Effect typing

Sam Lindley

The University of Edinburgh

SPLV 2024

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

$$\mathsf{maybeFail} : \forall e. A!(e \uplus \{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \Rightarrow \mathsf{Maybe}\ A!e$$
$$\mathsf{allChoices} : \forall e. A!(e \uplus \{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}\}) \Rightarrow \mathsf{List}\ A!e$$

Effect polymorphism

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

Example: choice and failure

$$\begin{aligned}\mathsf{maybeFail} &: \forall e.A!(e \uplus \{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \Rightarrow \mathsf{Maybe}\ A!e \\ \mathsf{allChoices} &: \forall e.A!(e \uplus \{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}\}) \Rightarrow \mathsf{List}\ A!e\end{aligned}$$

With explicit type applications we may write:

$$\textbf{handle}\ (\textbf{handle}\ \mathsf{drunkTosses}\,2\ \textbf{with}\ \mathsf{maybeFail}\,\{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}\})\ \textbf{with}\ \mathsf{allChoices}\,\emptyset$$

or

$$\textbf{handle}\ (\textbf{handle}\ \mathsf{drunkTosses}\,2\ \textbf{with}\ \mathsf{allChoices}\,\{\mathsf{fail} : a.1 \twoheadrightarrow a\})\ \textbf{with}\ \mathsf{maybeFail}\,\emptyset$$

Effect polymorphism via row polymorphism

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

Effect polymorphism via row polymorphism

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

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

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 $\ell_1 : A_1, ..., \ell : A_n$

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

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 ($\mathrm{Pre}(A)$) 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 ($\mathrm{Pre}(A)$) or absent (Abs)

Duplicate labels disallowed

Example:

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

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:

$$
\begin{aligned}
&\mathsf{maybeFail} : \forall(e : \mathsf{Row}).\\
&\qquad A!(\{\mathsf{fail} : (a : \mathsf{Type}).1 \twoheadrightarrow a; e\}) \Rightarrow \mathsf{Maybe}\ A!\{; e\}\\
&\mathsf{allChoices} : \forall(e : \mathsf{Row}).\\
&\qquad A!(e \uplus \{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}; e\}) \Rightarrow \mathsf{List}\ A!\{; e\}
\end{aligned}
$$

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):

$$\begin{aligned}&\textbf{handle}\ (\textbf{handle}\ \mathsf{drunkTosses}\,2\ \textbf{with}\ \mathsf{maybeFail}\,\{\textcolor{blue}{\mathsf{choose}} : 1 \twoheadrightarrow \mathsf{Bool}\}\ \mathsf{Abs})\\ &\textbf{with}\ \mathsf{allChoices}\,\emptyset\ \mathsf{Abs}\end{aligned}$$

Handler composition with row polymorphism

Instantiating an effect variable supports handler composition

Rémy style (explicit instantiation):

$$\textbf{handle}\ (\textbf{handle}\ \mathsf{drunkTosses}\, 2\ \textbf{with}\ \mathsf{maybeFail}\, \{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}\}\, \mathsf{Abs}) \\ \textbf{with}\ \mathsf{allChoices}\, \emptyset\, \mathsf{Abs}$$

Leijen style (explicit instantiation):

$$\textbf{handle}\ (\textbf{handle}\ \mathsf{drunkTosses}\, 2\ \textbf{with}\ \mathsf{maybeFail}\, \{\mathsf{choose} : 1 \twoheadrightarrow \mathsf{Bool}\}) \\ \textbf{with}\ \mathsf{allChoices}\, \emptyset$$

Example: abstracting over an exception handler

Rémy style:

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

Example: abstracting over an exception handler

Rémy style:

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

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

Example: abstracting over an exception handler

Rémy style:

$$
\begin{array}{l}
\text{catch} : (1 \to b!\{\text{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{\text{fail} : p; e\}) \to b!\{\text{fail} : p; e\} \\
\text{catch}\ m\ h = \textbf{handle}\ m()\ \textbf{with} \\
\qquad\qquad\qquad \textbf{return}\ x \mapsto x \\
\qquad\qquad\qquad \langle \text{fail}\ () \rangle \quad \mapsto h\ ()
\end{array}
$$

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

Leijen style:

$$
\begin{array}{l}
\text{catch} : (1 \to b!\{\text{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\} \\
\text{catch}\ m\ h = \textbf{handle}\ m()\ \textbf{with} \\
\qquad\qquad\qquad \textbf{return}\ x \mapsto x \\
\qquad\qquad\qquad \langle \text{fail}\ () \rangle \quad \mapsto h\ ()
\end{array}
$$

Example: abstracting over an exception handler

Rémy style:

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

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

Leijen style:

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

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

Invisible effect polymorphism

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

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):

$$
\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\} \\
&\mathsf{map} : (a \to b!\{; e\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{; e\}
\end{aligned}
$$

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):

$$\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\} \\
&\mathsf{map} : (a \to b!\{; e\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{; e\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \to (1 \to b!\{\}) \to b!\{\} \\
&\mathsf{map} : (a \to b!\{\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{\}
\end{aligned}$$

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):

$$
\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\} \\
&\mathsf{map} : (a \to b!\{; e\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{; e\}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \to (1 \to b!\{\}) \to b!\{\} \\
&\mathsf{map} : (a \to b!\{\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{\}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \to (1 \to b) \to b \\
&\mathsf{map} : (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b
\end{aligned}
$$

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):

$$\begin{aligned}&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a; e\}) \to (1 \to b!\{; e\}) \to b!\{; e\}\\&\mathsf{map} : (a \to b!\{; e\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{; e\}\end{aligned}$$

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

$$\begin{aligned}&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \to (1 \to b!\{\}) \to b!\{\}\\&\mathsf{map} : (a \to b!\{\}) \to \mathsf{List}\ a \to \mathsf{List}\ b!\{\}\end{aligned}$$

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

$$\begin{aligned}&\mathsf{catch} : (1 \to b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}) \to (1 \to b) \to b\\&\mathsf{map} : (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b\end{aligned}$$

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

$$\mathsf{reads} : \mathsf{List\ Nat} \to a!\{\mathsf{get} : 1 \twoheadrightarrow \mathsf{Nat}\} \Rightarrow a!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}$$

$$\begin{aligned} \mathsf{reads}\,([]) = \mathbf{return}\,x &\mapsto x \\ \langle \mathsf{get}\,() \to r\rangle &\mapsto \mathsf{fail}\,() \end{aligned} \qquad \begin{aligned} \mathsf{reads}\,(n :: ns) = \mathbf{return}\,x &\mapsto x \\ \langle \mathsf{get}\,() \to r\rangle &\mapsto r\ ns\ n \end{aligned}$$

$$\begin{aligned} &\mathsf{maybeFail} : b!\{\mathsf{fail} : a.1 \twoheadrightarrow a\} \Rightarrow \mathsf{Maybe}\ b \\ &\mathsf{maybeFail} = \mathbf{return}\,x \quad \mapsto \mathsf{Just}\,x \\ &\qquad\qquad\quad \langle \mathsf{fail}\,() \to r\rangle \mapsto \mathsf{Nothing} \end{aligned}$$

Effect pollution example

$$\mathrm{bad} : \mathrm{List}\ b \to (1 \to b!\{\mathsf{get} : 1 \twoheadrightarrow \mathrm{Nat}, \mathsf{fail} : a.1 \twoheadrightarrow a\}) \to \mathrm{Maybe}\ b$$
$$\mathrm{bad}\ ns\ t = \textbf{handle}\ (\textbf{handle}\ t\,()\ \textbf{with}\ \mathrm{reads}\, ns)\ \textbf{with}\ \mathrm{maybeFail}$$

Effect pollution example

$$\mathsf{bad} : \mathsf{List}\ b \to (1 \to b!\{\mathsf{get} : 1 \twoheadrightarrow \mathsf{Nat}, \mathsf{fail} : a.1 \twoheadrightarrow a\}) \to \mathsf{Maybe}\ b$$
$$\mathsf{bad}\ ns\ t = \mathbf{handle}\ (\mathbf{handle}\ t\ ()\ \mathbf{with}\ \mathsf{reads}\ ns)\ \mathbf{with}\ \mathsf{maybeFail}$$

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

Effect pollution example

$$\text{bad} : \text{List}\ b \to (1 \to b!\{\text{get} : 1 \twoheadrightarrow \text{Nat}, \text{fail} : a.1 \twoheadrightarrow a\}) \to \text{Maybe}\ b$$
$$\text{bad}\ ns\ t = \textbf{handle}\ (\textbf{handle}\ t\ ()\ \textbf{with}\ \text{reads}\ ns)\ \textbf{with}\ \text{maybeFail}$$

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

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

Effect pollution example

$$\begin{array}{l} \mathrm{bad} : \mathrm{List}\ b \to (1 \to b!\{\mathsf{get} : 1 \twoheadrightarrow \mathrm{Nat}, \mathsf{fail} : a.1 \twoheadrightarrow a\}) \to \mathrm{Maybe}\ b \\ \mathrm{bad}\ ns\ t = \textbf{handle}\ (\textbf{handle}\ t\ ()\ \textbf{with}\ \mathrm{reads}\ ns)\ \textbf{with}\ \mathrm{maybeFail} \end{array}$$

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

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

The aim is to define

$$\mathrm{good} : \mathrm{List}\ b \to (1 \to b!\{\mathsf{get} : 1 \twoheadrightarrow \mathrm{Nat}\}) \to \mathrm{Maybe}\ b$$

by composing reads and maybeFail such that

$$\mathrm{good}\,[1,2]\,(\lambda().\mathsf{get}\,() + \mathsf{fail}\,()) : \mathrm{Maybe\ Nat}!\{\mathsf{fail} : a.1 \twoheadrightarrow a\}$$

performs the fail operation.

Effect encapsulation

Two solutions to the effect pollution problem:

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

$$\begin{aligned}&\mathsf{good} : \mathsf{List}\ b \to (1 \to b!\{\mathsf{get} : 1 \twoheadrightarrow \mathsf{Nat}\}) \to \mathsf{Maybe}\ b\\ &\mathsf{good}\ ns\ t = \mathbf{handle}\ (\mathbf{handle}\ (\langle \mathsf{fail} \rangle\ (t\ ()))\ \mathbf{with}\ \mathsf{reads}\ ns)\ \mathbf{with}\ \mathsf{maybeFail}\end{aligned}$$

  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

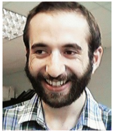
$$\frac{\Delta; \Gamma \vdash M : A!\{R\}}{\Delta; \Gamma \vdash \langle \mathsf{op} \rangle\ M : A!\{\mathsf{op} : B \twoheadrightarrow C; R\}}$$

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.
—anonymous graffiti, via Kurt Vonnegut

Frank

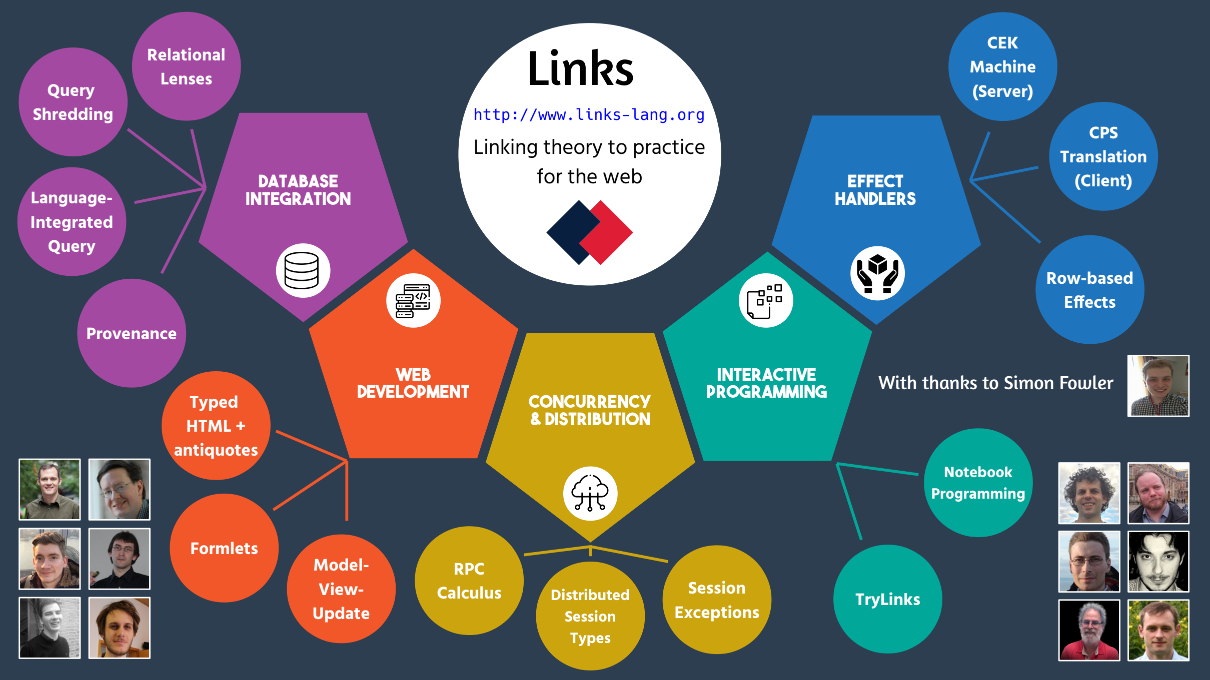
[Lindley, McBride, McLaughlin, POPL 2017, "Do be do be do"]
[Convent, Lindley, McBride, McLaughlin, JFP 2019, "Doo bee doo bee doo"]

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
DATABASE INTEGRATION
Relational Lenses
Query Shredding
Language-Integrated Query
Provenance
WEB DEVELOPMENT
Typed HTML + antiquotes
Formlets
Model-View-Update
CONCURRENCY & DISTRIBUTION
RPC Calculus
Distributed Session Types
Session Exceptions
INTERACTIVE PROGRAMMING
Notebook Programming
TryLinks
EFFECT HANDLERS
CEK Machine (Server)
CPS Translation (Client)
Row-based Effects
With thanks to Simon Fowler

Handlers in Links and Frank (demo)

Demos

Effect typing scalability challenges

Effect encapsulation

Linearity

Generativity

Indexed effects

Equations