf{maybeFail}\,\mathsf{Tosses}\,\{\mathsf{Choose}\}\,(\lambda().\\ \quad\quad \mathsf{drunkTosses}\,2))\end{array}$$

Example: abstracting over an exception handler

Rémy style:

$$\begin{aligned}
&\mathsf{catch} : (1 \to a\,!\,\{\mathsf{fail} : c.1 \twoheadrightarrow c; e\}) \times (1 \to a\,!\,\{\mathsf{fail} : p; e\}) \to a\,!\,\{\mathsf{fail} : p; e\} \\
&\mathsf{catch}\,(m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad \langle \mathsf{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

Example: abstracting over an exception handler

Rémy style:

$$\begin{aligned}
&\mathsf{catch} : (1 \to a\,!\,\{\mathsf{fail} : c.1 \twoheadrightarrow c; e\}) \times (1 \to a\,!\,\{\mathsf{fail} : p; e\}) \to a\,!\,\{\mathsf{fail} : p; e\} \\
&\mathsf{catch}\,(m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad \langle \mathsf{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

If h can itself fail then p is instantiated to Pre(a.1 ↠ a)

Example: abstracting over an exception handler

Rémy style:

$$\begin{aligned}
&\mathsf{catch} : (1 \to a \,!\, \{\mathsf{fail} : c.1 \twoheadrightarrow c; e\}) \times (1 \to a \,!\, \{\mathsf{fail} : p; e\}) \to a \,!\, \{\mathsf{fail} : p; e\} \\
&\mathsf{catch}\ (m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad \langle \mathsf{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

If h can itself fail then p is instantiated to Pre(a.1 ↠ a)

Leijen style:

$$\begin{aligned}
&\mathsf{catch} : (1 \to a \,!\, \{\mathsf{Fail}; e\}) \times (1 \to a \,!\, e) \to a \,!\, e \\
&\mathsf{catch}\ (m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad\qquad \langle \mathsf{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

Example: abstracting over an exception handler

Rémy style:

$$\begin{aligned}
&\text{catch} : (1 \to a \,!\, \{\text{fail} : c.1 \twoheadrightarrow c; e\}) \times (1 \to a \,!\, \{\text{fail} : p; e\}) \to a \,!\, \{\text{fail} : p; e\} \\
&\text{catch}\ (m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad \langle \text{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

If h can itself fail then p is instantiated to Pre(a.1 ↠ a)

Leijen style:

$$\begin{aligned}
&\text{catch} : (1 \to a \,!\, \{\text{Fail}; e\}) \times (1 \to a \,!\, e) \to a \,!\, e \\
&\text{catch}\ (m, h) = \textbf{handle}\ m\,()\ \textbf{with} \\
&\qquad\qquad \textbf{return}\ x \mapsto x \\
&\qquad\qquad \langle \text{fail}\ () \rangle \ \mapsto h\,()
\end{aligned}$$

If h can itself fail then e is instantiated to $\{\text{Fail}; e'\}$ for some e', which means the type of m is $(1 \to b \,!\, \{\text{Fail}, \text{Fail}; e'\})$

Example: effect pollution

Effects

$$\mathsf{Get} = \mathsf{get} : 1 \twoheadrightarrow \mathsf{Nat} \qquad \mathsf{Fail} = \mathsf{fail} : c.1 \twoheadrightarrow c$$

Handlers

$$\mathsf{reads} : \mathsf{List}\ \mathsf{Nat} \times (1 \to a\ !\ \{\mathsf{Get}; e\}) \to a\ !\ \{\mathsf{Fail}; e\}$$

$$\begin{array}{l}\mathsf{reads}\,([], m) = \\ \quad \mathbf{handle}^{\dagger}\ m\,()\ \mathbf{with} \\ \qquad \mathbf{return}\ x \quad \mapsto x \\ \qquad \langle \mathsf{get}\,() \to r \rangle \mapsto \mathsf{fail}\,()\end{array} \qquad \begin{array}{l}\mathsf{reads}\,(n :: ns, m) = \\ \quad \mathbf{handle}^{\dagger}\ m\,()\ \mathbf{with} \\ \qquad \mathbf{return}\ x \quad \mapsto x \\ \qquad \langle \mathsf{get}\,() \to r \rangle \mapsto \mathsf{reads}\,(ns, \lambda().r\ n)\end{array}$$

$$\begin{array}{l}\mathsf{maybeFail} : (1 \to a\ !\ \{\mathsf{Fail}; e\}) \to \mathsf{Maybe}\ a\ !\ e \\ \mathsf{maybeFail}\ m = \\ \quad \mathbf{handle}^{\dagger}\ m\,()\ \mathbf{with} \\ \qquad \mathbf{return}\ x \quad \mapsto \mathsf{Just}\ x \\ \qquad \langle \mathsf{fail}\,() \to r \rangle \mapsto \mathsf{Nothing}\end{array}$$

Example: effect pollution

$$\text{bad} : \text{List}\ a \times (1 \to a\ !\ \{\text{Get}, \text{Fail}; e\}) \to \text{Maybe}\ a\ !\ e$$
$$\text{bad}\ (ns, f) = \text{maybeFail}\ (\lambda().\text{reads}\ (ns, f))$$

Example: effect pollution

$$\begin{aligned}&\mathsf{bad} : \mathsf{List}\ a \times (1 \to a\ !\ \{\mathsf{Get}, \mathsf{Fail}; e\}) \to \mathsf{Maybe}\ a\ !\ e\\&\mathsf{bad}\,(ns, f) = \mathsf{maybeFail}\,(\lambda().\mathsf{reads}\,(ns, f))\end{aligned}$$

$$\mathsf{bad}\,([1,2], \lambda().\mathsf{get}\,() + \mathsf{fail}\,()) : \mathsf{Maybe\ Nat} \Longrightarrow \mathsf{Nothing}$$

Example: effect pollution

$$\begin{aligned}&\mathsf{bad} : \mathsf{List}\ a \times (1 \to a\ !\ \{\mathsf{Get}, \mathsf{Fail}; e\}) \to \mathsf{Maybe}\ a\ !\ e\\&\mathsf{bad}\,(ns, f) = \mathsf{maybeFail}\,(\lambda().\mathsf{reads}\,(ns, f))\end{aligned}$$

$$\mathsf{bad}\,([1,2], \lambda().\mathsf{get}\,() + \mathsf{fail}\,()) : \mathsf{Maybe\ Nat} \Longrightarrow \mathsf{Nothing}$$

How can we encapsulate the use of fail as an intermediate effect?

Example: effect pollution

$$\begin{aligned}&\mathsf{bad} : \mathsf{List}\ a \times (1 \to a\ !\ \{\mathsf{Get}, \mathsf{Fail}; e\}) \to \mathsf{Maybe}\ a\ !\ e\\&\mathsf{bad}\ (ns, f) = \mathsf{maybeFail}\ (\lambda().\mathsf{reads}\ (ns, f))\end{aligned}$$

$$\mathsf{bad}\ ([1,2], \lambda().\mathsf{get}\ () + \mathsf{fail}\ ()) : \mathsf{Maybe}\ \mathsf{Nat} \Longrightarrow \mathsf{Nothing}$$

How can we encapsulate the use of fail as an intermediate effect?

The aim is to define

$$\mathsf{good} : \mathsf{List}\ a \times (1 \to a\ !\ \{\mathsf{Get}; e\}) \to \mathsf{Maybe}\ a\ !\ e$$

by composing reads and maybeFail such that

$$\mathsf{good}\ ([1,2], \lambda().\mathsf{get}\ () + \mathsf{fail}\ ()) : \mathsf{Maybe}\ \mathsf{Nat}\ !\ \{\mathsf{Fail}; e\}$$

performs the fail operation.

Effect encapsulation

Two solutions to the effect pollution problem:

- Mask the intermediate effect (only works for Leijen-style row-typing)

$$\begin{aligned} &\text{good} : \text{List}\ a \times (1 \to a\ !\ \{\text{Get}; e\}) \to \text{Maybe}\ a\ !\ e \\ &\text{good}\,(ns, m) = \text{maybeFail}(\lambda().\text{reads}\,(ns, \lambda().\langle\text{fail}\rangle(m\,()))) \end{aligned}$$

  Frank, Koka, and Helium support this approach.
  [Biernacki, Piróg, Polesiuk, Sieczkowski, POPL 2018, "Handle with care"]
  [Convent, Lindley, McBride, McLaughlin, JFP 2019, "Doo bee doo bee doo"]

- Add support for fresh effects

  Helium and Links support this approach.
  [Biernacki, Piróg, Polesiuk, Sieczkowski, POPL 2019, "Abstracting algebraic effects"]

Effect masking

$$\frac{\Gamma \vdash M : A \,!\, \{R\}}{\Gamma \vdash \langle \mathsf{op} \rangle\ M : A \,!\, \{\mathsf{op} : B \twoheadrightarrow C; R\}}$$

A form of **weakening** for effects

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical

(they typically **use** the function arguments)

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Leijen style

$$\text{map} : (a \to b \,!\, e) \times \text{List}\ a \to \text{List}\ b \,!\, e$$
$$\text{catch} : (1 \to b \,!\, \{\text{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e$$
$$\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}; e\}) \to a \,!\, \{\text{Fail}; e\}$$

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Leijen style

$$\text{map} : (a \to b \,!\, e) \times \text{List}\ a \to \text{List}\ b \,!\, e$$
$$\text{catch} : (1 \to b \,!\, \{\text{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e$$
$$\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}; e\}) \to a \,!\, \{\text{Fail}; e\}$$

Do we really need all of these effect variables?

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Leijen style

$$\text{map} : (a \to b \,!\, e) \times \text{List}\ a \to \text{List}\ b \,!\, e$$
$$\text{catch} : (1 \to b \,!\, \{\text{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e$$
$$\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}; e\}) \to a \,!\, \{\text{Fail}; e\}$$

Do we really need all of these effect variables?

The **eee** problem [Martin Odersky, Vianna, March 2026]

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Leijen style

$$\begin{aligned}
&\text{map} : (a \to b \,!\, e) \times \text{List}\ a \to \text{List}\ b \,!\, e \\
&\text{catch} : (1 \to b \,!\, \{\text{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e \\
&\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}; e\}) \to a \,!\, \{\text{Fail}; e\}
\end{aligned}$$

Do we really need all of these effect variables? No!

The **eee** problem [Martin Odersky, Vianna, March 2026]

Syntactic sugar — omit all effect variables as they are all the same

$$\begin{aligned}
&\text{map} : ((a \to b \,!\, \{\}) \times \text{List}\ a) \to \text{List}\ b \,!\, \{\} \\
&\text{catch} : (1 \to b \,!\, \{\text{Fail}\}) \times (1 \to b \,!\, \{\}) \to b \,!\, \{\} \\
&\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}\}) \to a \,!\, \{\text{Fail}\}
\end{aligned}$$

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Syntactic sugar — omit all effect variables as they are all the same

$$\begin{aligned}
&\mathsf{map} : ((a \to b \,!\, \{\}) \times \mathsf{List}\ a) \to \mathsf{List}\ b \,!\, \{\} \\
&\mathsf{catch} : (1 \to b \,!\, \{\mathsf{Fail}\}) \times (1 \to b \,!\, \{\}) \to b \,!\, \{\} \\
&\mathsf{reads} : \mathsf{List}\ \mathsf{Nat} \times (1 \to a \,!\, \{\mathsf{Get}\}) \to a \,!\, \{\mathsf{Fail}\}
\end{aligned}$$

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Syntactic sugar — omit all effect variables as they are all the same

$$\begin{aligned}
&\text{map} : ((a \to b\,!\,\{\}) \times \text{List}\ a) \to \text{List}\ b\,!\,\{\} \\
&\text{catch} : (1 \to b\,!\,\{\text{Fail}\}) \times (1 \to b\,!\,\{\}) \to b\,!\,\{\} \\
&\text{reads} : \text{List Nat} \times (1 \to a\,!\,\{\text{Get}\}) \to a\,!\,\{\text{Fail}\}
\end{aligned}$$

Empty polymorphic effects need not be written at all

$$\begin{aligned}
&\text{map} : ((a \to b) \times \text{List}\ a) \to \text{List}\ b \\
&\text{catch} : (1 \to b\,!\,\{\text{Fail}\}) \times (1 \to b) \to b \\
&\text{reads} : \text{List Nat} \times (1 \to a\,!\,\{\text{Get}\}) \to a\,!\,\{\text{Fail}\}
\end{aligned}$$

Invisible effect polymorphism

Key insight: higher-order function effect variables are almost always identical
(they typically **use** the function arguments)

Syntactic sugar — omit all effect variables as they are all the same

$$\begin{aligned}
&\text{map} : ((a \to b \,!\, \{\}) \times \text{List}\ a) \to \text{List}\ b \,!\, \{\} \\
&\text{catch} : (1 \to b \,!\, \{\text{Fail}\}) \times (1 \to b \,!\, \{\}) \to b \,!\, \{\} \\
&\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}\}) \to a \,!\, \{\text{Fail}\}
\end{aligned}$$

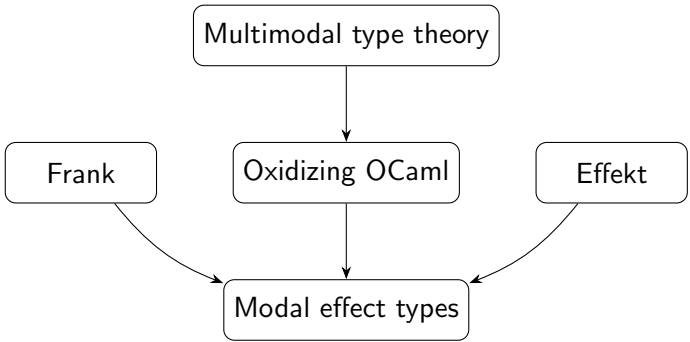
Empty polymorphic effects need not be written at all

$$\begin{aligned}
&\text{map} : ((a \to b) \times \text{List}\ a) \to \text{List}\ b \\
&\text{catch} : (1 \to b \,!\, \{\text{Fail}\}) \times (1 \to b) \to b \\
&\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}\}) \to a \,!\, \{\text{Fail}\}
\end{aligned}$$

Need explicit syntax for closed rows ($\emptyset$), but uncommon with row polymorphism

Modal effect types

Inspiration



Key ideas

- **decouple** effect typing from function arrows
- track effects through an **ambient effect context**
- use **modalities** to modify the ambient effect context locally

Modal effect types

Leijen style

$$\text{map} : (a \to b \,!\, e) \times \text{List}\ a \to \text{List}\ b \,!\, e$$
$$\text{catch} : (1 \to b \,!\, \{\text{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e$$
$$\text{reads} : \text{List Nat} \times (1 \to a \,!\, \{\text{Get}; e\}) \to a \,!\, \{\text{Fail}; e\}$$

Modal effect types

Leijen style

$$\begin{aligned}
&\mathsf{map} : (a \to b \,!\, e) \times \mathsf{List}\ a \to \mathsf{List}\ b \,!\, e \\
&\mathsf{catch} : (1 \to b \,!\, \{\mathsf{Fail}; e\}) \times (1 \to b \,!\, e) \to b \,!\, e \\
&\mathsf{reads} : \mathsf{List\ Nat} \times (1 \to a \,!\, \{\mathsf{Get}; e\}) \to a \,!\, \{\mathsf{Fail}; e\}
\end{aligned}$$

Modal effect types

$$\begin{aligned}
&\mathsf{map} : [](((a \to b) \times \mathsf{List}\ a) \to \mathsf{List}\ b) \\
&\mathsf{catch} : [](\langle\mathsf{Fail}\rangle(1 \to b) \times (1 \to b) \to b) \\
&\mathsf{reads} : [\mathsf{Fail}](\mathsf{List\ Nat} \times \langle\mathsf{Get}\rangle(1 \to a) \to a)
\end{aligned}$$

Top-level pure modalities need not be written at all

$$\begin{aligned}
&\mathsf{map} : ((a \to b) \times \mathsf{List}\ a) \to \mathsf{List}\ b \\
&\mathsf{catch} : \langle\mathsf{Fail}\rangle(1 \to b) \times (1 \to b) \to b \\
&\mathsf{reads} : [\mathsf{Fail}](\mathsf{List\ Nat} \times \langle\mathsf{Get}\rangle(1 \to a) \to a)
\end{aligned}$$

Modal effect types address the **eee** problem with no effect polymorphism at all!

Modal effect types references

Modal effect types
Tang, White, Dolan, Hillerström, Lindley, Lorenzen; OOPSLA 2025

Rows and capabilities as modal effects
Tang, Lindley; POPL 2026

Prototype implementation
Boussa, Tang, Lindley; Edinburgh 2026
(https://github.com/4y8/modal-effect-types)

MET fundamentals

Types	$A ::= 1 \mid A \times B \mid A \to B \mid \mu A \mid \ldots$
Signatures	$\Sigma ::= \cdot \mid \ell : A \twoheadrightarrow B$
Modalities	$\mu ::= [E] \mid \langle D \rangle$
Effect contexts	$D, E ::= \cdot \mid \ell, E$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x :_\mu A \mid \Gamma, \blacksquare_\mu$

Applying and composing modalities

Application

$$[E](F) = E$$
$$\langle D \rangle(F) = D, F$$

Composition

$$\mu \circ [E] = [E] \qquad [E] \circ \langle D \rangle = [D, E] \qquad \langle D_1 \rangle \circ \langle D_2 \rangle = \langle D_2, D_1 \rangle$$

$$(\mu \circ \nu)(E) = \nu(\mu(E))$$

Contexts identified up to elimination of trivial locks and composition of adjacent locks

$$\Gamma, \blacksquare_{\langle\rangle}, \Gamma' = \Gamma, \Gamma'$$
$$\Gamma, \blacksquare_{\mu}, \blacksquare_{\nu}, \Gamma' = \Gamma, \blacksquare_{\mu \circ \nu}, \Gamma'$$

Locks determine available effects

$$
\begin{aligned}
\mathsf{locks} &: \mathsf{Ctx} \to \mathsf{Modality} \\
\mathsf{locks}(\cdot) &= \langle\rangle \\
\mathsf{locks}(\Gamma, x :_\mu A) &= \mathsf{locks}(\Gamma) \\
\mathsf{locks}(\Gamma, \text{🔒}_\mu) &= \mathsf{locks}(\Gamma) \circ \mu
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{effects} &: \mathsf{Ctx} \to \mathsf{EffCtx} \\
\mathsf{effects}(\Gamma) &= (\mathsf{locks}(\Gamma))(\cdot)
\end{aligned}
$$

Typing rules

$$\textsc{T-Var}\quad \frac{\vdash_A \mu \Rightarrow \mathsf{locks}(\Gamma') \mathbin{@} \mathsf{effects}(\Gamma)}{\Gamma, x :_\mu A, \Gamma' \vdash x : A}$$

$$\textsc{T-Abs}\quad \frac{\Gamma, x :_{\langle\rangle} A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B}$$

$$\textsc{T-App}\quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\textsc{T-Mod}\quad \frac{\Gamma, \blacksquare_\mu \vdash V : A}{\Gamma \vdash \mathbf{mod}_\mu V : \mu A}$$

$$\textsc{T-LetMod}\quad \frac{\Gamma \vdash V : \mu A \qquad \Gamma, x :_\mu A \vdash M : B}{\Gamma \vdash \mathbf{let\ mod}_\mu\ x = V\ \mathbf{in}\ M : B}$$

Typing rules

T-Do

$$\frac{\Sigma \ni \ell : A \twoheadrightarrow B \qquad \mathsf{effects}(\Gamma) = \ell, E \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{do}\ \ell\ N : B}$$

T-Handle

$$\frac{\begin{array}{c}\Sigma \ni \ell : A' \twoheadrightarrow B' \\ \Gamma, \blacksquare_{\langle \ell \rangle} \vdash M : A \\ \Gamma, x :_{\langle \ell \rangle} A \vdash N : B \\ \Gamma, p :_{\langle\rangle} A', r :_{\langle\rangle} B' \to B \vdash N' : B\end{array}}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto N;\ \ell\ p\ r \mapsto N'\} : B}$$

Kinding rules

$$\frac{}{\vdash 1 : \mathsf{Abs}} \qquad \frac{\vdash A_1 : K_1 \qquad \vdash A_2 : K_2}{\vdash A_1 \times A_2 : K_1 \sqcup K_2} \qquad \frac{\vdash A_1 : K_1 \qquad \vdash A_2 : K_2}{\vdash A_1 \to A_2 : \mathsf{Any}}$$

$$\frac{\vdash A : K}{\vdash [E]A : \mathsf{Abs}} \qquad \frac{\vdash A : K}{\vdash \langle D \rangle A : \mathsf{Any}}$$

Subkinding

$$\mathsf{Abs} \leq \mathsf{Any}$$

A type has absolute kind iff all function types or relative modalities are guarded by an absolute modality.

Promoting modalities

$$\frac{E, E' = \mu(F)}{\vdash_A [E] \Rightarrow \mu \mathbin{@} F} \qquad \frac{\vdash A : \mathsf{Abs}}{\vdash_A \mu \Rightarrow \nu \mathbin{@} E}$$