Higher-ranked Exception Types

Work-in-Progress

Ruud Koot

Department of Information and Computing Sciences
Utrecht University

March 26, 2015

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.

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

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 $\tau\langle e\rangle$ for the exception type of a term of type τ that can evaluate to $\bot_\chi$ for some exception label $\chi \in e$.
- The fully annotated exception type for *map* would be:

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

Motivation

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

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

- If you want to be pedantic:

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

Motivation

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

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

- If you want to be pedantic:

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

Motivation

- The exception type

$$(\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_1 \cup e_2\rangle) \to [\alpha\langle e_1\rangle]\langle e_4\rangle \to [\beta\langle e_1 \cup e_2 \cup e_3\rangle]\langle e_4\rangle$$

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

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

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

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

  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

$$\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_1 \cup e_2\rangle,$$

  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:

$$\forall e_2\ e_3.(\forall e_1.\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle) \\ \quad \to (\forall e_4\ e_5.[\alpha\langle e_4\rangle]\langle e_5\rangle \to [\beta\langle e_2\ e_4\ \cup\ e_3\rangle]\langle e_5\rangle)$$

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

Motivation

- Given the following functions:

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

- 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:

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

as desired.

Intermezzo: Simply-typed λ-calculus

Types

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

Terms

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

Values

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

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)

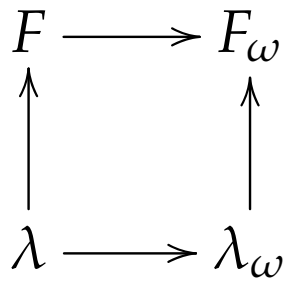
For any term t we have that $t \longrightarrow^* v$ (in a finite number of steps).

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.



- $id \;:\; B \to B$
  $id = \lambda x : B.x$
- $id \;:\; \forall \alpha :: *.\alpha \to \alpha$
  $id = \Lambda \alpha : *.\lambda x : \alpha.x$
- $Id \;::\; * \Rightarrow *$
  $Id = \lambda \alpha :: *.\alpha$
  $id \;:\; B \to Id\ B$
  $id = \lambda x : B.x$
- $Id \;::\; * \Rightarrow *$
  $Id = \lambda \alpha :: *.\alpha$
  $id \;:\; \forall \alpha :: *.\alpha \to Id\ \alpha$
  $id = \Lambda \alpha : *.\lambda x : \alpha.x$

- Omitted: the axis for dependent types.

Intermezzo: System F_ω

Types

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

Kinds

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

Intermezzo: System F_ω

Terms

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

Values

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

Intermezzo: System F_ω

Kinding Note the similarity between the *type* system of the simply typed λ-calculus.

$$\frac{}{\Gamma, \alpha :: \kappa \vdash \alpha :: \kappa}\ [\text{K-Var}] \qquad \frac{\Gamma, \alpha :: \kappa_1 \vdash \tau_2 :: \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau_2 :: \kappa_1 \Rightarrow \kappa_2}\ [\text{K-Abs}]$$

$$\frac{\Gamma \vdash \tau_1 :: \kappa_1 \Rightarrow \kappa_2 \quad \Gamma \vdash \tau_2 :: \kappa_1}{\Gamma \vdash \tau_1\ \tau_2 :: \kappa_2}\ [\text{K-App}]$$

$$\frac{\Gamma \vdash \tau_1 :: * \quad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \to \tau_2 :: *}\ [\text{K-Arrow}] \qquad \frac{\Gamma, \alpha :: \kappa \vdash \tau :: *}{\Gamma \vdash \forall \alpha :: \kappa . \tau :: *}\ [\text{K-Forall}]$$

Intermezzo: System F_ω

Type equivalence

$$\frac{}{\tau \equiv \tau}\ [\text{Q-Refl}] \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1}\ [\text{Q-Symm}]$$

$$\frac{\tau_1 \equiv \tau_2 \quad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}\ [\text{Q-Trans}] \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}\ [\text{Q-Arrow}]$$

$$\frac{\tau_1 \equiv \tau_2}{\forall \alpha :: \kappa.\tau_1 \equiv \forall \alpha :: \kappa.\tau_2}\ [\text{Q-Forall}]$$

$$\frac{\tau_1 \equiv \tau_2}{\lambda \alpha :: \kappa.\tau_1 \equiv \lambda \alpha :: \kappa.\tau_2}\ [\text{Q-Abs}] \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}\ [\text{Q-App}]$$

$$\frac{}{(\lambda \alpha :: \kappa.\tau_1)\ \tau_2 \equiv [\alpha \mapsto \tau_2]\ \tau_1}\ [\text{Q-Beta}]$$

Intermezzo: System F_ω

Typing In addition to the rules for the simply typed λ-calculus:

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

$$\frac{\Gamma, \alpha :: \kappa_1 \vdash t_2 : \tau_2}{\Gamma \vdash \Lambda \alpha :: \kappa_1 . t_2 : \forall \alpha :: \kappa_1 . \tau_2} \text{ [T-TyAbs]}$$

$$\frac{\Gamma \vdash t_1 : \forall \alpha :: \kappa_1 . \tau_1 \quad \Gamma \vdash \tau_2 :: \kappa_1}{\Gamma \vdash t_1 \langle \tau_2 \rangle : [\alpha \mapsto \tau_2] \tau_1} \text{ [T-TyApp]}$$

$$\frac{\Gamma \vdash t : \tau_1 \quad \tau_1 \equiv \tau_2 \quad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash t : \tau_2} \text{ [T-Eq]}$$

Intermezzo: System F_ω

Evaluation In addition to the rules for the simply typed λ-calculus:

$$\frac{t \longrightarrow t'}{\Lambda\alpha : \kappa.t \longrightarrow \Lambda\alpha : \kappa.t'}\ [\text{E-TyAbs}] \qquad \frac{t \longrightarrow t'}{t\ \langle\tau\rangle \longrightarrow t'\ \langle\tau\rangle}\ [\text{E-TyApp}]$$

$$\frac{}{(\Lambda\alpha : \kappa.t)\ \langle\tau\rangle \longrightarrow [\tau/\alpha]\, t}\ [\text{E-TyBeta}]$$

Intermezzo: System F_ω

Metatheory

- We still have *progress*, *preservation* and *decidability* (of type checking).
- Proofs rely on the structure of the types and type equivalence relation and thus the properties (especially *normalization* and *uniqueness of normal forms*) of the simply typed λ-calculus.

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, constants)

$$\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTy} \times \mathbf{Exn}$$

$$\mathcal{R}\ \Gamma\ \Delta\ x \quad = \Gamma_x$$

$$\mathcal{R}\ \Gamma\ \Delta\ c_\tau \ = \langle \bot_\tau ; \varnothing \rangle$$

$$\mathcal{R}\ \Gamma\ \Delta\ \lightning^{\ell}_{\tau} = \langle \bot_\tau ; \{\ell\} \rangle$$

Technicalities: Reconstruction (abstractions)

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTy} \times \mathbf{Exn} \\
\mathcal{R}\ \Gamma\ \Delta\ (\lambda x : \tau.t) = \\
\quad \mathbf{let}\ \langle \widehat{\tau}_1; e_1; \overline{e_i : \kappa_i} \rangle = \mathcal{C}\ \varnothing\ \tau \\
\qquad\quad \langle \widehat{\tau}_2; \varphi_2 \rangle \qquad\quad = \mathcal{R}\ (\Gamma, x : \widehat{\tau}_1\ \&\ e_1)\ (\Delta, \overline{e_i : \kappa_i})\ t \\
\quad \mathbf{in}\ \ \langle \forall \overline{e_i : \kappa_i}.\widehat{\tau}_1 \langle e_1 \rangle \to \widehat{\tau}_2 \langle \varphi_2 \rangle; \varnothing \rangle
\end{array}
$$

Technicalities: Completion

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

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

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

$$\frac{\vdash \tau_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \triangleright \overline{e_j :: \kappa_j} \quad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau}_2 \;\&\; \varphi_2 \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \to \tau_2 : \forall \overline{e_j :: \kappa_j}.\, (\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle) \;\&\; e\ \overline{e_i} \triangleright e :: \overline{\kappa_j \Rightarrow \mathrm{EXN}}, \overline{e_k :: \kappa_k}} \text{ [C-Arr]}$$

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

Technicalities: Completion

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

Technicalities: Completion

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

Technicalities: Completion

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

Technicalities: Completion

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

Technicalities: Completion

- $(\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool}$
- $$\forall e_2 :: \textsc{exn} \Rightarrow \textsc{exn}.\ \forall e_3 :: \textsc{exn}.\ \left(\forall e_1 :: \textsc{exn}.\ \widehat{\mathrm{bool}}\langle e_1\rangle \xrightarrow{e_3} \widehat{\mathrm{bool}}\langle e_2\ e_1\rangle\right) \to \widehat{\mathrm{bool}}\langle e_4\ e_2\ e_3\rangle\ \&\ e_5$$
- $e_4 :: (\textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}, e_5 :: \textsc{exn}$

Technicalities: Reconstruction (applications)

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTy} \times \mathbf{Exn} \\
\mathcal{R}\ \Gamma\ \Delta\ (t_1\ t_2) = \\
\quad \mathbf{let}\ \langle \widehat{\tau}_1; \varphi_1 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\tau}_2; \varphi_2 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\qquad \langle \widehat{\tau}_2' \langle e_2' \rangle \to \widehat{\tau}' \langle \varphi' \rangle; \overline{e_i : \kappa_i} \rangle = \mathcal{I}\ \widehat{\tau}_1 \\
\qquad \theta = [e_2' \mapsto \varphi_2] \circ \mathcal{M}\ \varnothing\ \widehat{\tau}_2\ \widehat{\tau}_2' \\
\quad \mathbf{in}\ \langle \llbracket \theta \widehat{\tau}' \rrbracket_\Delta; \llbracket \theta \varphi' \cup \varphi_1 \rrbracket_\Delta \rangle
\end{array}
$$

Technicalities: Matching

$$
\begin{array}{lll}
\mathcal{M} :: \mathbf{Env} \to \mathbf{Ty} \to \mathbf{Ty} \to \mathbf{Subst} & & \\
\mathcal{M}\ \Sigma\ \widehat{\mathrm{bool}} & \widehat{\mathrm{bool}} & = Id \\
\mathcal{M}\ \Sigma\ (\forall e :: \kappa.\widehat{\tau}_1) & (\forall e :: \kappa.\widehat{\tau}'_1) & = \mathcal{M}\ (\Sigma, e \mapsto \kappa)\ \widehat{\tau}_1\ \widehat{\tau}'_1 \\
\mathcal{M}\ \Sigma\ (\widehat{\tau}_1^{e_1} \to \widehat{\tau}_2^{\chi_2})\ (\widehat{\tau}_1^{e_1} \to \widehat{\tau}_2'^{(e_0\ \overline{e_j})}) & & = \\
\quad \left[e_0 \mapsto (\overline{\lambda e_j :: \Sigma(e_j).\ \chi_2})\right] \circ \mathcal{M}\ \Sigma\ \widehat{\tau}_2\ \widehat{\tau}'_2 & & \\
\mathcal{M}\ \Sigma\ \_ & \_ & = \mathbf{fail}
\end{array}
$$

Technicalities: Matching — Example

- $\mathcal{M}\ [e_1 :: \mathbf{EXN}, e_2 :: \mathbf{EXN} \Rightarrow \mathbf{EXN}, e_3 :: \mathbf{EXN}]$
  $(\widehat{\mathrm{bool}}\langle e_1 \rangle \rightarrow \widehat{\mathrm{bool}}\langle e_2\ e_1\ \cup\ e_3 \rangle)$
  $(\widehat{\mathrm{bool}}\langle e_1 \rangle \rightarrow \widehat{\mathrm{bool}}\langle e_0\ e_1\ e_2\ e_3 \rangle)$

Technicalities: Matching — Example

- $\mathcal{M}\ [e_1 :: \mathbf{EXN}, e_2 :: \mathbf{EXN} \Rightarrow \mathbf{EXN}, e_3 :: \mathbf{EXN}]$
  $(\widehat{\mathrm{bool}}\langle e_1 \rangle \rightarrow \widehat{\mathrm{bool}}\langle e_2\ e_1\ \cup\ e_3 \rangle)$
  $(\widehat{\mathrm{bool}}\langle e_1 \rangle \rightarrow \widehat{\mathrm{bool}}\langle e_0\ e_1\ e_2\ e_3 \rangle)$

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

Technicalities: Reconstruction (fixpoints)

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTy} \times \mathbf{Exn} \\
\mathcal{R}\ \Gamma\ \Delta\ (\mathbf{fix}\ t) = \\
\quad \mathbf{let}\ \langle \widehat{\tau}; \varphi \rangle \qquad\qquad\qquad\qquad\qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t \\
\qquad \langle \widehat{\tau}' \langle e' \rangle \to \widehat{\tau}'' \langle \varphi'' \rangle; \overline{e_i : \kappa_i} \rangle = \mathcal{I}\ \widehat{\tau} \\
\quad \mathbf{in}\ \ \langle \widehat{\tau}_0; \varphi_0; i \rangle \leftarrow \langle \bot_{\lfloor \widehat{\tau}' \rfloor}; \emptyset; 0 \rangle \\
\qquad \mathbf{do}\ \theta \qquad\qquad\qquad\qquad \leftarrow [e' \mapsto \varphi_i] \circ \mathcal{M}\ \emptyset\ \widehat{\tau}_i\ \widehat{\tau}' \\
\qquad\quad \langle \widehat{\tau}_{i+1}; \varphi_{i+1}; i \rangle \leftarrow \langle [\![\theta \widehat{\tau}'']\!]_\Delta; [\![\theta \varphi'']\!]_\Delta; i+1 \rangle \\
\qquad \mathbf{until}\ \langle \widehat{\tau}_i; \varphi_i \rangle \equiv \langle \widehat{\tau}_{i-1}; \varphi_{i-1} \rangle \\
\qquad \mathbf{return}\ \langle \widehat{\tau}_i; [\![\varphi \cup \varphi_i]\!]_\Delta \rangle
\end{array}
$$

Technicalities: $\lambda^{\cup}$

Types

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

Terms

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

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: $\lambda^{\cup}$

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

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:

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

- 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:

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

- 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

Paper and prototype

A prototype implementation of the type inference algorithm and an early draft of the paper are available from:

- `https://github.com/ruudkoot/phd`

Related work

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.