Refining the Delta Debugging of Type Errors

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

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

ABSTRACT

Understanding the cause of a type error can be challenging; for over 30 years researchers have proposed many sophisticated solutions that hardly made it into practice. Previously we presented a simple method for locating the cause of a type error in a functional program. Our method applies Zeller's isolating delta debugging algorithm, using the compiler as a black box: Simple line-based program slicing searches for a type error location. To improve speed, we incorporated a pre-processing stage for handling parse errors. In this paper we note that the method needs refining. We introduce a new algorithm that replaces the previous pre-processing of parse errors with on-request handling. We implemented the algorithm and evaluated it for Haskell and OCaml programs to demonstrate that it is language agnostic.

CCS CONCEPTS

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

KEYWORDS

Type Error, Error diagnosis, Blackbox, Delta Debugging

ACM Reference Format:
Joanna Sharrad and Olaf Chitil. 2021. Refining the Delta Debugging of Type Errors. In *IFL '21: 33rd Symposium on Implementation and Application of Functional Languages, September 01–03, 2021, Online.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

All programmers want fewer bugs that are easily identified and quick to fix. However, type errors in statically typed functional languages such as Haskell are notoriously awkward to locate. The compiler message provides little help when it reports the type error far from the actual ill-typed line. For example, for a program given by Chen and Erwig in their benchmark suite [2]:

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1.
```

the Glasgow Haskell Compiler (version 8.4.3) gives line 1 as the error location. However, Chen and Erwig, as the oracle, a programmer with knowledge of where the error occurs, tell us that the error is actually in line 2: `[0]` should be `0`.

For over 30 years, researchers have proposed many sophisticated solutions. Hardly any made it into practice; we believe this is because scaling these solutions to full programming languages such as Haskell and maintaining them with every change to the language is too much work. Hence we started developing a simple tool that uses the compiler as a blackbox, not duplicating any parsing or type checking.

Our first tool [19] implements Zeller's general *isolating delta debugging* algorithm [27] for type error debugging. The algorithm isolates a fault in a program by considering subsets of the program, called *configurations*. We chose a configuration to be any subset of the lines of the original ill-typed program. The algorithm computes two configurations; one is a well-typed configuration that is a subset of the other, ill-typed configuration. The difference between the two configurations is a cause of the type error. The isolating delta debugging algorithm starts with the empty, trivially well-typed program and the original ill-typed program; the algorithm then shrinks the difference between these two configurations iteratively. In every iteration the difference between the two configurations is halved; one half is added to the well-typed configuration, and the other half is removed from the ill-typed one. The black-box compiler then checks both new configurations. If one of them is well-typed, it becomes the new, bigger, well-typed configuration; if one of them is ill-typed, it becomes the new, smaller ill-typed configuration.

Besides well-typed and ill-typed, the black box compiler may also report a different error for a configuration, e.g. a parse error or an error for using an unknown identifier. In all these cases, delta debugging calls the configuration *unresolved*. If in an iteration all modified configurations are unresolved, then the algorithm tries further configurations by dividing the difference between the two original configurations by 4, 8, etc. Delta debugging terminates when it cannot reduce the difference between the two configurations any further.

The more unresolved configurations the algorithm encounters, the slower it becomes. We found that parse errors cause most unresolved configurations; hence we developed an algorithm termed *moiety* that creates only configurations that do not cause parse errors in the subsequently applied isolating delta debugging algorithm [18]. Although the algorithm combination speeds up locating a type error substantially, it is still too slow in practice. Moiety sends each line of the original ill-typed program separately to the blackbox compiler and for a typical module of 400 lines that can take around 13 minutes.

IFL '21, September 01–03, 2021, Online

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Hence in this paper we introduce a new algorithm, *good-omens*, which merges the idea of moiety into the *isolating delta debugging*. The *isolating delta debugging* algorithm uses good-omens by request. Suppose a split leads to a configuration yielding a parse error. In that case, our new algorithm uses the idea of moiety to find a better split as well as recording poor splits to avoid them in future iterations.

In this paper we make the following contributions:

- We present the good-omens algorithm for locating type errors. The algorithm merges parts of the moiety algorithm with the *isolating delta debugging* algorithm. (Section 3).
- We implement our new algorithm in an agnostic type error debugger named Eclectic and evaluate against our previous debugger Elucidate for run-time and result quality. (Section 4)
- We show that our debugger is truly language agnostic by evaluating two functional programming languages, Haskell and OCaml. (Section 4)

2 ILLUSTRATING BY EXAMPLE

Our debugger will accept only an ill-typed program as input; so let us consider a variant of our example from the Introduction. This program has a single type error on line 2:

```
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)
```

The two initial configurations for the *isolating delta debugging* algorithm are as follows. Our initial well-typed configuration is the empty program, shown on the left; the algorithm will add lines from our ill-typed program, thus maximising the well-typed configuration. Our initial ill-typed configuration is the complete original ill-typed program; the algorithm will remove lines, minimising the ill-typed configuration. Trivially the well-typed configuration is a subset of the ill-typed configuration.

initial well-typed

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

initial ill-typed

```
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)
```

Next, the *isolating delta debugging* algorithm splits the difference between the configurations in half. We remove the second half of the difference from our ill-typed configuration and add it to our well-typed configuration:

Iter. 1: modified well-typed

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

Iter. 1: modified ill-typed

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

We send both configurations to a blackbox compiler. We use only the message returned by the compiler, which tells us whether a configuration has a type error (*fails*), compiles successfully (*passes*), contains a 'Parse Error on Input' (*parseInput*) or causes any other error (*unresolved*). Our modified well-typed configuration on the left is *unresolved* and our modified ill-typed configuration on the right *fails*. Hence the modified ill-typed configuration becomes the new, smaller, ill-typed configuration, while the (empty) well-typed configuration remains unchanged.

Iter. 1 result: well-typed

```
1
2
3
4
5
```

Iter. 1 result: ill-typed

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
```

The next iteration of *isolating delta debugging* again creates two modified configurations:

Iter. 2: modified well-typed

```
1
2
3
4 plus :: Int -> Int -> Int
5 plus = (+)
```

Iter. 2: modified ill-typed

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1
4
5
```

The left program *passes* and the right one *fails*. *Isolating delta debugging* prioritises *passing*, and hence the modified well-typed configuration becomes the new, larger, well-typed configuration while the ill-typed configuration remains unchanged.

Iter. 2 result: well-typed

```
1
2
3
4 plus :: Int -> Int -> Int
5 plus = (+)
```

Iter. 2 result: ill-typed

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
```

The next iteration of *isolating delta debugging* again splits the difference between the well- and ill-typed configuration and modifies both configurations:

Iter. 3: modified well-typed

```
1
2
3     1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
```

Iter. 3: modified ill-typed

```
1 f x = case x of
2   0 -> [0]
3
4
5
```

This time we get a 'Parse Error on Input' for the left configuration. Hence the debugger calls the good-omens algorithm. The good-omens algorithm adds back lines to the configuration with the 'Parse Error on Input', in this case, our well-typed configuration, just before the line that caused the parse error. So the first iteration of good-omens adds line 2 back to the well-typed program:

Good-omens iteration 1

```
1
2     0 -> [0]
3     1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
```

We send this to our blackbox compiler, and again we receive a 'Parse Error on Input', this time for line 2. So the next iteration adds line 1 back:

Good-omens iteration 2

```
1 f x = case x of
2     0 -> [0]
3     1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
```

We receive a fail result from the blackbox compiler. The good-omens algorithm terminates and in addition to the two configurations returns the information that lines 1 to 3 form a moiety. A *moiety* is a set of consecutive lines that shall not be split by isolating delta debugging, because splitting would just cause a parse error.

Isolating delta debugging started with the assumption that lines can be split anywhere in the configuration, that is, each line is a separate moiety ($\{\{1\}, \{2\}, \ldots, \{9\}$). A call of the good-omens algorithm refines that information for future splittings by the isolating delta algorithm; here good-omens updates the moieties to $\{\{1, 2, 3\}, \{4\}, \{5\}, \ldots, \{9\}\}$.

The good-omens algorithm returns exactly the same well-typed and ill-typed configuration as the earlier iteration 2 of isolating delta debugging. Because lines 1 to 3 form a moiety, the difference between the two configurations cannot be reduced further. So isolating delta debugging terminates with this result: the type error location is within the difference of the two configurations, that is, within lines 1 to 3.

3 ECLECTIC

Eclectic is a modified version of our previous debugger, supporting the same features [19]; however, unlike its predecessor, the new debugger implements three core elements:

(1) The modified *Isolating Delta Debugging* algorithm
(2) The*Good-Omens* algorithm
(3) The *Agnostic* behaviour

Figure 1 shows how the *isolating delta debugging* and good-omens algorithms flow together. Next, a description of each aspect and how they relate in more detail.

3.1 Delta Debugging

The delta debugging algorithm has formed the backbone of our tools from the beginning due to its ability to mimic how programmers naturally debug without anything more than compiler output [18, 19]. The process goes as such: discover a bug, modify the source code, and recompile to see if we have achieved the desired outcome bug-free code. Andreas Zeller automated this technique into the *Simplifying Delta Debugging algorithm* [26, 27]. The algorithm works by modifying source code depending on the level of granularity. By default, granularity is set at 2, meaning the algorithm divides the program in half and sends each side to a test function. The results that this test function can provide are one of the following Fail (×), Pass (√), and Unresolved (?).

If we receive a *Fail*, on either half, the algorithm calls itself recursively whilst the granularity increases with an *Unresolved*. The algorithm stops when it can no longer divide the program. However, the *Simplifying Delta Debugging algorithm* can sometimes report results larger than wanted. Zellers response to this was the *Isolating Delta Debugging* algorithm which is the one we utilise within our Eclectic tool [5, 27, 28].

Isolating delta debugging builds upon the *Simplifying Delta Debugging algorithm* by considering all outcomes of the test function. Previously we were only really interested in *Fail* results to minimise the program down to the wrong location. We also want to use the *Pass* results to provide a set of maximum faultless locations. To do so, we will work on two copies of our program, known as configurations.

Zeller made his algorithm abstract, which allowed for ease of application on other domains, such as type error debugging, successful. So in our case, the pass configuration contains a well-typed version of our program, an empty program, and our fail configuration contains the ill-typed program. The fail configuration still employs the *Simplifying Delta Debugging algorithm* for us; this means minimising the source code by removing lines. In contrast, the pass configuration adds lines back to the empty program. Each configuration gets sent to the test function for the result, and our test function is a blackbox compiler. The compiler lets us know if we have received any matches to the *Fail*, *Pass*, or *Unresolved* results. For us, the new mappings of the results are Type Error *(Fail)* (×), Successful Compile *(Pass)* (√), and Any other non-type error *(Unresolved)* (?).

Just like the simplifying delta debugging algorithm, if the *Isolating delta debugging* algorithm receives a *Type Error (Fail)* or *Successful Compile (Pass)* it will continue to call itself recursively. Otherwise, an *Any other error (Unresolved)* result will again increase

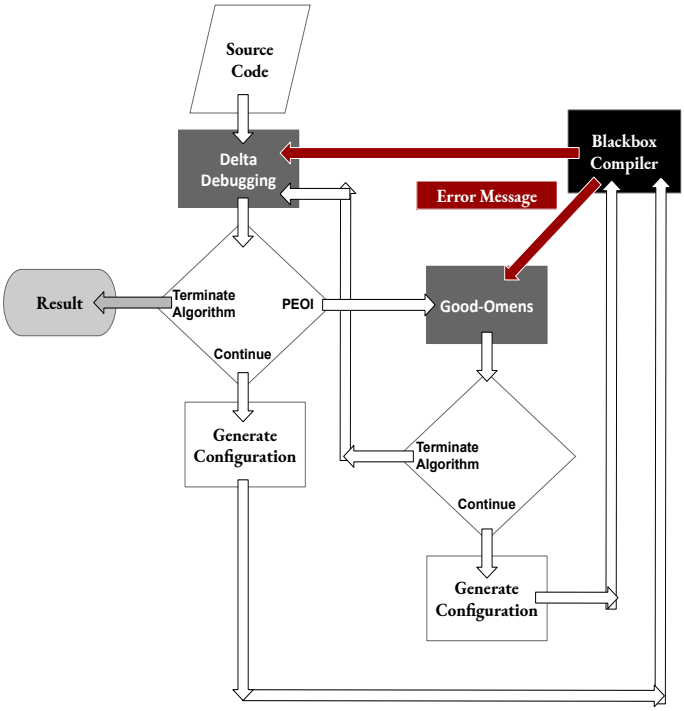

Figure 1: The flow of the Eclectic debugger

the granularity until it is no longer viable. Once the algorithm terminates, we have two configurations, one that contains only failing lines and the other only passing.

It is the intersection of these two configurations of which generates the result of which location is ill-typed. For example, below lines 1, 2, and 3 appear in the failing lines but not in the passing. The intersection of the two configurations here is lines 1 to 3.

Only Failing Lines

```
1 f x = case x of
2   0 -> [0]
3   1 -> 1
4
5
6
7
8
9
```

Only Passing Lines

```
1
2
3
4 plus :: Int -> Int -> Int
5 plus = (+)
6
7
8
9
```

3.2 Modified Isolating Delta Debugging

Delta Debugging is easily applied to the domain of type error debugging. However, to integrate the good-omens algorithm, some modifications had to be applied. Algorithm 1 shows an outline of the new modified *Isolating delta debugging* algorithm, again using pseudocode and concentrating on a subset of the algorithm that applies the choices. The changes have been highlighted.

ALGORITHM 1: Changes to Delta Debugging for Good-Omens

```
while j < n: do
  i = (j + offset) % n
  next_c_pass = listunion(c_pass, deltas[i])
  next_c_fail = listminus(c_fail, deltas[i])

  if test(next_c_fail) == ParseInput or test(next_c_pass) == ParseInput:
    good-omens(next_c_fail, next_c_pass, moieties)
  else if test(next_c_fail) == FAIL and n == 2:
    c_fail = next_c_fail
    n = 2; offset = 0; break
  else if test(next_c_fail) == PASS:
    c_pass = next_c_fail
    n = 2; offset = 0; break
  else if  test(next_c_pass) == FAIL:
    c_fail = next_c_pass
    n = 2; offset = 0; break
  else if test(next_c_fail) == FAIL:
    c_fail = next_c_fail
    n = max(n - 1, 2); offset = i; break
  else if test(next_c_pass) == PASS:
    c_pass = next_c_pass
    n = max(n - 1, 2); offset = i; break
  else
    j = j + 1 # Try next subset
  end if
end while
```

Here we have added an additional clause that detects for 'parse error on input' results. This means that our mapped results from the blackbox compiler now look like the following Type Error *(Fail)*, Successful Compile *(Pass)*, Any other non-type error *(Unresolved)*, and Parse error on input *(ParseInput)*.

These new results are applied as follows; first, we send our ill-typed program to the *Isolating delta debugging* algorithm. At this stage, it works as previously described, creating two configurations, one that is empty and one that is the entire ill-typed program. It recursively modifies each configuration with either more or fewer lines, requesting the results of the changes from the blackbox compiler until we receive a 'ParseInput' result. We then call the good-omens algorithm with the current configurations. Once the good-omens algorithm terminates, it returns a set of valid and invalid splitting points, or moieties, for the ill-typed programs source code and the *Isolating delta debugging* algorithm starts again from its last position. However, this time when dividing the configurations, it uses the valid splitting points for guidance. The *Isolating delta debugging* algorithm then recursively calls itself again until it either terminates or receives another 'ParseInput' result.

3.3 Good-Omens Algorithm

In a previous paper, we introduced the algorithm Moiety [18]. The algorithm works by pre-processing each line of an ill-typed program before reaching the *Isolating delta debugging* algorithm. The design of this pre-processing is to eliminate all 'Parse Errors on Input' by generating moieties, also described as sets of line numbers that are valid splitting points in the program. For the debugger, Eclectic, we modified the moiety algorithm to become the on-request good-omen algorithm seen in algorithm 2. The debugger no longer needed to check each line for a 'Parse Error on Input'; however still needs to generate sets of moieties, this time representing both valid and invalid splitting points.

ALGORITHM 2: The Good-Omens Algorithm

```
while l ! = n: do
  l = l - 1 # Add prior line to current line
  if test(next_c_fail) == FAIL:
    join current line to prior line in moieties list
    deltaDebugging(moietyList)
  else if test(next_c_fail) == PASS:
    join current line to prior line in moieties list
    deltaDebugging(moietyList)
  else if  test(next_c_pass) == UNRESOLVED:
    join current line to prior line in moieties list
    deltaDebugging(moietyList)
  else
    l = l - 1 # ParseInput result - Try next line
  end if
end while
```

Section 2 ran though a full example of the Eclectic tool. Here an example will just show how the good-omens algorithm works. Recollect that the example program caused *Isolating delta debugging* to call the good-omens algorithm at this point. The ill-typed configuration is on the left and the well-typed on the right:

Step 1: ill-typed

```
1 f x = case x of
2   0 -> [0]
3
4
5
6
7
8
9
```

Step 1: well-typed

```
1
2
3   1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
6
7
8
9
```

The cause of the 'Parse Error on Input' on the right configuration is currently the invalid split between lines 3 and 2. The *isolating delta debugging* provides the good-omens algorithm with both of the above configurations. It also provides the current set of moieties. In the case of the example, this is:

{{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}},

there are no invalid splitting points.

Just as *isolating delta debugging* does, good-omens works on both configurations, however only the one that contains the 'Parse Error on Input' is sent to the blackbox compiler to check for a new result. Each time the debugger calls the good-omens algorithm, it works on the current configuration using the 'ParseInput' line number for guidance. The good-omens algorithm generates a new configuration by moving the next line before line 3 from left-hand configuration to the right:

Step 2: ill-typed

```
1 f x = case x of
2
3
4
5
6
7
8
9
```

Step 2: well-typed

```
1
2   0 -> [0]
3   1 -> 1
4 plus :: Int -> Int -> Int
5 plus = (+)
6
7
8
9
```

The good-omens algorithm completes this process, and the algorithm calls the blackbox compiler with the new configuration. The results show the configuration still has a 'Parse Error on Input'. This time the issue is between lines 2 and 1. Again the good-omens algorithm produces a new configuration:

Step 3: ill-typed

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

Step 3: well-typed

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

Good-omens again calls the blackbox compiler and receives a Fail result. The algorithm has removed the 'Parse Error on Input', and the set of moieties looks like this:

$\{\{1, 2, 3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$

The algorithm now returns the moieties list to the *Isolating delta debugging* algorithm.

3.4 Agnostic Debugging with a blackbox compiler

We explained earlier that delta debugging requires a test function; ours is a blackbox compiler. A blackbox compiler is accessed externally; only our tool and algorithms have no connection or knowledge of the internal components. Similar to delta debugging, employing a blackbox compiler is a direct mapping of the natural debugging technique. The programmer only has access to the compilers error message, and so do we.

From the error message, we get all the information needed to apply our result descriptions (Fail, Pass, Unresolved, ParseInput) which the algorithms use to decide which path to take. In the case of *Isolating delta debugging*, this is how it will next split up the two configurations or if it needs to call the good-omens algorithm. In contrast, the good-omens algorithm uses it to check if the generation of our splitting points, moieties, are valid or not.

The compiler we use as a blackbox for our primary evaluation of our tool is the Glasgow Haskell Compiler (GHC). However, as all we are using the compiler for is the error message, we have the benefit of being able to keep our tool separate. Disjointing our tool and algorithms from the compiler of a language allows it to be agnostic. An agnostic debugger's benefit is the ability to have only one debugger for many languages. We see this in our agnostic evaluation, where we will use Ocamlc and GHC as our blackbox compilers.

We propose that an agnostic debugging tool has the following trait no awareness of language-specific details or syntax within its source code. However, there are certain aspects within this trait our tool might need to know when debugging. Our solution is to provide the tool with external setting files to capture a particular programming language's nuances.

This style of agnostic behaviour, using external sources, is similar to software that uses languages settings. The software itself does not contain over 7000 different languages but instead relies on placeholders that call the correct languages from an external language file similar to this short example:

```
putStrLn langHelp
```

Listing 1: Software using language translation

```
langHelp = "Help"
```

Listing 2: External Language File - English

```
langHelp = "Hilfe"
```

Listing 3: External Language File - German

Applying this style of language setting behaviour to type error debugging solutions compliments the reasoning for using a *blackbox compiler*. Not only do users of the solution avoid any modification to the compiler, but it also allows for ease of introducing new or updated programming languages without making any changes to the solution.

The example above shows how some software treats multiple spoken languages, using placeholders within the source code that match a terminal within a "language configuration". When finding a match, the contents of the terminal replaces the placeholder. The agnostic type error debugger in this chapter will do the same for programming language-specific terminology. When using the "language configurations" style, each spoken language has a separate "settings file", with the correct language recognised by a setting within the program itself. The agnostic type error debugger also uses separate files for each programming language. However, modifying the debugger every time a new programming language needs debugging goes against the core agnostic trait. As such, the type error debugger also pattern matches the "settings file". For the matching of the programming language to "settings file" to occur, the debugger needs one argument, that of the command for compiling a program in the users chosen programming language or build tool. For instance,

```
agnosticDebugger ghc -o myProgram myProgram.hs
agnosticDebugger cabal build myProgram.hs
agnosticDebugger ocamlc -o myProgram myProgram.ml
```

Would match the first run of the debugger, "agnosticDebugger", with the "settings file" for the Glasgow Haskell Compiler, and the second with the "settings file" for Cabal, and lastly with the "settings file" of OCamlc. Whatever the first argument for the agnostic debugger is will be the "settings file" that is invoked, whilst all the arguments after the first are used as standard, meaning the programmer can use any flags or program names they wish.

Now that the type error debugger knows what "settings file", and thus what programming language it will use, the substitution can happen. In figure 2 the full "settings files" for GHC are presented.

There are several exceptions that the type error debugger needs to not remove from the generated configurations to reduce blackbox compiler calls and to allow for substitution; these exceptions need to be listed in the "settings file". In figure 2, there are two sections, one for singular lines, in section `###EXCEPTIONS###`, and one for multi-line, in section `###MULTI_EXCEPTIONS###`, with commas separate both sets. However, those with multi-lines are placed within braces, so the type error debugger knows when these begin and end. However, exceptions are not the only pieces of information an agnostic solution needs to recognise. Recall that the blackbox compiler returns a result that is categorised by the type error debugger. Two of those categories are Fail, the configuration contains a type error, and ParseInput, the configuration contains a 'Parse Error on Input'. The previous type error debuggers used key terms from the output of the compilers error message to categorise the configurations correctly. Thus, knowledge of if the error contains the terms type error or parse is error is necessary. Unfortunately, there is no way to "discover" the substitutions for a programming language, nor are they the same for all statically typed function languages, so

```
###FILE_TYPE### -
hs

###TYPE_ERRORS### -
type, type , type:, type-variable

###TYPE_IGNORE### -
parse error, type signature, type constructor

###PARSE_ERRORS### -
parse error on input

###PARSE_IGNORE### -

###EXCEPTIONS### -
--,import

###MULTI_EXCEPTIONS### -
({-;-})
```

Figure 2: GHC Settings

again, these need to appear in the "settings file" as seen in figure 2 under sections ###TYPE_ERRORS### and ###PARSE_ERRORS###.

4 EVALUATION

In Section 2, our new method is presented, making our pre-processing algorithm work on a request only basis. We hypothesise that by combining *isolating delta debugging* and moiety algorithms, we should see a reduction in the time taken to locate type errors.

To show that our hypothesis is correct, we need to evaluate our method on a large data-set of ill-typed programs. In a previous paper, we designed one such data-set based on the real-world program Pandoc[1] [17, 18]. This 'scalability data-set' contains 80 modules of Pandoc, each with a manually inserted singular type error. The modules range in size from 32 to 2305 lines of code, giving us a good overview of how our tool effects programs of different sizes. We compare the results of this evaluation against our previous debugger, Elucidate, whose results have also been re-captured on a PC running Ubuntu Linux 20.04 with an AMD Ryzen 7 3800X, 32GB RAM and a Samsung 850 SSD.

4.1 Reduction of time

Question: *Can combining the isolating delta debugging, and moiety algorithms speed up the time taken to locate type errors?*

Let us look at Figure 3. Along the x-axis are our 80 modules from the scalability data-set, and the y-axis represents the time taken in seconds. To make the graph easier to read, we have omitted two tests and have placed them in the separate Figure 4, for Elucidate only, tests 79 at 2532 seconds(42 minutes 12 seconds) and test 80 at 2496 seconds (41 minutes 36 seconds).

Overall our new combined algorithms have significantly reduced the time taken to locate type errors. On average, we reduced the run-time by 1 minute 37 seconds. However, the most drastic differences are in our modules with over 200 lines. The most impressive was modules 79 and 80, which took over 40 minutes to return their results using our new method; both reduced by over 38 minutes (2310 seconds).

Unfortunately, not all of the tests successfully reduced the time taken. One such example is module 38, shown in more detail in Figure 5, which had the worse time increase at 482 seconds (8 minutes 2 seconds) over Elucidate. These increases on only some of the results are understandable. It is easy to assume that a reduction of time occurs because the debugger is no longer pre-processing entire programs linearly. However, this assumption excludes that applying the pre-processing algorithm, Moiety, compared to the 'on request' algorithm, Good-Omens, can cause *isolating delta debugging* to generate the configurations differently. Having the *isolating delta debugging* algorithm traversing different paths can increase the overall number of the results, particularly Unresolved outcomes. Each extra result is an additional call to the blackbox compiler, which raises the run-time. Figure 6 shows this increase in run-time and calls to the blackbox compiler, the category of the compiler results is on the x-axis, and the number of times each result is received on the y-axis. Here, Eclectic, on all result categories, has increase calls. This increase in compiler calls corresponds to all 21 out of 80 modules, which increased this evaluation's run-time.

As mentioned, we see that the addition of Unresolved and 'Parse Errors on Input' outcomes increases the time taken for the debugger. We currently have no way of predicting those outcomes before the debugger runs, especially agnostically.

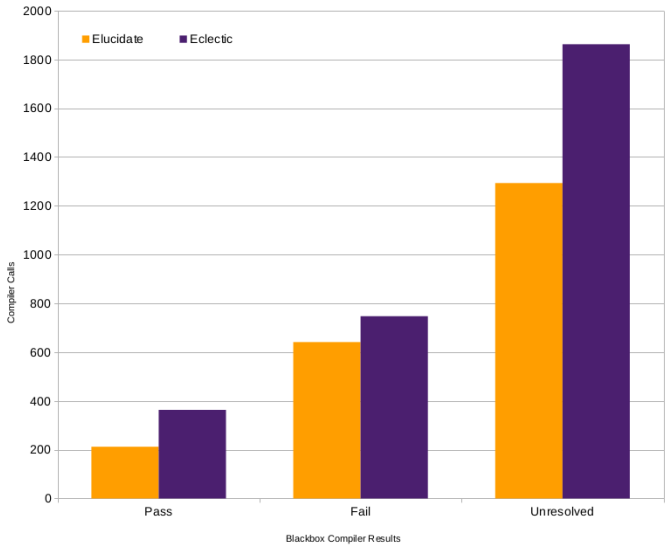

Figure 6: An increase of compiler calls leads to an increase of run-time

4.2 The quality of the debugger

In the previous section, we showed that our method successfully reduced the time taken to locate type errors. However, it is essential to show that a type error debugging tool has overall quality. In a previous paper, we introduced a framework to quantify the quality

[1]Pandoc is a popular Haskell library for markup conversion.

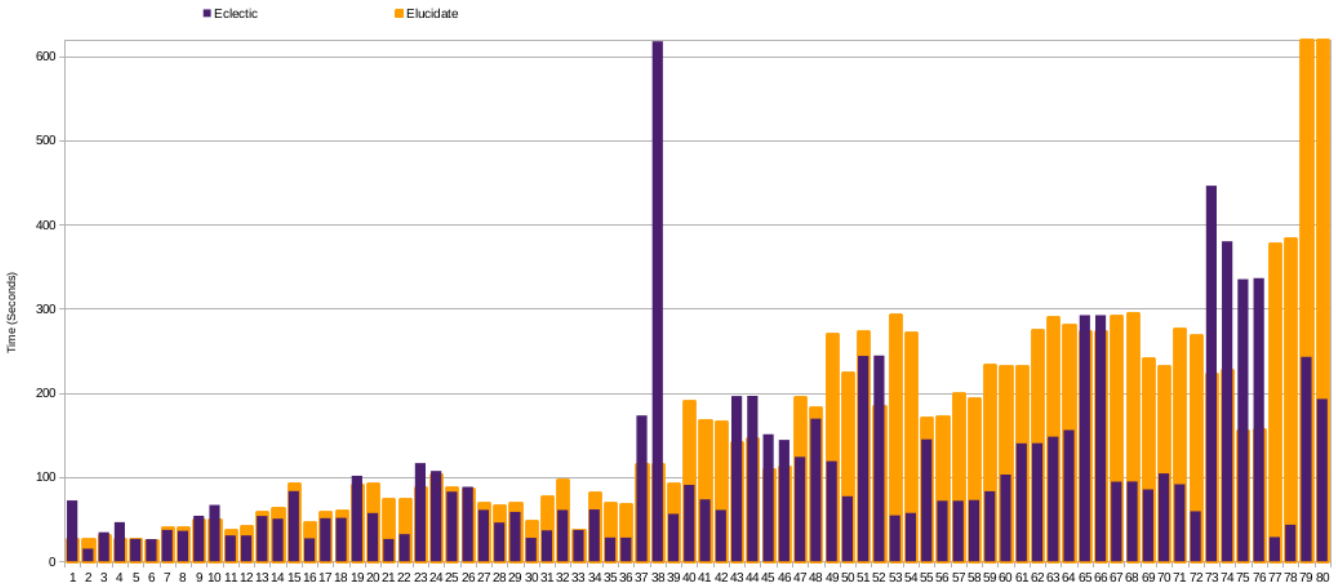

Figure 3: Elucidate and Eclectic - Run Time

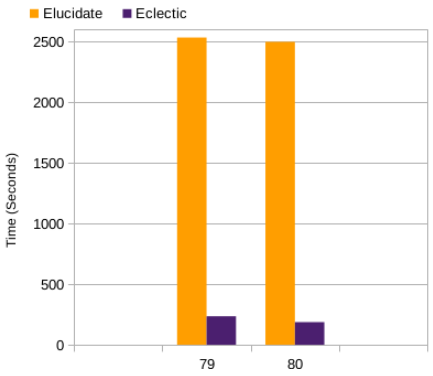

Figure 4: Programs 79-80

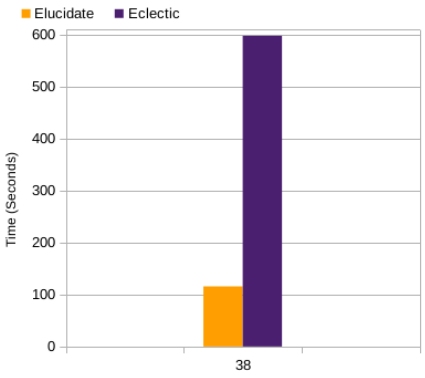

Figure 5: Program 38

of a debugger, and here we apply that framework to our tool [18]. The framework consists of four sections Accuracy, Recall, Precision and the F_1 Score. Commonly in type error debugging evaluations use recall only, the number of successful tests. However, it is also helpful to apply the other three sections to give a more rounded evaluation. Accuracy shows us the number of type error locations we correctly returned compared to those incorrectly returned. Precision tells us how many of the lines we have returned are correct, and the F_1 Score is the harmonic mean between recall and precision.

Metric	Elucidate	Eclectic
Accuracy	88%	83%
Recall	59%	79%
Precision	14%	11%
F_1 Score	19%	18%

Table 1: Framework Results - Average for the reduction of time evaluation

Table 1 shows the results of applying the framework to Eclectic. Here, along with Figure 7, the recall metric shows that the debugger increases from 59% to 79% on the number of correct locations; Eclectic located 63 out of 80 errors compared to Elucidate at 47. If the evaluation just used this metric, the new debugger would look significantly better on results and time reduction.

However, I want to provide a more authentic depiction of the debuggers. Unfortunately, that does not put Eclectic in a good light. Accuracy, precision, and F_1 Score are lower than the previous debugger. The lower than expected results are due to an increase in the returned line locations in 35 out of the 80 tests that contained a larger number of incorrect results. Module 70 contains an example of this. Elucidate returns the correct answer with one

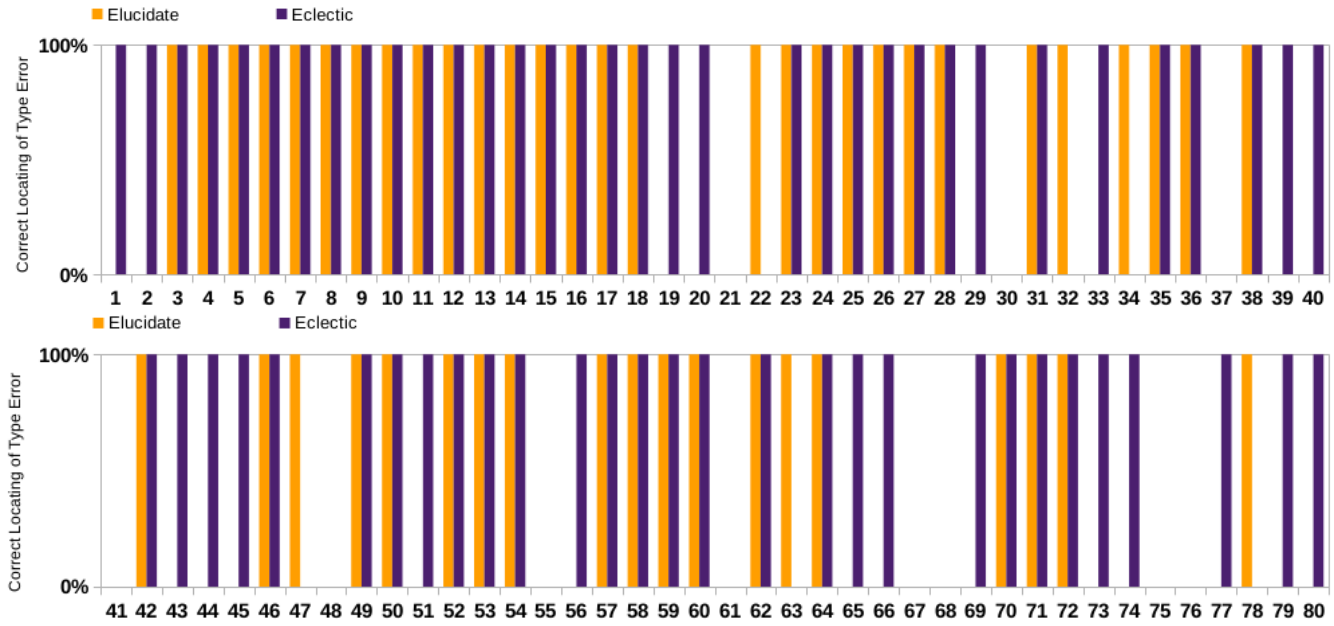

Figure 7: Recall data shows that the new debugger, Eclectic, locates 16 more type errors correctly, than the previous debugger Elucidate

reported line, while Eclectic does this in four lines. The reason for this discrepancy is the implementation of the good-omens algorithm. Currently, when calling the algorithm, an additional branch is generated. This branch is useful as it allows for more than one type error to be discovered, as seen in module 70's results. Elucidate returns only line 265, which in itself is a one-line function. However, Eclectic returns a three-line function `{49,50,51}` and the single line function at `{265}`. The need to discover more than one type error is subject to opinion and future work will see if this feature is more of a hindrance than a help. However, in some cases, the evaluation does get the opposite effect. When looking at module 77, Elucidate returns 28 results, each a different line number, and all 28 are incorrect. On the same module, Eclectic returns fewer results at two line numbers and gets the correct location of the type error.

Though these extra line results do not affect the core goal of reducing the debuggers time-taken, it reduces their quality. Further investigation is needed to iron out this problem.

4.3 Agnostic Debugging

As mentioned in Section 3.4, the Eclectic debugger and its algorithms are agnostic; its application can span multiple programming languages. To show that this agnostic behaviour works, we have evaluated our debugger on an additional statically typed language OCaml.

We converted 11 ill-typed Haskell programs from a set collated by Chen and Erwig [2] to OCaml. We used these benchmarks to see if we could successfully apply our debugger to another language. A successful application would show promising results for type error discovery in the new language.

Table 2 shows the debugger's results to both the Haskell and OCaml source code. For Haskell, we used the Glasgow Haskell Compiler as our blackbox, and for OCaml, the blackbox is OCamlc.

Metric	Haskell	OCaml
Accuracy	37%	49%
Recall	73%	73%
Precision	34%	64%
F_1 Score	44%	68%

Table 2: Framework - Average for the agnostic evaluation

For recall, the number of times the debugger correctly reported the line the error occurs on, and we see the debugger has identical results for 8 out of the 11 benchmarks. However, this is where the similarities stop. We see in Table 2 that for Accuracy, Precision, and F_1 Score, the OCaml language's results are more beneficial. The results outcome is due to the debugger not calling the good-omens algorithm. The lack of calls to the algorithm with OCaml happens because it does not have an equivalent to Haskells 'Parse Error on Input'[2]. Unfortunately, as shown our previous paper, the absence of an algorithm to remove these errors stunts the debugger's ability to scale to more extensive programs [18, 19]. More research is needed to see if OCamls lack of a 'Parse Error on Input' category would hinder the agnostic debuggers scalability when applied to other programming languages. However, the results do not affect the outcome that the debugger is proven to be wholly agnostic.

[2]To the best of the author's knowledge.

4.4 Summary

The evaluation proved that our new algorithms successfully locate type errors within our tool in a timely fashion on average. In the most favourable result, Eclectic reduced the time taken by over 38 minutes. We also managed to discover more correct locations of type errors with Eclectic than with Elucidate, as seen with our recall metric. However, when we gathered a more detailed look at our tool, it was clear to see that we struggled with a lower F_1 Score. Our short evaluation of agnostic debugging was a success, with evidence that there is a possibility of type error debuggers being agnostic in the future. Altogether, we have succeeded in reducing the tools time on average to locate type errors and shown that agnosticism works. However, we have also encountered further work within accuracy and precision, proving the necessity of a type error debugging tool framework.

5 RELATED WORK

Type error debugging research has a vast history which covers a variety of solutions over a span of thirty-plus years [1, 3, 4, 6, 10, 13, 15, 16, 20, 21, 23–25, 29]. However, these solutions tended to need either a modified compiler or patch. All of our tools, on which we have based this current work, use agnostic algorithms and a blackbox compiler and so bucked this trend [18, 19]. The core agnostic algorithm we have continued to use is Delta Debugging [5, 26–28]. The Delta Debugging algorithm automates the way programmers debug their software. The algorithm initially worked on a single configuration, faulty program; this version is the Simplifying Delta Debugging algorithm. However, it was not long until, Zeller the designer of Delta Debugging, released an improved version that worked on two configurations, *isolating delta debugging* algorithm. It is the *isolating delta debugging* algorithm of which our work uses. A core part of Delta Debugging is the necessity of a test function. This test function can be anything that produces results that can guide the algorithms path. In our case, we use a blackbox compiler, specifically the Glasgow Haskell Compiler (GHC). For us, a blackbox compiler is a compiler used the same way as a programmer does for accessing error messages. Unlike other solutions that suggest using a compiler as a blackbox, we do not need to apply any modifications allowing us to be a completely separate entity [8, 9, 12, 22]. Though we successfully combined *isolating delta debugging* algorithm and a blackbox compiler to locate type errors with a rate of 27 percentage points over GHC we found that our debugger would have issues scaling to larger programs [18]. Our answer was to look at the input given to the *isolating delta debugging* algorithm, as a configuration, before running. Though we were not the first to look in this direction, we were the first to do so with a pre-processing algorithm for type errors [7, 11, 14].

6 CONCLUSION AND FUTURE WORK

We presented our method of combining a modified *isolating delta debugging* algorithm, the moiety algorithm, a blackbox compiler, and the good-omens algorithm to locate type errors. Though successful in locating type errors, our previous tool had too low run-times [18]. Our new agnostic tool, Eclectic, addresses this problem of speed. Taking the strengths of both the *isolating delta debugging* and moiety algorithms, we united them with our new good-omens algorithm. Previously the self-contained moiety algorithm acted as a pre-processor for *isolating delta debugging*. Moiety generated 'parse error on input'-free configurations for the *isolating delta debugging* algorithm. These configurations allowed *isolating delta debugging* to know valid splitting points, locations that are available to be split without introducing an error. However, pre-processing came with a price: each line had to be type-checked against the compiler, leading to linear run-time. In contrast, Eclectic allows the *isolating delta debugging* algorithm to request valid splitting points only when it observes a 'parse error on input'. This change gives us an average reduction in the run-time of 1 minute 37 seconds.

Unfortunately, using a previously presented framework to quantify the quality of a type error debugger, the changes that made this significant difference in time slightly reduced the debugger quality overall compared to our previous implementation [18]. In recall, the most commonly used metric in type error debugging, we gain a positive 20% accurate locating of type errors on our previous debugger. However, the overall outcome of the tool shown by the F_1 Score returned 1% fewer. The explanation for this discrepancy is that the type error locations contain more lines of code than previously. Thirty-five of the eighty tests yielded larger locations, averaging ten lines more than Elucidate due to the algorithm's implementation. The current implementation allows for branches of the algorithm to find more than one type error in the code; however, we need to investigate if this is a beneficial aspect.

Along with the successful reduction of time, we also provide a short evaluation of the debuggers agnostic behaviour. This evaluation showed that we could apply our debugger and its algorithms to more than one programming language. However, though we received good results, we would like to complete a more in-depth investigation of agnostic debugging, with more extensive evaluations and a more comprehensive range of languages tested.

Concerning future work, we will first be looking into reducing these larger locations. We will not be able to reduce many to one location due to the moieties job of not allowing non-valid splitting points. However, we can investigate how the *isolating delta debugging* algorithm's differing chosen path causes this disparity and removes the ability to discover more than one type error at a time. Also, the tool works best for larger programs, with those at the shorter end not benefiting in reducing time. For this, we will be looking at heuristics to decide if to call the pre-processing or 'on request' algorithm. One such solution would resort back to the pre-processing Moiety algorithm if the programs source code is under a specific size.

Lastly, we would like to implement an empirical study using real programmers on the debuggers' ability to locate type errors and its agnostic features.

REFERENCES

[1] Karen L Bernstein and Eugene W Stark. 1995. *Debugging type errors.* Technical Report.
[2] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *POPL 2014*. ACM, 583–594.
[3] Sheng Chen and Martin Erwig. 2014. Guided Type Debugging. In *Functional and Logic Programming - 12th International Symposium*. 35–51.
[4] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ICFP 2001*. 193–204.
[5] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *27th International Conference on Software Engineering*. 342–351.

[6] 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.

[7] Christian Gram Kalhauge and Jens Palsberg. 2019. 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*. ACM, 556–566.

[8] 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

[9] Benjamin S. Lerner, Dan Grossman, and Craig Chambers. 2006. Seminal: searching for ML type-error messages. In *Proceedings of the ACM Workshop on ML*. 63–73.

[10] Bruce J McAdam. 1999. On the unification of substitutions in type inference. *Lecture notes in computer science* 1595 (1999), 137–152.

[11] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *ICSE '06*. ACM, 142–151.

[12] Zvonimir Pavlinovic. 2014. General Type Error Diagnostics Using MaxSMT. 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.

[14] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *PLDI 2012* (Beijing, China). ACM, 335–346.

[15] Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming, 12th International Symposium*. 1–16.

[16] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *ICFP 2016*. ACM, 228–242.

[17] Joanna Sharrad. 2021. Pandoc for evaluation of type error debuggers. Retrieved March 1, 2021 from https://github.com/JoannaSharrad/TypeErrorDebuggingScalabilityDataSet

[18] Joanna Sharrad and Olaf Chitil. 2020. Scaling Up Delta Debugging of Type Errors. In *Trends in Functional Programming: 21st International Symposium, TFP 2020, Krakow, Poland,*. Springer.

[19] Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *IFL 2018* (Lowell, MA, USA). ACM, 13–24.

[20] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. 72–83.

[21] 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.

[22] Kanae Tsushima and Kenichi Asai. 2012. An Embedded Type Debugger. In *IFL 2012*. 190–206.

[23] Kanae Tsushima and Olaf Chitil. 2014. Enumerating Counter-Factual Type Error Messages with an Existing Type Checker. In *PPL2014*.

[24] Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. 2020. Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker. In *IFL 2019: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. ACM.

[25] Mitchell Wand. 1986. Finding the Source of Type Errors. In *POPL 1986*. ACM, 38–43.

[26] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference*. 253–267.

[27] Andreas Zeller. 2009. *Why Programs Fail Guide to Systematic Debugging, 2nd Edition*. Academic Press.

[28] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[29] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. Diagnosing type errors with class. In *PLDI 2015*. ACM, 12–21.