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:
  - $divideByZero\ x = x\ /\ 0$
  - $head\ (x :: xs) = x$
  - ...
- 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.

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

Motivation

- Currently compilers do a local and syntactic analysis.

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

- "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):

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

- Program invariants can ensure incomplete pattern matches never fail.

Motivation

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

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

- 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

Imprecise exception semantics

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

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

Imprecise exception semantics

- If the scrutinee of a ... excpetion-finding mode
- implication for the analysis: cannot separate data and expcetion flow phases