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:

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

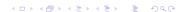- What should be the correct value of e?

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.

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

Motivation

- Instead of τ **throws** e, write τ^e for a type τ that can evaluate to $\perp_\chi$ for some $\chi \in e$.

- The fully annotated exception type for *map* would be:

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

Motivation

- Instead of τ **throws** e, write τ^e for a type τ that can evaluate to $\perp_\chi$ for some $\chi \in e$.

- The fully annotated exception type for *map* would be:

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

- If you want to be pedantic:

$$map :: \forall \alpha\ \beta\ e_1\ e_2\ e_3\ e_4.$$
$$((\alpha^{e_1} \to \beta^{(e_1 \cup e_2)})^{e_3} \to ([\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4})^{\varnothing})^{\varnothing}$$

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:

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

- If you want to be pedantic:

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

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:

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

- 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 EXN $\rightarrow$ EXN.

Motivation

- Given the following functions:

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

$$id \quad :: \forall e.\alpha^e \xrightarrow{\varnothing} \alpha^e$$

$$const\ \bot_{\mathbf{E}} :: \forall e.\alpha^e \xrightarrow{\varnothing} \beta^{\{\mathbf{E}\}}$$

- 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:

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

as desired.

Intermezzo: Simply-typed λ-calculus

Types

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

Terms

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

Values

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

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.

$$
\begin{array}{ccc}
F & \longrightarrow & F_\omega \\
\uparrow & & \uparrow \\
\lambda & \longrightarrow & \lambda_\omega
\end{array}
$$

- $id \; : \; B \to B$
  $id = \lambda x : B.x$

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

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

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

- Omitted: the axis for dependent types.

Intermezzo: System F_ω

Types

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

Kinds

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

Intermezzo: System F_ω

Terms

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

Values

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

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: Reconstruction (variables)

reconstruct $\widehat{\Gamma}\ x =$
   let $(\widehat{\tau}, \chi) = \widehat{\Gamma}\ (x)$
      e be fresh
   in $(\widehat{\tau}, e, \{\chi \subseteq e\})$

Technicalities: Reconstruction (abstractions)

$$\begin{aligned}
&\textit{reconstruct } \widehat{\Gamma} \ (\lambda x : \tau.t) = \\
&\quad \textbf{let } (\widehat{\tau_1}, \overline{e_i :: \kappa_i}) = \textit{complete } \tau \ \varnothing \\
&\qquad e_1 \textit{ be fresh} \\
&\qquad (\widehat{\tau_2}, e_2, C_1) = \textit{reconstruct } (\widehat{\Gamma}, x \mapsto (\widehat{\tau_1}, e_1)) \ t \\
&\qquad X = \{e_1\} \cup \{\overline{e_i}\} \cup \textit{fv } \widehat{\Gamma} \\
&\qquad \chi_2 = \textit{solve } C_1 \ X \ e_2 \\
&\qquad \widehat{\tau} = \forall e_1 :: \textsc{exn}.\overline{\forall e_i :: \kappa_i}. \ \widehat{\tau_1}^{e_1} \to \widehat{\tau_2}^{\chi_2} \\
&\qquad e \textit{ be fresh} \\
&\quad \textbf{in } (\widehat{\tau}, e, \varnothing)
\end{aligned}$$

Technicalities: Completion

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

$$\overline{e_i :: \kappa_i} \vdash \textbf{bool} : \widehat{\textbf{bool}} \,\&\, e \, \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow_{\textbf{EXN}} \quad \text{[C-Bool]}$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau} \,\&\, \chi \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau} \,\textbf{throws}\, \chi] \,\&\, e \, \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow_{\textbf{EXN}}, \overline{e_j :: \kappa_j}} \quad \text{[C-List]}$$

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

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

Technicalities: Completion

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

Technicalities: Completion

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

Technicalities: Completion

- **bool** $\rightarrow$ **bool**
- $\forall e_1 :: \text{EXN}. \, \widehat{\text{bool}}^{e_1} \rightarrow \widehat{\text{bool}}^{(e_2 \; e_1)} \; \& \; e_3$
- $e_2 :: \text{EXN} \Rightarrow \text{EXN}, e_3 :: \text{EXN}$

Technicalities: Completion

- **bool** $\rightarrow$ **bool** $\rightarrow$ **bool**
-

    $\forall e_1 ::$ EXN. $\widehat{\text{bool}}^{e_1} \rightarrow$
    $$(\forall e_4 :: \text{EXN. } \widehat{\text{bool}}^{e_4} \xrightarrow{e_2 \ e_1} \widehat{\text{bool}}^{(e_5 \ e_1 \ e_4)}) \ \& \ e_3$$

- $e_2 ::$ EXN $\Rightarrow$ EXN, $e_3 ::$ EXN, $e_5 ::$ EXN $\Rightarrow$ EXN $\Rightarrow$ EXN

Technicalities: Completion

- $(\textbf{bool} \to \textbf{bool}) \to \textbf{bool}$
-
    $$\forall e_2 :: \textsc{exn} \Rightarrow \textsc{exn}. \ \forall e_3 :: \textsc{exn}.$$
    $$\left(\forall e_1 :: \textsc{exn}. \ \widehat{\textrm{bool}}^{e_1} \xrightarrow{e_3} \widehat{\textrm{bool}}^{(e_2 \ e_1)} \right) \to \widehat{\textrm{bool}}^{(e_4 \ e_2 \ e_3)} \ \& \ e_5$$

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

$$reconstruct \; \widehat{\Gamma} \; (t_1 \; t_2) =$$
$$\textbf{let} \; (\widehat{\tau}_1, e_1, C_1) = reconstruct \; \widehat{\Gamma} \; t_1$$
$$(\widehat{\tau}_2, e_2, C_2) = reconstruct \; \widehat{\Gamma} \; t_2$$
$$\widehat{\tau}_2^{\prime e_2^{\prime}} \to \widehat{\tau}'^{\chi'} = instantiate \; \widehat{\tau}_1$$
$$\theta = [e_2^{\prime} \mapsto e_2] \circ match \; \varnothing \; \widehat{\tau}_2 \; \widehat{\tau}_2^{\prime}$$
$$e \; be \; fresh$$
$$C = \{...\}$$
$$\textbf{in} \; (\widehat{\tau}, e, C)$$

Technicalities: Matching

$$match :: \textbf{Env} \rightarrow \textbf{Ty} \rightarrow \textbf{Ty} \rightarrow \textbf{Subst}$$
$$match \; \Sigma \; \widehat{\text{bool}} \qquad \widehat{\text{bool}} \qquad = Id$$
$$match \; \Sigma \; (\forall e :: \kappa.\widehat{\tau}_1) \quad (\forall e :: \kappa.\widehat{\tau}_1') \qquad = match \; (\Sigma, e \mapsto \kappa) \; \widehat{\tau}_1 \; \widehat{\tau}_1'$$
$$match \; \Sigma \; (\widehat{\tau}_1^{e_1} \rightarrow \widehat{\tau}_2^{\chi_2}) \; (\widehat{\tau}_1^{e_1} \rightarrow \widehat{\tau}_2'^{(e_0 \; \overline{e_j})}) =$$
$$\left[e_0 \mapsto (\overline{\lambda e_j :: \Sigma(e_j). \; \chi_2}) \right] \circ match \; \Sigma \; \widehat{\tau}_2 \; \widehat{\tau}_2'$$
$$match \; \Sigma \; \_ \qquad\qquad \_ \qquad\qquad = \textbf{fail}$$

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

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

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

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

$$\tau \in \mathbf{Ty} \qquad ::= \quad \mathcal{P} \qquad \qquad \text{(base type)}$$
$$| \quad \tau_1 \to \tau_2 \qquad \text{(function type)}$$

Terms

$$t \in \mathbf{Tm} \qquad ::= \quad x, y, \dots \qquad \qquad \text{(variable)}$$
$$| \quad \lambda x : \tau.t \qquad \text{(abstraction)}$$
$$| \quad t_1 \; t_2 \qquad \qquad \text{(application)}$$
$$| \quad \varnothing \qquad \qquad \text{(empty)}$$
$$| \quad \{c\} \qquad \qquad \text{(singleton)}$$
$$| \quad t_1 \cup t_2 \qquad \qquad \text{(union)}$$

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: $\lambda^\cup$

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

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:

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

- This does not hold if we have observable exceptions and track them precisely.

  - Counterexample: Take $e_1 = \perp_{\textbf{E}_1}$ and $e_2 = \perp_{\textbf{E}_2}$.

- 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:

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

- This does not hold if we have observable exceptions and track them precisely.
  - Counterexample: Take $e_1 = \perp_{E_1}$ and $e_2 = \perp_{E_2}$.

- 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.