Delta Debugging Type Errors with a Blackbox Compiler

Joanna Sharrad
University of Kent
Canterbury, UK
jks31@kent.ac.uk

Olaf Chitil
University of Kent
Canterbury, UK
oc@kent.ac.uk

Meng Wang
University of Bristol
Bristol, UK
meng.wang@bristol.ac.uk

ABSTRACT

Debugging type errors is a necessary process that programmers, both novices and experts alike, face when using statically typed functional programming languages. All compilers often report the location of a type error inaccurately. This problem has been a subject of research for over thirty years. We present a new method for locating type errors: We apply the Isolating Delta Debugging algorithm coupled with a blackbox compiler. We evaluate our implementation for Haskell by comparing it with the output of the Glasgow Haskell Compiler; overall we obtain positive results in favour of our method of type error debugging.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Theory of computation → Program analysis**;

KEYWORDS

Type Error, Error diagnosis, Blackbox, Delta Debugging, Haskell

ACM Reference Format:
Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3310232.3310243

1 INTRODUCTION

Compilers for Haskell, OCaml and many other statically typed functional programming languages produce type error messages that can be lengthy, confusing and misleading, causing the programmer hours of frustration during debugging. One role of such a type error message is to tell the programmer the location of a type error within the ill-typed program. Although there has been over thirty years of research [8, 22] on how to improve the way we locate type conflicts and present them to the programmer, type error messages can be misleading. We can trace the cause of inaccurate type error location to an advanced feature of functional languages: type inference. A typical Haskell or OCaml program contains only little type information: definitions of data types, some type signatures for top-level functions and possibly a few more type annotations.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$$15.00
https://doi.org/10.1145/3310232.3310243

Type inference works by generating constraints for the type of every expression in the program and solving these constraints. An ill-typed program is just a program with type constraints that have no solution. Because the type checker cannot know which program parts and thus constraints are correct, that is, agree with the programmer's intentions, it may start solving incorrect constraints and therefore assume wrong types early on. Eventually, the type checker may note a type conflict when considering a constraint that is actually correct.

1.1 Variations of an Ill-Typed Programs

Consider the following Haskell program from Stuckey et al. [17]:

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

The program defines a function that shall insert an element into an ordered list, but the program is ill-typed. Stuckey et al. state that thefi rst line is incorrect and should instead look like below:

```
1 insert x [] = [x]
```

The Glasgow Haskell Compiler (GHC) version 8.2.2 wrongly gives the location of the type error as (part of) line two.

```
2 insert x (y:ys) | x > y    = y : insert x ys
```

Let us see how GHC comes up with this wrong location. GHC derives type constraints and immediately solves them as far as possible. It roughly traverses our example program line by line, starting with line 1. The type constraints for line 1 are solvable and yield the information that `insert` is of type $\alpha \rightarrow [\beta] \rightarrow \alpha$. Subsequently in line 2 the expression `x > y` yields the type constraint that `x` and `y` must have the same type, so together with the constraints for the function arguments `x` and `(y:ys)`, GHC concludes that `insert` must be of type $\alpha \rightarrow [\alpha] \rightarrow \alpha$. Finally, the occurrence of `insert x ys` as subexpression of `y : insert x ys` means that the result type of insert must be the same list type as the type of its second argument. So `insert x ys` has both type $[\alpha]$ and type α, a contradiction reported as type error.

Our program contains no type annotations or signature, meaning we have to infer all types. Surely adding a type signature will ensure that GHC returns the desired type error location? Indeed for

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

GHC identifies the type error location correctly:

```
2 insert x [] = x
```

However, a recent study showed that type signatures are often wrong, causing 30% of all type errors [23]! GHC trusts that a given type signature is correct and hence for

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

GHC wrongly locates the cause in line 2 again:

```
2 insert x (y:ys) | x > y   = y : insert x ys
```

In summary we see that the order in which type constraints are solved determines the reported type error location. There is nofi xed order to always obtain the right type error location and requiring type annotations in the program does not help.

As a consequence researchers developed type error slicing [7, 16], which determines a minimal unsatisfiable type constraint set and reports all program parts associated with these constraints as type error slice. However, practical experience showed that these type error slices are often quite big [7] and thus they do not provide the programmer with sufficient information for correcting the type error. Our aim is to determine a smaller type error location, a single line in the program. [1]

1.2 Our Method

Our method is based on the way programmers systematically debug errors without additional tools. The programmer removes part of the program or adds previously removed parts back in. They check for each such variant of the program whether the error still exists or has gone. By doing this systematically, the programmer can determine a small part of the program as the cause of the error.

This general method was termed *Delta Debugging* by Zeller [24]. Specifically, we apply the *Isolating Delta Debugging algorithm*, which determines two variants of the original program that capture a minimal difference between a correct and an erroneous variant of the program. Eventually our method produces the following result:

Result of our type error location method

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

This program listing with different highlighting shows that the type error location is in line 1 and that line 1 and 2 together cause the type error; that is, even without line 3 this program is ill-typed.

The Isolating Delta Debugging algorithm has two prerequisites: an input that can be modified repeatedly and a means of inquiring whether these modifications were successful. We fulfil thefirst prerequisite by employing the raw source code of the programmer's ill-typed program. We work directly on the program text rather than the abstract syntax tree. We make modifications that generate new variants of the program ready for testing to see whether they remain ill-typed. To examine whether they are indeed ill-typed or not, we employ the compiler as a black box. We do not use any location information included in any type error message of the compiler. This black box satisfies the second prerequisite of the Isolating Delta Debugging algorithm.

Once implemented in our tool Gramarye that works on Haskell programs and uses the Glasgow Haskell Compiler as blackbox, we can apply our method to any ill-typed program, no matter how many type errors it contains, to locate one type error. Once our approach has the correct location, the programmer canfi x it and reuse the tool tofi nd further type errors.

We evaluated Gramarye against the Glasgow Haskell Compiler using thirty programs containing single type errors and 870 programs generated to include two type errors.

Our paper makes the following contributions:

- We describe how to apply the Isolating Delta Debugging algorithm to type errors (Section 2).
- We use the compiler as a true black box; it can easily be replaced by a different compiler (Section 3.2).
- We implement the method in a tool called Gramarye that directly manipulates Haskell source code (Section 3.3).
- We evaluate our method against the Glasgow Haskell Compiler (Section 4).

Our evaluation shows an improvement in reporting type errors for many programs and demonstrates that our approach has promise in thefi eld of type error debugging.

2 AN ILLUSTRATION OF OUR METHOD

Figure 1 gives an overview of the Gramarye framework. It indicates the steps taken to locate type errors in an ill-typed program.

We start with a single ill-typed Haskell program. This program must contain a type error; otherwise we reject it. Here we work with the original ill-typed program of the Introduction.

From this program, we obtain two programs that the *Isolating Delta Debugging* algorithm will work with. One is the ill-typed program, from which the algorithm removes lines that are irrelevant for the type error. The algorithm aims to minimise this program. The other program is the empty program, which is definitely well-typed. The algorithm moves lines from the ill-typed program to the well-typed program; the algorithm aims to maximise the well-typed program. So we start with:

Step 1: well-typed program

```
1
2
3
```

Step 1: ill-typed program

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

[1] There is no fair comparison of size. Our method determines a line as error cause, no matter how long the line. All existing type error slicing algorithms produce slices built from small subexpressions; usually, however, these subexpressions are distributed over many, not necessarily adjacent, lines.

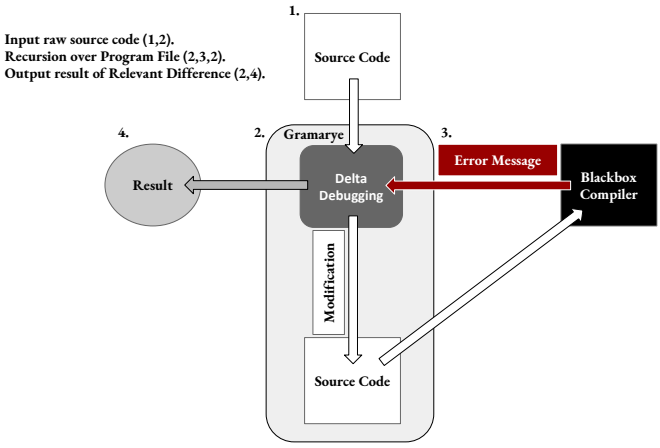

Figure 1: The Gramarye Framework

Now we move a line from the ill-typed program to the well-typed program. We pick line 3, obtaining two new program variants:

Step 1: modified well-typed program

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

Step 1: modified ill-typed program

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

We then send these two programs to the black box compiler for type checking:

- Step 1: modified well-typed program: unresolved.
- Step 1: modified ill-typed program: ill-typed.

The modified well-typed program is not a syntactically valid Haskell program; the compiler yields a parse error. So our black box compiler may yield one of three possible results:

(1) unresolved; compiler yields a non-type error
(2) ill-typed; compiler yields a type error
(3) well-typed; compilation successful

We cannot use an unresolved program for locating a type error, but each of the other two possible results are useful. Our modified ill-typed program is smaller than our original ill-typed program. We now know that the modified variant is ill-typed too, so we can replace our ill-typed program for the next step:

Step 2: well-typed program

```
1
2
3
```

Step 2: ill-typed program

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

The algorithm now repeats: Again we move a single line from the ill-typed program to the well-typed program. Let us pick line 2:

Step 2: modified well-typed program

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

Step 2: modified ill-typed program

```
1 insert x [] = x
2
3
```

Again we type check these two programs:

- Step 2: modified well-typed program: well-typed.
- Step 2: modified ill-typed program: well-typed.

Because both variants are well-typed and larger than the previous well-typed program, we can use either of them as new well-typed program. We pick the modified well-typed program and thus obtain;

Step 3: well-typed program

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

Step 3: ill-typed program

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

The well-typed and ill-typed programs differ by only a single line, and hence our algorithm terminates.

Thefi nal result is that the difference between well-typed and ill-typed program, here line 1, is the location of the type error. Because the ill-typed program contains only lines 1 and 2 of the original program, we also know that only lines 1 and 2 are needed to make the program ill-typed. Thus we obtain the output shown in the Introduction. If we want to add a compiler type error message for further explanations, we can pick the one we received for the minimised ill-typed program. The error message may be clearer than for the original, larger program.

The isolating delta debugging method is non-deterministic. Often different choices lead to the samefi nal result, but not always. Zeller argues that this non-determinism does not matter and that one result provides insightful debugging information to the programmer [25]. Hence his algorithm is deterministic, making arbitrary choices. Our implementation follows his algorithm and for our example makes the choices described here.

The algorithm is based on an ordering of programs, where a program is just a sequence of strings. A program P_1 is less or equal a program P_2 if they have the same number of lines and for every line, the line content is either the same for both programs, or the line is empty in P_1. All programs that we consider are between the well-typed and ill-typed programs that we start with. Thefinal well-typed and ill-typed programs have minimal distance, that is, they either differ by just one line or programs between them are unresolved; that is, they are not syntactically valid programs.

In this example, in each step, we moved only a single line from the ill-typed to the well-typed program. For programs with hundreds of lines, this simple approach would be expensive in time due to the number of programs needing consideration. Hence we use the full Isolating Delta Debugging algorithm which starts with moving either thefi rst or second half of the program from the ill-typed to the well-typed program. If both modified programs are unresolved, then we increase the granularity of modifications from moving half the program to moving a quarter of the program. In general, every time both modified programs are unresolved, we half the size of our modifications. This increase of granularity can continue until only a single line is modified.

Zeller analysed the complexity of the Isolating Delta Debugging algorithm. In our case complexity is the number of calls to the black box compiler in relation to the number of lines of the original program. In the worst case, if most calls yield unresolved, the number of calls is quadratic. In the best case, when no call yields unresolved, the number of calls is logarithmic [25].

3 IMPLEMENTATION

As illustrated in Figure 1, our Gramarye tool has four components;

- Delta Debugging.
- Blackbox Compiler.
- Source Code Modification.
- Result Processing.

We shall next describe each of the components in greater detail.

3.1 Delta Debugging

Zeller [6, 24–26] defined *Delta Debugging*, a debugging method that systematically applies the scientific approach of *Hypothesis-Test-Result*. When programmers debug, theyfi rst use the error message to conjecture a possible cause (hypothesis), then modify the program and recompile (test), and lastly use the outcome of the recompilation (result) to either repeat with a new, improved hypothesis, or terminate with the hypothesis having been proved.

Zeller does not only consider locating a cause in a defective program, but alternatively locating a cause in an input to a program that causes a failure at runtime or locating a cause in the runtime state of a program execution. He abstracts program/input/state by talking about configurations and differences between configurations. In our method a configuration is a program and a difference is a removal of some lines (replacement by empty lines). Our programs are input for the type checker, but the defect(s) that we look for are not in the type checker, but the program.

Essential for delta debugging is the existence of a testing function [24] that places a configuration into one of the following three categories:

(1) Unresolved (?).
(2) Fail (×).
(3) Pass (√).

For our method the testing function is the type checker of the compiler. We use domain specific terminology for the three categories:

(1) Unresolved (any non-type error).
(2) Ill-typed.
(3) Well-typed.

Zeller presents two delta debugging methods. He refers to them as Simplifying and Isolating [25].

3.1.1 Simplifying Delta Debugging. The algorithm determines a smaller variant of the given faulty (ill-typed) program. The result is minimal in that removing any further line makes the program pass (well-typed). The algorithm works by removing parts of a failing program until it no longer fails. The last failing variant of the program is the result of the algorithm. The minimality allows us to surmise that all parts of the program left must contribute to the error. The Simplifying Delta Debugging algorithm has the same disadvantage as program slicing algorithms for type errors [7, 16]: it often reports a rather large program. The second Delta Debugging algorithm, *Isolating*, aims to reduce the size of the reported program slice further.

3.1.2 Isolating Delta Debugging. Isolating Delta Debugging employs the Simplifying algorithm to generate a minimal faulty (ill-typed) program. At the same time the algorithm also produces a maximal passing (well-typed) program. The maximal program is created by taking a passing program (usually starting with the empty program) and adding lines from the faulty program until the program is faulty. The difference between the maximal passing and the minimal faulty program is then considered as cause of the fault. We chose the Isolating Delta Debugging algorithm, because this difference is substantially smaller than a minimal faulty program. Furthermore, Zeller states that in practice the Isolation algorithm is much more efficient than the Simplification algorithm [25].

3.1.3 Granularity. The Isolating Delta Debugging algorithm consists of two parts: granularity and moving of program parts. In our initial illustration of our method we moved individual lines; however, within the algorithm a granularity parameter determines how many lines of code we move between our ill-typed and well-typed programs: the number of lines is difference in the number of lines of the two programs divided by the granularity. A granularity of 2 means that the algorithm resembles a binary chop algorithm, it repeatedly divides the ill-typed program in half. The Isolating Delta Debugging algorithm starts with granularity set to 2, but depending on the results of the testing function, granularity can grow and shrink.

To understand granularity we apply the algorithm to an eight-line program that has a type error on line 8. Only when the testing function returns 'unresolved', the granularity may increase. Hence we assume that lines 1 to 3 and lines 4 and 5 of our program belong together: any program that contains only some lines of these two sets yields 'unresolved'.

At step 1 our well-typed, respectively ill-typed, program have the following lines:

```
{}  {1,2,3,4,5,6,7,8}
```

Because granularity is 2 and the two programs differ by 8 lines, we move thefi rst 8/2 = 4 lines from the ill-typed program to the well-typed one and then test both modified programs:

```
{1,2,3,4}?  {5,6,7,8}?
```

Both programs are unresolved. We cannot move any other 4 lines (we only ever move adjacent lines). Hence granularity is doubled from 2 to 4. So now we move 8/4 = 2 lines from the step 1 ill-typed program to the well-typed program. Wefi rst try

```
{1,2}?  {3,4,5,6,7,8}?
```

Both are unresolved, so we try the next two lines:

```
{3,4}?  {1,2,5,6,7,8}?
```

Again both are unresolved, so we continue with another two lines:

```
{5,6}?  {1,2,3,4,7,8}?
```

Still unresolved, so we try:

```
{7,8}×  {1,2,3,4,5,6}√
```

Finally our test function gives a different results. We reset granularity to 2. We select the well-typed program for Step 2; thus we have now the following well-typed and ill-typed program:

```
{1,2,3,4,5,6}  {1,2,3,4,5,6,7,8}
```

The two programs differ by 2 lines. We move 2/2 = 1 line from the ill-typed program to the well-typed program:

```
{1,2,3,4,5,6,7}√  {1,2,3,4,5,6,8}×
```

Thefi rst program is well-typed, the second ill-typed. Granularity stays at 2. We select the ill-typed program for Step 3:

```
{1,2,3,4,5,6}  {1,2,3,4,5,6,8}
```

Because the two programs differ only by one line, our algorithm stops. We have identified the single line difference, line 8, as the cause of the type error.

3.1.4 Choices. We noted already in Section 2 that in principle delta debugging is non-deterministic, but we follow Zeller's deterministic implementation [25]. In the preceding section on granularity we stated how the algorithm changes granularity and in which order the algorithm moves lines between the well-typed and ill-typed program. Finally Algorithm 1 shows how the test results for the modified well-typed and ill-typed program decide the choice of the next well-typed and ill-typed program.

ALGORITHM 1: Choices in Isolating Delta Debugging

```
testModProgWell = test(modProgWell)
testModProgIll = test(modProgIll)
if testModProgIll == IllTyped && granularity == 2 then
  | progIll = modProgIll
else if testModProgIll == WellTyped then
  | progWell = modProgIll
else if testModProgWell == IllTyped then
  | progIll = modProgWell
else if testModProgIll == IllTyped then
  | progIll = modProgIll
else if testModProgWell == WellTyped then
  | progWell = modProgWell
- - else: both modified programs are unresolved
try another program modification or terminate
```

3.2 A Blackbox Compiler

We use a compiler as a blackbox, an entity of which we only know the input and the output. Anything that happens within the blackbox remains a mystery to us. Compilers naturally lend themselves to this usage, taking an input (source code), and returning an output: a successfully compiled program or an error. The compiler we chose to use as a blackbox is the Glasgow Haskell Compiler (GHC), which is widely used by the Haskell community. As we can exploit GHC to gather type checking information without the need to alter the compiler itself, we can keep our tool separate. Not modifying the compiler has many benefits; changes made by the compiler developers will not affect the way our method works, users of our tool can avoid downloading a specialist compiler, or having the hassle of patching an existing one. Avoiding modification of the compiler also means our method is not restricted to the Haskell language, giving scope to expand to other functional languages.

We use our blackbox compiler as a type checker. In each iteration of the Isolating Delta Debugging algorithm, we determine the status of our modified ill-typed and well-typed programs as described in Section 2. When using the blackbox compiler, our tool receives the same output a programmer would when they are using GHC. Though the result of compiling with GHC gives a message that includes many details, we are only interested in whether our programs are well-typed, using this information to categorise as

we discuss in Section 3.1. Depending on the categories returned, we modify the source code of our programs in different ways, and again send them to the blackbox compiler for further type checking.

3.3 Source Code Manipulation

When programmers manually debug, they edit their source code directly, looking at where the error is suggested to occur and making changes in the surrounding area. We are also directly manipulating the source code, modifying our programs using the line numbers determined by the Isolating Delta Debugging algorithm. One significant bonus to the strategy of directly changing the source code is that it keeps our approach very simple. As we do not work on the Abstract Syntax Tree (AST) we do not need to parse our source code with each modification, allowing us to avoid making changes to an existing compiler or creating our own parser. Not editing the AST also means we can stay true to the programmer's original program, keeping personal preferences in layout intact by using empty lines as placeholders. In future work this will allow us to provide error messages that refer to the original program.

3.4 Processing the Results

The idea is that if one program is well-typed and the other ill-typed, then the source of the type error lies within the difference of the two; the relevant difference [25]. After the Isolating Delta Debugging algorithm has completed, two programs are left. If a line number does not make an appearance in both of these programs, then we report it as a relevant difference. Reporting whole lines also means that we can easily evaluate how successful we are in locating type errors.

4 EVALUATION

In Section 2 we have shown how we can successfully locate the correct line number of a type error. However, though positive for the example program we have used throughout, a more thorough evaluation was needed to determine the strength of our method in type error locating.

We chose to evaluate our method, implemented in our debugger Gramayre, against a benchmark of programs specially engineered to contain type errors. The programs collated by Chen and Erwig [3] were used to evaluate their Counter-Factual approach to type error debugging. In all, there are 121 programs in the CE benchmark, but not all had what Chen and Erwig called the 'oracle', the knowledge of where the type error lay. Though a program could have many correct solutions to remove a type error, we needed to know the correct location of where the type error occurred to evaluate accurately; so we removed all programs that did not specify the exact cause. To make our evaluation more compact, programs that were ill-typed in similar ways were also removed, reducing our set of test programs to thirty. However, we also wanted to see whether our method could report multiple type errors. To do so we took each pair of different programs from our 30 test programs, and joined the two programs together. Thus we generated a further 870 programs for the evaluation.

Our evaluation answers the following questions;

(1) We apply our method to Haskell programs that each contain a single type error. Do we see improvement in locating the errors compared to the Glasgow Haskell Compiler? (Section 4.1)
(2) We apply our method to Haskell programs that each contain two type errors. Do we see improvement in locating these errors compared to the Glasgow Haskell Compiler? (Section 4.2)
(3) Does our method return a smaller set of type error locations compared to the Glasgow Haskell Compiler? (Section 4.3)

We chose to compare our approach against GHC 8.2.2. We are using GHC as a blackbox compiler within our own tool, but as we use it solely as a type checker we do not use the line numbers that it reports, and thus these line numbers have no interference with our evaluation. GHC and our tool take the CE benchmarks, and type check each one; this results in a set of line numbers suggested as cause of the type error. To judge the success of locating the type error in the tests we have chosen to use the same criterion as Wand [22]. Wand states that even if we get multiple locations returned, the method is classed as a success if the exact location of the type error is within these. As both our tool and GHC can report multiple line numbers for one type error; we use Wand's criterion to allow us to take into consideration all line numbers returned, and not just thefirst.

4.1 Singular Type Error Evaluation

(1) We apply our method to Haskell programs that each contain a single type error. Do we see improvement in locating the errors compared to the Glasgow Haskell Compiler?

Each program of ourfi rst set contains one single type error; if the line number reported matches the 'oracle' response, then our result is accurate. The graph in Figure 2 shows for all 30 ill-typed programs whether Gramarye and GHC correctly discover the position of the type error. The results are positive. Out of the 30 ill-typed programs we accurately locate 23 (77%) of the type errors, compared to 15(50%) for GHC.

In some cases, multiple line numbers were returned. The primary cause of multiple line numbers are large expressions, especially those consisting of several key words, such as If-Then-Else or Let-In expressions. For example, in

Listing 1: Expression over two lines

```
1 doRow (y:ys) r = (if y < r && y > (r-dy) then '*'
2                           else ' ') : doRow r ys
```

our tool identifies both lines as the cause of the type error. In this example GHC wrongly suggests thefi rst line as causing the error. These large expressions are one area for future work.

The evaluation also drew attention to two individual programs that needed more investigating. Initially our debugger failed to discover a type error in program 24 yet GHC did successfully. On inspection program 24 contained a mistake in the original benchmark programs. The program contained two type errors, and not the singular error we were expecting. Removal of one of the type errors returned the results we would expect, with both our debugger and GHC successfully locating the errors. Due to this anomalies we decided to check all programs for multiple type errors. Program 9 also contained two errors. Oncefi xed our debugger was no

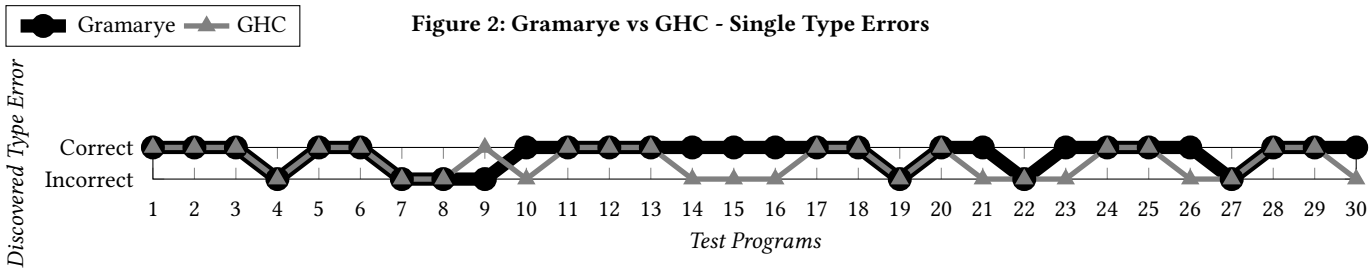

Figure 2: Gramarye vs GHC - Single Type Errors

longer successful infi nding the type error. This is due to Program 9 containing an unnecessary call to a variable bound to foldl.

Listing 2: Program 9

```
1 foldleft = foldl
2 intList = [12, 3]
3 zero = 0.0
4 addReciprocals total i = total + (1.0 / i)
5 totalOfReciprocals = foldleft zero addReciprocals intList
```

Our debugger returns line 1 as faulty, yet the type error is on line 5 where the variables 'zero' and 'addReciprocals' should be swapped. If we instead use foldl directly rather than via a variable, our debugger thenfi nds the correct broken line number.

In singular discovery our method has a 27 percentage point success rate over GHC when locating type errors in Haskell source.

4.2 Multiple Type Errors Evaluation

(2) We apply our method to Haskell programs that each contain two type errors. Do we see improvement in locating these errors compared to the Glasgow Haskell Compiler?

To evaluate the locating of multiple errors we merged our singular programs. This gave us programs that each had two self-contained type errors. Self-contained can be described as having two separate functions that do not interact with each other, however both functions containing a single type error. In Listing 3, thefi rst function has an error on line 2 and the second function on line 6, but neither type error affects the other;

Listing 3: Multiple Type Error Example

```
1 addList ls s = if s `elem` ls then ls else s : ls
2 v5 = addList"a "["b"]
3
4 sumLists = sum2 . map sum2
5 sum2 [] = []
6 sum2 (x:xs) = x + sum2 xs
```

Listing 3 is just one of the programs we generated that contains multiple type errors. We created these by merging the CE benchmark programs. Each set of programs includes the earlier source code with the addition of another CE program attached to the bottom. In all, we generated 870 new ill-typed programs to test.

The success criteria for reporting an accurate discovery of the position of a type error in an ill-typed program that contains multiple errors is similar to what we used for singular errors. The only difference being, that though we have two errors per program we only need one error to be reported to for a success.

Table 1, shows one set of results from a mergedfi le. Thefirst column lists the program number that we are using as the base and the second column indexes the number of the program we merged to the end of the source code. Under the Gramayre and GHC columns, we use ticks and crosses to denote if either correctly reports a type errors location, under this, we total the number of correct matches as a percentage.

With this particular combination of CE benchmark programs, we can see that Gramaryefi nds 42 percentage points more than GHC when locating type error positions. However, this is not always the case. Table 2, provides the total results for all of our programs as an average. The average was generated by combining the results of each of the groups of programs. Column one lists the base program, and the last two columns show the percentage of how accurate our tool and GHC were at locating type errors.

In total, we can see that Gramaryefi nds 7 percentage points fewer type errors in our multi-error programs than GHC. The Isolating Delta Debugging algorithm restricts Gramarye to always locate just one type error, thefi rst it comes across. Once it has found this error, the algorithm assumes the job is complete and does not check any further. However, GHC reports many type error messages giving GHC an advantage over our debugger. The more error messages reported the more chance GHC has to report a correct line. We can this effect in the results. Currently, we would expect the programmer tofi x one type error at a time, and so will repeatedly use the tool after each implementedfi x. Reporting many errors message is only an advantage in this evaluation, not when debugging type errors as a whole.

4.3 Precise Type Error Evaluation

(3) Does our method return a smaller set of type error locations compared to the Glasgow Haskell Compiler?

Though our criteria for success allowed us to check multiple returned line numbers for the correct type error position, reporting many lines to the programmer is not ideal. As we aimed to return just a singular line number as the cause of the type error, an additional evaluation criteria allowed us to pinpoint how specific our tool is compared to GHC. All of the programs we tested had a single type error on a distinct line; our new rule specified that if either

Table 1: Testing a program with two type errors.

Program	Merged	Gramarye	GHC
15	1	✓	✓
15	2	✓	✓
15	3	✓	✓
15	4	✓	×
15	5	✓	✓
15	6	✓	✓
15	7	✓	×
15	8	×	×
15	9	✓	×
15	10	✓	×
15	11	✓	✓
15	12	✓	✓
15	13	✓	✓
15	14	✓	×
15	16	✓	×
15	17	✓	✓
15	18	✓	✓
15	19	✓	×
15	20	✓	✓
15	21	✓	×
15	22	×	×
15	23	✓	×
15	24	✓	✓
15	25	✓	×
15	26	✓	×
15	27	×	×
15	28	✓	✓
15	29	✓	✓
15	30	✓	×
Total		90%	48%

Gramarye or GHC returned a single accurate location, then they were classed as having a "precise success".

Table 3 shows all the programs that had a single type error; a tick denotes if either Gramarye or GHC accurately report a single line number as being the cause of the type error. A report of multiple lines means a cross is displayed, even if a report of a correctly located type error was within them.

Our method had a positive outcome when locating a single line as the cause of the fault. Gramarye reported accurately 16 times (53%), and, GHC does slightly worse at 12 times(40%).

When evaluating programs that included multiple self-contained type errors, we had a slightly different criteria, judging "precise success" under the following rules;

- A single line number containing the location of error one.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList"a "["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

Table 2: Overall testing of programs with two type errors.

Program	Gramarye	GHC
1	69%	100%
2	62%	100%
3	72%	97%
4	72%	52%
5	66%	100%
6	72%	100%
7	62%	48%
8	66%	52%
9	72%	55%
10	62%	52%
11	62%	100%
12	66%	100%
13	62%	100%
14	76%	52%
15	90%	48%
16	62%	52%
17	69%	100%
18	69%	100%
19	79%	21%
20	66%	100%
21	79%	45%
22	38%	45%
23	66%	52%
24	59%	100%
25	62%	100%
26	69%	55%
27	62%	52%
28	69%	100%
29	62%	100%
30	62%	52%
Average	67%	74%

- A single line number containing the location of error two.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList"a "["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

- Two line numbers containing the location of both error one and two.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList"a "["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

All other results, even those that include the correct location, are recorded as failing the "precise success" criterion of discovering type errors. Table 4 presents the test programs that contained two type

Table 3: "precise success" on single type errors.

Program	Gramarye	GHC
1	✓	✓
2	×	✓
3	×	×
4	×	×
5	×	✓
6	×	✓
7	×	×
8	×	×
9	✓	×
10	✓	×
11	×	✓
12	✓	✓
13	✓	✓
14	×	×
15	×	×
16	✓	×
17	✓	×
18	✓	✓
19	×	×
20	✓	✓
21	✓	×
22	×	×
23	✓	×
24	✓	×
25	✓	✓
26	✓	×
27	×	×
28	✓	✓
29	×	✓
30	✓	×
Total	53%	40%

Table 4: "precise success" on programs with two type errors.

Program	Gramarye	GHC
1	48%	38%
2	45%	38%
3	52%	7%
4	48%	0%
5	41%	41%
6	34%	38%
7	41%	0%
8	41%	0%
9	48%	0%
10	48%	0%
11	45%	41%
12	45%	48%
13	41%	38%
14	31%	0%
15	14%	0%
16	41%	0%
17	48%	0%
18	45%	38%
19	10%	0%
20	41%	34%
21	69%	0%
22	21%	0%
23	38%	0%
24	41%	0%
25	45%	34%
26	45%	0%
27	41%	3%
28	45%	41%
29	34%	34%
30	45%	0%
Average	41%	16%

errors. The name of the original program along with the percentage of type error locations deemed to be a "precise success" are shown.

Analysing Table 4 we can see that our method is again successful in reporting the correct type error location using just one line number with 41% accuracy compared to GHC with 16%. GHC tends to report as many line numbers it feels are associated with the type error, very much like slicing. However, our method works on returning the smallest number of lines meaning we achieve a higher rate of receiving only one location at a time.

4.4 Efficiency Evaluation

Our debugger can successfully locate type errors in Haskell source code, however it also needs to be efficient. Efficiency is an important aspect for any programmer wanting to use our tool, so we evaluate it against the following questions:

(1) How many calls to the compiler are needed?
(2) How long does Gramarye take tofi nd the type error?

The evaluation ran on a computer containing an AMD Phenom X4 965, 32GB RAM and a Samsung 850 Solid State Drive, whilst running Ubuntu Linux 16.04 LTS. Table 5 shows the program we are evaluating, how many lines of code each contains, the "clock-time" that the programmer will experience when using the debugger, the number of calls our debugger makes to GHC, and the "clock-time" time for GHC to return the result. On average our debugger took 3.359 seconds to provide a location, which meant calling GHC as a blackbox 11 times.

For programs with single type errors we can see that the majority of the "clock-time" is caused by calling GHC to type check. Reducing the number of calls to the blackbox compiler would increase the efficiency of the debugger. As we are working on a line by line basis we known that there is a risk of producing many unresolved programs that all need to be type-checked. In Table 5 we can also see which categories the programs we type-checked were placed into. In total the programs were categorised as 97 well-typed, 85 ill-typed, and 81 containing unresolved errors, such as a parse failure. The 81 calls to the type checker that were the result of unresolved programs are clearly a source of efficiency drain on the debugger, and a starting point of investigation for future work in this area.

In Table 6 we have condensed the results into averages for each group of programs, denoting the groups by the program number

Table 5: efficiency on programs with one type error.

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	5	2.3	6	0.3	3	2	1
2	8	4.7	14	0.3	2	1	6
3	8	3.0	8	0.3	3	4	1
4	8	2.7	8	0.2	4	2	2
5	5	2.7	8	0.2	2	2	2
6	8	3.2	10	0.3	3	2	3
7	6	2.6	8	0.3	5	3	0
8	5	2.2	6	0.2	3	2	1
9	8	2.9	8	0.3	5	3	0
10	10	5.6	16	0.3	5	4	4
11	5	2.7	8	0.3	2	2	2
12	6	2.5	6	0.2	2	3	1
13	6	2.2	6	0.3	3	3	0
14	8	4.3	18	0.2	2	4	8
15	10	4.3	20	0.2	2	4	8
16	8	4.3	14	0.2	4	3	4
17	5	2.4	6	0.3	4	2	0
18	7	2.1	6	0.2	3	2	1
19	20	9.4	44	0.3	3	6	11
20	8	2.6	8	0.3	4	2	2
21	7	2.8	8	0.3	4	3	1
22	12	2.8	8	0.3	3	3	2
23	8	4.0	14	0.2	4	3	4
24	6	2.7	8	0.2	3	4	1
25	7	4.1	14	0.3	3	2	5
26	7	6.8	24	0.3	4	4	9
27	5	2.2	6	0.3	4	2	0
28	6	2.0	6	0.3	3	3	0
29	5	2.5	8	0.2	2	2	2
30	5	2.2	6	0.3	3	3	0
Average	7	3.4	11	0.3	97	85	81

that was used to generate the programs. When evaluating the programs with multiple type errors we see a similar outcome to the single type errors, with GHC calls tightly associated with the debugger "clock-time". The worst result we received debugging a program with multiple errors took 28.672 seconds and called GHC 110 times. However on average the debugger took 4 seconds and 14 calls to return a type error location.

4.5 Summary

Overall, our evaluation has proven positive towards our method of type error debugging. From the testing, our strength lies in the reporting of singular type errors, be that one per program or the reporting of one instance of type error amongst many. Our results compared to GHC when testing more than one type error in a program prove to be less positive. This is due to the debugger pinpointing a singular type error each time it is run. However, we believe giving an accurate location over a broad suggestion is preferential.

5 RELATED WORK

Type error debugging has taken many forms over the past thirty years; we will not be able to cover all of them. Some core categories within type error debugging include: Slicing [7, 14, 18], Interaction [4, 5, 15, 16, 21], Type Inference Modification [1, 11], and working with Constraints [13, 27]. These solutions are complicated to implement, relying on programmers to patch their compiler or use a bespoke one. However, others do not provide an implementation to use at all, and in the cases where there is an implementation, they are no longer maintained to work with the latest compiler [9].

Delta Debugging is one solution that allows for separation from the compiler, and is the name for two algorithms, one that simplifies and another that isolates[6, 24–26]. There is only one demonstration of the application of the Simplifying Delta Debugging algorithm to types errors, namely an implementation in the Liquid Haskell type checker [19].

Prior works that mention using the idea of a black box include: using the compiler's type inferencer as a black box to construct a type tree to use to debug the program [20], and having an SMT solver as a blackbox to return the satisfiable set of constraints to

Table 6: efficiency on programs with two type errors.

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	11	3.4	11	0.3	3	4	2
2	14	5.3	18	0.3	4	5	6
3	14	3.7	12	0.4	3	5	3
4	14	4.0	14	0.3	3	4	4
5	11	3.5	12	0.3	3	4	3
6	14	4.1	15	0.3	3	4	4
7	12	3.3	11	0.3	3	4	2
8	11	3.1	11	0.3	3	4	2
9	14	4.0	14	0.3	3	4	4
10	16	4.4	16	0.3	3	4	5
11	11	3.2	11	0.3	3	4	2
12	11	3.4	11	0.3	3	4	2
13	12	3.3	11	0.3	3	4	2
14	12	3.9	13	0.5	2	4	4
15	15	4.9	17	0.5	26	4	6
16	14	5.2	19	0.3	4	4	6
17	11	3.4	11	0.3	3	4	2
18	13	3.1	11	0.3	3	4	2
19	26	9.4	34	0.4	3	6	14
20	14	3.6	12	0.3	3	4	3
21	13	3.4	11	0.3	4	4	2
22	18	4.3	15	0.3	4	5	4
23	14	5.2	18	0.3	4	4	6
24	12	3.1	10	0.3	3	4	2
25	13	3.3	11	0.3	3	4	3
26	13	3.2	11	0.3	3	4	3
27	11	3.3	11	0.3	3	4	2
28	12	3.3	11	0.3	3	4	2
29	11	3.7	12	0.3	3	4	3
30	11	3.4	11	0.3	3	4	2
Average	14	3.4	13	0.4	3	4	4

show type errored expressions[12], and SEMINAL [9, 10]. SEMINAL, along with previous solutions of using a blackbox compiler, actually makes modifications to an existing compiler. This requires the programmer to apply a patch; however, that patch is no longer maintained for the latest Ocaml compiler. SEMINAL works by modifying the Abstract Syntax Tree (AST), adding and removing expressions. It returns a location of the type error along with a suggested fix.

Inspired by SEMINAL is an approach that talks about altering source code with a constraint-free tool, however though the author refers to source code modification, the implementation works directly on the AST [14]. Another tool TypeHope also discusses changing the source code of a program to stay true to how a programmer debugs. However, again, the implementation edits the AST [2]. At this point, as far as the authors know, modifying source code directly is a new approach in the type error debuggingfield.

6 CONCLUSION AND FUTURE WORK

Our method combines the Isolating Delta Debugging algorithm, a black box compiler and direct source code modification to locate type errors. Our tool Gramarye implements the method for Haskell using the Glasgow Haskell compiler as a black box. From our evaluation we have gathered positive results that support our method for type error debugging. For single type errors our tool gives a 27 percentage points improvement over GHC. However, for two separate type errors in a single program GHC was 9 percentage points more successful. When returning only a precise line number for type errors, our method proved positive with 53% for locating singular type errors, and 41% when applied to a program that contained two type errors. A significant practical advantage of our method is that our tool Gramarye has only a small GHC-specific component and thus can easily be modified for other programming languages and compilers.

In the future we will be looking at where Gramarye did well and what its points of failure were. We will then use the outcome of the investigation to improve our algorithm for type error debugging. We will study closer the non-determinism of our method: can we sometimes determine whether one choice is better than another? After we have improved our method to determine the correct line number, we intend to increase the granularity of the tool further to eventually modify programs by single characters instead of lines, thus identifying subexpressions that cause type errors. On the

theoretical side, there is clearly a close link between our method and methods described in the literature that perform type error slicing based on minimal unsolvable constraint sets. We want to formalise that link.

Additional improvements to the tool outside of the algorithm would also be useful. An improved GUI, though not necessary for seeing if our approach is beneficial, does open up the options of not only combining with other methodologies that rely on interaction but also testing with real-life participants. We also would like to conduct empirical research of our solution in combination with evaluating against collected student programs to affirm our strategy.

REFERENCES

[1] Karen L Bernstein and Eugene W Stark. 1995. *Debugging type errors (full version).* Technical Report. State University of New York at Stony Brook, Stony Brook, NY 11794-4400 USA. https://pdfs.semanticscholar.org/814c/164c88ba7dd22e7e501cdd1a951586a3117b.pdf

[2] Bernd Braßel. 2004. Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*. https://www.informatik.uni-kiel.de/~mh/wlp2004/final_papers/paper13.ps

[3] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 583–594. https://doi.org/10.1145/2535838.2535863

[4] Sheng Chen and Martin Erwig. 2014. Guided Type Debugging. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

[5] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*. 193–204. https://doi.org/10.1145/507635.507659

[6] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 342–351. https://doi.org/10.1145/1062455.1062522

[7] Christian Haack and Joe B. Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.* 50, 1-3 (2004), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

[8] Gregory F. Johnson and Janet A. Walz. 1986. A Maximum-Flow Approach to Anomaly Isolation in Unification-Based Incremental Type Inference. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 44–57. https://doi.org/10.1145/512644.512649

[9] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 425–434. https://doi.org/10.1145/1250734.1250783

[10] Benjamin S. Lerner, Dan Grossman, and Craig Chambers. 2006. Seminal: searching for ML type-error messages. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. 63–73. https://doi.org/10.1145/1159876.1159887

[11] Bruce J McAdam. 1999. On the unification of substitutions in type inference. *Lecture notes in computer science* 1595 (1999), 137–152. https://link.springer.com/chapter/10.1007/3-540-48515-5_9

[12] Zvonimir Pavlinovic. 2014. General Type Error Diagnostics Using MaxSMT. (2014). https://pdfs.semanticscholar.org/1c14/7bc9f51cc950596dbc3e7cc5121202d160da.pdf

[13] Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electr. Notes Theor. Comput. Sci.* 312 (2015), 197–213. https://doi.org/10.1016/j.entcs.2015.04.012

[14] Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*. 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[15] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 228–242. https://doi.org/10.1145/2951913.2951915

[16] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. 72–83. https://doi.org/10.1145/871895.871903

[17] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. 80–91. https://doi.org/10.1145/1017472.1017486

[18] Frank Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (2001), 5–55. https://doi.org/10.1145/366378.366379

[19] A Tondwalkar. 2016. *Finding and Fixing Bugs in Liquid Haskell.* Master's thesis. University of Virginia. https://pdfs.semanticscholar.org/79b4/22959847253c40aff25c228205372d9ebc60.pdf

[20] Kanae Tsushima and Kenichi Asai. 2012. An Embedded Type Debugger. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*. 190–206. https://doi.org/10.1007/978-3-642-41582-1_12

[21] Kanae Tsushima and Olaf Chitil. 2014. Enumerating Counter-Factual Type Error Messages with an Existing Type Checker. In *16th Workshop on Programming and Programming Languages, PPL2014*. http://kar.kent.ac.uk/49007/

[22] Mitchell Wand. 1986. Finding the Source of Type Errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 38–43. https://doi.org/10.1145/512644.512648

[23] Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did?. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[24] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*. 253–267. https://doi.org/10.1007/3-540-48166-4_16

[25] Andreas Zeller. 2009. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition.* Academic Press. http://store.elsevier.com/product.jsp?isbn=9780123745156&pagename=search

[26] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[27] Danfeng Zhang, Andrew C Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. *Diagnosing Haskell type errors.* Technical Report. Technical Report http://hdl.handle. net/1813/39907, Cornell University. https://pdfs.semanticscholar.org/d32f/81a5c1706e225e2255b72c1e4b41f799e8f1.pdf

Received May 2018