Explaining Type Errors

Brent A. Yorgey
Hendrix College
yorgey@hendrix.edu

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

Harley D. Eades III
Augusta University
harley.eades@gmail.com

Every beginning student of programming—that is, every student with the ill fortune of having a language with a static type system foisted upon them by a well-intentioned yet sadistic instructor—is well-acquainted with the Dreaded Type Error Message:

```
Couldn't match expected type (t, b0) with actual type Int
In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f (3 :: Int), namely
    (\ p -> fst p)
```

Why do type error messages have to be so terrifying? Can't we do a better job explaining type errors to programmers?

We propose two interrelated theses:

1. We ought to move away from static error "messages" and towards *interactive error explanations*.
2. We ought to consider the problem of generating error explanations in a more *systematic, disciplined, and formal way*. Explaining errors to users shouldn't just be relegated to the status of an "engineering issue", but ought to have all the tools of programming language theory and practice applied to it.

1 The curse of information

Consider a standard version of the simply typed lambda calculus shown in Figure 1, with some arbitrary set of base types B and typing annotations on lambda-bound variables.

$$
\begin{aligned}
t &::= x \mid \lambda x{:}\tau.\ t \mid t_1\ t_2 \\
\tau &::= B \mid \tau_1 \to \tau_2 \\
\Gamma &::= \cdot \mid \Gamma, x : \tau
\end{aligned}
$$

$$
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ t : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}
$$

Figure 1. The simply typed lambda calculus

A typical implementation of a type checker for this language recurs through the structure of a term, adding bindings to the context as it recurs through lambdas, and checking that the types match up appropriately at each application. If the types don't match, some sort of error message is generated:

if $(\tau_1 \neq \tau_1')$
  then throwError "Mismatch: expected $\{\tau_1\}$, actual $\{\tau_1'\}$"
else ...

Of course, this error message lacks any sort of context. One problem is that it may be difficult for the programmer to even figure out which part of their program the error corresponds to, especially if the mismatch happened deep inside a large term. This problem is not too hard to solve, by retaining information about the source code locations of terms, and possibly by making use of appropriate editor support for highlighting the locations of reported error messages; of course, most real-world language implementations actually do this.

However, a deeper problem is that even if the programmer knows *where* in their program the error message came from, they may not understand *why* it is an error: for example, where did the two types in question come from, and why does the type checker think they ought to be equal?

A natural reaction to this problem is to add more information to the error message: for example, highlighting a larger portion of the term containing the problematic subterm, including the types of variables mentioned in the term, or even giving suggestions about potential fixes. However, this is likely to be unhelpful in the long run:

- How do we decide which information to include in a given message? If we include all possible information (whatever that even means!), the message will be impossibly large. On the other hand, if we are more selective, we may end up omitting information which would have been helpful—not to mention that the helpfulness of any particular information varies depending on the individual programmer and their background.
- Even if we somehow figure out which information would be most helpful to include, paradoxically, it may not be helpful to include it! Beginners, in particular, may be *less* likely to read large error messages—even when those error messages contain information that would genuinely help them—because the messages seem overwhelming or intimidating. Experts also may prefer less information, because in many cases they do not need it, and they would rather be able to see more error messages on the screen at once.

It seems we can't win: programmers will be confused if there is not enough information about what went wrong; but if we try to present more information, it is likely to be unhelpful, overwhelming, or both.

2 Interactivity to the rescue

The problem of how much information to include in error messages is actually a red herring. The real problem is that for reasons of history and technical convenience, we are stuck thinking in a framework that is terminal-based and oriented towards batch processing. Even the term "error *message*" itself seems to presuppose this mode of interaction.

Suppose, instead, that the programmer is allowed to *explore errors interactively*: they are initially presented with a concise description of the error, and then they can incrementally explore additional information—for example, by expanding nodes in a tree corresponding to questions they might want to ask. This solves both problems outlined above:

$$\frac{\forall \tau_1.\ \Gamma \not\vdash t_1 : \tau_1 \to \tau_2}{\Gamma \not\vdash t_1\ t_2 : \tau_2}\ \textsc{LhsTy}\!\!\frownie$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \not\vdash t_2 : \tau_1}{\Gamma \not\vdash t_1\ t_2 : \tau_2}\ \textsc{RhsTy}\!\!\frownie$$

Figure 2. Some untyping rules for the STLC

- We do not have to decide what information to include up front; we can think of the error explanation as a lazy tree containing all the information we could ever conceivably generate about the error and its causes. The programmer then gets to interactively select exactly the information they want to see.
- The programmer is less likely to be overwhelmed, since the initial message they are shown can be kept short and to the point. Even if they end up looking at the same information that a static, batch error message might have included, processing and assimilating the information will still be psychologically easier when they are able to explore the information incrementally.

3 How to explain things

So, how do we go about generating such error explanations? It is here that we think the tools of programming language theory can be fruitfully applied. On the face of it, type errors do not seem like a very "off the beaten track" topic at all. There was a spate of work on explaining type errors in the 1990's and early 2000's (for example, Beaven and Stansifer (1993); Chitil (2001); Simon et al. (2000)), but this line of work seems to have mostly died out. Though there is ongoing work on tracking type and error *provenance* (Augustsson (2014) is a representative example), we are not aware of more recent work taking a principled, PL-style approach to the problem of explaining type errors to programmers.

When a program has a valid type, how do we explain it? The answer to this is well-known in the PL community: a *typing derivation* is a (constructive) proof that a given term has a given type, and a constructive proof is simply a detailed, logically rigorous explanation. If we want a type inference algorithm to explain itself, we can have it return an entire typing derivation, not just a type.

So, when a program *doesn't* type check, how should we explain it? Given the discussion in the previous paragraph, the answer ought to be clear: with an *untyping derivation*, that is, a constructive proof of *un*typability! We can extract information from this to show to the programmer, or we can simply let them interactively explore it. That is, to be clear, we propose that an untyping derivation itself should form the core of the tree structure making up an error explanation (though actually it should probably be more like a zipper onto the untyping derivation, since the explanation should start "focused in" on the specific location of an error, rather than on the top-level program).

Let's look at a simple example. Figure 2 shows just two of the rules for untyping derivations for the simply typed lambda calculus; we pronounce $\Gamma \not\vdash t : \tau$ as "t does not have type τ in context Γ". Rules LhsTy☹ and RhsTy☹ enumerate the ways that an application can fail to have a given type: $t_1\ t_2$ does not have type τ_2 if either t_1 does not have an appropriate function type, or if t_1 does have an appropriate type but t_2 does not have the right type to be its argument.

How do we know when we have the definition of untyping derivations correct for a given type system? We can justify a particular definition of untyping by proving some metatheorems relating it to typing. For example, we certainly want a metatheorem of the form

$$\forall t \tau.\ (\Gamma \not\vdash t : \tau) \implies \neg(\Gamma \vdash t : \tau).$$

For some type systems we can prove the converse as well. If we want an untyping derivation to somehow encompass all possible errors in a term, rather than just picking one (corresponding to the difference between a type checking algorithm that stops as soon as an error is encountered and one which tries to continue and collect additional errors), we could try to prove a uniqueness theorem—for a given triple of (Γ, t, τ) there is at most one untyping derivation.

There are many remaining questions:

- There is still a lot of latitude in designing the rules for untyping derivations. For example, one could imagine adding another rule for the STLC:

  $$\frac{\Gamma \vdash t_1 : \tau \qquad \textsc{NotArrow}(\tau)}{\Gamma \not\vdash t_1\ t_2 : \tau_2}$$

  This rule says that an application does not have a type if its LHS *does* have a type, but not an arrow type. This rule is actually subsumed by rule LhsTy☹, but might be more comprehensible to programmers in cases when it applies. Is there a systematic way to approach the task of choosing rules for building untyping derivations?
- Is there some way to *mechanically* derive untyping derivation rules from the rules of a type system? De Morgan-type laws may play a starring role, but there are likely to be subtleties, given considerations such as the previous bullet point.
- Although building appropriate untyping rules seems straightforward for natural-deduction-style systems, it becomes much trickier when trying to explain unification failures; the precise algorithms used to do unification seem to matter quite a bit. What is the right way to explain unification failures to programmers?

4 Proposal

In our talk, we will (a) motivate interactive error explanations and untyping derivations, (b) present several variant systems of untyping rules for the STLC, with appropriate metatheorems all formalized in Agda, (c) share some preliminary thoughts on how to explain unification errors.

References

Lennart Augustsson. 2014. Type Error Provenance. https://www.youtube.com/watch?v=rdVqQUOvxSU.

Mike Beaven and Ryan Stansifer. 1993. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems (LOPLAS)* 2, 1-4 (1993), 17–30.

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM, Firenze, Italy, 182–196. http://www.cs.kent.ac.uk/pubs/2001/1811

Axel Simon, Olaf Chitil, and Frank Huch. 2000. Typeview: A Tool for Understanding Type Errors. In *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, Markus Mohnen and Pieter Koopman (Eds.). Aachener Informatik-Bericht 00-7, RWTH Aachen, Aachen, Germany, 182–196. http://www.cs.kent.ac.uk/pubs/2000/1899