Higher-Ranked Exception Types

Work-in-Progress

Ruud Koot

Department of Information and Computing Sciences
Utrecht University

September 2, 2014

Motivation

- Types should not lie; we would like to have *checked exceptions* in Haskell:

$$map :: (\alpha \to \beta) \to [\alpha] \to [\beta] \ \mathbf{throws}\ e$$

- What should be the correct value of e?

Motivation

Assigning accurate exception types is complicated by:

Higher-order functions Exceptions raised by higher-order functions depend on the exceptions raised by functional arguments.

$$map :: (\alpha \to \beta \ \mathbf{throws}\ e_1) \to [\alpha] \to [\beta]\ \mathbf{throws}\ (e_1 \cup e_2)$$

Non-strict evaluation Exceptions are embedded inside values.

$$\begin{aligned} map :: &(\alpha\ \mathbf{throws}\ e_1 \to \beta)\ \mathbf{throws}\ e_2 \\ &\to [\alpha\ \mathbf{throws}\ e_3]\ \mathbf{throws}\ e_4 \to [\beta\ \mathbf{throws}\ e_5]\ \mathbf{throws}\ e_6 \end{aligned}$$

Motivation

- Instead of τ **throws** e, write τ^e for a type τ that can evaluate to $\bot_\chi$ for some $\chi \in e$.
- The fully annotated exception type for *map* would be:

$$
\begin{aligned}
&map :: (\alpha^{e_1} \to \beta^{(e_1 \,\cup\, e_2)})^{e_3} \to [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \,\cup\, e_2 \,\cup\, e_3)}]^{e_4} \\
&map = \lambda f.\lambda xs.\ \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad\quad \begin{array}{lcl} [\,] & \mapsto & [\,] \\ (y : ys) & \mapsto & f\ y : map\ f\ ys \end{array}
\end{aligned}
$$

Motivation

- Instead of τ **throws** e, write τ^e for a type τ that can evaluate to $\bot_\chi$ for some $\chi \in e$.
- The fully annotated exception type for *map* would be:

$$
\begin{aligned}
&\mathit{map} :: (\alpha^{e_1} \to \beta^{(e_1 \,\cup\, e_2)})^{e_3} \to [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \,\cup\, e_2 \,\cup\, e_3)}]^{e_4} \\
&\mathit{map} = \lambda f.\lambda xs.\ \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad\qquad [\,] \qquad\quad \mapsto [\,] \\
&\qquad\qquad\qquad (y : ys) \mapsto f\ y : \mathit{map}\ f\ ys
\end{aligned}
$$

- If you want to be pedantic:

$$
\begin{aligned}
&\mathit{map} :: \forall \alpha\ \beta\ e_1\ e_2\ e_3\ e_4. \\
&\quad ((\alpha^{e_1} \to \beta^{(e_1 \,\cup\, e_2)})^{e_3} \to ([\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \,\cup\, e_2 \,\cup\, e_3)}]^{e_4})^{\emptyset})^{\emptyset}
\end{aligned}
$$

Motivation

- Instead of τ **throws** e, write τ^e for a type τ that can evaluate to $\bot_\chi$ for some $\chi \in e$.
- The fully annotated exception type for *map* would be:

$$
\begin{aligned}
&\mathit{map} :: (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \to [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4} \\
&\mathit{map} = \lambda f.\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\qquad [\,] \qquad\quad \mapsto [\,] \\
&\qquad\qquad\qquad (y : ys) \mapsto f\ y : \mathit{map}\ f\ ys
\end{aligned}
$$

- If you want to be pedantic:

$$
\begin{aligned}
&\mathit{map} :: \forall \alpha\ \beta\ e_1\ e_2\ e_3\ e_4. \\
&\quad (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \xrightarrow{\emptyset} [\alpha^{e_1}]^{e_4} \xrightarrow{\emptyset} [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4}
\end{aligned}
$$

Motivation

- The exception type

$$map :: (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \,\cup\, e_2)}) \to [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \,\cup\, e_2 \,\cup\, e_3)}]^{e_4}$$

  is not as accurate as we would like.
- Consider the instantiations:

$$\begin{aligned} &map\ id && :: [\alpha^{e_1}]^{e_4} \to [\alpha^{e_1}]^{e_4} \\ &map\ (const\ \bot_{\mathbf{E}}) && :: [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \,\cup\, \{\mathbf{E}\})}]^{e_4} \end{aligned}$$

- A more appropriate type for $map\ (const\ \bot_{\mathbf{E}})$ would be

$$map\ (const\ \bot_{\mathbf{E}}) :: [\alpha^{e_1}]^{e_4} \to [\beta^{\{\mathbf{E}\}}]^{e_4}$$

  as it cannot propagate exceptional elements inside the input list to the output list.

Motivation

- The problem is that we have already committed the first argument of *map* to be of type

$$\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)},$$

  i.e. it propagates exceptional values from the its input to the output while possibly adding additional exceptional values.
- This is a worst-case scenario: it is sound but inaccurate.

Motivation

- The solution is to move from Hindley–Milner to F_ω, introducing *higher-ranked types* and *type operators*.
  - Recall that System F_ω replicates the *simply typed λ-calculus* on the type level.
- This gives us the expressiveness to state the exception type of *map* as:

$$\begin{aligned}&\forall e_2\ e_3.(\forall e_1.\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2\ e_1)})\\&\quad \to (\forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{(e_2\ e_4\ \cup\ e_3)}]^{e_5})\end{aligned}$$

- Note that e_2 is an *exception operator* of kind $\textsc{exn} \to \textsc{exn}$.

Motivation

- Given the following functions:

$$\begin{aligned}
map \quad & :: \forall e_2\, e_3.(\forall e_1.\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2\, e_1)}) \\
& \quad \to (\forall e_4\, e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{(e_2\, e_4\, \cup\, e_3)}]^{e_5}) \\
id \quad & :: \forall e.\alpha^e \xrightarrow{\emptyset} \alpha^e \\
const\ \bot_{\mathbf{E}} & :: \forall e.\alpha^e \xrightarrow{\emptyset} \beta^{\{\mathbf{E}\}}
\end{aligned}$$

- Applying *id* or $const\ \bot_{\mathbf{E}}$ to *map* will give rise the the instantiations $e_2 \mapsto \lambda e.e$, respectively $e_2 \mapsto \lambda e.\{\mathbf{E}\}$.
- This gives us the exception types:

$$\begin{aligned}
map\ id \quad & :: \forall e_4\, e_5.[\alpha^{e_4}]^{e_5} \to [\alpha^{e_4}]^{e_5} \\
map\ (const\ \bot_{\mathbf{E}}) & :: \forall e_4\, e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{\{\mathbf{E}\}}]^{e_5}
\end{aligned}$$

as desired.

Technicalities

- Due to their syntactic weight, higher-ranked exception type only seem useful if they can be infered automatically.
- Unlike for HM type inference is undecidable in F_ω.
- However, the exception types are annotations piggybacking on top of an underlying type system.
- Holdermans and Hage [HH10] showed type inference is decidable for a higher-ranked annotated type system with type operators performing control-flow analysis.

Technicalities

1. Perform Hindley–Milner type inference to reconstruct the underlying types.
2. Run a second inference pass to reconstruct the exception types.
   2.1 Collect a set of subtyping constraints.
   2.2 In case of a λ-abstraction $\lambda x : \tau.e$, we *complete* the type τ to an exception type.
   2.3 In case of an application we *match* the types of the formal and actual parameter.
3. Solve the generated subtyping constraints.

Technicalities: Reconstruction (variables)

$$
\begin{array}{l}
\mathit{reconstruct}\ \widehat{\Gamma}\ x = \\
\quad \mathbf{let}\ (\widehat{\tau}, \chi) = \widehat{\Gamma}\ (x) \\
\qquad\quad e\ \mathit{be\ fresh} \\
\quad \mathbf{in}\ \ (\widehat{\tau}, e, \{\chi \subseteq e\})
\end{array}
$$

Technicalities: Reconstruction (abstractions)

$$
\begin{array}{l}
\mathit{reconstruct}\ \widehat{\Gamma}\ (\lambda x : \tau . t) = \\
\quad \mathbf{let}\ (\widehat{\tau}_1, \overline{e_i :: \kappa_i}) = \mathit{complete}\ \tau\ \emptyset \\
\qquad e_1\ \mathit{be\ fresh} \\
\qquad (\widehat{\tau}_2, e_2, C_1)\ = \mathit{reconstruct}\ (\widehat{\Gamma}, x \mapsto (\widehat{\tau}_1, e_1))\ t \\
\qquad X\ = \{e_1\} \cup \{\overline{e_i}\} \cup fv\ \widehat{\Gamma} \\
\qquad \chi_2 = \mathit{solve}\ C_1\ X\ e_2 \\
\qquad \widehat{\tau}\ = \forall e_1 :: \textsc{exn}. \overline{\forall e_i :: \kappa_i}.\ \widehat{\tau}_1^{e_1} \to \widehat{\tau}_2^{\chi_2} \\
\qquad e\ \mathit{be\ fresh} \\
\quad \mathbf{in}\ \ (\widehat{\tau}, e, \emptyset)
\end{array}
$$

Technicalities: Completion

- The completion procedure adds as many quantifiers and type operators as possible to a type.

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\mathrm{bool}} \mathbin{\&} e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \mathrm{EXN}}\ [\text{C-Bool}]$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau} \mathbin{\&} \chi \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau}\ \mathbf{tw}\ \chi] \mathbin{\&} e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \mathrm{EXN}, \overline{e_j :: \kappa_j}}\ [\text{C-List}]$$

$$\frac{\vdash \tau_1 : \widehat{\tau}_1 \mathbin{\&} \chi_1 \triangleright \overline{e_j :: \kappa_j} \quad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau}_2 \mathbin{\&} \chi_2 \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \to \tau_2 : \forall \overline{e_j :: \kappa_j}.(\widehat{\tau}_1\ \mathbf{tw}\ \chi_1 \to \widehat{\tau}_2\ \mathbf{tw}\ \chi_2) \mathbin{\&} e\ \overline{e_i} \triangleright e :: \overline{\kappa_j} \Rightarrow \mathrm{EXN}, \overline{e_k :: \kappa_k}}\ [\text{C-Arr}]$$

Figure : Type completion ($\Gamma \vdash \tau : \widehat{\tau} \mathbin{\&} \chi \triangleright \Gamma'$)

Technicalities: Completion

- $\cdot \vdash \mathbf{bool} : \widehat{\text{bool}} \;\&\; e_1 \triangleright e_1 :: \textsc{exn}$

Technicalities: Completion

- $\cdot \vdash \mathbf{bool} : \widehat{\text{bool}}\ \&\ e_1 \triangleright e_1 :: \textsc{exn}$
- $e_1 :: \textsc{exn} \vdash \mathbf{bool} : \widehat{\text{bool}}\ \&\ e_2\ e_1 \triangleright e_2 :: \textsc{exn} \Rightarrow \textsc{exn}$

Technicalities: Completion

- $\mathbf{bool} \to \mathbf{bool}$
- $\forall e_1 :: \textsc{exn}.\ \widehat{\mathrm{bool}}^{e_1} \to \widehat{\mathrm{bool}}^{(e_2\ e_1)}\ \&\ e_3$
- $e_2 :: \textsc{exn} \Rightarrow \textsc{exn}, e_3 :: \textsc{exn}$

Technicalities: Completion

- $\mathbf{bool} \to \mathbf{bool} \to \mathbf{bool}$
- $$\forall e_1 :: \textsc{exn}.\ \widehat{\text{bool}}^{e_1} \to \\ (\forall e_4 :: \textsc{exn}.\ \widehat{\text{bool}}^{e_4} \xrightarrow{e_2\ e_1} \widehat{\text{bool}}^{(e_5\ e_1\ e_4)}) \ \&\ e_3$$
- $e_2 :: \textsc{exn} \Rightarrow \textsc{exn}, e_3 :: \textsc{exn}, e_5 :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}$

Technicalities: Completion

- $(\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool}$
-

$$\begin{aligned}&\forall e_2 :: \textsc{exn} \Rightarrow \textsc{exn}.\ \forall e_3 :: \textsc{exn}.\\ &\quad \left(\forall e_1 :: \textsc{exn}.\ \widehat{\mathrm{bool}}^{e_1} \xrightarrow{e_3} \widehat{\mathrm{bool}}^{(e_2\ e_1)}\right) \to \widehat{\mathrm{bool}}^{(e_4\ e_2\ e_3)}\ \&\ e_5\end{aligned}$$

- $e_4 :: (\textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}, e_5 :: \textsc{exn}$

Technicalities: Reconstruction (applications)

$$
\begin{array}{l}
\mathit{reconstruct}\ \widehat{\Gamma}\ (t_1\ t_2) = \\
\quad \textbf{let}\ (\widehat{\tau}_1, e_1, C_1) \ = \mathit{reconstruct}\ \widehat{\Gamma}\ t_1 \\
\qquad\quad (\widehat{\tau}_2, e_2, C_2) \ = \mathit{reconstruct}\ \widehat{\Gamma}\ t_2 \\
\qquad\quad \widehat{\tau}_2'^{e_2'} \to \widehat{\tau}'^{\chi'} \ = \mathit{instantiate}\ \widehat{\tau}_1 \\
\qquad\quad \theta = [e_2' \mapsto e_2] \circ \mathit{match}\ \emptyset\ \widehat{\tau}_2\ \widehat{\tau}_2' \\
\qquad\quad e\ \mathit{be\ fresh} \\
\qquad\quad C = \{...\} \\
\quad \textbf{in}\ (\widehat{\tau}, e, C)
\end{array}
$$

Technicalities: Matching

$$
\begin{array}{llll}
\mathit{match} :: \mathbf{Env} \rightarrow \mathbf{Ty} \rightarrow \mathbf{Ty} \rightarrow \mathbf{Subst} & & & \\
\mathit{match}\ \Sigma\ \widehat{\text{bool}} & \widehat{\text{bool}} & & = \mathit{Id} \\
\mathit{match}\ \Sigma\ (\forall e :: \kappa . \widehat{\tau}_1) & (\forall e :: \kappa . \widehat{\tau}'_1) & & = \mathit{match}\ (\Sigma, e \mapsto \kappa)\ \widehat{\tau}_1\ \widehat{\tau}'_1 \\
\multicolumn{3}{l}{\mathit{match}\ \Sigma\ (\widehat{\tau}_1^{e_1} \rightarrow \widehat{\tau}_2^{\chi_2})\ (\widehat{\tau}_1^{e_1} \rightarrow \widehat{\tau}_2^{\prime (e_0\ \overline{e_j})})} & = \\
\multicolumn{4}{l}{\quad \left[e_0 \mapsto (\overline{\lambda e_j :: \Sigma(e_j).}\ \chi_2) \right] \circ \mathit{match}\ \Sigma\ \widehat{\tau}_2\ \widehat{\tau}'_2} \\
\mathit{match}\ \Sigma\ \_ & \_ & & = \mathbf{fail}
\end{array}
$$

Technicalities: Matching — Example

- $\mathit{match}\ [e_1 :: \textsc{exn}, e_2 :: \textsc{exn} \Rightarrow \textsc{exn}, e_3 :: \textsc{exn}]$
  $(\mathrm{b\widehat{oo}l}^{e_1} \to \mathrm{b\widehat{oo}l}^{(e_2\ \ e_1\ \cup\ e_3)})\ (\mathrm{b\widehat{oo}l}^{e_1} \to \mathrm{b\widehat{oo}l}^{(e_0\ \ e_1\ \ e_2\ \ e_3)})$

Technicalities: Matching — Example

- $match\ [e_1 :: \textsc{exn}, e_2 :: \textsc{exn} \Rightarrow \textsc{exn}, e_3 :: \textsc{exn}]$
  $(\widehat{\text{bool}}^{e_1} \rightarrow \widehat{\text{bool}}^{(e_2\ e_1\ \cup\ e_3)})\ (\widehat{\text{bool}}^{e_1} \rightarrow \widehat{\text{bool}}^{(e_0\ e_1\ e_2\ e_3)})$

- $[e_0 \mapsto \lambda e_1 :: \textsc{exn}.\lambda e_2 :: \textsc{exn} \Rightarrow \textsc{exn}.\lambda e_3 :: \textsc{exn}.e_2\ e_1 \cup e_3])$

Technicalities: Constraint solving

- Solving subtyping constraints can be done using a fixed-point iteration.
- To decide we have reached a fixed point we need an equality on types.
- But types are now a simply typed λ-calculus.

Technicalities: $\lambda^{\cup}$

Types

$$
\begin{array}{llll}
\tau \in \mathbf{Ty} & ::= & \mathcal{P} & \text{(base type)} \\
 & \mid & \tau_1 \to \tau_2 & \text{(function type)}
\end{array}
$$

Terms

$$
\begin{array}{llll}
t \in \mathbf{Tm} & ::= & x, y, \ldots & \text{(variable)} \\
 & \mid & \lambda x : \tau . t & \text{(abstraction)} \\
 & \mid & t_1 \ t_2 & \text{(application)} \\
 & \mid & \emptyset & \text{(empty)} \\
 & \mid & \{c\} & \text{(singleton)} \\
 & \mid & t_1 \cup t_2 & \text{(union)}
\end{array}
$$

Values Values v are terms of the form

$$\lambda x_1{:}\tau_1.\cdots\lambda x_i{:}\tau_i.\{c_1\}\cup(\cdots\cup(\{c_j\}\cup(x_1\ v_{11}\cdots v_{1m}\cup(\cdots\cup x_k\ v_{k1}\cdots v_{kn}))))$$

Technicalities: $\lambda^{\cup}$

$$
\begin{array}{rcll}
(\lambda x:\tau.t_1)\ t_2 & \longrightarrow & [t_2/x]\,t_1 & & (\beta\text{-reduction}) \\
(t_1 \cup t_2)\ t_3 & \longrightarrow & t_1\ t_3 \cup t_2\ t_3 & & \\
(\lambda x:\tau.t_1) \cup (\lambda x:\tau.t_2) & \longrightarrow & \lambda x:\tau.\,(t_1 \cup t_2) & & (\text{congruences}) \\
x\ t_1 \cdots t_n \cup x'\ t_1' \cdots t_n' & \longrightarrow & x\ \ (t_1 \cup t_1') \cdots (t_n \cup t_n') & & \\
(t_1 \cup t_2) \cup t_3 & \longrightarrow & t_1 \cup (t_2 \cup t_3) & & (\text{associativity}) \\
\emptyset \cup t & \longrightarrow & t & & \\
t \cup \emptyset & \longrightarrow & t & & (\text{unit}) \\
x \cup x & \longrightarrow & x & & \\
x \cup (x \cup t) & \longrightarrow & x \cup t & & \\
\{c\} \cup \{c\} & \longrightarrow & \{c\} & & (\text{idempotence}) \\
\{c\} \cup (\{c\} \cup t) & \longrightarrow & \{c\} \cup t & & \\
x\ t_1 \cdots t_n \cup \{c\} & \longrightarrow & \{c\} \cup x\ t_1 \cdots t_n & & (1) \\
x\ t_1 \cdots t_n \cup (\{c\} \cup t) & \longrightarrow & \{c\} \cup (x\ t_1 \cdots t_n \cup t) & & (2) \\
x\ t_1 \cdots t_n \cup x'\ t_1' \cdots t_n' & \longrightarrow & x'\ t_1' \cdots t_n' \cup x\ t_1 \cdots t_n & \text{if } x' \prec x & (3) \\
x\ t_1 \cdots t_n \cup (x'\ t_1' \cdots t_n' \cup t) & \longrightarrow & x'\ t_1' \cdots t_n' \cup (x\ t_1 \cdots t_n \cup t) & \text{if } x' \prec x & (4) \\
\{c\} \cup \{c'\} & \longrightarrow & \{c'\} \cup \{c\} & \text{if } c' \prec c & (5) \\
\{c\} \cup (\{c'\} \cup t) & \longrightarrow & \{c'\} \cup (\{c\} \cup t) & \text{if } c' \prec c & (6)
\end{array}
$$

Technicalities: $\lambda^{\cup}$

Conjecture

The reduction relation $\longrightarrow$ *preserves meaning.*

Conjecture

The reduction relation $\longrightarrow$ *is strongly normalizing.*

Conjecture

The reduction relation $\longrightarrow$ *is locally confluent.*

Corollary

The reduction relation $\longrightarrow$ *is confluent.*

Corollary

The $\lambda^{\cup}$*-calculus has unique normal forms.*

Corollary

Equality of $\lambda^{\cup}$*-terms can be decided by normalization.*

Problems

- Not sound w.r.t. *imprecise exception semantics*.
- Making it sound negates the precision gained by higher-ranked types.
- Need to move to a more powerful constraint language.
  - In previous work we used conditionals/implications and a somewhat ad hoc non-emptyness guard.
  - Now I want to look at *Boolean rings*, which look more well-behaved.
  - May make more sense to use *equational unification* instead of constraints.

Problems: Imprecise exception semantics

- In an optimizing compiler we want the following equality, called the *case-switching transformation*, to hold:

$$
\begin{array}{l}
\forall e_i.\textbf{if } e_1 \textbf{ then} \\
\quad\quad \textbf{if } e_2 \textbf{ then } e_3 \textbf{ else } e_4 \\
\quad \textbf{else} \\
\quad\quad \textbf{if } e_2 \textbf{ then } e_5 \textbf{ else } e_6 \equiv \textbf{if } e_2 \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } e_1 \textbf{ then } e_3 \textbf{ else } e_5 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } e_1 \textbf{ then } e_4 \textbf{ else } e_6
\end{array}
$$

- This does not hold if we have observable exceptions and track them precisely.
  - Counterexample: Take $e_1 = \bot_{\mathbf{E_1}}$ and $e_2 = \bot_{\mathbf{E_2}}$.
- Introduce some “imprecision”: If the guard can reduce to an exceptional value, then pretend both branches get executed.

Problems: Imprecise exception semantics

- In an optimizing compiler we want the following equality, called the *case-switching transformation*, to hold:

$$
\begin{array}{l}
\forall e_i.\mathbf{if}\ e_1\ \mathbf{then} \\
\quad\quad \mathbf{if}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \\
\quad \mathbf{else} \\
\quad\quad \mathbf{if}\ e_2\ \mathbf{then}\ e_5\ \mathbf{else}\ e_6 \equiv \begin{array}[t]{l} \mathbf{if}\ e_2\ \mathbf{then} \\ \quad \mathbf{if}\ e_1\ \mathbf{then}\ e_3\ \mathbf{else}\ e_5 \\ \mathbf{else} \\ \quad \mathbf{if}\ e_1\ \mathbf{then}\ e_4\ \mathbf{else}\ e_6 \end{array}
\end{array}
$$

- This does not hold if we have observable exceptions and track them precisely.
  - Counterexample: Take $e_1 = \bot_{\mathbf{E_1}}$ and $e_2 = \bot_{\mathbf{E_2}}$.
- Introduce some “imprecision”: If the guard can reduce to an exceptional value, then pretend both branches get executed.

References

Stefan Holdermans and Jurriaan Hage, *Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators*, Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, 2010, pp. 63–74.

Andrew J. Kennedy, *Type inference and equational theories*, Tech. Report LIX-RR-96-09, Laboratoire D'Informatique, École Polytechnique, 1996.