Effect typing

Sam Lindley

The University of Edinburgh

OPLSS 2022
Effect polymorphism

To support flexible composition of effectful programs we need **effect polymorphism**.
Effect polymorphism

To support flexible composition of effectful programs we need **effect polymorphism**.

Example: choice and failure

\[
\text{maybeFail} : \forall e. A!(e \uplus \{\text{fail : } a.1 \rightarrow a\}) \Rightarrow \text{Maybe } A!e
\]

\[
\text{allChoices} : \forall e. A!(e \uplus \{\text{choose : } 1 \rightarrow \text{Bool}\}) \Rightarrow \text{List } A!e
\]
Effect polymorphism

To support flexible composition of effectful programs we need **effect polymorphism**.

Example: choice and failure

\[
\text{maybeFail} : \forall e. A!(e \uplus \{\text{fail} : a.1 \to a\}) \Rightarrow \text{Maybe } A!e
\]

\[
\text{allChoices} : \forall e. A!(e \uplus \{\text{choose} : 1 \to \text{Bool}\}) \Rightarrow \text{List } A!e
\]

With explicit type applications we may write:

\[
\text{handle (handle drunkTosses 2 with maybeFail \{choose : 1 \to \text{Bool}\}) with allChoices } \emptyset
\]

or

\[
\text{handle (handle drunkTosses 2 with allChoices \{fail : a.1 \to a\}) with maybeFail } \emptyset
\]
Effect polymorphism via row polymorphism

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

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

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 value types $\ell_1 : A_1, \ldots, \ell : A_n$

Row polymorphism supports abstracting over the rest of a row type: $\ell : A_1, \ldots, \ell : A_n; \rho$

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

Originally row polymorphism was 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 value types $\ell_1 : A_1, \ldots, \ell : A_n$

Row polymorphism supports abstracting over the rest of a row type: $\ell : A_1, \ldots, \ell : A_n; \rho$

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

Originally row polymorphism was 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 \( A \) (\( \text{Pre}(A) \)) or absent (\( \text{Abs} \))

Duplicate labels disallowed
Rémy-style row polymorphism

Rows as maps from labels to type-level maybes — each label is either present with type $A$ ($\text{Pre}(A)$) or absent ($\text{Abs}$)

Duplicate labels disallowed

Example:

\[
\text{maybeFail} : \forall (e : \text{Row}_{\{\text{fail}\}}, (p : \text{Presence}).
A!(\{\text{fail} : (a : \text{Type}).1 \rightarrow a; e\}) \Rightarrow \text{Maybe } A!\{\text{fail} : p; e\}
\]

allChoices : \forall (e : \text{Row}_{\{\text{choose}\}}, (p : \text{Presence}).
A!(e \uplus \{\text{choose} : 1 \rightarrow \text{Bool}; e\}) \Rightarrow \text{List } A!\{\text{choose} : p; e\}
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:

\[
\text{maybeFail} : \forall (e : \text{Row}). \quad A!\left(\{\text{fail} : (a : \text{Type}).1 \to a; e\}\right) \Rightarrow \text{Maybe} \ A\!\{; e\}
\]

\[
\text{allChoices} : \forall (e : \text{Row}). \quad A!(e \uplus \{\text{choose} : 1 \to \text{Bool}; e\}) \Rightarrow \text{List} \ A\!\{; e\}
\]
Handler composition with row polymorphism

Instantiating an effect variable supports handler composition
Instantiating an effect variable supports handler composition

Rémy style (explicit instantiation):

```haskell
handle (handle drunkTosses 2 with maybeFail {choose : 1 ↠ Bool} Abs) with allChoices ∅ Abs
```
Handler composition with row polymorphism

Instantiating an effect variable supports handler composition

Rémy style (explicit instantiation):

```
handle (handle drunkTosses 2 with maybeFail {choose : 1 \rightarrow Bool} Abs)
with allChoices ∅ Abs
```

Leijen style (explicit instantiation):

```
handle (handle drunkTosses 2 with maybeFail {choose : 1 \rightarrow Bool})
with allChoices ∅
```
Example: abstracting over an exception handler

Rémy style:

catch : (1 → b!{fail : a.1 ↠ a; e}) → (1 → b!{fail : p; e}) → b!{fail : p; e}
catch \ m \ h = handle \ m() \ with
    \ return \ x ↦ x
    \ ⟨fail ()⟩ ↦ h ()
Example: abstracting over an exception handler

Rémy style:

\[
\text{catch : } (1 \to b!\{\text{fail : } a.1 \to a; e\}) \to (1 \to b!\{\text{fail : } p; e\}) \to b!\{\text{fail : } p; e\}
\]

\[
\text{catch } m \ h = \text{handle } m() \ \text{with}
\]

\[
\begin{align*}
\text{return } x & \mapsto x \\
\langle \text{fail } () \rangle & \mapsto h ()
\end{align*}
\]

If \( h \) can itself fail then \( p \) is instantiated to \( \text{Pre}(a.1 \to a) \)
Example: abstracting over an exception handler

Rémy style:

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a; e\}) \rightarrow (1 \to b!\{\text{fail} : p; e\}) \rightarrow b!\{\text{fail} : p; e\}
\]
\[
\text{catch } m \; h = \text{handle } m() \; \text{with}
\]
\[
\begin{align*}
\text{return } x & \mapsto x \\
\langle \text{fail } () \rangle & \mapsto h ()
\end{align*}
\]

If \( h \) can itself fail then \( p \) is instantiated to \( \text{Pre}(a.1 \to a) \)

Leijen style:

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a; e\}) \rightarrow (1 \to b!\{; e\}) \rightarrow b!\{; e\}
\]
\[
\text{catch } m \; h = \text{handle } m() \; \text{with}
\]
\[
\begin{align*}
\text{return } x & \mapsto x \\
\langle \text{fail } () \rangle & \mapsto h ()
\end{align*}
\]
Example: abstracting over an exception handler

Rémy style:

\[
\text{catch} : (1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a; e\}) \rightarrow (1 \rightarrow b!\{\text{fail} : p; e\}) \rightarrow b!\{\text{fail} : p; e\}
\]

\[
\text{catch } m \ h = \text{handle } m() \ \text{with}
\]

\[
\begin{align*}
\text{return } x & \mapsto x \\
\langle \text{fail } () \rangle & \mapsto h ()
\end{align*}
\]

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

Leijen style:

\[
\text{catch} : (1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a; e\}) \rightarrow (1 \rightarrow b!\{; e\}) \rightarrow b!\{; e\}
\]

\[
\text{catch } m \ h = \text{handle } m() \ \text{with}
\]

\[
\begin{align*}
\text{return } x & \mapsto x \\
\langle \text{fail } () \rangle & \mapsto h ()
\end{align*}
\]

If \( h \) can itself fail then \( e \) is instantiated to (\text{fail} : a.1 \rightarrow a; e') for some \( e' \), which means the type of \( m \) is \((1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a, \text{fail} : a.1 \rightarrow a; e'\})\)
Invisible effect polymorphism

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

Example (Leijen style):

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \mapsto a\}; e}\]

\[
\text{map} : (a \to b!\{}\to \text{List} a \to \text{List} b!\{}\]

And further that empty polymorphic effects need not be written at all:

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \mapsto a\}) \to (1 \to b!\{}\to b!\{}
\]

\[
\text{map} : (a \to b!\{}\to \text{List} a \to \text{List} b!\{}
\]

We do now need to use explicit syntax to denote a closed row (\(\emptyset\)), but with row-based effect typing closed rows are uncommon.
Invisible effect polymorphism

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

Example (Leijen style):

\[
\begin{align*}
\text{catch} : (1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a; e\}) & \rightarrow (1 \rightarrow b!\{; e\}) \rightarrow b!\{; e\} \\
\text{map} : (a \rightarrow b!\{; e\}) & \rightarrow \text{List} a \rightarrow \text{List} b!\{; e\}
\end{align*}
\]
Invisible effect polymorphism

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

Example (Leijen style):

\[
\text{catch} : (1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a; e\}) \rightarrow (1 \rightarrow b!\{; e\}) \rightarrow b!\{; e\}
\]
\[
\text{map} : (a \rightarrow b!\{; e\}) \rightarrow \text{List a} \rightarrow \text{List b!\{; e\}}
\]

We adopt a convention that omitted effect variables are all the same

\[
\text{catch} : (1 \rightarrow b!\{\text{fail} : a.1 \rightarrow a\}) \rightarrow (1 \rightarrow b!\{\}) \rightarrow b!\{
\]
\[
\text{map} : (a \rightarrow b!\{\}) \rightarrow \text{List a} \rightarrow \text{List b!\{}}
\]

And further that empty polymorphic effects need not be written at all:

\[
\text{catch} : (1 \rightarrow b!\{\}) \rightarrow (1 \rightarrow b!\{\}) \rightarrow b!\{
\]
\[
\text{map} : (a \rightarrow b!\{\}) \rightarrow \text{List a} \rightarrow \text{List b!\{}}
\]

We do not need to use explicit syntax to denote a closed row (\(\emptyset\)), but with row-based effect typing closed rows are uncommon
Invisible effect polymorphism

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

Example (Leijen style):

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a; e\}) \to (1 \to b!\{}; e\}) \to b!\{}; e\}
\]
\[
\text{map} : (a \to b!\{}; e\}) \to \text{List } a \to \text{List } b!\{}; e\}
\]

We adopt a convention that omitted effect variables are all the same

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a\}) \to (1 \to b!\{}\} \to b!\{}\}
\]
\[
\text{map} : (a \to b!\{}\} \to \text{List } a \to \text{List } b!\{}\}
\]

And further that empty polymorphic effects need not be written at all:

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a\}) \to (1 \to b) \to b
\]
\[
\text{map} : (a \to b) \to \text{List } a \to \text{List } b
Invisible effect polymorphism

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

Example (Leijen style):

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\}
\]

\[
\text{map} : (a \to b!\{; e\}) \to \text{List } a \to \text{List } b!\{; e\}
\]

We adopt a convention that omitted effect variables are all the same

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a\}) \to (1 \to b\{\}) \to b\{\}
\]

\[
\text{map} : (a \to b!\{\}) \to \text{List } a \to \text{List } b!\{}
\]

And further that empty polymorphic effects need not be written at all:

\[
\text{catch} : (1 \to b!\{\text{fail} : a.1 \to a\}) \to (1 \to b) \to b
\]

\[
\text{map} : (a \to b) \to \text{List } a \to \text{List } b
\]

We do now need to use explicit syntax to denote a closed row (\(\emptyset\)), but with row-based effect typing closed rows are uncommon
Effect pollution example

Handlers

reads : List Nat → a!{get : 1 → Nat} ⇒ a
reads ([]) = return x ↦ x
   ⟨get () → r⟩ ↦ fail ()
reads (n :: ns) = return x ↦ x
   ⟨get () → r⟩ ↦ r ns n

maybeFail : b!{fail : a.1 → a} ⇒ Maybe b
maybeFail = return x ↦ Just x
   ⟨fail () → r⟩ ↦ Nothing
Effect pollution example

bad : List b → (1 → b!{get : 1 → Nat, fail : a.1 → a}) → Maybe b
bad ns t = handle (handle t () with reads ns) with maybeFail
Effect pollution example

\[
\text{bad} : \text{List } b \rightarrow (1 \rightarrow b!\{\text{get} : 1 \rightarrow \text{Nat}, \text{fail} : a.1 \rightarrow a\}) \rightarrow \text{Maybe } b
\]

\[
\text{bad } ns \ t = \text{handle} (\text{handle } t () \ \text{with} \ \text{reads } ns) \ \text{with} \ \text{maybeFail}
\]

\[
\text{bad } [1, 2] (\lambda().\text{get} () + \text{fail} ()) : \text{Maybe Nat} \Rightarrow \text{Nothing}
\]
Effect pollution example

\[
\text{bad : List } b \rightarrow (1 \rightarrow b!\{\text{get} : 1 \rightarrow \text{Nat}, \text{fail} : a.1 \rightarrow a\}) \rightarrow \text{Maybe } b
\]

\[
\text{bad } ns \ t = \text{handle (handle } t () \text{ with reads } ns) \text{ with maybeFail}
\]

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

How can we encapsulate the use of \text{fail} as an intermediate effect?
Effect pollution example

\[ \text{bad : List } b \rightarrow (1 \rightarrow b!\{\text{get} : 1 \rightarrow \text{Nat}, \text{fail} : a.1 \rightarrow a\}) \rightarrow \text{Maybe } b \]

\[ \text{bad } ns \ t = \text{handle } (\text{handle } t() \ \text{with} \ \text{reads } ns) \ \text{with} \ \text{maybeFail} \]

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

How can we encapsulate the use of \text{fail} as an intermediate effect?

The aim is to define

\[ \text{good : List } b \rightarrow (1 \rightarrow b!\{\text{get} : 1 \rightarrow \text{Nat}\}) \rightarrow \text{Maybe } b \]

by composing \text{reads} and \text{maybeFail} such that

\[ \text{good } [1, 2] (\lambda(). \text{get}() + \text{fail}()) : \text{Maybe Nat!}\{\text{fail} : a.1 \rightarrow a\} \]

performs the \text{fail} operation.
Effect encapsulation

Two solutions to the effect pollution problem:

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

  \[
  \text{good : List } b \rightarrow (1 \rightarrow b!\{\text{get : 1 \rightarrow Nat}\}) \rightarrow \text{Maybe } b
  \]

\[
\text{good } ns \, t = \text{handle (handle (\langle \text{fail} \rangle \, (t \, ()))) with reads } ns \, \text{with maybeFail}
\]

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

\[ \Delta; \Gamma \vdash M : A!\{R\} \]

\[ \Delta; \Gamma \vdash \langle \text{op} \rangle M : A!\{\text{op} : B \rightarrow C; R\} \]

Akin to weakening for effects
Doo bee doo bee doo

Shall I be pure or impure?
—Philip Wadler

A value is. A computation does.
—Paul Blain Levy

‘To be is to do’—Socrates.
‘To do is to be’—Sartre.
‘Do be do be do’—Sinatra.
—anonymouse graffiti, via Kurt Vonnegut
Frank

[Lindley, McBride, McLaughlin, POPL 2017, “Do be do be do”]
[Convent, Lindley, McBride, McLaughlin, JFP 2019, “Doo bee doo bee doo”]

Frank is an unequivocally effect handler oriented research programming language

Key features include:

- invisible effect polymorphism
- call-by-handling
- multihandlers
- adjustments
- adaptors (a generalisation of mask)

Probably a misfeature: unusual syntax
Links
http://www.links-lang.org
Linking theory to practice for the web

- Query Shredding
- Language-Integrated Query
- Provenance
- Relational Lenses

DATABASE INTEGRATION

WEB DEVELOPMENT
- Typed HTML + antiquotes
- Formlets
- Model-View-Update
- RPC Calculus

CONCURRENCY & DISTRIBUTION
- Distributed Session Types

INTERACTIVE PROGRAMMING
- Session Exceptions
- TryLinks
- Notebook Programming

EFFECT HANDLERS

- CEK Machine (Server)
- CPS Translation (Client)
- Row-based Effects

With thanks to Simon Fowler
Handlers in Links and Frank (demo)
Effect typing scalability challenges

Effect encapsulation

Linearity

Generativity

Indexed effects

Equations