# Scaling Up Delta Debugging of Type Errors
## Category: Research

Joanna Sharrad(✉) and Olaf Chitil

University of Kent, Canterbury, UK
{jks31,oc}@kent.ac.uk

**Abstract.** Type error messages of compilers of statically typed functional languages are often inaccurate, making type error debugging hard. Many solutions to the problem have been proposed, but most have been evaluated only with short programs, that is, of fewer than 30 lines. In this paper we note that our own tool for delta debugging type errors scales poorly for large programs. In response we present a new tool that applies a new algorithm for segmenting a large program before the delta debugging algorithm is applied. We propose a framework for quantifying the quality of type error debuggers and apply it to our new tool demonstrating substantial improvement.

**Keywords:** Type error · Error diagnosis · Blackbox · Delta debugging · Haskell

## 1 Introduction

Type errors in statically typed functional languages such as Haskell, ML and OCaml are difficult to understand and repair. The type error message of a compiler gives a location in the ill-typed program, but this location is often far from the defect that needs to be repaired. In over 30 years numerous solutions have been proposed, but none has been widely adopted.

In our opinion the major reason for this non-adoption is the effort required for implementing proposed solutions for full programming languages and maintaining them in the face of evolving languages and compilers. Proposed solutions usually require new compiler front-ends, including new type inference implementations, or substantial modifications of existing compilers. We believe that a small improvement that requires little implementation and maintenance effort is much better than a big improvement that requires substantial effort. Hence it has been our goal to develop a type error debugger that uses the compiler as a true black box, that is, it calls the compiler as an external program.

In an earlier paper we presented and evaluated such a type error debugger [18]. Our debugger implements the *isolating delta debugging* algorithm [28] to locate the defective line in an ill-typed program. Our debugger works solely on a line-based principle, directly adding and removing the lines of the source code

J. Sharrad—PhD Student.


A. Byrski and J. Hughes (Eds.): TFP 2020, LNCS 12222, pp. 71–93, 2020.
https://doi.org/10.1007/978-3-030-57761-2_4

to generate configurations. These configurations, that is variants of the ill-typed program, are tested by calling the compiler. Our debugger does not duplicate compiler work such as parsing, instead it uses minimal information from the outcome of the compiler call; in particular, the only information the debugger uses is whether compilation succeeded (passed), failed with a type error (fail), or failed with some other error (unresolved). As a consequence such a debugger is mostly programming language agnostic.

We showed that our debugger yields good locations in reasonable time for a data-set sample of 121 ill-typed programs, named the CE benchmarks, that had been taken from papers on type error debugging [3]. However, unlike delta debugging of run-time failures, which was evaluated with large programs, successfully finding a fault in a 178,000 line program, all these programs are short; the longest has 23 lines [27]. So for many type error debugging methods proposed in the literature that use this and other data-sets, including our own debugger, it is unknown from their evaluations whether they scale for larger programs. To counter this we introduce a new data-set, named the *scalability* benchmarks, of 80 type errors that we introduced into the large program Pandoc. This data-set provides a starting point for evaluating the scalability of type error solutions.

### 1.1 Brief Example of the Line-Based Problem

As our debugger is line-based it is affected by where the *isolating delta debugging* algorithm chooses to split the source code. The *isolating delta debugging* algorithm tests a logarithmic[1] number of configurations if no outcome is unresolved. For example, an ill-typed program containing just one line will immediately locate the fault on that line from the first configuration whereas an ill-typed program containing 6 lines of code can take three configurations to locate the type error. However, as we have previously said we do not replicate parsing and so every line combination can be a possible configuration. This has the detrimental effect of causing many ill-formed variants; producing a significant number of unresolved results.

Take as a brief example this Haskell program from Stuckey et al. [21] that we used in our previous paper[18]:

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3                 | otherwise = x : y : ys
```

The program is ill-typed. The first line is incorrect the x should be a list of x. The Glasgow Haskell compiler[2] gives us line 2 as the incorrect line, whereas our previous debugger correctly points out line 1. However, even in this three line program we still receive unresolved results from the *isolating delta debugging* algorithm. For example the following configuration returns a parse error:

[1] With respect to the number of lines of the original ill-typed program.
[2] https://www.haskell.org/ghc, version 8.4.3.

```
1
2
3                 | otherwise = x : y : ys
```

The more outcomes are unresolved, the less efficient *isolating delta debugging* becomes, up to a quadratic number of configurations. "When using ... [isolating delta debugging], it is thus wise to keep unresolved test outcomes to a minimum, as this keeps down the number of tests required" [28]. All applications implementing *isolating delta debugging* try to minimise the number of configurations with unresolved test outcomes. In our application the root cause of most unresolved outcomes are parse errors. Building some kind of parser for our debugger would contradict our goals. Hence here we present an algorithm, Moiety, that calls the compiler as a black box. Moiety detects the lines within an ill-typed program that are valid splitting points. A moiety is a configuration of the original program that consists of consecutive lines that should not be split. If a moiety is split, then compilation will produce a parse error. In summary, the moiety information guides the *isolating delta debugging* algorithm to reduce unresolved test outcomes and thus reduce the time taken for the algorithm to run.

We implemented the new type error debugger, Elucidate20. It combines the new moiety algorithm with an *isolating delta debugging* algorithm that uses moiety information. The debugger locates a defective line of an ill-typed Haskell program, using the Glasgow Haskell compiler as black box. To debug large programs, Elucidate20, unlike our previous debugger, also supports multi-module programs and a standard project build tool.

In this paper we make the following contributions:

- We present the moiety algorithm, which generates, using the compiler as a black box, a set of moieties of the ill-typed program. That set determines the configurations for the *isolating delta debugging* (Sect. 3).
- We propose a framework for quantifying the quality of type error debuggers (Sect. 4).
- We introduce a new data-set of 80 ill-typed variants of the program Pandoc (Sect. 5)
- We evaluate Elucidate20 to see whether the moiety algorithm reduces unresolved results and thus the run-time of the *isolating delta debugging* algorithm. We use our new framework and scalability data-set (Sect. 5).

## 2 The Problem

### 2.1 Delta Debugging Type Errors

Let us briefly review what delta debugging is and how we applied it to type error debugging [18].

To locate the defect in an ill-typed program, many programmers simply remove (or comment out) some parts of the program and compile the smaller program. If the smaller program is also ill-typed, the procedure is repeated.

If the smaller program is not ill-typed, a different part of the previous program is removed. This shrinking by trial and error repeats until the program cannot shrink further, that is, no smaller program is ill-typed.

*Simplifying delta debugging* [27,28] is a greedy algorithm that automates this method. Simplifying delta debugging divides the program into two halves and tests each one. If one half is ill-typed, the algorithm calls itself recursively for that half. If neither half is ill-typed, it divides the program into four parts and tests each one. Again the algorithm calls itself recursively for any ill-typed part, but if none is ill-typed, it tries again by dividing the program into eight parts. When the program cannot be divided further, the algorithm stops with the last ill-typed program as result.

Recall that testing a program yields one of *three outcomes*: *fail* (ill-typed), *pass* (well-typed) or *unresolved* (any other error such as parse error or unbound identifier). For the simplifying delta debugging algorithm it does not matter whether an outcome is pass or unresolved, but for the *isolating delta debugging* algorithm, which we actually use, the difference is essential.

A program variant that may be tested is called a *configuration*. For type error debugging we made the same choice of configurations as many other implementations of delta debugging: we chose to always remove whole lines of the ill-typed program[3]. Hence a configuration is the original ill-typed program with some lines replaced by empty lines[4]. A configuration being a subconfiguration of another configuration is a natural partial order on configurations, with the empty configuration, consisting of many empty lines, being the minimum and the original ill-typed program being the maximum.

A minimal ill-typed program is often still big, because for every function or type that it uses it has to includes its definition, which is usually well-typed. To isolate a cause of the type error we want to exclude these well-typed definitions. Therefore we decided to use the *isolating delta debugging* algorithm for type error debugging.

The *isolating delta debugging* algorithm [6,28,29] works with a pair of configurations, a passing and a failing configuration, the former being a subconfiguration of the latter. The algorithm starts with the empty configuration as passing configuration and the ill-typed program as failing configuration. The algorithm divides the difference between the two configurations into two parts and tests the passing configuration with each of these parts added and the failing configuration with each of these parts removed. If any tested configuration yields a passing outcome, it can become the new passing configuration, if any tested configuration yields a failing outcome, it can become the new failing configuration; then the algorithm calls itself recursively with a new pair of configurations. If all of the tested configurations yield unresolved outcomes, the difference is divided instead into four, eight, etc. parts, similar to the simplifying delta debugging algorithm, until eventually a passing or failing configuration is found; if no further division

[3] Removing single characters is another choice presented by Zeller [28].

[4] Instead of removing the lines completely we still keep the empty lines to avoid undesirable changes of program layout.

is possible, the algorithm terminates. The algorithm does not specify how the difference between two configurations is divided into parts and there may be several passing and failing outcomes; thus the algorithm is non-deterministic; however, like any other implementation, ours makes a choice and thus is deterministic [1,12]. In every recursive call the passing configuration is a subconfiguration of the failing configuration (and both are subconfigurations of the original ill-typed program). Every recursive call reduces the difference between the two configurations, until the difference cannot be reduced any further.

The final result of *isolating delta debugging* is a pair of configurations, where the first configuration is a passing subconfiguration of the second failing configuration, such that there exists no passing or failing configuration between the two configurations. The algorithm is greedy to limit run-time and it is not guaranteed to return a pair of configurations with minimal difference.

The final pair of configurations is the result of the *isolating delta debugging* algorithm. The difference between the two configurations, which may be neither a passing nor a failing configuration, isolates a failure cause. This difference is the result of our type error debugger.

### 2.2 The Effect of Unresolved Outcomes

Because our definition of configuration is based on program lines, all complexity measures of type error debugging are with respect to the number of lines of the ill-typed program. For a given ill-typed program there exists an exponential number of configurations. Already finding a failing configuration of minimal size is known to be NP-complete [11].

In type error debugging nearly all run-time is spent in the tests made by the compiler. In general, the run-time of delta debugging is proportional to the number of tests made.[5]

We see from the description of delta debugging that if no test outcome is unresolved, it is basically a binary search. In contrast, frequent unresolved outcomes cause the algorithm to repeatedly divide (differences of) configurations into four, eight, etc. parts and make more tests. If every configuration is unresolved the algorithm starts to generate configurations that contain a single line until all lines of the program have been checked[6]. So as we already stated in the introduction, the *isolating delta debugging* algorithm has logarithmic time complexity if no outcome is unresolved and becomes less efficient, up to a quadratic time complexity, with many unresolved outcomes. Therefore any successful application of delta debugging makes some effort to avoid unresolved outcomes.

The issue with many unresolved outcomes can be shown more clearly within our earlier results [18]. These 900 programs were generated by concatenating pairs of some of the original small CE benchmark programs. For space reasons, we have ordered the 900 programs by number of lines and put them into 4 groups: the shortest 225 in the first group, the next 225 in the second group, etc.

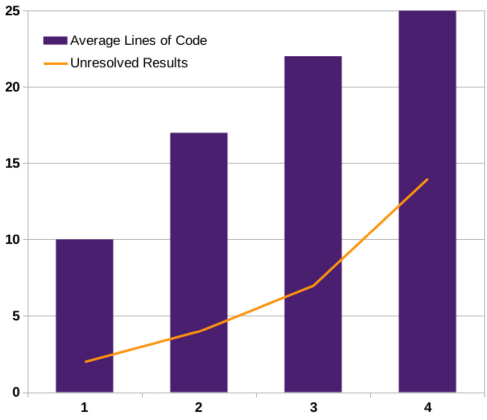
[5] This assumes similar run-time for every test, which may not be the case.

[6] A proof is available on page 408 of 'Why Programs Fail' [28].

Table 1 on the left shows the average outcomes and indicates that the number of unresolved results grows substantially with program size which can be seen more clearly with the a graphically representation on the right.

**Table 1.** Average number of unresolved outcomes compared to number of lines of code.

| # lines | # unresolveds |
|---|---|
| 10 | 2 |
| 17 | 4 |
| 22 | 7 |
| 25 | 14 |




However, we noted in the introduction that ill-typed programs that have been used to evaluate type error debuggers are short. The longest program in the CE benchmark suite [3] of 121 programs has just 23 lines. Such programs are good for studying how a type error debugger works and many of these programs are representative for the first programs written by novices learning a functional programming language. But they do not show us how a type error debugger will scale as not just novices need help with type error debugging, but also more experienced functional programmers who build useful, real-world programs.

In October 2019 we measured the top 100 Haskell programs on the popular public repository GitHub[7]. On average each program has 31872 lines of code, 138 modules, and 229 lines of code per module, far from the 23 lines mentioned above. Even though our type error debugger processes small programs in a few seconds, when applied to programs such as these that contain on average a few hundred lines, due to a hefty number of unresolved results, it could take substantially longer, which is unacceptable [18].

As already stated there is an obvious suspect for the high number of unresolved outcomes in larger programs: although splitting multiple equations of a single function definition yields well-formed definitions in Haskell, splitting a multi-line equation into half usually yields ill-formed programs; the same holds for multi-line type declarations, which often appear in larger programs, and case expressions with a branch per line. Many configurations are simply unparsable!

[7] https://github.com/search?l=Haskell&q=Haskell&s=stars&type=Repositories.

**Table 2.** Number of error messages giving unresolved outcome.

| Error message | # |
|---|---|
| The last statement in a 'do' block must be an expression | 4 |
| Variable not in scope | 4 |
| Not in scope: | 5 |
| Empty 'do' block | 5 |
| Parse error (incorrect indentation or mismatched brackets) | 7 |
| Empty list of alternatives in case expression | 8 |
| The type signature...lacks an accompanying binding | 16 |
| Parse error on input | 77 |
| Total | 126 |

To test our suspicion, we chose the most popular software from our Github results Pandoc, to initiate our *scalability* data-set. As a initial test we introduced a single type error in a single module. The ill-typed module has 87 lines and our debugger had 126 unresolved outcomes, which we categorise by error message of the Glasgow Haskell compiler in Table 2. Most error messages are related to parsing and "parse error on input" is by far the most frequent one.

## 3 The Moiety Algorithm and Delta Debugging

We always obtain a configuration that does not parse, if we split the original ill-typed program at certain consecutive lines. Given its dominance, we solely focus on the "parse error on input" error message. These indicate that parsing failed somewhere inside the configuration, whereas for example "parse error (incorrect indentation or mismatched brackets)" indicates that parsing fails at the end of the configuration. Concentrating on the former means our algorithm has the ability to distinguish between the two. So we use this information to first determine which lines that should never be separated as they will cause a "parse error on input" and then apply the delta debugging algorithm such that it never splits in these places.

We name our pre-processing algorithm *Moiety*; according to the Merriam-Webster dictionary a moiety is "one of the portions into which something is divided".[8] Moiety divides the ill-typed program into moieties, that is, what position in the source code we can split the lines. This is represented as a tuple, with the starting and ending points of the splits.

[8] https://www.merriam-webster.com/dictionary/moiety.

### 3.1 Illustration of the Algorithm

The moiety algorithm is designed to reduce unresolved, "parse error on input", results from large programs. However, to present how moiety works concisely we have to consider the following small ill-typed program[9]:

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
6 fib x = case x of
7    0 -> f x
8    1 -> f x
9    n -> fib (n-1) `plus` fib (n-2)
```

To limit runtime, the algorithm may only traverse the program once from beginning to end to produce its set of moieties; no line in the program is submitted to the compiler duplicate times. Moiety calls the compiler to test a program for whether it yields "parse error on input" or not. We show the tested program on the left and the test outcome and resulting moiety set on the right. We note that line 1 never yields "parse error on input" so start with line 2.

```
1
2       0 -> [0]
3
4
5
6
7
8
9
```

"parse error on input"

moieties:`()`

As line 2 produces a "parse error on input" it cannot be the starting line for a plausible split; and so we continue with line 3:

```
1
2
3    1 -> 1
4
5
6
7
8
9
```

"parse error on input"

moieties:`()`

[9] It should be noted when we talk about small programs in type error debugging we are discussing those that are used for evaluation and not those used for examples.

Like line 2, line 3 also cannot start a new moiety; we continue with line 4:

```
1
2                                   not "parse error on input"
3
4 plus :: Int -> Int -> Int         moieties:(3,4)
5
6
7
8
9
```

Line 4 is not a "parse error on input" so we can create a new moiety. We can successfully split line 4 from line 3; and so line 3 is our starting point and line 4 is our finishing points of our first moiety. Next line 5:

```
1
2                                   not "parse error on input"
3
4                                   moieties:(3,4) (4,5)
5 plus = (+)
6
7
8
9
```

Likewise, line 5 starts a new moiety as it can be split from line 4. We move on to line 6:

```
1
2                                   not "parse error on input"
3
4                                   moieties:(3,4) (4,5) (5,6)
5
6 fib x = case x of
7
8
9
```

So line 6 starts a new moiety too. We continue with line 7:

```
1
2                                   "parse error on input"
3
4                                   moieties:(3,4) (4,5) (5,6)
5
6
7       0 -> f x
8
9
```

At this point it is hopefully obvious that lines 8 and 9 each also gives the outcome "parse error on input" and so the algorithm finishes with the moieties `(3,4) (4,5) (5,6)`.

Working through the example shows how simple the moiety algorithm is: The algorithm tests every single line of the original ill-typed program whether it yields "parse error on input" or not. In case of the former, the line cannot be split from the preceding lines so no moiety can be generated. Otherwise it does start a new moiety. The result is an ordered set of moieties, two lines that can be successfully split.

### 3.2 Example of Isolating Delta Debugging with Moieties

In the subsequent *isolating delta debugging* algorithm moieties are never split, simply by redefining a configuration as a subset of moieties.

So in our example we have the moiety list `(3,4) (4,5) (5,6)`.

We start *isolating delta debugging* with the passing configuration {} and the failing configuration {`[1,2,3],[4],[5], [6,7,8,9]`}. As we can see our failing configuration, of source code line numbers, is now split using the moieties.[10] We divide the difference between the two configurations by two and hence test the configurations {`[1,2,3],[4]`} and {`[5],[6,7,8,9]`}. Both configurations give the outcome unresolved. Hence we have to divide the difference between our passing and failing configuration by four and test the configurations {`[1,2,3]`}, {`[4]`}, {`[5]`}, {`[6,7,8,9]`} and the configurations {`[4],[5],[6,7,8,9]`}, {`[1,2,3],[5],[6,7,8,9]`}, {`[1,2,3],[4],[6,7,8,9]`}, {`[1,2,3],[4],[5]`}. Our implementation happens to test {`[5]`} first and the test gives outcome pass.

Next, *isolating delta debugging* calls itself recursively with the new passing configuration {`[5]`} and the failing configuration {`[1,2,3],[4],[5],[6,7,8,9]`}. We divide the difference, which is 3 moieties, by two and hence test the configurations {`[1,2,3],[4],[5]`} and {`[5],[6,7,8,9]`}. The first configuration gives outcome fail.

Next, *isolating delta debugging* calls itself recursively with the old passing configuration {`[5]`} and the new failing configuration {`[1,2,3],[4],[5]`}.

[10] {`[1],[2], [3],[4],[5],[6],[7],[8],[9]`} represents the non-moiety failing configuration. Every line is an acceptable splitting location.

We divide the difference by two and hence test the configurations {`[1,2,3],[5]`} and {`[4],[5]`}. The first configuration gives outcome fail.

Finally, *isolating delta debugging* calls itself recursively with the old passing configuration {`[5]`} and the new failing configuration {`[1,2,3],[5]`}. Because the difference between the two configurations is only one moiety, the algorithm terminates with the these two configurations as result. Our type debugger returns the difference between these two configurations as the location of the defect: {1, 2, 3}. The actual type error is in line 2, but our type debugger can return at best a single moiety and its preceding lines.

### 3.3 Time Complexity

We designed the moiety algorithm to return a list of moieties in the shortest time possible, that is linear in the number of lines of the ill-typed program. We know that *isolating delta debugging* takes between logarithmic and quadratic time, now in the number of moieties. Because moieties avoid the most common type of unresolved outcome, we hope that overall the time complexity of type error debugging is close to linear.

## 4 A Framework for Type Error Debugging

In data science using model metrics such as Accuracy, Precision, and Recall are an accepted standard [19,26]. Yet within type error debugging evaluations only recall, whether a type error has been located correctly or not, run-time, and the authors personal goals are deemed important [4,16]. We disagree with using only one metric as it can bias results, and in later works authors seem to agree [17,30]. However, even though we are slowly seeing other metrics joining recall in type error debugging evaluations they are not representing the same formulas, and so we are proposing the following as a framework for future evaluations to allow for ease of solution comparison (Table 3).

**Table 3.** Terminology

| Shorthand | Longhand | Equivalents |
|---|---|---|
| **Data science** | | |
| TP | True Positive | |
| TN | True Negative | |
| FP | False Positive | |
| FN | False Negative | |
| **Our terms** | | |
| $R_L$ | Reported lines (number of lines returned) | TP + FP |
| $R_E$ | Reported errors (number of correct errors) | TP |
| $U_R$ | Unreported lines (number of correct unreported lines) | TN |
| L | Lines of code (total source code) | TN + TP + FN + FP |
| E | Errors (number of errors in the code) | TP + FN |

### 4.1 The Metrics

*Accuracy* tells us the typical distance from a measure to the optimum value. For our domain, number of lines correctly excluded plus correctly reported lines containing a type error. However, this is problematic as we receive a high number of True Negative answers, number of correct lines ignored, and so this is generally ignored in the type error debugging domain in favour of recall.

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} = \frac{U_R + R_E}{L} \tag{1}$$

*Recall,* aka sensitivity, is the measure of the quantity of elements correctly returned.

$$Recall = \frac{TP}{TP + FN} = \frac{R_E}{E} \tag{2}$$

For type errors this measures the number of errors that are reported correctly compared to the number of errors within the source code. As already noted, this metric is most used in type error debugging evaluations. It shows if a debugger can successfully discover the correct number of type errors within an ill-typed program. However, like Accuracy, it is not without fault as the following example will show.

Lets us assume we have an ill-typed program containing 8 lines (L = 8) and 1 type error (E = 1). We run a debugger and it returns all 8 lines of code as containing the type error ($R_L = 8$) and, obviously, returns the correct line location within this ($R_E = 1$). Most type error debugging evaluations do not mention the amount of lines returned, only if their debugger located the line correctly. If recall is used as the only metric in evaluations we end up being able to state that this example shows our debugger is 100% correct.

$$Recall = \frac{R_E}{E} = \frac{1}{1} = 100\% \tag{3}$$

This, to us, is obviously incorrect, yet the metric proves it to be true. To counter this issue Data Science employs another metric.

*Precision,* also known as positive predictive value, is the number of elements within the entire returned set of results.

$$Precision = \frac{TP}{TP + FP} = \frac{R_E}{R_L} \tag{4}$$

Mapped to our domain it is the number of correct lines of code reported by the debugger compared to the total number of lines returned. Precision allows us to see if we have returned the correct location as one single line versus returning a correct location within several lines.

Applying precision to our ongoing example we receive:

$$Precision = \frac{R_E}{R_L} = \frac{1}{8} = 12.5\% \tag{5}$$

As can be seen this is a significant difference from our results from recall, however it is also not practical to use Precision as a singular metric either due its reliance on False Positives, some of the lines returned do not contain a type error. This is where the Data Science domain employs the $F_1$ Score.

$F_1$ *Score* is calculated from the harmony mean of the two metrics *Recall* and *Precision*. This produces an accuracy measure that accounts for the imbalance of data within type error debugging, meaning the $F_1$ score is crucial in showing the true results of evaluations.

$$F_1 = 2\frac{Precision \cdot Recall}{Precision + Recall} = 2\frac{R_E}{E + R_L} \tag{6}$$

Now with our example we can see meaningful feedback for evaluation.

$$F_1 = 2\frac{R_E}{E + R_L} = 2\frac{1}{1 + 8} = 22\% \tag{7}$$

With this framework we can now generate easily comparable evaluations for future work in the type error debugging domain.

## 5 Evaluating Our Method

We now apply our method on a single real-world program to test scalability; Pandoc is a Haskell library for markup conversion, it has a total of 64,467 lines of code with an average of 430 lines of code per module in 150 modules. We place within Pandoc 80 individual type errors into 40 of its modules (using each module twice) of which each contain between 32 and 2305 lines of code (Table 4).

**Table 4.** Lines of code per module with associated errors

| Errors | LoC | Errors | LoC | Errors | LoC | Errors | LoC |
|---|---|---|---|---|---|---|---|
| {1, 2} | 32 | {21, 22} | 73 | {41, 42} | 156 | {61, 62} | 238 |
| {3, 4} | 37 | {23, 24} | 77 | {43, 44} | 167 | {63, 64} | 240 |
| {5, 6} | 45 | {25, 26} | 79 | {45, 46} | 187 | {65, 66} | 258 |
| {7, 8} | 48 | {27, 28} | 83 | {47, 48} | 192 | {67, 68} | 261 |
| {9, 10} | 48 | {29, 30} | 86 | {49, 50} | 204 | {69, 70} | 266 |
| {11, 12} | 52 | {31, 32} | 86 | {51, 52} | 205 | {71, 72} | 271 |
| {13, 14} | 58 | {33, 34} | 91 | {53, 54} | 212 | {73, 74} | 275 |
| {15, 16} | 58 | {35, 36} | 94 | {55, 56} | 213 | {75, 76} | 278 |
| {17, 18} | 65 | {37, 38} | 140 | {57, 58} | 214 | {77, 78} | 287 |
| {19, 20} | 68 | {39, 40} | 155 | {59, 60} | 227 | {79, 80} | 2305 |

The modules chosen were the first 39 in size order that contained code that could be made ill-typed. The last module was the largest module Pandoc contained at 2305 lines. The placement of the error was decided upon by a random

**Table 5.** Type error categories

| Category | Errors total |
|---|---|
| **Couldn't match...** | **79** |
| Rigid type variable bound by the type signature .. | 5 |
| In the ? field of a record ..In the expression .. | 3 |
| **...In the expression:?...** | **22** |
| In an equation ? .. | 7 |
| In a stmt of a 'do' block ? .. | 3 |
| In a case alternative ? .. | 7 |
| In the expression: ? .. | 5 |
| **...In the ? argument of ?...** | **20** |
| In the expression ? In an equation for ? ... | 7 |
| In a stmt of a 'do' block ? .. | 11 |
| In the ? argument of ?.. | 2 |
| **...In the pattern: ?...** | **3** |
| In a case alternative ? In the expression ? .. | 2 |
| In equation ? .. | 1 |
| **...is applied to...arguments ...** | **26** |
| Possible cause ? is applied to too many arguments .. | 3 |
| Probable cause ? is applied to too few arguments .. | 11 |
| The function ? is applied to ? argument/s .. | 12 |
| **Couldn't deduce...** | **1** |
| Arising from a use of ? from the context ? bound by the type signature | 1 |

number generator. If the line suggested was unsuitable for type error placement the generator was re-run. The type errors were inserted manually with no prior planning on the category of type error. The categories, listed by the individual error message presented by GHC, can be seen in Table 5. To note, all of the type errors inserted are Equality Errors as according to *TcErrors*[11].

We compare our debugger, Elucidate20, with Gramarye19. Gramarye19 is a modified version of our previous debugger Gramarye; and like Elucidate20 now supports the following features:

*Modular Programs.* The type error location of a compiler is unreliable, but our type error debugger assumes that the first module identified by the compiler as ill-typed does contain the type error location; our type error debugger works solely on that module. If a module causes the first compiler type error, then all modules directly or indirectly imported are well-typed. An identifier defined in an imported module may have a type that contradicts with how the identifier is used in the ill-typed module. However, even when both definition and use are

[11] TcErrors is part of the Glasgow Haskell Compiler and states that type errors fall into one of 4 groups; more information about this can be found in: https://github.com/JoannaSharrad/ghcErrorsDoc/blob/master/RoughGuidetoGHCTcErrors.pdf.

in the same module and the definition is typable, delta debugging will always identify the use of the identifier as the cause of the error, not the definition. So our treatment of modules is consistent with our general treatment of definition vs. use.

*Haskell Specific Language.* There are some language declarations that should be ignored when removing lines as they will always lead to an unresolved result. Hence our type error debugger leaves these declarations in all configurations tested by the delta debugging algorithm. The following are never the location of a type error; import declarations, single line comments, multi-line comments, and module declaration. Unfortunately, recognising lines with these declarations is specific to the programming language Haskell, and thus removes the agnostic status from the delta debugging algorithm.

*The Build Tool.* When measuring the top 100 Haskell programs on GitHub, we found that they all use Cabal[12] for packaging and building. Therefore our type error debugger has a flag to call the build tool cabal instead of the Glasgow Haskell compiler for testing. When cabal is used, the user has to state the target program instead of the ill-typed module.

Though the above have been added as features to both Gramarye19 and Elucidate20, the latter still keeps delta debugging free of the moiety pre-processing [18].

For this evaluation we ran our benchmarks on an AMD Phenom X4, 32 GB RAM, Samsung SSD 850, PC running Ubuntu 18.04LTS to answer the following questions:

1. Does the Moiety algorithm reduce the number of unresolved results?
2. Does the pre-processing affect the time taken by *Isolating Delta Debugging*?
3. Does applying the new framework quantify the quality of the debugger?

### 5.1 Reduction of Unresolved Results

*Question: Does the Moiety algorithm reduce the number of Unresolved results?*

The moiety algorithm produces a set of splitting locations in the source code. Our scalability benchmark contained a total of 16264 lines of code of which 16184 were places that the *isolating delta debugging* algorithm was allowed to split. Pre-processing the source code using the moiety algorithm we see that out of these 7953 (68%) were plausible splitting points. On average 39% of a single benchmark could not be split without causing a "parse error on input".

In Fig. 1 we can see the number of unresolved outcomes, on the y axis, for each of the 80 type errors in the scalability benchmark listed on the x axis. For the desired outcome we want the bar to be close to zero. For ease of reading we have capped Fig. 1 at a maximum of 170 unresolved results, however it is worth noting that Gramarye19 returned seven results higher than this with modules

[12] https://www.haskell.org/cabal/.

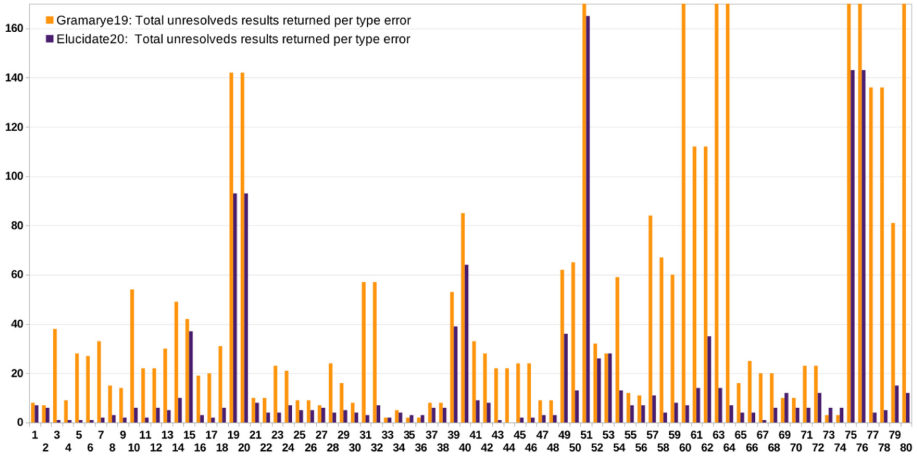



**Fig. 1.** Unresolved results per introduced type error

51, 60, 63, 64, 75, 76 and 80 returning 265, 395, 1436, 1436, 221, 221, and 504 unresolved results respectively. The highest outcome of unresolveds from Elucidate20 was 165, with its lowest being 0 compared to Gramarye19 with 2.

On average there are 16 unresolved outcomes per type error from Elucidate20 compared to Gramarye19 at 88; meaning a reduction of 72 calls to the blackbox compiler. The importance of reducing calls is seen in benchmark 64 a module with 240 lines of code; here Gramarye19 has 1436 unresolved outcomes and takes just over an hour to run the *isolating delta debugging* algorithm whereas Elucidate20 receives only 7 unresolved results and the time taken drops to just 36 s, a difference of around 52 min.

Elucidate20 has an significant impact, totalling a removal of 5743 unresolved outcomes from the entire benchmark, over Gramarye19. However, though we have seen, with benchmark 64, that the delta debugging Run-Time can be reduced does the Moiety algorithm make a reduction to all of our benchmarks?

### 5.2 The Run-Time Speeds

*Question: Does the pre-processing effect the time taken by Isolating Delta Debugging?*

With the unresolved results minimised we hypotheses that the time taken by delta debugging should reduce. In Fig. 2 we show the outcome of just the run-time of delta debugging (excluding pre-processing) in seconds on the y axis, and again each type error on the x axis. As in Sect. 5.1 we have again modified the figure so that we can see the data more clearly by dropping off the most extreme results of Gramarye19 in type errors 60, 63, 64, and 80 who returned run-time results of 1295 s (21 m 35 s), 4299 s (1 h 11 m 39 s), 4201 s (1 h 10 m 1 s), and 1482 s (24 m 42 s) each. The highest result from Elucidate20 is 436 s (7 m 16 s), with the lowest being recorded at 16 s compared to Gramarye19 at 21 s.

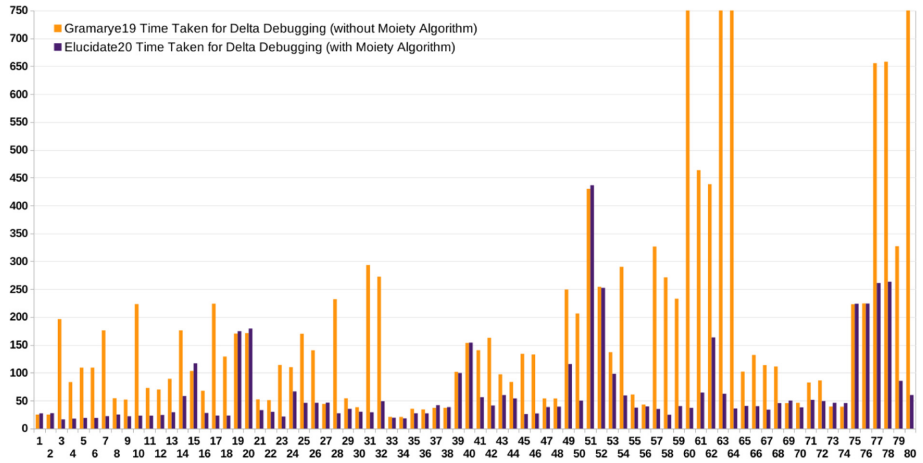



**Fig. 2.** Delta debugging run-time

On average Gramarye19 took 285 s (4 m 45 s) to run the *isolating delta debugging* algorithm, 219 (3 m 39 s) more than Elucidate20 at 66 (1 m 6 s) showing a clear link between total unresolved outcomes received and the time taken to locate a type error. In total Elucidate20 reduced the time taken by *isolating delta debugging* algorithm for the entire benchmark by 4 h 52 m 8 s.

However, when running a debugger the user experiences the entire process not just the algorithm locating the type errors. Our pre-processing is linear, based on lines of code in the program, and the length equals the amount of calls we need to make to the blackbox compiler. Gramarye19 with its lack of moiety algorithm takes on average 303 (5 m 3 s) compared to Elucidate20 at 419 (6 m 59 s). It is clear to see that when using our moiety algorithm we are around a minute slower than our previous debugger. This issue with pre-processing is down to the calling of the compiler as a blackbox. In the case of the scalability benchmark we are calling the build tool Cabal. As an example, if we take our worst case result, benchmark 79, we can see that we reduce the run-time of the *isolating delta debugging* algorithm from 327 s (5 m 27 s) to 85 s (1 m 25 s), however the user-time is increased from 330 s (5 m 30) to the awful 4888 s (1 h 21 m 28 s). If we look at this benchmark it is 2306 lines of code and every call to Cabal takes around 2 s. If we apply 2 s exactly to every line of code we can see that we get a result of 4612 s (1 h 16 m 52 s) close to our worst case benchmark. However, the pre-processing method does have occasional successes in improving overall debugging time, with Elucidate20 reducing the user-time for some of our benchmarks by over an hour.

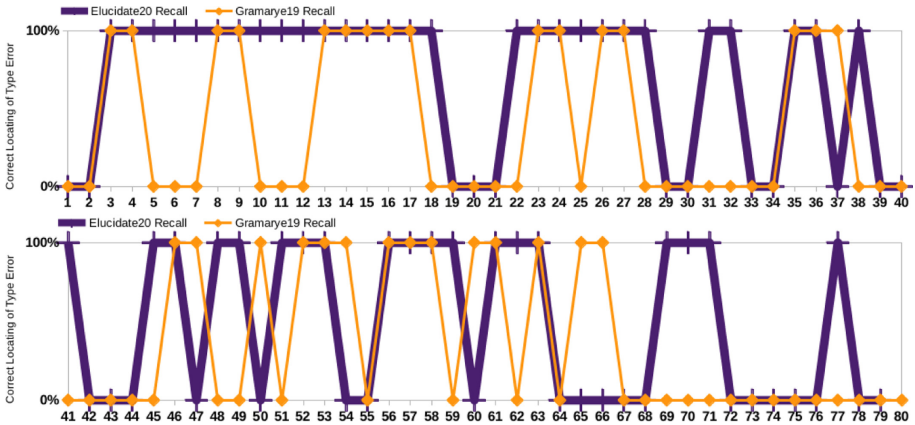



**Fig. 3.** Recall

### 5.3 Applying the Framework

In Fig. 3 and Table 6 we present the data from applying the framework to our results. We display the outcome of *recall* in more depth as to mimic other type error debugging evaluations. The two graphs show all 80 modules on the x axis and if the type error they contained were either correctly located (100%) or not (0%) or the y axis.

The framework results table shows the average outcome for all four of our metrics. The higher the percentage the more desirable.

**Table 6.** Framework results - average

| Metric | Gramarye19 | Elucidate20 |
|---|---|---|
| Accuracy | 94% | 88% |
| Recall | 38% | 59% |
| Precision | 16% | 14% |
| $F_1$ Score | 20% | 19% |

*Question: Does applying the new framework quantify the quality of the debugger?* Recall shows us if the debugger has returned the correct type error specified. As we only have a single type error per benchmark our result is binary. Elucidate20 correctly locates 59% (47/80) of the type errors compared to Gramarye19 which returns fewer correct type errors at 38% (30/80). This rise in correct results from Elucidate20 is directly linked to the pre-processing of the source code. Firstly, as we are passing a new configuration to delta debugging, setting out how to split our lines, we have the chance to generate an alternative pathway of modifications leading to different results from our Blackbox compiler; as the path that the debugger takes relies on these outcomes an alternative result can

happen. Secondly as our method does not allow the splitting of lines outside the moieties we gain the bias of returning a greater set of locations and so increasing our chances of success. As described in Sect. 4 this bias can allow us to return 100 results as suggested locations with an ill-typed program of only 100 lines; we can say that this would not make a suitable solution and so is countered with the precision metric.

In the Table 6 we see that indeed Gramarye19 is more precise than Elucidate20; however overall this only accounts for a difference of 2% points meaning we need to invoke the $F_1$ score for an accurate reading.

The $F_1$ score blends our metric results, Recall and Precision, to form a true overview of the results, as already mentioned this is the harmony mean of the two metrics. With this set of benchmarks we receive a 1% difference between the presented debugger Elucidate20 and the previous Gramarye19, with the latter providing a higher $F_1$ score. This outcome is not surprising; the precision of Elucidate20 is hampered by the moiety algorithm. However, we do not see this as a negative; it was our aim to avoid causing unresolveds and as such these are the most precise result we can currently return for the specific benchmarks utilised in this evaluation. This outcome was also positive evidence that shows the importance of using more than one metric when evaluation debugging solutions, and works well to indicate that many metrics are needed to present the true quality of a type error debugger.

### 5.4 Summary

Applying the moiety algorithm successfully reduced the number of unresolved outcomes significantly. This in turn reduced the time taken for the *isolating delta debugging* algorithm to run by an average of around 3 min. However, for the actual time the user experiences we must include the pre-processing that moiety provides. In doing so we found that calling the build tool Cabal as our blackbox compiler clearly gave unsatisfying results and that work is needed to reduce the time of each blackbox compiler call. When applying the framework we found that using the de facto recall metric did show improved results for Elucidate20. However, when we added the metrics precision and $F_1$ score from the framework a more accurate picture was presented with Elucidate20's results being slightly lower than Gramarye19.

In all, we have improved the time taken by *isolating delta debugging*, we have detected further work for reducing the time-taken by calling a Blackbox Compiler, and have shown the need for the framework to quantify the quality of type error debuggers in the future.

## 6 Related Work

Type Error Debugging research has a long and fruitful history starting in the eighties[25]. It spans many solutions in a variety of categories each specialising on their own core ideas [2,4,5,7,10,13,15,16,20,22,24,30]. However, these solutions

rarely contain a through evaluation and when they do it does not attempt to directly evaluate on large programs with type errors. Instead the evaluations aim for success on small programs, typically of the size that first-time programmers would produce. For example from a recent paper *'Learning to blame: localizing novice type errors with data-driven diagnosis'* though the evaluation mentions the usage of both accuracy and recall the authors state; "We acknowledge, of course, that students are not industrial programmers and our results may not translate to large-scale software development..." [17] and in a well-known type error debugging paper *Counter-Factual Typing for Debugging Type Errors* the authors say "...the numbers do not tell much about how the systems would perform in everyday practice." [4]. One general method of debugging that has been applied to a 178,000 line program is Delta Debugging. Defined by Zeller in 1999, delta debugging comes in two forms Simplifying and Isolating and is applied to a general debugging domain rather than specific categories of errors [6,27–29]. In our previous paper we applied Zeller's work specifically to type errors in functional languages employing the compiler as a blackbox [18]. A Blackbox Compiler differs from other Blackbox solutions mentioned in prior literature(Blackbox Type Checkers, Blackbox Type Inference [9,15,23]) as it treats the entire compiler as an external entity rather than a component of it. This method of only taking external cues, such as whether a program is ill or well-typed, avoids users having to patch or download a specific compiler to explicitly improve type error discovery. Though combining a Blackbox Compiler and *isolating delta debugging* to the domain of type errors returned positive results reducing unresolveds was seen to be beneficial future work. One option for reducing the unresolveds was the modification of the delta debugging configuration. Generating Configurations to avoid invalid inputs for delta debugging is not new [11,14]. The closest to our work observes that modifying lines of source code can and will generate broken code that will still need to be sent to the test function causing debugging times to increase [8]. In *Binary Reduction of Dependency Graphs* the authors aim to reduce these invalid inputs by using dependency graphs to map the smallest set of classes that are invalid without each other, reference's to other classes, in Java bytecode. Their dependency analysis is specific for Java and they only use the *simplifying delta debugging* algorithm.

## 7 Conclusion and Future Work

We presented a method of combining *Isolating Delta Debugging* and a blackbox compiler to locate type errors. Most solutions in type error debugging do not evaluate on large programs, those that have more than 30 lines of code, and so we aimed to target these. However, when applying *isolating delta debugging* to locate type errors in these large programs we can receive outcomes that are unresolved, it can split source code in a way that causes parse errors, that reduce the efficiency. We introduce an algorithm that pre-processes an ill-typed program to eliminate these parse error; in particular 'parse errors on input'. Our pre-processing algorithm, Moiety, presents where in the source code the lines can

be split to avoid causing a parse errors. These moieties are then used as a configuration for delta debugging to reduce the unresolveds caused by parse errors, which in turn is linked to the time taken in delta debugging large programs.

To test the success of our solution on locating type errors in large programs we introduced the scalability benchmarks, a set of 80 ill-typed large programs within the real-world program Pandoc, and a framework based on Data Science standards. The evaluation comprised of comparing our original debugger, Gramarye19, that used *isolating delta debugging* to locate type errors and our new debugger, Elucidate20, that also include the pre-processing algorithm moiety. In the first part of the evaluation we saw if a reduction of unresolved results and a decrease in the *isolating delta debugging* algorithms run-time could be achieved. Elucidate20 on average returned 72 fewer unresolveds per benchmark reducing the time taken for *isolating delta debugging* to run by an average of 216 s (3 m 36 s). The best case reduction of time was from over an hour to 7 s, however, the overall time the user experiences was a priority too. Here, with the combination of moiety and *isolating delta debugging*, Elucidate20 did take longer than our previous debugger to locate type errors on average with an increase of 100 s (1 m 40 s), however, when looking at individual benchmarks Elucidate20 did reduce some user-times by more than an hour. In the second part of our evaluation we employed our new suggested framework. We noted that one metric within the framework, recall, is the most commonly used in our domain and showed a positive result for Elucidate20 with a 21% points increase in locating a type error compared to Gramarye19. However, the reason for the framework is to improve the ability to quantify the quality of type error debuggers and when the entire framework is applied it shows that the difference between Elucidate20 and Gramarye19 drops to just 1% point. This significant difference in results shows that just applying the traditional recall metric is not satisfactory for evaluations in this field and the application of the framework on future type error debugging solutions is needed to be able to report clearer results, and comparisons between solutions.

For future work an increase of the categories of parse errors we treat with the pre-processing along with adding other errors such as *Variables not in Scope* is a concrete direction; as the moiety algorithm already works though individual lines adding these will not increase the overheads and has the possibility of reducing the time delta debugging takes further down. It is also clear that though pre-processing speeds up delta debugging it also, on average, slows the overall run-time of the debugger. Reducing the time it takes to generate a list of moieties would be extremely beneficial. We would also want to increase our scalability benchmarks to include more than one core program as this will remove any bias away from how a programmer may specifically layout out their source code. Lastly, though we applied our method to Haskell programs, our debugger is nearly language agnostic. Delta Debugging and the Moiety algorithm are not specific for the programming language, allowing for a reasonable modification towards an agnostic debugger in the future.

**Acknowledgements.** We would like to thank all of the reviewers for their thorough feedback, which we incorporated into this paper.

## References

1. Artho, C.: Iterative delta debugging. STTT - Softw. Tools Technol. Transf. **13**(3), 223–246 (2011). https://doi.org/10.1007/s10009-010-0139-9
2. Bernstein, K.L., Stark, E.W.: Debugging type errors. Technical report, November 1995
3. Chen, S., Erwig, M.: Counter-factual typing for debugging type errors. In: Jagannathan, S., Sewell, P. (eds.) POPL 2014, pp. 583–594. ACM (2014)
4. Chen, S., Erwig, M.: Guided type debugging. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 35–51. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_3
5. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: Pierce, B.C. (ed.) ICFP 2001, pp. 193–204. ACM (2001)
6. Cleve, H., Zeller, A.: Locating causes of program failures. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) 27th International Conference on Software Engineering, pp. 342–351. ACM (2005)
7. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. Sci. Comput. Program. **50**(1–3), 189–224 (2004)
8. Kalhauge, C.G., Palsberg, J.: Binary reduction of dependency graphs. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 556–566. ACM (2019)
9. Lerner, B.S., Grossman, D., Chambers, C.: Seminal: searching for ML type-error messages. In: Proceedings of the ACM Workshop on ML, pp. 63–73 (2006)
10. McAdam, B.J.: On the unification of substitutions in type inference. In: Hammond, K., Davie, T., Clack, C. (eds.) IFL 1998. LNCS, vol. 1595, pp. 137–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48515-5_9
11. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: ICSE 2006, pp. 142–151. ACM (2006)
12. Misherghi, G.S., HDD, Z.S.: Hierarchical delta debugging. Ph.D. thesis, University of California, Davis (2007). https://pdfs.semanticscholar.org/a337/e5ba5b18cc45fd4517b90c5ac92e8052b6d3.pdf
13. Rahli, V., Wells, J.B., Pirie, J., Kamareddine, F.: Skalpel: a type error slicer for standard ML. Electron. Notes Theor. Comput. Sci. **312**, 197–213 (2015)
14. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI 2012, pp. 335–346. ACM (2012)
15. Schilling, T.: Constraint-free type error slicing. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32037-8_1
16. Seidel, E.L., Jhala, R., Weimer, W.: Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In: ICFP 2016, pp. 228–242. ACM (2016)
17. Seidel, E.L., Sibghat, H., Chaudhuri, K., Weimer, W., Jhala, R.: Learning to blame: Localizing novice type errors with data-driven diagnosis. CoRR abs/1708.07583 (2017). http://arxiv.org/abs/1708.07583
18. Sharrad, J., Chitil, O., Wang, M.: Delta debugging type errors with a blackbox compiler. In: IFL 2018, pp. 13–24. ACM (2018)

19. Shung, K.P.: Accuracy, precision, recall or f1? November 2019. https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9
20. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 72–83. ACM (2003)
21. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. In: Nilsson, H. (ed.) Workshop on Haskell 2004, pp. 80–91. ACM (2004). https://doi.org/10.1145/1017472.1017486
22. Tip, F., Dinesh, T.B.: A slicing-based approach for locating type errors. ACM Trans. Softw. Eng. Methodol. **10**(1), 5–55 (2001)
23. Tsushima, K., Asai, K.: An embedded type debugger. In: Hinze, R. (ed.) IFL 2012. LNCS, vol. 8241, pp. 190–206. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41582-1_12
24. Tsushima, K., Chitil, O.: Enumerating counter-factual type error messages with an existing type checker. In: PPL 2014 (2014)
25. Wand, M.: Finding the source of type errors. In: POPL 1986, pp. 38–43. ACM (1986)
26. Witten, I., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, Burlington (2005)
27. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/SIGSOFT FSE -1999. LNCS, vol. 1687, pp. 253–267. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48166-4_16
28. Zeller, A.: Why Programs Fail Guide to Systematic Debugging, 2nd edn. Academic Press, Cambridge (2009)
29. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)
30. Zhang, D., Myers, A.C., Vytiniotis, D., Peyton Jones, S.L.: Diagnosing type errors with class. In: Grove, D., Blackburn, S. (eds.) PLDI 2015, pp. 12–21. ACM (2015)