Higher-Ranked Exception Types

(Extended abstract of work-in-progress)

Ruud Koot [*]

Department of Computing and Information Sciences
Princetonplein 5, 3584 CC, Utrecht
Utrecht University
r.koot@uu.nl

1. Motivation

An important design decision underlying the Haskell programming language is that "types should not lie": functions should behave as if they were mathematical functions on the domain and range stated by their type; any side-effects the function wishes to perform must be made explicit by giving it an appropriate monadic type. Surprisingly to some, then, Haskell does not feature *checked exceptions*. Any exception that may be raised—whether explicitly by a call to the the *error* function, or implicitly by a pattern match failure—is completely invisible in the type. This is unfortunate as during program verification such a type would give the programmer an explicit list of proof obligations he must fulfill: for each exceptions it must be shown that either the exception can never be raised or only be raised in truly exceptional circumstances.

While some programming languages with more mundane type systems do state the exceptions functions may raise in their type signatures, assigning accurate exception types to functions in a Haskell program is complicated by:

Higher-order functions Exceptions raised by higher-order functions depend on the exceptions raised by functional arguments.

Non-strict evaluation Exceptions are not a form of control flow, but are values that can be embedded inside other values.

Writing the set of exceptions that may be raised when evaluating an expression to weak head normal form as a superscript on its type, a fully annotated exception type for *map* would be:

$$\begin{aligned}
&\mathit{map}\ :\ \forall \alpha\ \beta\ e_1\ e_2\ e_3\ e_4.\\
&\quad (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \xrightarrow{\emptyset} [\alpha^{e_1}]^{e_4} \xrightarrow{\emptyset} [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4}\\
&\mathit{map} = \lambda f.\lambda xs.\ \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \qquad\ \mapsto [\,]\\
&\qquad (y : ys) \mapsto f\ y : \mathit{map}\ f\ ys
\end{aligned}$$

However, this exception type is not as accurate as we would like it to be. Consider the applications:

$$\begin{aligned}
&\mathit{map}\ \mathit{id} &&: [\alpha^{e_1}]^{e_4} \to [\alpha^{e_1}]^{e_4}\\
&\mathit{map}\ (\mathit{const}\ \bot_{\mathbf{E}}) &&: [\alpha^{e_1}]^{e_4} \to [\beta^{(e_1 \cup \{\mathbf{E}\})}]^{e_4}
\end{aligned}$$

While the exception type for *map id* is as expected, a more appropriate type for *map* (*const* $\bot_{\mathbf{E}}$) would be

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

as it cannot possibly propagate any exceptional elements inside the input list to the output list.

[*] This material is based upon work supported by the Netherlands Organisation for Scientific Research (NWO).

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 input to the output while possibly adding some additional exceptional values. This is a worst-case scenario: it is sound but inaccurate.

The solution is to move from Hindley–Milner to System F_ω, introducing *higher-ranked exception types* and *exception operators*. This gives us the expressiveness to state the exception type of *map* as:

$$\begin{aligned}
&\forall e_2\ e_3.(\forall e_1.\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2\ e_1)})\\
&\quad \to (\forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{(e_2\ e_4\ \cup\ e_3)}]^{e_5})
\end{aligned}$$

Note that e_2 has kind EXN $\to$ EXN. Given the following functions:

$$\begin{aligned}
&\mathit{map} &&: \forall e_2\ e_3.(\forall e_1.\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2\ e_1)})\\
& &&\quad \to (\forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{(e_2\ e_4\ \cup\ e_3)}]^{e_5})\\
&\mathit{id} &&: \forall e.\alpha^{e} \xrightarrow{\emptyset} \alpha^{e}\\
&\mathit{const}\ \bot_{\mathbf{E}} &&: \forall e.\alpha^{e} \xrightarrow{\emptyset} \beta^{\{\mathbf{E}\}}
\end{aligned}$$

Applying *map* to *id* or *const* $\bot_{\mathbf{E}}$ 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:

$$\begin{aligned}
&\mathit{map}\ \mathit{id} &&: \forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \to [\alpha^{e_4}]^{e_5}\\
&\mathit{map}\ (\mathit{const}\ \bot_{\mathbf{E}}) &&: \forall e_4\ e_5.[\alpha^{e_4}]^{e_5} \to [\beta^{\{\mathbf{E}\}}]^{e_5}
\end{aligned}$$

as desired.

2. Approach and Related Work

Due to their syntactic weight, higher-ranked exception types only seem useful if they can be inferred automatically. Unlike for Hindley–Milner, type inference is undecidable in System F_ω. However, the exception types are annotations piggybacking on top an underlying type system. Holdermans and Hage (2010) show that type inference is decidable for a higher-ranked annotated type system with type operators performing control-flow analysis.

Their analysis proceeds in three stages:

1. Hindley–Milner type inference is used to generate a fully type-annotated syntax tree.
2. An Algorithm $\mathcal{W}$-derived method is used to infer the control-flow types. The infered types are "completions" of the underlying types: they have the same (type contructor) shape as the underlying types, but have a maximal number of quantifiers and type operators with a maximal number of arguments placed

where possible; together with a set of set-inclusion constraints between type variables expressing the control-flow.

3. The generated constraints are solved using a fixed-point iteration.

To adapt their approach for our purposes a number of problems needs to be overcome:

1. During the constraint-solving phase of the algorithm equality between types needs to be decided to determine whether a fixed-point has been reached or not. As types in System F_ω are terms in the simply typed λ-calculus, a number of β-reduction steps may need to be performed to bring them into normal form. Additionally, the set-inclusion constraints introduce ACI1 equalities between the terms. The authors have axiomatized these equalities, but do not give a concrete procedure for deciding equality. We wish to use a general result by Breazu-Tannen (1988) to show the combination of these two systems gives rise to a normalizing and confluent rewrite system that can indeed be used to decide equality.

2. Haskell has an *imprecise exception semantics* that is necessary to validate a number of program transformations applied by optimizing compilers, such as the case-switching transformation:

```
∀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
```

   (Note that this transformation does not hold if we have observable exceptions and track them precisely: as a counterexample, take $e_1 = \bot_{\mathbf{E_1}}$ and $e_2 = \bot_{\mathbf{E_2}}$.)

   The imprecise exception semantics states that if evaluating the condition of a **case**- or **if** − **then** − **else**-expression raises an exception, then it is also admissible to raise an exception that would have been been raised by evaluating any of the consequents. A similar rule applies when applying an argument to an exceptional function value.

   While it would be sound to assume any conditional can always raise an exception when evaluated, this assumption would negate any additional precision gained by the higher-ranked types and more. Set-inclusion constraints do not allow us to adequately express the imprecise exception semantics. In an earlier approach we tried extending the set-inclusion constraints with conditionals, but the procedure deciding constraint entailment turned out to be rather *ad hoc* and unsatisfactory and introduced an additional level of approximation in order to retain termination.

   Instead we propose moving from a constraint-based inference algorithm to one employing equational unification. In particular the equational theory of Boolean rings (Wright 1991; Kennedy 1996) seems sufficiently expressive and has many desirable properties: unification is decidable and a most general unifier exists; there are confluent and normalizing rewrite systems to reduce terms into a normal form.

References

V. Breazu-Tannen. Combining algebra and higher-order types. In *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 82–90, 1988. doi: 10.1109/LICS.1988.5103.

S. Holdermans and J. Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 63–74, 2010. ISBN 978-1-60558-794-3.

A. J. Kennedy. Type inference and equational theories. Technical Report LIX-RR-96-09, Laboratoire D'Informatique, École Polytechnique, 1996.

D. A. Wright. A new technique for strictness analysis. In S. Abramsky and T. Maibaum, editors, *TAPSOFT '91*, volume 494 of *Lecture Notes in Computer Science*, pages 235–258. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-53981-0.

A. Requested Information

Research advisor Jurriaan Hage (`j.hage@uu.nl`)

ACM member number 1602867

Category Graduate