Type-based Exception Analysis

for Non-strict Higher-order Functional Languages with Imprecise Exception Semantics

Ruud Koot Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

January 14, 2015

Motivation

- "Well-typed programs do not go wrong."

Motivation

- "Well-typed programs do not go wrong."
- Except:
  - $\mathit{reciprocal}\ x = 1\ /\ x$
  - $\mathit{head}\ (x : xs) = x$
  - ...
- Practical programming languages allow functions to be *partial*.

Motivation

- Requiring all functions to be total may be undesirable.
  - Dependent types are heavy-weight.
  - Running everything in the *Maybe* monad does not solve the problem, only moves it.
  - Some partial functions are *benign*.
- We do want to warn the programmer something may go wrong at run-time.

Motivation

- Currently compilers do a local and syntactic analysis.

$$\begin{array}{l} head :: [\alpha] \rightarrow \alpha \\ head\ xs = \mathbf{case}\ xs\ \mathbf{of}\ \{(y : ys) \rightarrow y\} \end{array}$$

Motivation

- Currently compilers do a local and syntactic analysis.

$$\mathit{head} :: [\alpha] \to \alpha$$
$$\mathit{head}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{(y : ys) \to y\}$$

- "The problem is in *head* and *every* place you call it!"

$$\mathit{main} = \mathit{head}\ [1, 2, 3]$$

- Worse are non-escaping local definitions.

Motivation

- The canonical example by Mitchell & Runciman (2008):

$$
\begin{array}{l}
\mathit{risers} :: \mathit{Ord}\ \alpha \Rightarrow [\alpha] \to [[\alpha]] \\
\mathit{risers}\ [\,] \qquad\qquad = [\,] \\
\mathit{risers}\ [x] \qquad\quad\ = [[x]] \\
\mathit{risers}\ (x_1 : x_2 : \mathit{xs}) = \\
\quad \textbf{if}\ x_1 \leqslant x_2\ \textbf{then}\ (x_1 : y) : \mathit{ys}\ \textbf{else}\ [x_1] : (y : \mathit{ys}) \\
\qquad \textbf{where}\ (y : \mathit{ys}) = \mathit{risers}\ (x_2 : \mathit{xs})
\end{array}
$$

- Computes monotonically increasing subsegments of a list:

$$\mathit{risers}\ [1,3,5,1,2] \longrightarrow^{*} [[1,3,5],[1,2]]$$

- Program invariants can ensure incomplete pattern matches never fail.

Motivation

- Instead use a semantic approach: “where can exceptions flow to?”

$$
\begin{aligned}
&\mathit{head} :: [\,\alpha\,] \to \alpha \\
&\mathit{head}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{\,[\,] \to \lightning\,;(y : \mathit{ys}) \to y\,\}
\end{aligned}
$$

- Simultaneously need to track data flow to determine which branches are not taken.
- Using a type-and-effect system, the analysis is still modular.

Basic idea: data flow

- We can then assign to each of the three individual branches of *risers* the following types:

$$
\begin{aligned}
&\mathit{risers}_1 && :: \forall\alpha.\mathit{Ord}\ \alpha \Rightarrow [\alpha]^{\mathbf{N}} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{N}} \\
&\mathit{risers}_2, \mathit{risers}_3 && :: \forall\alpha.\mathit{Ord}\ \alpha \Rightarrow [\alpha]^{\mathbf{C}} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{C}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{risers}\ :: \mathit{Ord}\ \alpha \Rightarrow [\alpha] \to [[\alpha]] \\
&\mathit{risers}\ [\,] && = [\,] \\
&\mathit{risers}\ [x] && = [[x]] \\
&\mathit{risers}\ (x_1 : x_2 : xs) = \\
&\quad \mathbf{if}\ x_1 \leqslant x_2\ \mathbf{then}\ (x_1 : y) : ys\ \mathbf{else}\ [x_1] : (y : ys) \\
&\qquad \mathbf{where}\ (y : ys) = \mathit{risers}\ (x_2 : xs)
\end{aligned}
$$

Basic idea: data flow

- We can then assign to each of the three individual branches of *risers* the following types:

$$\begin{array}{ll} risers_1 & :: \forall \alpha . Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{N}} \rightarrow [[\alpha]^{\mathbf{N} \sqcup \mathbf{C}}]^{\mathbf{N}} \\ risers_2, risers_3 & :: \forall \alpha . Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{C}} \rightarrow [[\alpha]^{\mathbf{N} \sqcup \mathbf{C}}]^{\mathbf{C}} \end{array}$$

- From the three individual branches we may infer:

$$risers :: \forall \alpha . Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{N} \sqcup \mathbf{C}} \rightarrow [[\alpha]^{\mathbf{N} \sqcup \mathbf{C}}]^{\mathbf{N} \sqcup \mathbf{C}}$$

- Adding *polyvariance* gives us a more precise result:

$$risers :: \forall \alpha \beta . Ord\ \alpha \Rightarrow [\alpha]^{\beta} \rightarrow [[\alpha]^{\mathbf{N} \sqcup \mathbf{C}}]^{\beta}$$

Basic idea: exception flow

- A compiler will translate the partial function *head* into:

$$\begin{aligned} \mathit{head}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} & \\ & [\,] \mapsto \lightning^{\mathbf{pattern\text{-}match\text{-}failure}} \\ & (y : \mathit{ys}) \mapsto y \end{aligned}$$

- which can be assigned the exception type:

$$\mathit{head} :: \forall \tau \alpha \beta . [\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha \sqcup \beta \sqcup \mathbf{pattern\text{-}match\text{-}failure}}$$

Basic idea: exception flow

- A compiler will translate the partial function *head* into:

$$\begin{array}{l} head\ xs = \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad\qquad [\,] \qquad\ \ \mapsto \lightning^{\mathbf{pattern\text{-}match\text{-}failure}} \\ \qquad\qquad\qquad (y : ys) \mapsto y \end{array}$$

- which can be assigned the exception type:

$$head :: \forall \tau \alpha \beta . [\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha \sqcup \beta \sqcup \mathbf{pattern\text{-}match\text{-}failure}}$$

- This type tells us that *head* might always raise a **pattern-match-failure** exception!
- Introduce a dependency of the exception flow on the data flow of the program:

$$head :: \forall \tau \alpha \beta \gamma . [\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha \sqcup \beta \sqcup \gamma}$$
$$\mathbf{with}\ \{\mathbf{N} \sqsubseteq \beta \Rightarrow \mathbf{pattern\text{-}match\text{-}failure} \sqsubseteq \gamma\}$$

Imprecise exception semantics

- Non-strict languages can have an *imprecise exception semantics* (Peyton Jones *et al*, 1999).
  - Can non-deterministically raise one from a set of exceptions.
  - Necessary for the soundness of certain program transformations, e.g. the case-switching transformation:

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

Imprecise exception semantics

- If an exception can be raised by pattern matching, we need to continue evaluating all branches.
- Implication for the analysis: cannot separate data and exception flow phases.

Algorithm

- Assumes program is well-typed in underlying type system.
- Algorithm $\mathcal{W}$-like constraint generation phase.
- Worklist/fixpoint-based constraint solver.

Types and constraints

- Annotated types:

$$\tau \quad ::= \quad \alpha \quad | \quad \tau_1 \xrightarrow{\alpha} \tau_2 \quad | \quad \tau_1 \times^{\alpha} \tau_2 \quad | \quad [\tau]^{\alpha}$$

- *Conditional constraints* and *indices* model dependence between data flow and exception flow:

$$\begin{array}{lcl} c & ::= & g \Rightarrow r \\ g & ::= & \Lambda_\iota \sqsubseteq_\iota \alpha \quad | \quad \exists_\iota \alpha \quad | \quad g_1 \vee g_2 \quad | \quad \mathbf{true} \\ r & ::= & \Lambda_\iota \sqsubseteq_\iota \alpha \quad | \quad \alpha_1 \sqsubseteq_\iota \alpha_2 \quad | \quad \tau_1 \leqslant_\iota \tau_2 \end{array}$$

- Asymmetric to keep solving tractable.

A type rule (case-expressions)

$$\frac{\begin{array}{c} C;\Gamma \vdash e_1 : [\tau_1]^{\alpha_1} \quad C;\Gamma \vdash e_2 : \tau_2 \\ C;\Gamma, x_1 : \tau_1, x_2 : [\tau_1]^{\beta} \vdash e_3 : \tau_3 \\ C \Vdash \mathbf{N} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi \alpha_1 \Rightarrow \tau_2 \leqslant_{\delta\chi} \tau \\ C \Vdash \mathbf{C} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi \alpha_1 \Rightarrow \tau_3 \leqslant_{\delta\chi} \tau \\ C \Vdash \alpha_1 \sqsubseteq_\chi \lceil \tau \rceil \quad C \Vdash \mathbf{N} \sqcup \mathbf{C} \sqsubseteq_\delta \beta \quad C \Vdash \alpha_1 \sqsubseteq_\chi \beta \end{array}}{C;\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} : \tau} \text{[T-CASE]}$$

Operators

- Operators have a constraint set corresponding to their abstract interpretation:

$$\omega_{\div}(\alpha_1, \alpha_2, \alpha) \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \mathbf{0} \sqsubseteq_\delta \alpha_2 \Rightarrow \{\textbf{div-by-0}\} \sqsubseteq_\chi \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_1 \wedge \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_1 \wedge \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \wedge \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \wedge \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \mathbf{0} \sqsubseteq_\delta \alpha \end{array} \right\}$$

- Can make a trade-off between precision and accuracy.

Operators

Definition

An operator constraint set $\omega_\oplus$ is said to be *consistent* with respect to an operator interpretation $[\![\cdot \oplus \cdot]\!]$ if, whenever $C; \Gamma \vdash n_1 : \alpha_1$, $C; \Gamma \vdash n_2 : \alpha_2$, and $C \Vdash \omega_\oplus(\alpha_1, \alpha_2, \alpha)$ then $C; \Gamma \vdash [\![n_1 \oplus n_2]\!] : \alpha'$ with $C \Vdash \alpha' \leqslant_{\delta\chi} \alpha$ for some α'.

Definition

An operator constraint set $\omega_\oplus$ is *monotonic* if, whenever $C \Vdash \omega_\oplus(\alpha_1, \alpha_2, \alpha)$ and $C \Vdash \alpha_1' \sqsubseteq_{\delta\chi} \alpha_1$, $C \Vdash \alpha_2' \sqsubseteq_{\delta\chi} \alpha_2$, $C \Vdash \alpha \sqsubseteq_{\delta\chi} \alpha'$ then $C \Vdash \omega_\oplus(\alpha_1', \alpha_2', \alpha')$.

Metatheory

Theorem (Conservative extension)

If e is well-typed in the underlying type system, then it can be given a type in the annotated type system.

Theorem (Progress)

If $C; \Gamma \vdash e : \sigma$ then either e is a value or there exist an e', such that for any ρ with $C \vdash \Gamma \bowtie \rho$ we have $\rho \ \vdash \ e \longrightarrow \ e'$.

Theorem (Preservation)

If $C; \Gamma \vdash e : \sigma_1$, $\rho \ \vdash \ e \longrightarrow \ e'$ and $C \vdash \Gamma \bowtie \rho$ then $C; \Gamma \vdash e' : \sigma_2$ with $C \Vdash \sigma_2 \leqslant_{\delta\chi} \sigma_1$.

Polyvariant recursion

- To precisely type *risers* (i.e. infer that no exception can be raised if not already present in input) we need *polyvariant recursion* (i.e. polymorphic recursion restricted to annotations).
- This poses algorithmic difficulties w.r.t. termination.
  - Variable elimination technique by Dussart, Henglein & Mossin (1995) does not work due to conditional constraints.
  - Restricting the number of fresh variables generated makes the analysis fairly unpredictable.
  - Terminating before a fixpoint has been reached invalidates the soundness result, but may not be a problem in practice.
- The *ad hoc* nature of the constraint language comes back to bite us; currently looking at more well-behaved constraint languages (Boolean rings).

Questions?