Refining the Delta Debugging of Type Errors

Joanna Sharrad
jo@sharrad.co.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, black box, delta debugging

ACM Reference Format:
Joanna Sharrad and Olaf Chitil. 2021. Refining the Delta Debugging of Type Errors. In *33rd Symposium on Implementation and Application of Functional Languages (IFL '21), September 1–3, 2021, Nijmegen, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3544885.3544888

1 INTRODUCTION

All programmers want to easily identify and quickly fix the bugs they have. However, the cause of a type error in a statically typed functional language such as Haskell is notoriously awkward to locate. The compiler message provides little help when it reports the type error far from its actual cause. For example, for a Haskell 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 what the programmer intended, 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, because it is too much work to scale these solutions to full programming languages such as Haskell and maintain them with every change to the language. Hence we developed a simple tool that uses the compiler as a black box, not duplicating any type checking or even parsing.

Andreas Zeller's delta debugging [26–28] formalises the method that many programmers use to locate the cause of a bug in a program: they systematically remove or comment out parts of the program and test each such slice of the program, called *configuration* by him, whether it has the bug or not. Eventually, a small part of the original program is identified as the location of the cause.

Delta debugging has been the backbone of our approach from the beginning [19]. We chose a configuration to be a subset of lines of the original ill-typed program. This line-based approach avoids the need for parsing. In type error debugging the test of a configuration is a call to the compiler: a configuration may *pass* (type check and compile), *fail* (yield a type error) or be *unresolved* (yield any other compiler error, e.g. parse error or unknown identifier).

The more unresolved configurations delta debugging encounters, the more configurations it needs to test and thus the slower it becomes. We found that parse errors cause most unresolved configurations. Hence we developed an algorithm termed *moiety* that initially processes the ill-typed program to create information that the subsequently applied delta debugging algorithm uses to create only configurations that do not cause parse errors [18]. Although the sequential composition of both algorithms 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 black box compiler and for a typical module of 400 lines that can take 13 minutes [18].

In this paper, we introduce a new algorithm, *good-omens*, which is based on our original moiety algorithm but tests only a couple of lines. Our new delta debugging algorithm calls good-omens when needed. We make the following contributions:

- We present a new variant of the delta debugging algorithm for locating type errors and the good-omens algorithm (Section 3).
- We implement our new algorithms in a new type error debugger named Eclectic and evaluate against our previous debugger Elucidate for run-time and result quality (Section 5).
- We show that our debugger is truely language- and compiler-agnostic by evaluating its application to two functional programming languages, Haskell and OCaml (Sections 4 and 5).

IFL '21, September 1–3, 2021, Nijmegen, Netherlands

ACM ISBN 978-1-4503-8644-9/21/09…$15.00
https://doi.org/10.1145/3544885.3544888

2 ILLUSTRATING BY EXAMPLE

Our debugger requires an ill-typed program as input; so let us consider an extended variant of our example from the Introduction. This program has a single type error caused by 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)
```

Our *delta debugging* algorithm works on two configurations, a well-typed one and an ill-typed one. 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 *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 black box compiler. We use only the message returned by the compiler, which tells us whether a configuration has a type error (*fail*), compiles successfully (*pass*), contains a 'Parse Error on Input' (*noparse*) 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 *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*. *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 *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 is told that line 3 caused the parse error, and the algorithm will check lines just before it. So the first iteration of *good-omens* sends line 2 to the black box compiler.

Good-omens iteration 1

```
2   0 -> [0]
```

Again we receive a 'Parse Error on Input'. So the next iteration checks line 1:

Good-omens iteration 2

```
1 f x = case x of
```

That line on its own does not parse either, but the parse error is not in line 1, and it is not a 'Parse Error on Input'-error. Hence the good-omens algorithm terminates and returns the information that lines 1 to 3 form a moiety.

A *moiety* is a set of consecutive line numbers that shall not be split by delta debugging, because splitting would just cause a parse error. 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 to 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 to the delta debugging algorithm. Delta debugging continues with the updated moieties. At the end of iteration 2 delta debugging already produced two configurations which differ only by lines 1 to 3. Because these three lines form a moiety, delta debugging cannot reduce the difference between the two configurations any further. So delta debugging terminates with the result that the type error location is within the difference of the two configurations, that is, within lines 1 to 3.

3 THE ALGORITHMS

Developing delta debugging of type errors was a journey. In our first paper [19] we adapted and evaluated the isolating delta debugging algorithm for type error debugging. In our second paper [18] we improved the speed of delta debugging for larger ill-typed programs through pre-processing with a novel moiety algorithm. Finally, in this paper we improve the overall speed of the type error debugging process through a tight integration of moieties and delta debugging.

3.1 Delta Debugging of Type Errors

We chose delta debugging as basis for our work on locating type errors, because it is a simple method that uses a black box test with few assumptions on it. Delta debugging has been used to locate the causes of many types of run-time errors; to apply it to type error debugging we use a compiler for the black box test and interpret its result appropriately as pass, fail and unresolved.

Delta debugging is a search through *configurations*; for type error debugging a configuration is a slice of the ill-typed program. At the start of our research we chose a configuration to be any subset of lines of the ill-typed program. Thus development and continued maintenance of a parser are not required, meeting our original aim. Also a character-based approach would give too many configurations to search and a line is a sufficiently precise error location.

Zeller first defined the *simplifying delta debugging* algorithm [26–28] for shrinking a buggy program. We could use this algorithm for type error debugging to obtain a minimal ill-typed program. Minimal means that removing any further line would yield a well-typed or unresolved program. However, even a minimal ill-typed program is often large, because ill-typed definitions use many variables whose well-typed definitions the program needs to include. Therefore we use Zeller's later *isolating delta debugging* algorithm.

3.2 Isolating Delta Debugging

The *isolating delta debugging* algorithm [5, 27, 29] isolates the cause of a type error by computing two configurations: a well-typed configuration and an ill-typed configuration; the former is a subset of the latter. The small 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 iteratively grows the first configuration and shrinks the second, so that the difference between these two configurations shrinks.

In every iteration the difference between the two configurations is halved. A half is added to the well-typed configuration and removed from the ill-typed one. Doing the same with the other half, another two configurations are obtained. The black-box compiler checks the 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. This divisor is called *granularity*. Delta debugging terminates when it cannot reduce the difference between the two configurations any further.

3.3 Moieties Determine Configurations

It is well known that isolating delta debugging takes time logarithmic in the size of the input if no configuration is unresolved, because the granularity remains 2 throughout and the algorithm is basically a binary search [27]. However, if many unresolved configurations are met, then isolating delta debugging can require up to quadratic time [27]. Our implementation proved to be slow for larger ill-typed programs, because most tested configurations are unresolved, nearly always because of parse errors.

We found a way to avoid most unparseable configurations while keeping to our aim. We observed that the Glasgow Haskell compiler can produce different parse error messages, but by far the most commonly occurring one in our experiments was 'Parse Error on Input'. That error means that unexpectedly the given line is not the start of a top-level declaration (equation defining a function, type signature declaration, type definition, etc.). A parseable program module is a sequence of top-level declarations (including an optional module declaration at the beginning). Including some but not all lines of a top-level declaration in a configuration is highly likely to yield an unparseable, unresolved configuration.

We allowed the black box compiler to return a new fourth category *noparse* for the 'Parse Error on Input' of the Glasgow Haskell compiler[1].

[1] Other sorts of parse errors occur rarely and are still classified as unresolved

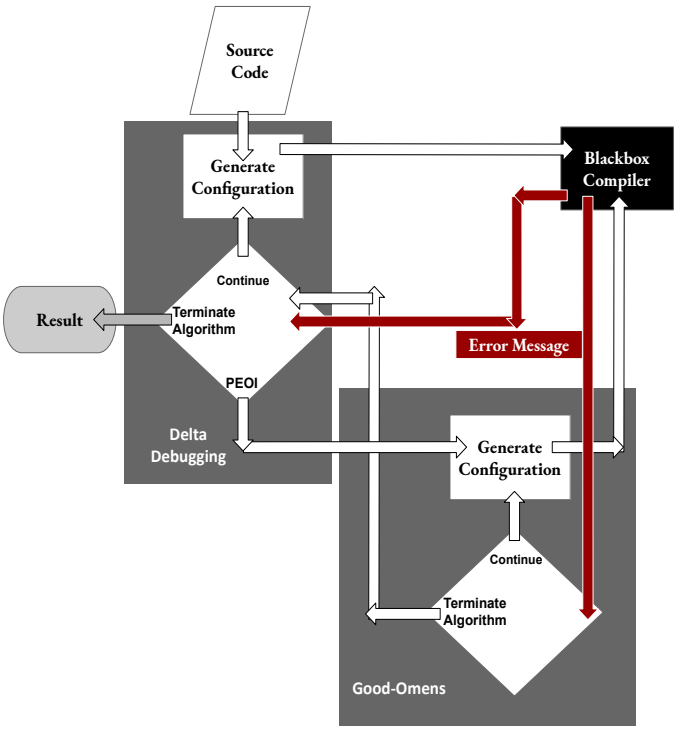

Figure 1: Informal overview of the control-flow of the algorithms in the Eclectic debugger

On the side of the type error debugger, we introduced the concept of a *moiety*, a set of consecutive lines that shall not be separated when forming a configuration. We created the moiety algorithm, which given an (ill-typed) program as input, produces a list of moieties. The subsequently run isolating delta debugging algorithm creates only configurations that obey the moiety list. No configuration ever yields *noparse*. Some configurations may still not be parseable and hence unresolved, but experiments showed that far fewer configurations are unresolved, and hence the isolating delta debugging algorithm is much faster. Unfortunately, the moiety algorithm proved to take substantial time to initially process the ill-typed program [18].

3.4 Delta Debugging with Good-Omens

In this paper, we present a new variant of isolating delta debugging that, whenever it comes across a configuration with a parse error, calls an algorithm we call *good-omens* that determines a single moiety around the parse error location. So moieties are determined on-request, and the set of configurations that are searched by the isolating delta debugging algorithm continuously reduces based on the current knowledge of moieties.

Figure 1 gives an informal overview of the control-flow of the *isolating delta debugging* and the *good-omens* algorithms working together.

3.5 Our Isolating Delta Debugging Algorithm

Algorithm 1 is the new isolating delta debugging algorithm. Our description of the algorithm keeps close to Zeller's Python implementation [27]. For simplicity, we removed the original *offset* variable, and we follow his later implementation [28] in using a Boolean variable *unres* to check whether the while loop is left via a break statement. We focus on explaining the differences to Zeller's implementation.

For us, a configuration is just a set of line numbers; the actual line content of each line number is in the variable *cont*. Hence the isolating delta debugging function dd has three parameters. It returns a tuple of two configurations.

In line 2, the list of moieties, *moieties*, is initialised such that every line is a separate moiety. Line 3 sets the granularity n to 2. If testing all modified configurations in the while loop returns unresolved, then the granularity is doubled in line 44.

In line 5, the difference *delta* between the two current configurations is determined. If given the constraints of the current moiety list that *delta* cannot be split into n (the granularity) parts, then the algorithm terminates in line 7; otherwise, the result of the split is assigned to the array *deltas* in line 8.

Each iteration of the while loop tests a modification of the pass and fail configuration. Lines 16 to 24 are our main addition to Zeller's algorithm. If one of the two modified configurations does not parse, then our good-omens algorithm is called with the number of the line in which parsing failed. Isolating delta debugging will afterwards continue with an updated moiety list.

Algorithm 1: Delta Debugging of a type error

```
1  define dd (cPass, cFail, cont)
2    moieties ← initialMoieties(cont)
3    n ← 2
4    loop
5      delta ← cMinus(cFail, cPass)
6      if n > numOfMoieties (delta, moieties) then
7        return (cPass, cFail)
8      deltas ← cSplit(delta, moities, n)
9      unres ← True
10     j ← 0
11     while j < n do
12       nextCPass = cPlus(cPass, deltas[j])
13       nextCFail = cMinus(cFail, deltas[j])
14       resNextCFail ← test(nextCFail, cont)
15       resNextCPass ← test(nextCPass, cont)
16       if resNextCFail == NOPARSE then
17         moieties ←
             go(line(resNextCFail), moieties, cont)
18         unres ← False
19         break
20       resNextCPass ← test(nextCPass, cont)
21       if resNextCPass == NOPARSE then
22         moieties ←
             go(line(resNextCPass), moieties, cont)
23         unres ← False
24         break
25       else if resNextCFail == PASS then
26         cPass ← nextCFail
27         n ← 2; unres ← False; break
28       else if resNextCPass == FAIL then
29         cFail ← nextCPass
30         n ← 2; unres ← False; break
31       else if resNextCFail == FAIL then
32         cFail ← nextCFail
33         n ← max(n − 1, 2); unres ← False; break
34       else if resNextCPass == PASS then
35         cPass ← nextCPass
36         n ← max(n − 1, 2); unres ← False; break
37       else -- try next part of delta
38         j ← j + 1
39     end while
40     if unres then -- all configurations are unresolved
41       if n >= numOfMoieties (delta, moieties) then
42         return (cPass, cFail)
43       else -- increase granularity
44         n ← min(n ∗ 2, numOfMoieties(delta, moieties))
45   end loop
46 end define
```

Algorithm 2: Good-Omens determines one moiety

```
1 define go (l, moieties, cont)
2   moiety ← mkMoiety(l)
3   l ← l − 1
4   while test (cLine (cont,l)) == NOPARSE do
5     addLine(l, moiety)
6     l ← l − 1
7   end while
8   return updateMoiety (moiety, moieties)
9 end define
```

3.6 The Good-Omens Algorithm

The good-omens algorithm is an on-request variant of our moiety algorithm. The good-omens algorithm go is shown as Algorithm 2. Besides the number of a line which yields a noparse error, it also takes the current list of moieties and the content of the original ill-typed program as parameters.

The algorithm creates a new moiety that consists only of the line specified by the first parameter. Subsequently a while loop checks each preceding line whether the black box compiler yields noparse. All such lines are added to the moiety until a line is found that does not yield noparse; in extremis that will be the first line[2]. Finally the newly created moiety is integrated with the old list of moieties, which in practice means combining several moieties of the old list into one new moiety in the new list. That list is the result of the algorithm.

4 AGNOSTIC DEBUGGING

For our type error debugger the compiler is a black box, because the debugger does not need any knowledge of the compiler's internal workings; the debugger only produces the compiler's input and uses its output. The compiler is a test of a configuration that either succeeds (*pass*) or the (first) error message categorises the test as *fail, noparse* or otherwise *unresolved.*

The debugger needs a small amount of language- and compiler-specific information to produce the compiler's input and categorise its output. Our past debuggers included code to work on Haskell programs with the Glasgow Haskell compiler (GHC) or the build tool Cabal. Now we build a *language- and compiler-agnostic type error debugger*, that is, the debugger contains no language- or compiler-specific code. This one debugger can be used for many programming languages and compilers. In Section 5.3 we will evaluate it also for OCaml programs using the OCamlc compiler. We provide the agnostic debugger with external settings files that contain necessary programming languages' and compilers' nuances. Thus also new language features or an updated compiler do not require any changes to the debugger.

To select the correct settings file, the debugger uses one argument, that of the command for compiling (or building) a program in the user's chosen programming language. For instance:

[2] Algorithm dd starts with an ill-typed program that does parse.

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

The first line runs the debugger *agnosticDebugger* with the settings file for the Glasgow Haskell Compiler, the second line with the settings file for the Cabal build tool used by many Haskell projects and the final one with the settings file for OCamlc. The first argument for the agnostic debugger determines (the name of) the settings file, whilst all the arguments after the first are passed to every call of the compiler, meaning the programmer can use any flags or program names they wish.

In Figure 2 the full settings file for GHC is presented.

```
###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

If compilation fails, then the type error debugger uses key phrases in the (first) error message to categorise the configuration. In the settings file `###TYPE_ERRORS###` heads the phrases for *fail* and `###PARSE_ERRORS###` heads the phrases for *noparse*. The headings `###TYPE_IGNORE###` and `###PARSE_IGNORE###` contain phrases that would trigger a *fail* or *noparse* but should not. If compilation of a configuration fails, but it is neither *fail* nor *noparse*, then it is *unresolved*.

In principle no language-specific information is required to produce a configuration, as it is just a subset of lines of the original program. However, to reduce the number of black box compiler calls, the type error debugger should never remove any comments from the source. Similarly, import declarations should never be removed, because imported modules are always well-typed but removing the declarations is likely to yield an unresolved configuration because of undefined identifiers. Hence the phrases indicating any such exceptions need to be listed in the settings file. In Figure 2 heading `###EXCEPTIONS###` is for singular lines and heading `###MULTI_EXCEPTIONS###` for multi-lines. For multi-line exceptions the phrases are paired in parentheses, so the type error debugger knows when such an exception begins and ends.

5 EVALUATION

Our new type error debugger is called *Eclectic*. It supports the same features as our previous type error debuggers [19], but it implements three core elements:

(1) the modified *isolating delta debugging* algorithm
(2) the *good-omens* algorithm
(3) the *agnostic* behaviour

Our previous type error debugger using the moiety algorithm as a pre-processor is called *Elucidate*.

We hypothesise that compared to Elucidate, Eclectic reduces 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[3] [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 affects 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.

5.1 Reduction of Time

Question: *Can integrating the moiety algorithm into the isolating delta debugging 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: Elucidate takes 2532 seconds (42 minutes and 12 seconds) for test 79 and takes 2496 seconds (41 minutes and 36 seconds) for test 80.

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

Unfortunately, not all of the tests successfully reduced the time taken. One such exception is module 38, shown in more detail in Figure 5, which had the worst 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 traverse different paths can increase the overall number of the results, particularly *unresolved* outcomes. Each

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

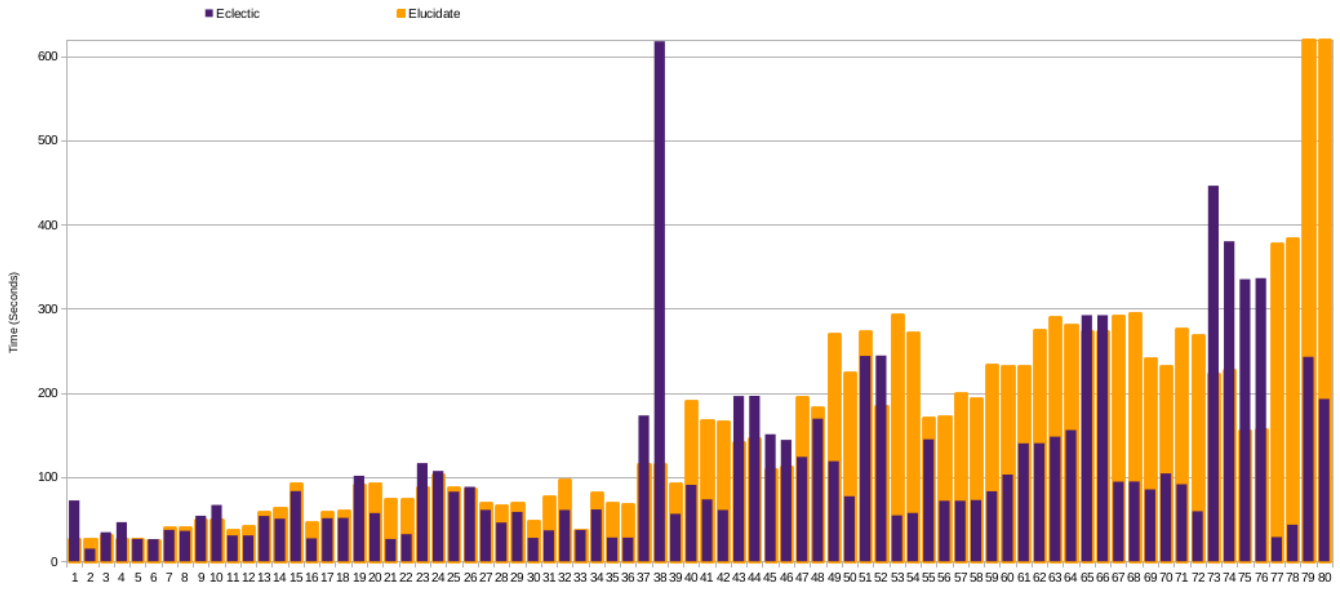

Figure 3: Elucidate and Eclectic - Run Time

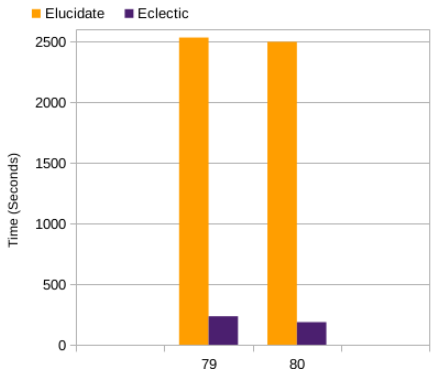

Figure 4: Run Time — Programs 79&80

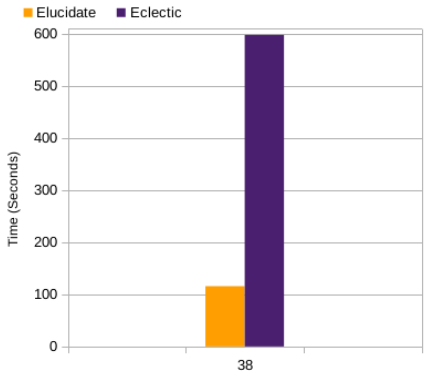

Figure 5: Run Time — Program 38

extra result is an additional call to the black box compiler, which raises the run-time. Figure 6 shows this increase in run-time and calls to the black box 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 increased calls. This increase in compiler calls matches the 21 out of 80 modules that increased the debugger's run-time.

As mentioned, we see that the increase of *unresolved* and *noparse* outcomes increases the time taken for the debugger. We currently have no way of predicting those outcomes before the debugger runs, especially agnostically.

5.2 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 debugger has overall quality. In a previous paper, we introduced a framework to quantify the quality of a debugger, and here we apply that framework to our tool [18]. The framework consists of the four qualities *accuracy*, *recall*, *precision* and the F_1 *score*. Commonly, type error debugging evaluations use only *recall*, the percentage of successful tests. However, it is also helpful to apply the other three qualities, to give a more rounded evaluation. *Accuracy* shows us the number of type error locations that the debugger correctly returned compared to those incorrectly returned. *Precision* tells us how many of the lines the debugger returned are correct, and the F_1 *score* is the harmonic means of recall and precision.

Table 1 and Figure 7 show the results of applying the framework to the old Elucidate and the new Eclectic, using our 80 modules. Recall increases from 59% to 79% ; more precisely, Eclectic located 63 out of 80 errors compared to Elucidate at 47. If the evaluation

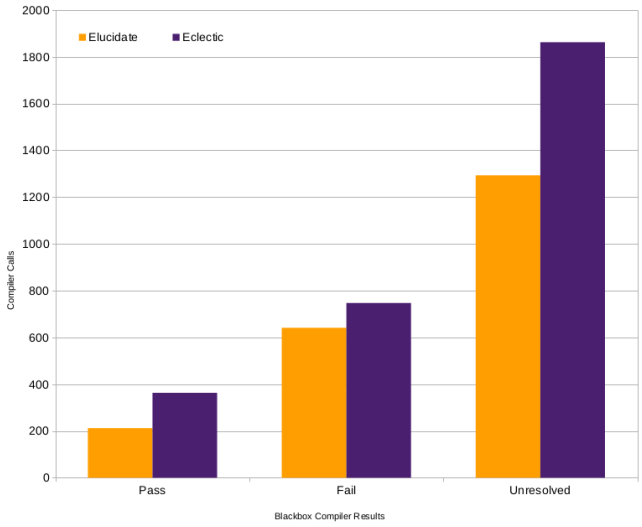

Figure 6: An increase of compiler calls leads to an increase of run-time

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

Table 1: Framework Results - Average for the 80 modules

just used this quality, the new debugger would look significantly better on quality and time reduction.

However, we 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 for the previous debugger. The lower than expected results are due to an increase in the number of returned lines: in 35 out of the 80 tests Eclectic returned a larger number of incorrect results than Elucidate. Module 70 is an example of: Elucidate returns the correct answer in one line, while Eclectic does so 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 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 we 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 just two lines and these includes the correct location of the type error.

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

5.3 Agnostic Debugging

As mentioned in Section 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 for both Haskell and OCaml for all 11 benchmark programs. For Haskell we used the Glasgow Haskell Compiler as our black box, and for OCaml the black box 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

The debugger actually produces identical results for 8 out of the 11 benchmarks. Recall, the number of times the debugger correctly reports the line the error occurs on, is identical for both languages. However, this is where the similarities stop. For accuracy, precision, and F_1 Score the results are better for OCaml than for Haskell. This 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 Haskell's 'Parse Error on Input'[4]. Unfortunately, as shown in our previous paper, the absence of an algorithm to remove these parse errors stunts the debugger's ability to scale to longer programs [18, 19]. More research is needed to see if OCaml's lack of a 'Parse Error on Input' category would hinder the agnostic debugger's scalability when applied to other programming languages. However, the results do not affect the outcome that the debugger is proven to be wholly agnostic.

5.4 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 challenges within accuracy and precision, proving the necessity of a type error debugging tool framework.

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

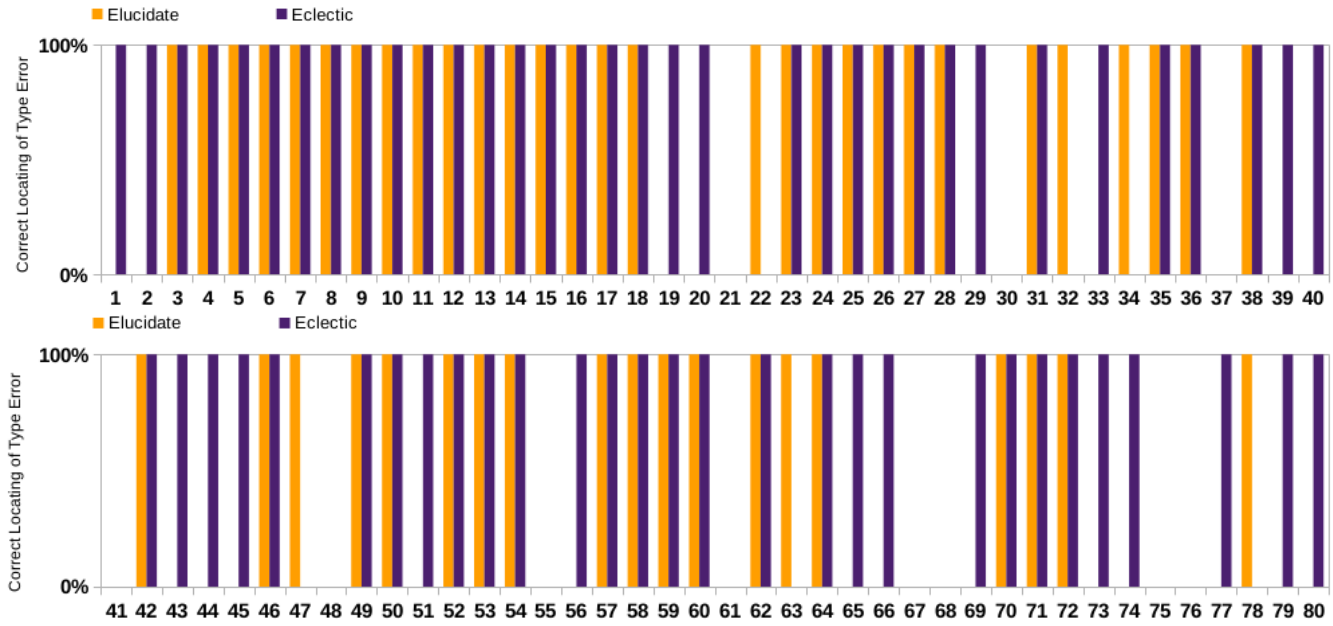

Figure 7: Recall data shows that the new debugger, Eclectic, correctly locates 16 type errors more than the previous debugger, Elucidate

6 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, 30]. 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 black box compiler and so bucked this trend [18, 19]. The core agnostic algorithm we have continued to use is *delta debugging* [5, 26, 27, 29]. The delta debugging algorithm automates the way programmers debug their software. The algorithm initially worked on a single configuration, the faulty program; this version is called 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, the *isolating delta debugging* algorithm. Our work uses this *isolating delta debugging* algorithm. A core part of delta debugging is the need for a test function. This test function can be anything that produces results that can guide the algorithm's path. In our case, we use a black box compiler, specifically the Glasgow Haskell Compiler (GHC). For us, a black box 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 black box, we do not need to apply any modifications, allowing our debugger to be a completely separate entity [8, 9, 12, 22]. Though we successfully combined the *isolating delta debugging* algorithm and a black box 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 delta debugging type errors [7, 11, 14].

7 CONCLUSION AND FUTURE WORK

We presented our method of integrating our moiety algorithm into a modified *isolating delta debugging* algorithm locate type errors. Though successful in locating type errors, our previous tool had too long run-times [18]. Our new debugger Eclectic addresses this problem of speed and additionally is language- and compiler-agnostic. Taking the strengths of both the *isolating delta debugging* and the moiety algorithms we united them through our new good-omens algorithm. Previously the self-contained moiety algorithm acted as a pre-processor for *isolating delta debugging*. Moiety generated *noparse*-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 compiled separately, leading to linear run-time. In contrast, Eclectic allows the *isolating delta debugging* algorithm to request valid splitting points only when it comes across a *noparse* configuration. This change gives us an average reduction in 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 percentage points in accurate locating of type errors on our previous debugger. However, the overall outcome of the tool shown by the F_1 Score is 1 percentage point lower. 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 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 debugger's 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 the on-request algorithm. One such solution would resort back to the pre-processing moiety algorithm if the program's source code was under a specific size.

Lastly, we would like to implement an empirical study using real programmers on the debugger's 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: A Guide to Systematic Debugging, 2nd Edition*. Academic Press.

[28] Andreas Zeller. 2021. Reducing Failure-Inducing Inputs. In *The Debugging Book*. CISPA Helmholtz Center for Information Security. https://www.debuggingbook.org/html/DeltaDebugger.html Retrieved 2021-09-03 14:44:52+02:00.

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

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