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

(Abstract of POPL '14 submission)

Ruud Koot Jurriaan Hage

Department of Computing and Information Sciences
Utrecht University
{r.koot,j.hage}@uu.nl

Abstract

Most statically typed functional programming languages allow programmers to write partial functions: functions that are not defined on all the elements of the domain they are claimed to work on by their type. Thus in practise, well-typed programs can—and *do*—still go wrong. A compiler should warn the programmer about the places in the program where such wrongness may occur.

Contemporary compilers for functional languages employ a local and purely syntactic analysis to detect incomplete **case**-expressions—those that do not cover all possible patterns of constructors allowed for by the type of the scrutinee, the source of most partiality in a program. As programs often maintain invariants on their data—restricting the potential values of the scrutinee to a subtype of its given or inferred type—many of these incomplete **case**-expressions are in actuality completely benign. Such an analysis will thus report many false positives, overwhelming the programmer.

We develop and prove the correctness of a constraint-based type system that detects harmful sources of partiality by accurately tracing the flow of both exceptions—the manifestation of partiality gone wrong—and ordinary data through the program, as well as the dependencies between them. The latter is crucial for usable precision, but has been omitted from previously published exception analyses.

Our contributions include the following:

- We develop a type-driven—and thus *modular*—exception analysis that tracks data flow in order to give accurate warnings about exceptions raised due to pattern-match failures.
- Accuracy is achieved through the simultaneous use of *subtyping* (modelling data flow), *conditional constraints* (modelling control flow), *parametric polyvariance* (to achieve context-sensitivity) and *polyvariant recursion* (to avoid poisoning).
- The analysis works for *call-by-name* languages with an *imprecise exception semantics*. Such a semantics is necessary to justify several program transformations applied by optimizing compilers for call-by-name languages with distinguishable exceptions
- We give an operational semantics for imprecise exceptions and prove the analysis sound with respect to this semantics.
- The analysis presented in this paper is implemented in a prototype and, in addition to a pen-and-paper proof, the metatheory has been mostly mechanized.

1. Motivation

Many algorithms maintain invariants on the data structures they use, that cannot easily be encoded into their types. These invariants often ensure that certain incomplete **case**-expressions are guaranteed not to cause a pattern-match failure.

An example of such an algorithm is the $risers$ function from (Mitchell and Runciman 2008), computing monotonically increasing subsegments of a list:

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

For example:

$$risers\ [1,3,5,1,2] \longrightarrow^* [[1,3,5],[1,2]].$$

The irrefutable pattern in the **where**-clause in the third alternative of $risers$ expects the recursive call to return a non-empty list. A naive analysis might raise a warning here. If we think a bit longer, however, we see that we also pass the recursive call to $risers$ a non-empty list. This means we will end up in either the second or third alternative inside the recursive call. Both the second alternative and both branches of the **if**-expression in the third alternative produce a non-empty list, satisfying the assumption we made earlier and allowing us to conclude that this functional is total and will never raise a pattern-match failure exception. (Raising or propagating an exception because we pattern-match on exceptional values present in the input still belongs to the possibilities, though.)

2. Informal overview

We formulate our analysis in terms of a constraint-based type and effect system (Talpin and Jouvelot 1994; Abadi et al. 1999).

2.1 Data flow

How would we capture the informal reasoning we used in Section 1 to convince ourselves that $risers$ does not cause a pattern-match failure using a type system? A reasonable first approach would be to annotate all list types with the kind of list it can be: **N** if it must be an empty lists, a lists that that necessary has a nil-constructor at the head of its spine; **C** if it must be a non-empty list having a cons-constructor at its head; $\mathbf{N} \sqcup \mathbf{C}$ if it can be either. We can

then assign to each of the three individual branches of $risers$ the following types:

$$risers_1 : \forall\alpha.Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{N}} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{N}}$$
$$risers_2 : \forall\alpha.Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{C}} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{C}}$$
$$risers_3 : \forall\alpha.Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{C}} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{C}}$$

From the three individual branches we may infer:

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

Assigning this type to $risers$ will still let us believe that a pattern-match failure may occur in the irrefutable pattern in the **where**-clause, as this type tells us any invocation of $risers$—including the recursive call in the **where**-clause—may evaluate to an empty list. The problem is that $risers_1$—the branch that can never be reached from the call in the **where**-clause—is *poisoning* the overall result. *Polyvariance* (or *property polymorphism*) can rescue us from this precarious situation, however. We can instead assign to each of the branches, and thereby the overall result, the type:

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

In the recursive call to $risers$ we know the argument passed is a non-empty list, so we can instantiate β to **C**, informing us that the result of the recursive call will be a non-empty list as well and guaranteeing that the irrefutable pattern-match will succeed. There is one little subtlety here, though: in a conventional Hindley–Milner type system we are not allowed, or even able, to instantiate β to anything, as the type is kept monomorphic for recursive calls. We, therefore, have to extend our type system with *polyvariant recursion*. While inferring polymorphic recursive types is undecidable in general (Kfoury et al. 1993; Henglein 1993)—and, being a program analysis, we cannot rely on any programmer-supplied annotations—earlier research (Tofte and Talpin 1994; Dussart et al. 1995; Rittri 1995; Leroy and Pessaux 2000) has shown that this special case of polyvariant recursion is often both crucial to obtain adequate precision and feasible to infer automatically.

2.2 Exception flow

The intentention of our analysis it to track the exceptions that may be raised during the execution of a program. As with the data flow we express this set of exceptions as an annotation on the type of a program. For example, the program:

$$f\ x = x \div 0$$

should be given the exception type:

$$f : \forall\alpha.\mathbb{Z}^{\alpha} \xrightarrow{\emptyset} \mathbb{Z}^{\alpha\sqcup\mathbf{div\text{-}by\text{-}zero}}$$

This type explains that f is a function accepting an integer as its first and only parameter. As we are working in a call-by-name language, this integer might actually still be a thunk that raises an exception from the set α when evaluated. The program then divides this argument by zero, returning the result. While the result will be of type integer, this operation is almost guaranteed to raise a division-by-zero exception. It is *almost* guaranteed and not completely guaranteed to raise a division-by-zero exception, as the division operator is strict in both of its arguments and might thus force the left-hand side argument to be evaluated before raising the **div-by-zero** exception. This evaluation might then in turn cause an exception from the set α to be raised first. The complete result type is thus an integer with an exception annotation consisting of the union (or *join*) of the exception set α on the argument together with an additional exception **div-by-zero**. Finally, we note that there is an empty exception set annotating the function space constructor, indicating that no exceptions will be raised when evaluating f to closure.

While this approach seems promising at first, it is not immediately adequate for our purpose: detecting potential pattern-match failures that may occur at run time. Consider the following program:

$$head\ (x :: xs) = x$$

After an initial desugaring step, a compiler will translate this program into:

$$head\ xxs = \mathbf{case}\ xxs\ \mathbf{of}\ \{\,[] \mapsto \mathbf{pmf}; x :: xs \mapsto x\,\}$$

which can be assigned the exception type:

$$head : \forall\tau\alpha\beta.[\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha\sqcup\beta\sqcup\mathbf{pmf}}$$

This type tells us that $head$ might always raise a **pmf** exception, irrespective of what argument is applied to. Clearly, we won't be able to outperform a simple syntactic analysis in this manner. What we need is a way to introduce a dependency of the exception flow on the data flow of the program, so we can express that $head$ will only raise a **pmf** if it possible for the argument passed to it to be an empty list. We can do so by introducing conditional constraints into our type system:

$$head : \forall\tau\alpha\beta\gamma.[\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha\sqcup\beta\sqcup\gamma}\ \mathbf{with}\ \{\mathbf{N} \sqsubseteq \beta \Rightarrow \mathbf{pmf} \sqsubseteq \gamma\}$$

This type explains that $head$ will return an element of type τ that might—when inspected—raise any exception present in the elements of the list (α), the spine of the list (β) or from an additional set of exceptions (γ), with the constraint that if the list to which $head$ is applied is empty, then this exception set contains the **pmf** exception, otherwise it is taken to be empty. (We apologize for the slight abuse of notation—using the annotation β to hold both data and exception-flow information—this is remedied in the formal type system.)

References

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.

D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proceedings of the Second International Symposium on Static Analysis*, SAS '95, pages 118–135, London, UK, UK, 1995. Springer-Verlag.

F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, Apr. 1993.

X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, Mar. 2000.

N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM.

M. Rittri. Dimension inference under polymorphic recursion. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 147–159, New York, NY, USA, 1995. ACM.

J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, June 1994.

M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.