Higher-Ranked Exception Types

Ruud Koot

Utrecht University

May 15, 2014

Motivation

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

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

- 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.

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

Non-strict evaluation Exceptions are embedded inside values.

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

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:

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

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:

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

- If you want to be pedantic:

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

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:

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

- If you want to be pedantic:

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

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:

$$\forall e_2\ e_3.(\forall e_1.\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2\ e_1)}) \\ \rightarrow (\forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \rightarrow [\beta^{(e_2\ e_4\ \cup\ e_3)}]^{e_5})$$

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

Motivation

- Given the following functions:

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

- 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:

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

as desired.

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.

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\text{bool}}\ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_i} \Rightarrow \textsc{exn}}\ [\textsc{C-Bool}]$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau}\ \&\ \chi \rhd \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau}\ \mathbf{throws}\ \chi]\ \ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_i} \Rightarrow \textsc{exn}, \overline{e_j :: \kappa_j}}\ [\textsc{C-List}]$$

$$\frac{\vdash \tau_1 : \widehat{\tau}_1\ \&\ \chi_1 \rhd \overline{e_j :: \kappa_j} \quad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau}_2\ \&\ \chi_2 \rhd \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \to \tau_2 : \forall \overline{e_j :: \kappa_j}.(\widehat{\tau}_1\ \mathbf{throws}\ \chi_1 \to \widehat{\tau}_2\ \mathbf{throws}\ \chi_2)\ \ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_j} \Rightarrow \textsc{exn}, \overline{e_k :: \kappa_k}}\ [\textsc{C-Arr}]$$

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

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

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

Terms

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

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: $\lambda^{\cup}$

$$
\begin{aligned}
(\lambda x : \tau . t_1)\ t_2 &\longrightarrow [t_2 / x]\, t_1 && && (\beta\text{-reduction}) \\
(t_1 \cup t_2)\ t_3 &\longrightarrow t_1\ t_3 \cup t_2\ t_3 && && \\
(\lambda x : \tau . t_1) \cup (\lambda x : \tau . t_2) &\longrightarrow \lambda x : \tau .\, (t_1 \cup t_2) && && (\text{congruences}) \\
x\ t_1 \cdots t_n \cup x'\ t'_1 \cdots t'_n &\longrightarrow x\ \ (t_1 \cup t'_1) \cdots (t_n \cup t'_n) && && \\
(t_1 \cup t_2) \cup t_3 &\longrightarrow t_1 \cup (t_2 \cup t_3) && && (\text{associativity}) \\
\emptyset \cup t &\longrightarrow t && && \\
t \cup \emptyset &\longrightarrow t && && (\text{unit}) \\
x \cup x &\longrightarrow x && && \\
x \cup (x \cup t) &\longrightarrow x \cup t && && \\
\{c\} \cup \{c\} &\longrightarrow \{c\} && && (\text{idempotence}) \\
\{c\} \cup (\{c\} \cup t) &\longrightarrow \{c\} \cup t && && \\
x\ t_1 \cdots t_n \cup \{c\} &\longrightarrow \{c\} \cup x\ t_1 \cdots t_n && && (1) \\
x\ t_1 \cdots t_n \cup (\{c\} \cup t) &\longrightarrow \{c\} \cup (x\ t_1 \cdots t_n \cup t) && && (2) \\
x\ t_1 \cdots t_n \cup x'\ t'_1 \cdots t'_n &\longrightarrow x'\ t'_1 \cdots t'_n \cup x\ t_1 \cdots t_n && \text{if } x' \prec x && (3) \\
x\ t_1 \cdots t_n \cup (x'\ t'_1 \cdots t'_n \cup t) &\longrightarrow x'\ t'_1 \cdots t'_n \cup (x\ t_1 \cdots t_n \cup t) && \text{if } x' \prec x && (4) \\
\{c\} \cup \{c'\} &\longrightarrow \{c'\} \cup \{c\} && \text{if } c' \prec c && (5) \\
\{c\} \cup (\{c'\} \cup t) &\longrightarrow \{c'\} \cup (\{c\} \cup t) && \text{if } c' \prec c && (6)
\end{aligned}
$$

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 languages.
  - 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.

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.