

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:
 - ▶ *reciprocal* $x = 1 / x$
 - ▶ *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.

$head :: [\alpha] \rightarrow \alpha$

$head\ xs = \mathbf{case}\ xs\ \mathbf{of}\ \{ (y : ys) \rightarrow y \}$

Motivation

- ▶ Currently compilers do a local and syntactic analysis.

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

- ▶ “The problem is in *head* and *every* place you call it!”

$$main = head\ [1,2,3]$$

- ▶ Worse are non-escaping local definitions.

Motivation

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

$$\begin{aligned} \text{risers} &:: \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [[\alpha]] \\ \text{risers } [] &= [] \\ \text{risers } [x] &= [[x]] \\ \text{risers } (x_1 : x_2 : xs) &= \\ &\quad \text{if } x_1 \leq x_2 \text{ then } (x_1 : y) : ys \text{ else } [x_1] : (y : ys) \\ &\quad \text{where } (y : ys) = \text{risers } (x_2 : xs) \end{aligned}$$

- ▶ Computes monotonically increasing subsegments of a list:

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

- ▶ Program invariants can ensure incomplete pattern matches never fail.

Motivation

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

$$head :: [\alpha] \rightarrow \alpha$$
$$head\ xs = \mathbf{case}\ xs\ \mathbf{of}\ \{ [] \rightarrow \bot; (y : ys) \rightarrow y \}$$

- ▶ 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:

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

$$\begin{aligned} \text{risers} &:: \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [[\alpha]] \\ \text{risers } [] &= [] \\ \text{risers } [x] &= [[x]] \\ \text{risers } (x_1 : x_2 : xs) &= \\ &\quad \text{if } x_1 \leq x_2 \text{ then } (x_1 : y) : ys \text{ else } [x_1] : (y : ys) \\ &\quad \text{where } (y : ys) = \text{risers } (x_2 : xs) \end{aligned}$$

Basic idea: data flow

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

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

- ▶ From the three individual branches we may infer:

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

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

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

Basic idea: exception flow

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

$$\begin{aligned} \text{head } xs &= \mathbf{case } xs \mathbf{ of} \\ &\quad [] \quad \mapsto \not\downarrow \mathbf{pattern-match-failure} \\ &\quad (y : ys) \mapsto y \end{aligned}$$

- ▶ which can be assigned the exception type:

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

Basic idea: exception flow

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

$$\begin{aligned} \text{head } xs &= \mathbf{case} \text{ } xs \mathbf{ of} \\ &\quad [] \quad \mapsto \not\downarrow \mathbf{pattern-match-failure} \\ &\quad (y : ys) \mapsto y \end{aligned}$$

- ▶ which can be assigned the exception type:

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

- ▶ 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:

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

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:

$\forall e_i.$ **if** e_1 **then**
 if e_2 **then** e_3 **else** e_4
else
 if e_2 **then** e_5 **else** $e_6 =$ **if** e_2 **then**
 if e_1 **then** e_3 **else** e_5
 else
 if e_1 **then** e_4 **else** e_6

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.

Constraint language

- ▶ *Conditional constraints* and *indices* model dependence between data flow and exception flow
- ▶ Asymmetric to keep solving tractable

$$c ::= g \Rightarrow r$$

$$g ::= \Lambda_l \sqsubseteq_l \alpha \mid \exists_l \alpha \mid g_1 \vee g_2 \mid \mathbf{true}$$

$$r ::= \Lambda_l \sqsubseteq_l \alpha \mid \alpha_1 \sqsubseteq_l \alpha_2 \mid \tau_1 \leqslant_l \tau_2$$

A type rule (case-expressions)

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

Questions?

Operators

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

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

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

Operators

Definition

An operator constraint set ω_{\oplus} is said to be *consistent* with respect to an operator interpretation $\llbracket \cdot \oplus \cdot \rrbracket$ 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 $C; \Gamma \vdash \llbracket n_1 \oplus n_2 \rrbracket : \alpha'$ with $C \Vdash \alpha' \leq_{\delta\chi} \alpha$ for some α' .

Definition

An operator constraint set ω_{\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 $C \Vdash \sigma_2 \leq_{\delta\chi} \sigma_1$.

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).