Higher-Ranked Exception Types

WORK-IN-PROGRESS

Ruud Koot

Department of Information and Computing Sciences
Utrecht University

June 12, 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:

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

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:

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

- If you want to be pedantic:

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

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:

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

- 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:

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

- 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:

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

as desired.

Intermezzo: Simply-typed λ-calculus

Types

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

Terms

$$\begin{array}{rcll} t \in \mathbf{Tm} & ::= & x, y, \ldots & \text{(variable)} \\ & \mid & \lambda x : \tau . t & \text{(abstraction)} \\ & \mid & t_1 \; t_2 & \text{(application)} \end{array}$$

Values

$$\begin{array}{rcll} v \in \mathbf{Val} & ::= & x, y, \ldots & \text{(free variable)} \\ & \mid & \lambda x : \tau . v & \text{(abstraction value)} \end{array}$$

Intermezzo: Simply-typed λ-calculus

Typing

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ [\text{T-Var}] \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \to \tau_2}\ [\text{T-Abs}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ [\text{T-App}]$$

Intermezzo: Simply-typed λ-calculus

Evaluation We perform *full β-reduction*, i.e. we also evaluate under binders.

$$\frac{t \longrightarrow t'}{\lambda x : \tau.t \longrightarrow \lambda x : \tau.t'} \text{ [E-ABS]}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \text{ [E-APP}_1\text{]} \qquad \frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \text{ [E-APP}_2\text{]}$$

$$\frac{}{(\lambda x : \tau.t_1)\ t_2 \longrightarrow [t_2/x]\ t_1} \text{ [E-BETA]}$$

Intermezzo: Simply-typed λ-calculus

Theorem (Progress)

A term t is either a value v, or we can reduce $t \longrightarrow t'$.

Theorem (Preservation)

If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Theorem (Confluence)

If $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then exists a term t' such that $t_1 \longrightarrow^* t'$ and $t_2 \longrightarrow^* t'$.

Theorem (Normalization)

For any term t we have that $t \longrightarrow^* v$ (in a finite number of steps).

Corollary (Uniqueness of normal forms)

If $t \longrightarrow^* v_1$ and $t \longrightarrow^* v_2$, then $v_1 \equiv v_2$.

Intermezzo: The lambda "cube"

- The simply-typed λ-calculus can be extended with *parametric polymorphism*, or *type operators*, or both.

$$
\begin{tikzcd}
F \arrow[r] & F_\omega \\
\lambda \arrow[u] \arrow[r] & \lambda_\omega \arrow[u]
\end{tikzcd}
$$

- $$\begin{aligned} id &: B \to B \\ id &= \lambda x : B.x \end{aligned}$$
- $$\begin{aligned} id &: \forall \alpha :: *.\alpha \to \alpha \\ id &= \Lambda \alpha : *.\lambda x : \alpha.x \end{aligned}$$
- $$\begin{aligned} Id &:: * \Rightarrow * \\ Id &= \lambda \alpha :: *.\alpha \end{aligned}$$
  $$\begin{aligned} id &: B \to Id\ B \\ id &= \lambda x : B.x \end{aligned}$$
- $$\begin{aligned} Id &:: * \Rightarrow * \\ Id &= \lambda \alpha :: *.\alpha \end{aligned}$$
  $$\begin{aligned} id &: \forall \alpha :: *.\alpha \to Id\ \alpha \\ id &= \Lambda \alpha : *.\lambda x : \alpha.x \end{aligned}$$

- Omitted: the axis for dependent types.

Intermezzo: System F_ω

Types

$$
\begin{array}{llll}
\tau \in \mathbf{Ty} & ::= & \alpha & \text{(type variable)} \\
 & \mid & \tau_1 \to \tau_2 & \text{(function type)} \\
 & \mid & \forall \alpha :: \kappa . \tau & \text{(universal type)} \\
 & \mid & \lambda \alpha :: \kappa . \tau & \text{(operator abstraction)} \\
 & \mid & \tau_1 \; \tau_2 & \text{(operator application)}
\end{array}
$$

Kinds

$$
\begin{array}{llll}
\kappa \in \mathbf{Kind} & ::= & * & \text{(base kind)} \\
 & \mid & \kappa_1 \Rightarrow \kappa_2 & \text{(operator kind)}
\end{array}
$$

Intermezzo: System F_ω

Terms

$$\begin{array}{lrll} t \in \mathbf{Tm} & ::= & x, y, \ldots & \text{(variable)} \\ & \mid & \lambda x : \tau . t & \text{(abstraction)} \\ & \mid & t_1 \; t_2 & \text{(application)} \\ & \mid & \Lambda \alpha :: \kappa . t & \text{(type abstraction)} \\ & \mid & t \; \langle \tau \rangle & \text{(type application)} \end{array}$$

Values

$$\begin{array}{lrll} v \in \mathbf{Val} & ::= & x, y, \ldots & \text{(free variable)} \\ & \mid & \lambda x : \tau . v & \text{(abstraction value)} \end{array}$$

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: Completion

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

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

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

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

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

Technicalities: Completion

- $a \vdash b : c \,\&\, d \triangleright e$

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

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

Terms

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

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: $\lambda^{\cup}$

$$(\lambda x : \tau.t_1)\ t_2 \longrightarrow [t_2/x]\, t_1 \qquad (\beta\text{-reduction})$$

$$(t_1 \cup t_2)\ t_3 \longrightarrow t_1\ t_3 \cup t_2\ t_3$$

$$(\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2) \longrightarrow \lambda x : \tau.\,(t_1 \cup t_2) \qquad (\text{congruences})$$

$$x\ t_1 \cdots t_n \cup x'\ t'_1 \cdots t'_n \longrightarrow x\ \ (t_1 \cup t'_1) \cdots (t_n \cup t'_n)$$

$$(t_1 \cup t_2) \cup t_3 \longrightarrow t_1 \cup (t_2 \cup t_3) \qquad (\text{associativity})$$

$$\emptyset \cup t \longrightarrow t$$

$$t \cup \emptyset \longrightarrow t \qquad (\text{unit})$$

$$x \cup x \longrightarrow x$$

$$x \cup (x \cup t) \longrightarrow x \cup t$$

$$\{c\} \cup \{c\} \longrightarrow \{c\} \qquad (\text{idempotence})$$

$$\{c\} \cup (\{c\} \cup t) \longrightarrow \{c\} \cup t$$

$$x\ t_1 \cdots t_n \cup \{c\} \longrightarrow \{c\} \cup x\ t_1 \cdots t_n \qquad (1)$$

$$x\ t_1 \cdots t_n \cup (\{c\} \cup t) \longrightarrow \{c\} \cup (x\ t_1 \cdots t_n \cup t) \qquad (2)$$

$$x\ t_1 \cdots t_n \cup x'\ t'_1 \cdots t'_n \longrightarrow x'\ t'_1 \cdots t'_n \cup x\ t_1 \cdots t_n \qquad \text{if } x' \prec x \qquad (3)$$

$$x\ t_1 \cdots t_n \cup (x'\ t'_1 \cdots t'_n \cup t) \longrightarrow x'\ t'_1 \cdots t'_n \cup (x\ t_1 \cdots t_n \cup t) \qquad \text{if } x' \prec x \qquad (4)$$

$$\{c\} \cup \{c'\} \longrightarrow \{c'\} \cup \{c\} \qquad \text{if } c' \prec c \qquad (5)$$

$$\{c\} \cup (\{c'\} \cup t) \longrightarrow \{c'\} \cup (\{c\} \cup t) \qquad \text{if } c' \prec c \qquad (6)$$

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:

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

- 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.

Problems: Imprecise exception semantics

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

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

- This does not hold if we have observable exceptions and track them precisely.
  - Counterexample: Take $e_1 = \perp_{\mathbf{E_1}}$ and $e_2 = \perp_{\mathbf{E_2}}$.
- 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 (New York, NY, USA), ICFP '10, ACM, 2010, pp. 63–74.
- Andrew J. Kennedy, *Type inference and equational theories*, Tech. Report LIX-RR-96-09, Laboratoire D'Informatique, École Polytechnique, 1996.