Blame Tracking and Type Error Debugging

Sheng Chen
University of Louisiana at Lafayette, USA
http://www.ucs.louisiana.edu/~sxc2311
chen@louisiana.edu

John Peter Campora III
University of Louisiana at Lafayette, USA
campora@louisiana.edu

Abstract

In this work, we present an unexpected connection between gradual typing and type error debugging. Namely, we illustrate that gradual typing provides a natural way to defer type errors in statically ill-typed programs, providing more feedback than traditional approaches to deferring type errors. When evaluating expressions that lead to runtime type errors, the usefulness of the feedback depends on *blame tracking*, the defacto approach to locating the cause of such runtime type errors. Unfortunately, blame tracking suffers from the *bias* problem for type error localization in languages with type inference. We illustrate and formalize the bias problem for blame tracking, present ideas for adapting existing type error debugging techniques to combat this bias, and outline further challenges.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases Blame tracking, type error debugging, gradual typing, type inference

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.2

Funding This work is partially supported by the NSF grant CCF-1750886.

1 Introduction

Static and dynamic typing have different strengths [38, 53, 54]. For example, static typing can detect more errors at compile time but delivers no runtime feedback for programs that have static errors, while dynamic typing can run any program and provide feedback but defers error detection to runtime. Gradual typing [45, 52, 47] is a new language design approach that can integrate both typing disciplines in a single language, allowing different interacting program regions to use static or dynamic typing as needed. Adopting gradual typing has been popular with both statically typed languages (for example C#[3]) and dynamically typed languages (for example JavaScript and Flow [8]). Additionally, gradual typing has been adapted to work with many advanced language features [44, 31, 46, 37, 16, 30].

In particular, much research has explored the interaction of gradual typing with type inference [46, 37, 16, 4, 5]. There are several reasons that this interaction has been explored. First, inference can benefit the usability of gradual typing without programmers having to manually modify numerous type annotations, as argued by [36]. Second, inferred types can be synchronized to help improve gradual typing performance [36, 37, 5, 57]. Third, inference can aid in the detection of inconsistencies in programs [16, 4, 30].

This paper explores an intriguing connection between gradual typing and type inference, the *blame tracking* in gradual typing and *type error debugging* in type inference. In gradual typing, well-typed programs may still encounter runtime type errors since type-checkers allow statically typed contexts to accept values produced by dynamically typed expressions. At runtime these dynamic values may not have the desired runtime types that the context expected, and consequently a runtime type error may be triggered. *Blame safety*, adapted from [15] by Wadler and Findler [61] and further developed in [1, 60, 43, 2], is a well accepted approach in the gradual typing community for indicating which code region is responsible for a runtime type error. Blame safety states that when a cast fails the blame is always assigned to the more dynamic part, under the slogan that "well-typed programs can't be blamed".

Type error localization provides a similar purpose in type error debugging. It specifies which code region is responsible for a static type error. Inaccurate type error localization produces poor error messages [62, 24], and type inference often leads to poor localization. Unlike in gradual typing where blame tracking is the de facto mechanism for enforcing blame safety, numerous approaches have been developed to improving error localization and error reporting in the presence of type inference errors in the last three decades [7, 21, 19, 51, 24, 28, 17, 14, 13, 10, 68, 33, 27, 9, 12, 32, 26, 55, 64, 65, 42, 11].

In this paper, we explore the relations between gradual typing and type error debugging. Our exploration includes the following directions. First, we investigate whether gradual typing and blame tracking improve type error debugging for type inference. Intuitively, we can turn a program with type errors into a well-typed gradual program by annotating certain expressions with dynamic types. We can then obtain and observe runtime behaviors of the program using gradual typing, which may give the users a better understanding of the type error. Through a few examples, we conjecture that gradual typing improves type error debugging, but blame tracking does not. We present the details in Section 2.

Second, by drawing an analogy from type error localization, we reflect the usefulness of blame tracking for providing debugging information when gradual programs encounter dynamic type errors. A fundamental difficulty in type error localization is that type inference is biased. Specifically, it tends to attribute the type error to a later part of the program's syntax tree while in fact the error may have been caused by an earlier part. Unfortunately, we observe that blame tracking is also biased. It tends to blame a later untyped region along the execution path leading to a dynamic type error, while the error may have been caused by an earlier region. We prove this problem in Section 3. Thus, we suggest that while blame tracking is well-accepted in the gradual typing community, its helpfulness in debugging is questionable, particularly in languages supporting type inference.

Third, as blame tracking suffers from the same problem as type inference does, we are interested in knowing the potential of adapting existing type error localization and debugging approaches to improve blame tracking. We discuss the potential of several approaches and highlight the challenges in adapting them in Section 4. We conclude in Section 5.

2 Gradual Typing and Blame Tracking for Type Error Debugging

This section investigates how gradual typing and blame tracking can help with type error debugging in Sections 2.1 and 2.2, respectively.

2.1 Gradual Typing for Type Error Debugging

We use the example factorial program from Figure 1 to illustrate how gradual typing provides new insights for type error debugging. The type error in Figure 1a is caused by the value `true`, which should instead be `1`. Types for the program are inferred by solving constraints that are generated by the structure of the program. For example, in each function call a constraint is generated ensuring that the argument type of the function must match the type of the argument, and similarly different branches in a single function are constrained into returning the same type. These constraints are collected and solved while traversing program structure. If the constraints can't be solved, then a type error is raised.

To make type inference feasible, constraint solving must be *most general* [59]. Most-general constraint solving pushes back failure detection as late as possible, and thus the reported location is likely not the real error cause. For example, Helium [20], a Haskell type error debugger, attributes the type error to `*`, later than the real error cause `true`, as can be seen from Figure 1b. The standard Haskell compiler GHC also suffers from the right-biased problem, although it reports the whole `else` branch as the error cause.

Most type error debugging approaches prevent the running of ill-typed programs. However, the ability to run ill-typed programs is believed to help program understanding [41, 58], which may in turn help programmers fix their type errors. A main approach that enables the execution of ill-typed programs is deferring type errors to runtime [58]. In a program that contains both well-typed and ill-typed functions, deferring type errors allows programmers to run functions that are not involved in the type error. It, however, provides little help to fix the type error itself. To illustrate, consider running the program in Figure 1a with GHC and deferred type errors enabled. After loading the function, we can invoke `fac` in GHCi. While it looks reasonable to expect the result `true` when we run `fac 0`, since the branch that will be executed does not contain a type error, GHCi actually dumps its deferred compile-time type error message, as shown in Figure 1d. This message is biased since it does not mention the real error cause `true`. The work by Seidel et al. [41] can also run ill-typed programs, but it supports much fewer language features.

An alternative to the previously mentioned approaches is to use gradual typing, facilitated by ascribing certain subexpressions with the dynamic type (denoted by ?). For `fac` in Figure 1a, we make it well-typed by ascribing ? to `*`, as shown in Figure 1c. This new expression, which we name `facG`, is well-typed because the dynamically typed operator (`*`:?) can accept values of any type as arguments. We run `facG` with the gradual evaluator developed by Miyazaki et al. [30], and the result is shown in Figure 1e. The output first shows

```
fac n = if n == 0 then true
                  else n * fac (n-1)
```

(a) An ill-typed function, adopted from [41].

```
(3,43): Type error in infix application
 expression        : n * fac (n - 1)
 operator          : *
   type            : a    -> a    -> a
   does not match  : Int -> Bool -> Bool
```

(b) Output from Helium [20] for `fac`.

```
facG n = if n == 0 then true
               else n (*:?) facG (n-1)
```

(c) A well-typed gradual program by ascribing `*` in Figure 1a to have a dynamic type ?.

```
*Main> fac 0
*** Exception: Fac.hs:3:41: error:
• Couldn't match expected type ‘Bool’
                   with actual type ‘Int’
• In the expression: n * fac (n - 1)
  In the expression:
    if n == 0 then True else n * fac (n - 1)
    ...
(deferred type error)
```

(d) Output from running `fac` with GHC 8.0.2 and deferring type errors enabled.

```
# facG 0;;
- : bool = true
```

(e) Result of running `facG` with the idea from [30].

Figure 1 An ill-typed function `fac` and outputs from various tools and approaches. To reconcile language differences, we use `true` for `True` in Haskell.

the type and then the result after the equals sign. Interestingly, gradual typing produces `true` for the expression `facG 0`. With this output, a keen programmer should already be able to fix the type error in `fac` because the factorial function should never return a boolean value. She may thus change `true` to `1` to remove the type error.

Because gradual typing generally provides more feedback when running “statically” ill-typed (but gradually well-typed) programs than deferred type errors, we believe that gradual typing does offer additional insights beyond existing type error debugging approaches. However, a main hindrance of adopting gradual typing for debugging type errors is that current implementations supporting gradual typing with type inference and parametric polymorphism are limited. The work in [30] (building on much previous work on some mix of polymorphism, inference, and blame [16, 1, 2, 22, 23]) provides an implementation, but it is restricted to a small set of language features.

A particular twist in using gradual typing to improve type error debugging is that the user needs to provide appropriate arguments to ill-typed functions. Specifically, gradual typing will show execution results only if the execution does not encounter a dynamic type error. For example, only when the argument to `facG` is `0` will the resulting `true` be shown. Otherwise, blaming information will be shown. In such cases, the usefulness of gradual typing for type error debugging depending on that of blame tracking, which we investigate in Section 2.2.

2.2 Blame Tracking for Type Error Debugging

Blame tracking specifies which subexpression should be blamed if the execution of an expression encounters a runtime error. The standard goal of blame tracking in gradual typing is to preserve blame safety [61, 60], which attributes the blame for runtime type errors to subexpressions with more dynamic types. We use the following table to illustrate the behaviors of blame safety. In the “Expressions” column, we use a `;` to denote a sequential expression. The “Blames” column are the expressions blamed after running the corresponding expression with the evaluator from [30]. Our goal in the expressions (1) through (4) is to use gradual typing to debug the type error in `fac` (Figure 1a). The goal of expressions (5) through (11) is to debug the type error in `(\x y -> if true then x else y) 2 false`.

Ids	Expressions	Blames
(1)	`fac1 n = if n == 0 then true else n (*:?) fac1 (n-1); fac1 1`	`*`
(2)	`fac2 n = if n == 0 then (true:?) else n * fac2 (n-1); fac2 1`	`true`
(3)	`fac3 n = if n == 0 then (true:?) else n (*:?) fac3 (n-1); fac3 1`	`*`
(4)	`fac4 n = if n > 0 then n (*:?) fac4 (n-1) else (true:?); fac4 1`	`*`
(5)	`(\x y -> if true then x else y) 2 (false:?)`	`false`
(6)	`(\x y -> if true then x else y) (2:?) false`	`2`
(7)	`(\x y -> if true then x else y) (2:?) (false:?)`	`false`
(8)	`(\x y -> if true then x else y) (false:?) (2:?)`	`2`
(9)	`(\x y -> if true then x else y) ((succ:?) 2) (false:?)`	`false`
(10)	`(\x y -> if true then x else y) ((succ:?) (2:?)) (false:?)`	`false`
(11)	`(\x y -> if true then x else y) ((\x -> x) (succ:?) 2) (false:?)`	`false`

From the table, we make two observations. First, when an expression contains only one subexpression with a dynamic type ?, then that subexpression will be blamed for causing the runtime type error. This is intuitive, because that subexpression is more dynamic than all other subexpressions and will alone be responsible for type mismatches at runtime. The expressions (1), (2), (5), and (6) in the table all belong to this case. Note that the anonymous function is statically-typed (since the parameters do not have the type ?), and the conditional branches are required to have the same type. As a result, the expressions (5) through (11) will raise blame, rather than returning the first argument. We observe that blame tracking does blame the subexpression that caused the type error if that subexpression happens to have the dynamic type, as in the expression (2) above. However, if the user knows where to add ? she probably already knows how to fix the type error.

Second, when an expression contains multiple subexpressions that have the dynamic type, then blame safety is *biased* in attributing the dynamic type error. Specifically, because blame safety is connected to the expression that triggers the runtime type error, it always blames the most recently encountered dynamic context in program execution, even if the true cause of the error was due to an expression evaluated much earlier. For example, in both expressions (3) and (4), the subexpression `true`:? is returned earlier than it is being used as a multiplicand to `*`:?. In other words, `true` is being executed before `*`:?. As a result, `*`:? is blamed in both subexpressions, regardless of their ordering in the conditional branches. Alternatively, we can view `true`:? as injecting `true` to a dynamic value, and when it is used in `*`:? a projection happens. This fits in our description of blame, since blame tracking always blames projections that follow injections.

In expressions (7) through (11), the anonymous function is first applied to the first argument, substituting the argument into `x` in the body and also instantiating the type of both parameters `x` and `y`. When it is applied to the second argument, its type is ensured to be the same as the instantiation. Therefore, we observe that the first argument is executed first and the second argument later. Unsurprisingly, blame safety always assigns the blame to the second argument. The expression (8) in the table demonstrates that that the ordering of types is responsible for determining the blamed expression. The expressions (9) through (11) confirm this observation, even as expressions become more complicated.

We observe that when an expression has multiple subexpressions with ?, blame tracking may provide little help to type error debugging. There is little context in the program that indicates that the blamed subexpression is the true cause of the type error. This phenomenon is well-understood in the type error debugging research community. For example, while `*` is blamed in the expressions (3) and (4), we already know that that does not cause the

$$
\begin{array}{rlll}
\text{Term variables} & x,\ y & \text{Type variables} & X,\ Y \\
\text{Type constants} & \iota & \text{Blame labels} & \ell \\
\text{Static types} & T \ ::= \ \iota \mid X \mid T \to T & \text{Gradual types} & U \ ::= \ ? \mid \iota \mid X \mid U \to U \\
\text{Ground types} & G \ ::= \ \iota \mid ?\to? & & \\
\text{Expressions} & e \ ::= \ x \mid c \mid op(e,e) \mid \lambda x\!:\!U.e \mid e\ e \mid e : U \Rightarrow^{\ell} U & & \\
\text{Values} & w \ ::= \ c \mid op(w,w) \mid \lambda x\!:\!U.e \mid w : U \to U \Rightarrow^{\ell} U \to U \mid w : G \Rightarrow^{\ell} ? & &
\end{array}
$$

Figure 2 Syntax of types, expressions, and values.

type error. In subexpressions (7) and (9) through (11), while `false` may be the error cause (because changing it to some integer value will in fact fix the type error), `2` or/and `succ` are equally likely to have caused the type error. For example, the user may have intended to use a boolean value where `2` is, or they may have intended to use a boolean valued function such as `even` or `odd` instead of `succ`.

Based on these two observations, we conclude that blame tracking offers very little additional help to type error debugging. Moreover, our second observation implies that blame safety may not be an ideal way to report causes for dynamic type errors, since it is biased. In Section 3 we prove that this bias is indeed a general problem.

3 Blame Tracking is Biased

To prove that blame tracking is biased, our high-level idea is to show that gradual program execution embodies type constraint generation and solving, which are well-known to be biased in the type inference research community. Our idea is inspired by the work of dynamic type inference (DTI) for gradual typing [30]. In combining gradual typing and type inference, some type variables are left undecided at compile time due to the interaction of dynamic types. Consider, for example, the expression $(\lambda x\!:\!?.x\ 2)\ (\lambda y.y)$. During type inference, the type, say Y, for the parameter y cannot be decided because it is solely required to be consistent with ? (which every type is consistent with). However, the choice of Y has a significant impact on the execution result. If Y is chosen to be `Int`, then the expression runs correctly. Otherwise, the expression leads to dynamic type errors. The challenge here is that it is statically difficult to decide that Y should be `Int`.

DTI addressed this issue by keeping Y as y's static type and deferring the instantiation of it to runtime. Eventually, $\lambda y.y$ will be applied to 2, making it clear that Y needs to be instantiated with `Int` to make this application succeed. Miyazaki et al. [30] proved that DTI is both sound and complete in the sense that if a term is evaluated successfully then some correct instantiation (such as instantiating Y with `Int` at compile time) of the term will execute successfully in the blame calculus [61, 1, 2].

This example illustrates that DTI mixes reductions and type instantiations without generating type constraints explicitly. To investigate whether existing type error localization and debugging approaches can help address the bias in blame tracking, we instead separate constraint generation (during program execution) and constraint solving (after the execution is finished). If constraint solving succeeds, then the program executes without raising blame in DTI. Otherwise, blame will be raised. The advantage of this separation is that it allows us to make a clear connection between constraint solving and blame tracking.

$$
\begin{array}{rcll}
op(w_1, w_2) & \longrightarrow_G & ([\![op]\!](w_1, w_2),\ \{\}) & \\
(\lambda x : U.e)\ w & \longrightarrow_G & (e[w/x],\ \{\}) & \\
w : \iota \Rightarrow^{\ell} \iota & \longrightarrow_G & (w,\ \{\}) & \\
w : ? \Rightarrow^{\ell} ? & \longrightarrow_G & (w,\ \{\}) & \\
(w_1 : U_1 \rightarrow U_2 \Rightarrow^{\ell} U_3 \rightarrow U_4)\ w_2 & \longrightarrow_G & ((w_1\ (w_2 : U_3 \Rightarrow^{\overline{\ell}} U_1)) : U_2 \Rightarrow^{\ell} U_4,\ \{\}) & \\
w : U \Rightarrow^{\ell} ? & \longrightarrow_G & (w : U \Rightarrow^{\ell} G \Rightarrow^{\ell} ?, \{\}) \quad (\text{if } U \neq ?, U \neq G, U \sim G) & \\
w : ? \Rightarrow^{\ell} U & \longrightarrow_G & (w : ? \Rightarrow^{\ell} G \Rightarrow^{\ell} U, \{\}) \quad (\text{if } U \neq ?, U \neq G, U \sim G) & \\
w : G_1 \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} G_2 & \longrightarrow_G & (w, \{G_1 \cdot \overset{\ell_2}{=} \cdot G_2\}) & \textsc{Meet} \\
w : \iota \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} X & \longrightarrow_G & (w, \{\iota \cdot \overset{\ell_2}{=} \cdot X\}) & \textsc{Base} \\
w : ? \rightarrow ? \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} X & \longrightarrow_G & (w : ? \rightarrow ? \Rightarrow^{\ell_2} X_1 \rightarrow X_2, \{X \cdot \overset{\ell_2}{=} \cdot X_1 \rightarrow X_2\}) & \textsc{Arrow}
\end{array}
$$

Figure 3 Reduction rules. X_1 and X_2 in the ARROW rule are fresh.

3.1 Syntax

We consider a cast calculus with the type and expression syntax given in Figure 2. The definition is standard compared to other cast calculi [30, 47]. In the figure, $e : U_1 \Rightarrow^{\ell} U_2$ denotes a cast that ensures e has the type U_2 at runtime and raises blame at location ℓ otherwise. Both the cast and the label ℓ are inserted while translating gradual programs into programs in a cast calculus [47, 30].

The main difference between our calculus and others is that we do not have an explicit `blame` construct and do not terminate programs early, while others do [30, 47]. To avoid early termination, we add a value form $op(w_1, w_2)$ that interprets possible computations like `3 + true` as the value `+(3,true)`. The main reason for this addition is that we do not want to terminate program execution once a cast error would be encountered but rather collect all type constraints until program reduction finishes. We will show that our calculus and corresponding reduction rules yield the same result as others–that is they succeed with the the same value or fail with the same blame label.

3.2 Dynamic Constraint Generation and Solving

The reduction and constraint generation rules are presented in Figure 3. Our reductions have the form $e_1 \longrightarrow_G (e_2, C)$, where C is a set of constraints. A constraint has the form $U_1 \cdot \overset{\ell}{=} \cdot U_2$, denoting that U_1 and U_2 are required to be the same type, and ℓ will be blamed if they can not be made the same. We write $e \longrightarrow_G^{*} (e_n, C)$ if $e \longrightarrow_G (e_1, C_1)$, $e_1 \longrightarrow_G (e_2, C_2)$, $\ldots$, and $e_{n-1} \longrightarrow_G (e_n, C_n)$, and $C = C_1 \cup C_2 \cup \cdots \cup C_n$.

Since our semantics is designed to collect all type constraints, our reduction rules differ from those in [30] only when typing constraints are generated, or when a primitive operation would produce an error. The rules are the same in all other cases. Specifically, the first seven rules in Figure 3 are mostly standard among other cast dynamic semantics [30, 47]. Since no typing constraints are generated, the constraint set is empty in these rules.

The MEET rule handles two cases that are dealt separately in other cast semantics. In the first case, G_1 and G_2 are the same, and the cast will be successful and will thus be dropped. In the second case, G_1 and G_2 are different, and the cast will fail and blame ℓ_2. When our approach attempts to solve the constraint $G_1 \cdot \overset{\ell_2}{=} \cdot G_2$, it will behave as one of these cases. Specifically, if G_1 and G_2 are the same, then the constraint solving succeeds. Otherwise, constraint solving fails and blames ℓ_2.

In Base, a dynamically typed value with primitive type ι can be projected into having type X if and only if X of is the same as ι. As a result, a constraint $\iota \cdot \overset{\ell_2}{=} \cdot X$ is generated. Similarly, in Arrow, when we project a value that has a function type $(? \to ?)$, our minimal expectation of the value is that it is a function $(U_1 \to U_2)$. Both of these rules are similar to those in [30], but they solve the constraint immediately while we collect and solve them later.

For a generated constraint set, the Robinson's unification algorithm [39] will suffice to solve it. For simplicity, we use $\mathcal{U}$ to denote that algorithm with following simple extensions. For any constraint set C, $\mathcal{U}(C)$ returns a substitution θ if constraint solving succeeds. When constraint solving fails, it returns (ℓ, θ), where ℓ is the label of the constraint that fails to solve and must be blamed, and θ is the substitution accumulated so far. Note that $\mathcal{U}$ solves the constraints in the ordering they are added to the constraint set, as what classic type inference does [35].

In general, our approach of separating reduction and constraint solving is correct, as expressed in the following theorem, where $\longrightarrow_D^*$ denotes the reduction relation that mixes constraint solving and reduction, as defined in [30].

▶ **Theorem 1** (Correctness of $\longrightarrow_G$ and $\mathcal{U}$). *Given any expression e, let $e \longrightarrow_G^* (w_1, C)$.*

- *$\mathcal{U}(C) = \theta$ if and only if $e \longrightarrow_D^* w_2$ and $\theta(w_1) = w_2$.*
- *$\mathcal{U}(C) = (\ell,\theta)$ if and only if $e \longrightarrow_D^*$ `blame` ℓ.*

Intuitively, the theorem states that both approaches compute the same correct result or blame the same location. The theorem can be proved by induction over the $\longrightarrow_G$ relation defined in Figure 3 and the relation $\longrightarrow_D$ in [30].

The following example shows the reduction sequences of $\longrightarrow_G^*$ (between the two rules) and $\longrightarrow_D^*$ (after the second rule) for the expression $(\lambda x\ y.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2{:}?)\ (5{:}?)$. Note, $\rightsquigarrow$ denotes the step of type inference and cast insertion. The two parameters have the same type variable X after type inference because (1) they have no ?s and normal type inference applies to them and the subexpressions using them and (2) they are the two branches of the same conditional, which are required to have the same type [16, 30].

$$\begin{array}{lr}
\quad (\lambda x\ y.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2{:}?)\ (5{:}?) & \\
\rightsquigarrow (\lambda x : X\ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2 : \texttt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X) & \\
\hline
\longrightarrow_G (\lambda x : X\ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ 5 & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_5}{=} \cdot X\} \\
\longrightarrow_G^* 2 & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_5}{=} \cdot X\} \\
\hline
\longrightarrow_D (\lambda x : \texttt{Int}\ y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} \texttt{Int}) & \{X \mapsto \texttt{Int}\} \\
\longrightarrow_D (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} \texttt{Int}) & \\
\longrightarrow_D (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ 5 & \\
\longrightarrow_D 2 &
\end{array}$$

For this example, the reduction $\longrightarrow_D^*$ produces the result 2. The relation $\longrightarrow_G^*$ also produces that result, but generates an additional constraint set $\{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_5}{=} \cdot X\}$, which has the solution $\{X \mapsto \texttt{Int}\}$ after being solved by the solver $\mathcal{U}$. Therefore, both reductions succeed and produce 2.

The following example shows the reduction sequences of $\longrightarrow_G^*$ (between the two rules) and $\longrightarrow_D^*$ (after the second rule) for the expression $(\lambda x\ y.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2{:}?)\ (\texttt{false}{:}?)$.

$$
\begin{array}{ll}
& (\lambda x\ y.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2{:}?)\ (\texttt{false}{:}?) \\
\rightsquigarrow & (\lambda x : X\ \ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2 : \texttt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} X)
\end{array}
$$

$$
\begin{array}{lll}
\hline
\longrightarrow_G & (\lambda x : X\ \ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G & (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G & (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ \texttt{false} & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_\texttt{f}}{=} \cdot X\} \\
\longrightarrow_G^* & 2 & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_\texttt{f}}{=} \cdot X\} \\
\hline
\longrightarrow_D & (\lambda x : \texttt{Int}\ \ y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} \texttt{Int}) & \{X \mapsto \texttt{Int}\} \\
\longrightarrow_D & (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} \texttt{Int}) & \\
\longrightarrow_D & (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{blame}\ \ell_\texttt{f}) & \\
\longrightarrow_D & \texttt{blame}\ \ell_\texttt{f} &
\end{array}
$$

For this example, the reduction $\longrightarrow_D^*$ blames the location $\ell_\texttt{f}$ because `false` does not have the expected type `Int` at runtime. The relation $\longrightarrow_G^*$ produces the result 2 with the constraint set $\{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_\texttt{f}}{=} \cdot X\}$. When solving this set in the ordering that constraints were added to this set, $\mathcal{U}$ fails to solve the second constraint because X will be updated to `Int` after the first constraint is solved. As a result, solving the second constraint leads the program label $\ell_\texttt{f}$ being blamed. Overall, both reductions have the same behavior of blaming $\ell_\texttt{f}$ for this example.

Although the two reductions blame same locations for expressions that have runtime type errors, the reduction $\longrightarrow_G^*$ can extract more useful information that can be exploited by existing type error debugging approaches to provide better blaming information (We briefly explore this idea in Section 4). The following example `cond3` illustrates this aspect. For the cast inserted program, the relation $\longrightarrow_D^*$ blames ℓ_2 and so does $\longrightarrow_G^*$. However, our reduction rules also collect the constraint set $C_{\texttt{cond}} = \{\texttt{Bool} \cdot \overset{\ell_\texttt{f}}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_5}{=} \cdot X\}$.

$$
\begin{array}{ll}
\texttt{cond3} = & (\lambda x\ y\ z.\textbf{if}\ \texttt{false}\ \textbf{then}\ x\ \textbf{else}\ (\textbf{if}\ \texttt{true}\ \textbf{then}\ y\ \textbf{else}\ z))\ (\texttt{false}{:}?)\ (2{:}?)\ (5{:}?) \\
\rightsquigarrow & (\lambda x : X\ \ y : X\ \ z : X.\textbf{if}\ \texttt{false}\ \textbf{then}\ x\ \textbf{else}\ (\textbf{if}\ \texttt{true}\ \textbf{then}\ y\ \textbf{else}\ z)) \\
& \qquad (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_\texttt{f}} ? \Rightarrow^{\ell_\texttt{f}} X)\ (2 : \texttt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X)
\end{array}
$$

Based on Theorem 1, we can reduce blame tracking to constraint generation and solving in our approach. Since constraint solving is biased [14, 25, 29, 66, 10, 17], blame tracking is also biased. The difference between type inference and gradual typing is that constraints are collected at compile time in the former while at runtime in the latter. This means that in type inference the bias happens along the abstract syntax traversal ordering while in gradual typing the bias happens along the execution ordering.

4 Type Error Debugging for Blame Tracking

The previous section shows that blame tracking is biased similar to type inference, albeit with constraint collection happening at different times. This inspires that existing work in type error debugging may be adapted to alleviate the bias problem in blame tracking.

Reordering constraint solving. A common idea to combat the bias in the standard unification algorithm is to reorder the unification problems being solved [14, 25, 29, 66]. For example, if we solve the constraints in $C_{\texttt{cond}}$ from the last to the first, then the location ℓ_f, corresponding to `false`, will be blamed. While these approaches can improve blame tracking in some cases, they are in general still biased in the orderings they solve constraint problems.

Error slicing. Instead of just blaming one location, type error slicing approaches [50, 17, 40, 48, 49, 63] highlight all of the program locations relevant to a type error. For example, for the expression `cond3` in Section 3, although only ℓ_2 looks to cause the dynamic type error, all three locations ℓ_f, ℓ_2, and ℓ_5 will be identified by slicing approaches due to the connection of the common type variable X in all constraints. The downside of slicing approaches is that the user still must determine the real error cause among all those identified.

Error localization. Many approaches [24, 21, 18, 20, 34, 33, 27, 67, 68, 69] have been developed to exploit context information to locate the most likely error location among a set of locations. From the constraint set $C_{\texttt{cond}}$, all these approaches will blame ℓ_f as the error cause because the type variable X is unified with `Bool` once but `Int` twice. This result makes sense because changing `false` modifies only one subexpression, whereas the other fix requires changing 2 and 5. Most of these approaches, however, lack a concrete message about how to fix the type error.

Discussion. Some challenges exist in adopting existing type error debugging for improving blame tracking, including: (1) current gradual typing implementations do not facilitate constraint collection at runtime, (2) the assumptions that hold for type error debugging may not hold for blame tracking, and (3) shortcomings with error debugging approaches will also be transferred to those for blame tracking.

The real challenge is that, in some situations, neither blame tracking nor type error debugging help to debug the error, illustrated by the following expression.

```
(\x y -> if true then x else y) ((\x -> (x:?)) succ (2:?)) (false:?)
```

Assume the user intended to use `even` instead of `succ` in the expression, meaning that `succ` is the error source. Similar to the expression (11) (Section 2.2), this expression will cause a dynamic type error. However, neither blame tracking nor a potentially adopted type error debugging approach will be able to locate `succ` as the error source since it is not annotated with a ?. Worse, since this expression is well-typed, no existing type error debugging approach can help, leaving it to the user to determine the real problem.

One may suggest to remove all ?s in gradual programs and employ existing type error debugging approaches to assign blame for such programs. This idea is particularly intriguing as recent work on a user study of gradual typing behaviors [56] suggests that both experienced and novice programmers value static feedback for programs that have runtime type errors. However, the suggested idea fails because it may report errors in programs that do not fail. To illustrate, consider the following expression `condxy`, adopted from [6].

$$\texttt{condxy} = (\lambda x\ (y:?).\mathbf{if}\ x\ \mathbf{then}\ \texttt{even}\ y\ \mathbf{else}\ \texttt{not}\ y)\texttt{true}\ 2$$

Assume `even` has the type `Int`$\rightarrow$`Bool` and `not` has the type `Bool`$\rightarrow$`Bool`. This gradual expression runs without blaming any subexpression. However, if we remove the ? for the parameter y, we receive an error that blames some subexpression in `condxy`. Consequently, this idea may yield too many false positive error reports.

5 Conclusion

Type error debugging for fixing and understanding type errors when type inference fails is a well-studied subject. Blame tracking, though relatively new, is a well-accepted mechanism for assigning blames to program locations when gradually-typed programs encounter runtime type errors. This paper explores connections between these two error debugging mechanisms, focusing on how one can help the other. A fundamental observation in our exploration is that blame tracking can be reduced to constraint collection and solving, the two main components of type inference, indicating that the well-accepted gradual typing error debugging mechanism suffers from the bias problem in type inference. This illustrates two problems. First, it limits the ability to use gradual typing as a type-error debugging approach similar to deferred type errors. Second, it means that blame tracking in general may not help programmers find the cause of their runtime type errors. This calls for more research into understanding and improving of blame tracking, particularly in gradual languages that employ type inference to recover type information.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. *SIGPLAN Not.*, 46(1):201–214, January 2011. `doi:10.1145/1925844.1926409`.

2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for Free for Free: Parametricity, with and Without Types. *Proc. ACM Program. Lang.*, 1(ICFP):39:1–39:28, August 2017. `doi:10.1145/3110283`.

3 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding Dynamic Types to C♯. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming*, pages 76–100, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

4 John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating Gradual Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18, New York, NY, USA, 2018. ACM.

5 John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. `doi:10.1145/3236793`.

6 Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. In *ACM SIGPLAN International Conference on Functional Programming*, ICFP 2017, 2017. To appear.

7 Christopher Chambers, Sheng Chen, Duc Le, and Christopher Scaffidi. The function, and dysfunction, of information sources in learning functional programming. *Journal of Computing Sciences in Colleges*, 28(1):220–226, 2012.

8 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA):48:1–48:30, October 2017. `doi:10.1145/3133872`.

9 S. Chen and M. Erwig. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming*, LNCS 8475, pages 35–51, 2014.

10 Sheng Chen and Martin Erwig. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM. `doi:10.1145/2535838.2535863`.

11 Sheng Chen and Martin Erwig. Systematic identification and communication of type errors. *Journal of Functional Programming*, 28:e2, 2018. `doi:10.1017/S095679681700020X`.

12 Sheng Chen, Martin Erwig, and Karl Smeltzer. Exploiting diversity in type checkers for better error messages. *Journal of Visual Languages & Computing*, 39:10–21, 2017. Special Issue on Programming and Modelling Tools. `doi:10.1016/j.jvlc.2016.07.001`.

13 Olaf Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ACM Int. Conf. on Functional Programming*, pages 193–204, September 2001.

14 Hyunjun Eo, Oukseh Lee, and Kwangkeun Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004. `doi:10.1007/BF03037279`.

15 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. `doi:10.1145/581478.581484`.

16 Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. `doi:10.1145/2676726.2676992`.

17 Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003.

18 Jurriaan Hage and Bastiaan Heeren. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages*, pages 199–216. Springer, 2007.

19 Jurriaan Hage and Peter Van Keeken. Mining for Helium. *Technical report UU-CS*, 2006.

20 Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM. `doi:10.1145/871895.871902`.

21 Bastiaan J. Heeren. *Top Quality Type Error Messages.* PhD thesis, Universiteit Utrecht, The Netherlands, September 2005. URL: `http://www.cs.uu.nl/people/bastiaan/phdthesis`.

22 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.*, 1(ICFP):40:1–40:29, August 2017. `doi:10.1145/3110284`.

23 Lintaro Ina and Atsushi Igarashi. Gradual Typing for Generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM. `doi:10.1145/2048066.2048114`.

24 Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986. `doi:10.1145/512644.512649`.

25 Oukseh Lee and Kwangkeun Yi. A Generalized Let-Polymorphic Type Inference Algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

26 B. Lerner, M. Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*, pages 425–434, 2007. `doi:10.1145/1250734.1250783`.

27 Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. A Practical Framework for Type Inference Error Explanation. In *OOPSLA*, pages 781–799, 2016.

28 Bruce J. McAdam. How to repair type errors automatically. In Kevin Hammond and Sharon Curtis, editors, *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), University of Stirling, Bridge of Allan, Scotland, August 22nd to 24th, 2001*, volume 3 of *Trends in Functional Programming*, pages 87–98. Intellect, 2001.

29 Bruce J McAdam. *Reporting Type Errors in Functional Programs.* PhD thesis, Larboratory for Foundations of Computer Science, The University of Edinburgh, 2002.

30 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. `doi:10.1145/3290331`.

31 Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *OOPSLA*, New York, NY, USA, 2017. ACM. `doi:10.1145/3133880`.

32 Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*, pages 15–26, 2003. `doi:10.1145/944705.944708`.

33 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding Minimum Type Error Sources. In *OOPSLA*, pages 525–542, 2014.

34 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Practical SMT-based Type Error Localization. In *ICFP*, pages 412–423, 2015.

35 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

36 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.*, 47(1):481–494, January 2012. `doi:10.1145/2103621.2103714`.

37 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *POPL*, 2015.

38 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM. `doi:10.1145/2635868.2635922`.

39 J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965. `doi:10.1145/321250.321253`.

40 Thomas Schilling. Constraint-Free type error slicing. In *Trends in Functional Programming*, pages 1–16. Springer, 2012.

41 Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic Witnesses for Static Type Errors. In *ACM SIGPLAN International Conference on Functional Programming*, 2016. to appear.

42 Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to Blame: Localizing Novice Type Errors with Data-driven Diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, October 2017. `doi:10.1145/3138818`.

43 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 425–435, New York, NY, USA, 2015. ACM. `doi:10.1145/2737924.2737968`.

44 Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *In Proceedings of the 24th European Symposium on Programming (ESOP'15)*, volume 9032 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2015. Springer-Verlag. `doi:10.1007/978-3-662-46669-8_18`.

45 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

46 Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. `doi:10.1145/1408681.1408688`.

47 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

48 Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 72–83, 2003. `doi:10.1145/871895.871903`.

49 Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91, 2004. `doi:10.1145/1017472.1017486`.

50 F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, January 2001. `doi:10.1145/366378.366379`.

51 Ville Tirronen, SAMUEL UUSI-MÄKELÄ, and VILLE ISOMÖTTÖNEN. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming*, 25:e11, 2015.

52 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. `doi:10.1145/1176617.1176755`.

53 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. `doi:10.1145/1328438.1328486`.

54 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SNAPL.2017.17`.

55 Kanae Tsushima and Olaf Chitil. A Common Framework Using Expected Types for Several Type Debugging Approaches. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, pages 230–246, Cham, 2018. Springer International Publishing.

56 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The Behavior of Gradual Types: A User Study. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 1–12, New York, NY, USA, 2018. ACM. `doi:10.1145/3276945.3276947`.

57 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 762–774, New York, NY, USA, 2017. ACM. `doi:10.1145/3009837.3009849`.

58 Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 341–352, 2012. `doi:10.1145/2364527.2364554`.

59 Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, September 2011.

60 Philip Wadler. A Complement to Blame. In *SNAPL*, 2015.

61 Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. `doi:10.1007/978-3-642-00590-9_1`.

62 Mitchell Wand. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986. `doi:10.1145/512644.512648`.

63 Jeremy Richard Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions.* PhD thesis, The University of Melbourne, January 2006.

64 Baijun Wu, John Peter Campora III, and Sheng Chen. Learning User Friendly Type-error Messages. *Proc. ACM Program. Lang.*, 1(OOPSLA):106:1–106:29, October 2017. `doi:10.1145/3133930`.

65 Baijun Wu and Sheng Chen. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.*, 1(OOPSLA):105:1–105:27, October 2017. `doi:10.1145/3133929`.

66 Jun Yang. *Improving Polymorphic Type Explanations.* PhD thesis, Heriot-Watt University, May 2001.

67 Danfeng Zhang and Andrew C. Myers. Toward General Diagnosis of Static Errors. In *ACM Symp. on Principles of Programming Languages*, pages 569–581, 2014.

68 Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing Type Errors with Class. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–21, 2015.

69 Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.*, 39(4):18:1–18:47, August 2017. `doi:10.1145/3121137`.