Delta Debugging Type Errors in Real-World Programs

Category: Research

Joanna Sharrad*[0000−0003−2334−8862] and Olaf Chitil[0000−0001−7986−9929]

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

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

Keywords: Type Error, Error Diagnosis, Blackbox, Delta Debugging, Haskell

1 Introduction

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

In our opinion the major reason for this non-adoption is the effort required for implementing proposed solutions for full programming languages and maintaining them in the face of evolving languages and compilers. Proposed solutions usually require new compiler front-ends, including new type inference implementations, or substantial modifications of existing compilers. We believe that a small improvement that requires little implementation and maintenance effort is much better than a big improvement that requires substantial effort. Hence it has been our goal to develop a type error debugger that uses the compiler as a true black box, that is, it calls the compiler as an external program. The debugger should not duplicate compiler work such as parsing. The debugger should use minimal information from the outcome of the compiler call; in particular, it

* PhD Student

should not rely on the details of its type error messages. As a consequence such a debugger is mostly programming language agnostic.

In an earlier paper we presented and evaluated such a type error debugger [15]. Our debugger implements the isolating delta debugging algorithm [24] to locate the defective line in an ill-typed program. Our debugger tests many configurations, that is variants, of the ill-typed program by calling the compiler; the only information the debugger uses is whether compilation succeeded (passed), failed with a type error (fail), or failed with some other error (unresolved).

We showed that our debugger yields good locations in reasonable time for 121 ill-typed programs that had been taken from papers on type error debugging [2]. However, all these programs are short; the longest has 23 lines. So for many type error debugging methods proposed in the literature it is unknown whether they scale for larger programs. We found that our debugger was unacceptably slow for large programs. The isolating delta debugging algorithm tests a logarithmic[1] number of configurations if no outcome is unresolved. The more outcomes are unresolved, the less efficient it becomes, up to a quadratic number of configurations. "When using ... [isolating delta debugging], it is thus wise to keep unresolved test outcomes to a minimum, as this keeps down the number of tests required" [24]. All applications of isolating delta debugging try to minimise the number of configurations with unresolved test outcomes. Additionally, Kalhauge and Palsberg [7] note that delta debugging as a greedy algorithm may fail to produce the best global solution. When delta debugging Java bytecode decompilers, they observe how unresolved test outcomes lead to poor debugging results. Many configurations have unresolved outcomes, because some dependencies of Java bytecode classes are missing. Their solution is to first produce a dependency graph of Java bytecode classes. That dependency graph then guides their delta debugging algorithm in choosing configurations of the Java bytecode classes to test, avoiding any configuration that would miss dependencies.

Most unresolved outcomes in our type error debugger are parse errors. Building some kind of parser for our debugger would contradict our goals. Hence here we present an algorithm, we name it Moiety, that calls the compiler as a black box. Moiety divides an ill-typed program into intervals of lines, we name them moieties. In delta debugging no configuration should contain only part of a moiety, because in that case compilation will fail with a parse error. The moiety information guides the isolating delta debugging algorithm. We reduce unresolved test outcomes and thus the total runtime of the type error debugger.

To debug real-world programs, we now support multi-module programs and a standard build tool. We implemented the new type error debugger, Elucidate, which combines the new Moiety algorithm with an isolating delta debugging algorithm that uses moiety information. The debugger locates a defective line of an ill-typed Haskell program, using the Glasgow Haskell compiler[2] as black box.

In this paper we make the following contributions:

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

- We present the Moiety algorithm, which generates, using the compiler as a black box, a list of moieties of the ill-typed program. That list determines the set of configurations searched by the isolating delta debugging algorithm. (Section 3).
- We show how to extend delta debugging of type errors to multi-module programs (Section 4).
- We propose a framework for evaluating type error debuggers (Section 5).
- We evaluate Elucidate in our framework, using a corpus of 80 ill-typed variants of the real-world program Pandoc (Section 6).

2 The Problem

2.1 Delta Debugging Type Errors

Let us briefly review what delta debugging is and how we applied it to type error debugging [15].

To locate the defect in an ill-typed program, many programmers simply remove (or comment out) some parts of the program and compile the smaller program. If the smaller program is also ill-typed, the procedure is repeated. If the smaller program is not ill-typed, a different part of the previous program is removed. This shrinking by trial and error repeats until the program cannot be shrunk further, that is, no smaller program is ill-typed.

Simplifying delta debugging [23, 24] is a greedy algorithm that automates this method. Simplifying delta debugging divides the program into two halves and tests each one. If one half is ill-typed, the algorithm calls itself recursively for that half. If neither half is ill-typed, it divides the program into four parts and tests each one. Again the algorithm calls itself recursively for any ill-typed part, but if none is ill-typed, it tries again by dividing the program into eight parts. When the program cannot be divided further, the algorithm stops with the last ill-typed program as result.

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

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

[3] Removing single characters is another popular choice.

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

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

The *isolating delta debugging* algorithm [5, 24, 25] works with a pair of configurations, a passing and a failing configuration, the former being a subconfiguration of the latter. The algorithm starts with the empty configuration as passing configuration and the ill-typed program as failing configuration. The algorithm divides the difference between the two configurations into two parts and tests the passing configuration with each of these parts added and the failing configuration with each of these parts removed. If any tested configuration yields a passing outcome, it can become the new passing configuration, if any tested configuration yields a failing outcome, it can become the new failing configuration; then the algorithm calls itself recursively with a new pair of configurations. If all of the tested configurations yield unresolved outcomes, the difference is divided instead into four, eight, etc. parts, similar to the simplifying delta debugging algorithm, until eventually a passing or failing configuration is found; if no further division is possible, the algorithm terminates. The algorithm does not specify how the difference between two configurations is divided into parts and there may be several passing and failing outcomes; thus the algorithm is nondeterministic; however, like any other implementation, ours makes a choice and thus is deterministic. In every recursive call the passing configuration is a subconfiguration of the failing configuration (and both are subconfigurations of the original ill-typed program). Every recursive call reduces the difference between the two configurations, until the difference cannot be reduced any further.

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

The final pair of configurations is the result of the isolating delta debugging algorithm. The difference between the two configurations, which may be neither a passing nor a failing configuration, isolates a failure cause. The result returned by our type error debugger is this difference, that is, one or more lines of the original ill-typed program.

2.2 Small and Real-World Programs

We noted in the introduction that ill-typed programs that have been used to evaluate type error debuggers are short. The longest program in a recent benchmark suite [2] of 121 programs has just 23 lines. Such programs are good for studying how a type error debugger works and many of these programs are representative for the first programs written by novices learning a functional programming language. However, not just novices need help with type error debugging, but

also more experienced functional programmers who build useful, real-world programs. So we measured in October 2019 the top 100 Haskell programs on the popular public repository GitHub[5]. On average each program has 31872 lines of code, 138 modules, and 229 lines of code per module. Our aim in this paper is to ensure that type error debugging works well for such real-world programs.

2.3 The Effect of Unresolved Outcomes on Delta Debugging

Because our definition of configuration is based on program lines, all complexity measures of type error debugging are with respect to the number of lines of the ill-typed program. For a given ill-typed program there exists an exponential number of configurations. Already finding a failing configuration of minimal size is known to be NP-complete [10].

In type error debugging nearly all runtime is spent in the tests made by the compiler. In general, the runtime of delta debugging is proportional to the number of tests made.[6]

We see from the description of delta debugging that if no test outcome is unresolved, it is basically a binary search. In contrast, frequent unresolved outcomes cause the algorithm to repeatedly divide (differences of) configurations into four, eight, etc. parts and make more tests. So as we already stated in the introduction, the isolating delta debugging algorithm has logarithmic time complexity if no outcome is unresolved. The more outcomes are unresolved, the less efficient it becomes, up to a quadratic time complexity [24].

Therefore any successful application of delta debugging makes some effort to avoid unresolved outcomes. Like other applications we decided to base our definition of configurations on removing complete lines instead, for example, of dividing on the character level, because lines often contain complete definitions of types or functions, and thus a configuration has a reasonable chance to be a syntactically correct program.

2.4 Unresolved Outcomes in Type Error Debugging

Our type error debugger processes small programs in a few seconds [15], but when we applied it to a program of a few hundred lines, it took an hour. That time is unacceptable. We just established that real-world Haskell programs consist of hundreds of modules, each of which can have a few hundred lines of code. Although for the moment we still consider only single module programs, these may have hundreds of lines of code.

Let us look again at our earlier results [15] for 900 programs that we had generated by concatenating pairs of some of our original benchmark programs. We order the 900 programs by number of lines and put them into 4 groups: the shortest 125 in the first group, the next 125 in the second group, etc. Table 1 shows the outcomes. These indicate that the number of unresolved outcomes grows substantially with program size.

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

[6] This assumes similar runtime for every test, which may not be the case.

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

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

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

Table 2. Number of error messages giving unresolved outcome.

There is an obvious suspect for the high number of unresolved outcomes in larger programs: although splitting multiple equations of a single function definition yields well-formed definitions in Haskell, splitting a multi-line equation into half usually yields ill-formed programs; the same holds for multi-line type declarations, which often appear in larger programs, and case expressions with a branch per line. Many configurations are simply unparsable! To test our suspicion, we introduced a single type error in a module of the real-word program Pandoc from GitHub. The ill-typed module has 87 lines and our debugger had 126 unresolved outcomes, which we categorise by error message of the Glasgow Haskell compiler in Table 2. Most error messages are related to parsing and "parse error on input" is by far the most frequent one. "parse error on input" indicates that parsing failed somewhere inside the configuration, whereas "parse error (incorrect indentation or mismatched brackets)" indicates that parsing fails at the end of the configuration.

3 The Moiety Algorithm and Delta Debugging

We always obtain a configuration that does not parse, if we split the original ill-typed program at certain consecutive lines. So we first determine these lines that should never be separated and then apply the delta debugging algorithm such that it never splits in these places. Given its dominance, we solely focus on the "parse error on input" error message.

We name our pre-processing algorithm *Moiety*; according to the Merriam-Webster dictionary a moiety is "one of the portions into which something is divided".[7] Moiety divides the ill-typed program into moieties, that is, intervals of lines. Instead of actually dividing the ill-typed program, it proves to be easier

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

to keep the moiety information separately from the ill-typed program, as an ordered list of line intervals.

3.1 Example Application of the Algorithm

To see how Moiety works, we consider the following ill-typed program:

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

To limit runtime, the algorithm may only traverse the program once from beginning to end to produce its list of moieties. Moiety calls the compiler to test a program for whether it yields "parse error on input" or not. We show the tested program on the left and the test outcome and resulting moiety list on the right. We begin with the program consisting only of the first line of our original program:

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

not "parse error on input"

`[[1]]`

Because parsing does not fail inside the program, the line starts our first moiety. We now look at line 2 and want to check whether it can start another new moiety:

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

"parse error on input"

`[[1,2]]`

The parse error states clearly that line 2 cannot start a new moiety; so we add the line to the preceding moiety. We continue with considering line 3:

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

"parse error on input"

```
[[1,2,3]]
```

Like line 2, line 3 also cannot start a new moiety; we add line 3 to the preceeding moiety. We continue with line 4:

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

not "parse error on input"

```
[[1,2,3],[4]]
```

So line 4 can start a new moiety and we update our moiety list accordingly. Next line 5:

```
1
2
3
4
5 plus = (+)
6
7
8
9
```

not "parse error on input"

```
[[1,2,3],[4],[5]]
```

Likewise, line 5 starts a new moiety. We move on to line 6:

```
1
2                                   not "parse error on input"
3
4                                   [[1,2,3],[4],[5],[6]]
5
6 fib x = case x of
7
8
9
```

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

```
1
2                                   "parse error on input"
3
4                                   [[1,2,3],[4],[5],[6,7]]
5
6
7       0 -> f x
8
9
```

At this point it is hopefully obvious that lines 8 and 9 each also gives the outcome "parse error on input" and so the algorithm finishes with the moieties `[[1,2,3],[4],[5],[6,7,8,9]]`.

3.2 The Algorithm

Working through the example shows how simple the Moiety algorithm is: The algorithm tests every single line of the original ill-typed program whether it yields "parse error on input" or not. In case of the former, the line cannot be the start of a moiety and becomes part of the moiety of the preceding line. Otherwise it does start a new moiety. The result is an ordered list of moieties.

We note that line 1 never yields "parse error on input" and hence testing it is actually unnecessary.

3.3 Isolating Delta Debugging with Segmentation Information

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

So in our example we have the moiety list `[[1,2,3],[4],[5],[6,7,8,9]]`.

We start isolating delta debugging with the passing configuration {} and the failing configuration {`[1,2,3],[4],[5],[6,7,8,9]`}. We divide the difference between the two configurations by two and hence test the configurations {`[1,2,3],[4]`} and {`[5],[6,7,8,9]`}. Both configurations give the outcome

unresolved. Hence we have to divide the difference between our passing and failing configuration by four and test the configurations {`[1,2,3]`}, {`[4]`}, {`[5]`}, {`[6,7,8,9]`} and the configurations {`[4],[5],[6,7,8,9]`}, {`[1,2,3],[5],[6,7,8,9]`}, {`[1,2,3],[4],[6,7,8,9]`}, {`[1,2,3],[4],[5]`}. Our implementation happens to test {`[5]`} first and the test gives outcome pass.

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

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

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

3.4 Time Complexity

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

4 Debugging Real-World Programs

Real-world programs do not just have many lines, but they nearly always consist of many modules and are compiled using a build tool.

4.1 Modular Programs

The type error location of a compiler is unreliable, but our type error debugger assumes that the first module identified by the compiler as ill-typed does contain the type error location; our type error debugger works solely on that module.

If a module causes the first compiler type error, then all modules directly or indirectly imported are well-typed. An identifier defined in an imported module may have a type that contradicts with how the identifier is used in the ill-typed module. However, even when both definition and use are in the same module

and the definition is typable, delta debugging will always identify the use of the identifier as the cause of the error, not the definition. So our treatment of modules is consistent with our general treatment of definition vs. use. Furthermore, identifiers exported by a module rarely have a wrong type, because their type is additionally stated by a type declaration.

4.2 Import Declarations

Because all imported modules are well-typed, an import declaration is never the location of a type error. However, removing an import declaration from a module usually leads to an unresolved outcome, because numerous identifiers lack definitions. Hence our type error debugger leaves all import declarations in all configurations tested by the delta debugging algorithm. Unfortunately, recognising lines with import declarations is specific to the programming language Haskell.

4.3 The Build Tool

When measuring the top 100 Haskell programs on GitHub, we found that they all use Cabal[8] for packaging and building. Therefore our type error debugger has a flag to call the build tool cabal instead of the Glasgow Haskell compiler for testing. When cabal is used, the user has to state the target program instead of the ill-typed module.

5 A Framework for Type Error Debugging

In data science using model metrics such as Accuracy, Precision, and Recall are an accepted standard [22, 16]. Yet within type error debugging evaluations only Recall is deemed important. We propose the following as a framework for future type error debugging evaluations.

5.1 The Metrics

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

$$Recall = \frac{TP}{TP + FN} = \frac{R_E}{E} \tag{1}$$

For us it measures the number of errors that are reported correctly compared to the number of errors within the source code. This metric is most used in type error debugging evaluations as it shows if a debugger can successfully discover the correct number of type errors within an ill-typed program.

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

Shorthand	Longhand	Equivalents
Standard Data Science		
TP	True Positive	
TN	True Negative	
FP	False Positive	
FN	False Negative	
Our Terminology		
R_L	Reported Lines (number of lines returned)	TP + FP
R_E	Reported Errors(number of correct errors)	TP
L	Lines of code (Total Source Code)	TN+TP+FN+FP
E	Errors (number of Errors in the code)	TP+FN

Table 3. Terminology

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

$$Precision = \frac{TP}{TP + FP} = \frac{R_E}{R_L} \tag{2}$$

Equal to the number of correct lines of code reported by the debugger compared to the total number of lines returned. The return of a correct location as one single line versus returning a correct location within several lines.

Accuracy tells us how close a measure is to another measure. Applied within our domain it unfortunately means we receive a high number of True Negatives, number of lines correctly excluded, and thus gives us an unfair balance.

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} = \frac{R_E}{L} \tag{3}$$

Though problematic in our area, the metric does show us how much of the source code contains type errors which in turn can be useful when combined with other data. However for the type error debugging domain we employ an alternative to accuracy, the F1 score.

F1 Score is the total accuracy of our tests; applied solely to have a balance between *precision* and *recall*. As there is an imbalance of data with type error debugging, F1 is crucial in showing the true results of evaluations.

$$F1 = 2\frac{Precision \cdot Recall}{Precision + Recall} = 2\frac{R_E}{E + R_L} \tag{4}$$

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

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

Table 4. Lines of Code per Module with Associated Errors

6 Evaluating our Method

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

We compare our presented debugger, Elucidate, with Gramarye2019(G19) using our framework from Section 5. Gramarye2019 is a modified version of our previous debugger Gramarye; As the latter did not support modular programs we added the functionality whilst keeping Delta Debugging free of the Moiety pre-processing [15].

For this evaluation we answer the following questions:

1. Does our debugger, Elucidate (E20), show positive results compared to Gramarye2019 (G19) in the categories of Recall, Precision, and Accuracy?
2. Does E20 reduce the number of Unresolved results compared to G19?
3. Does our pre-processing effect the time of Delta Debugging and that of which is felt by the user?

6.1 Applying the Framework

In Figure 1 we present the data from the evaluation. The x axis represents each individual error as in Table 4 with the y axis showing the results from the metrics in percentages. Recall shows if a type error has been correctly located (100%) or not (0%). Precision favours a higher percentage; 100% would represent a singular correct location being returned whereas a lower percentage would show a correct result within a set of incorrect locations. Accuracy in our graph shows what percentage of code contains an error; this metric for our evaluation is useful only to show that as our ill-typed program gets larger our chances of reporting a correct location diminishes and so instead we shall use the F1 score. As an

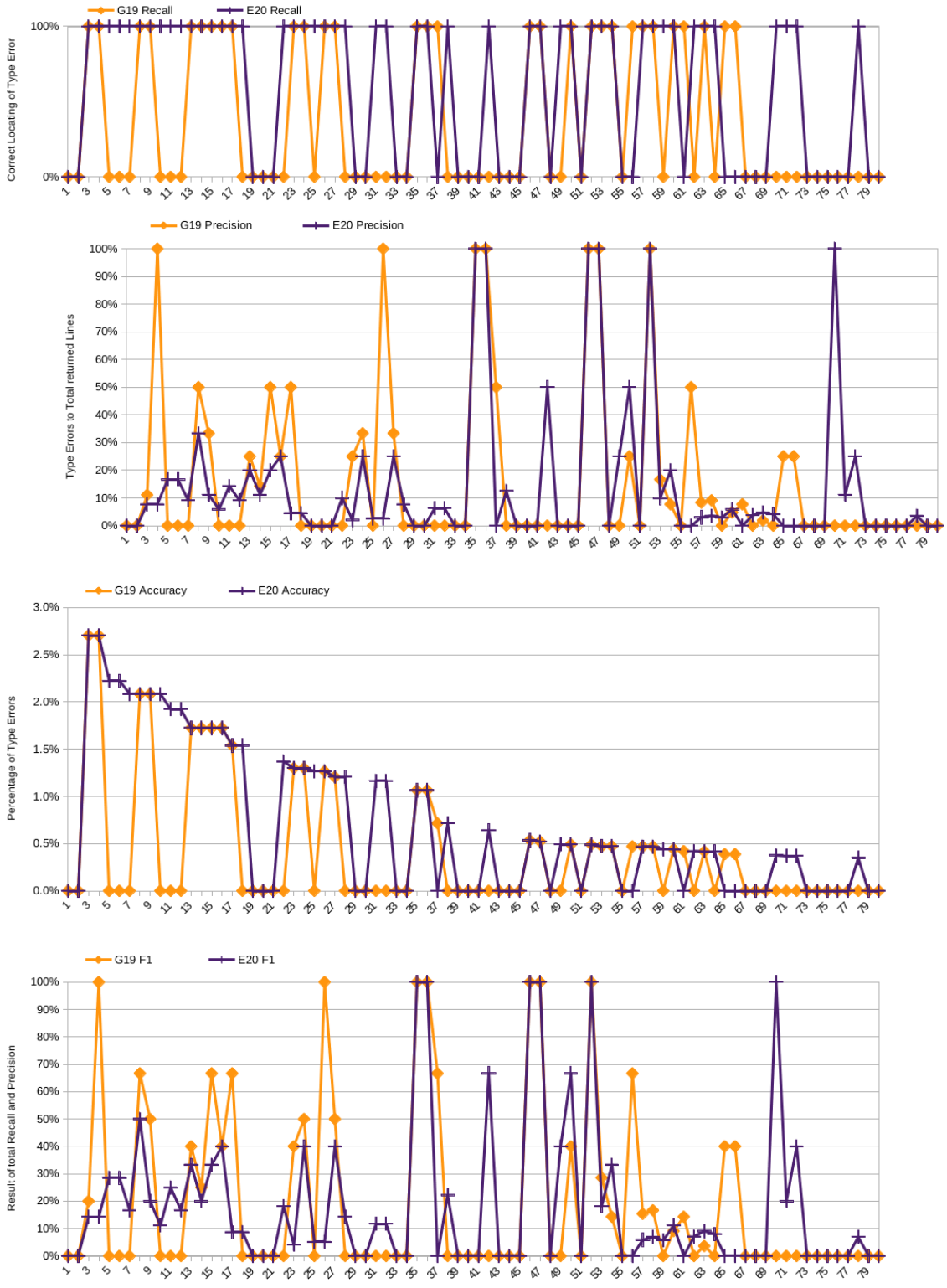

Fig. 1. Recall, Precision, Accuracy, and F1

alternative to Accuracy the F1 score fits our domain well as a balance between our two important metrics Recall and Precision; with F1 we prefer a higher percentage being returned.

Question: Does our debugger, Elucidate (E20), show positive results compared to Gramarye2019 (G19) in the categories of Recall, Precision, and Accuracy?

Recall shows us if the debugger has returned the correct type error specified. As we only have a single type error per benchmark our result is binary. Elucidate (E20) correctly locates 59% of the type errors compared to Gramarye2019 (G19) who returns fewer correct type errors at 38%. This rise in correct results from Elucidate is directly linked to the pre-processing of the source code. Firstly, as we are passing a new configuration to Delta Debugging, setting out how to split our lines, we have the chance to generate an alternative pathway of modifications leading to different results from our Blackbox compiler; as the path that the debugger takes relies on these outcomes an alternative result can happen. Secondly as our method does not allow the splitting of lines with reliances we inheritable gain the bias of returning a greater set of locations and so increasing our chances of success. This bias is countered with the the precision metric.

If we return 100 results as suggested locations and the ill-typed program only has 100 lines we can say that this would not make a suitable solution. The recall metric lets us do this without any hindrance so we need to combine it with another, precision. In the second graph of Figure 1 we see that indeed Gramarye2019 is more precise than Elucidate; however overall this only accounts for a difference of 1.78 percentage points meaning we need to invoke the F1 score for an accurate reading.

F1 blends our two graphs, recall and precision, to form a true overview of the results. With this set of benchmarks we receive a 1% difference between the presented debugger Elucidate and the previous Gramarye, with the latter providing a higher F1 score. This outcome is not surprising; the precision of Elucidate is hampered by the Moiety algorithm. However, we do not see this as a negative; it was our aim to avoid causing Unresolveds and as such these are the most precise result we can currently return for the specific benchmarks utilised in this evaluation.

6.2 Reduction of the Unresolved Outcomes

Question : Does E20 reduce the number of Unresolved outcomes compared to G19?

In Figure 6.2 we can see the number of unresolved outcomes for each benchmark. For ease of reading we have capped Figure 6.2 at a maximum of 280, however it is worth noting that Gramarye2019 returned four results higher than this at {60,395}, {80,504}, {63,1436}, and {64,1436} respectively.

On average there are 16 unresolveds per module from Elucidate20 compared to Gramarye19 at 88; meaning a reduction of 72 calls to the blackbox compiler.

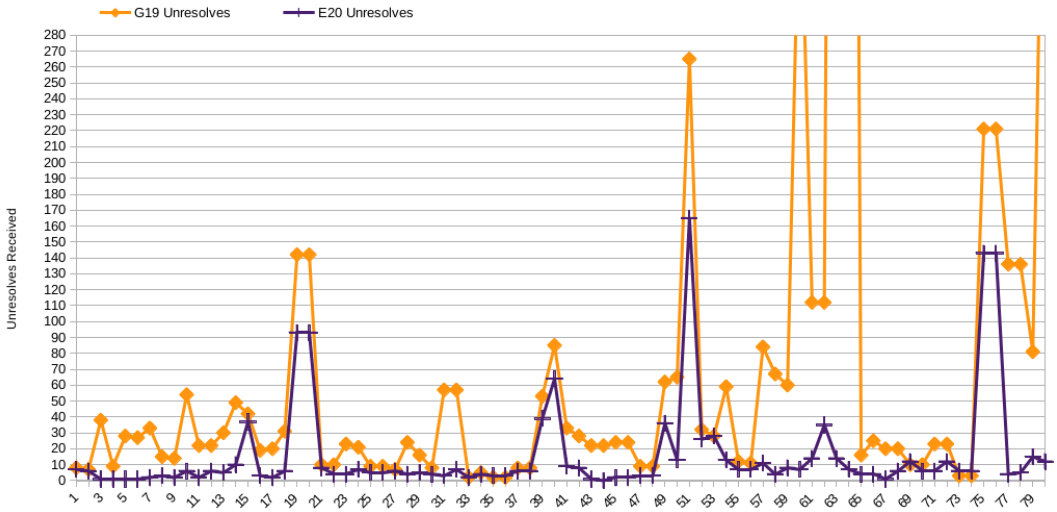

Fig. 2. Unresolved outcomes

The importance of reducing calls is seen in benchmark 64; here Gramarye19 has 1436 Unresolveds and takes over 70 minutes to return a type error location whereas Elucidate receives only 7 Unresolveds and the time taken drops to just over 9 minutes, a difference of around 61 minutes.

Elucidate20 has an significant impact on the reduction of unresolveds however as with recall and precision in Section 6.1 this outcome needs to combined with other metrics in our case the Delta Debugging Run-Time and the User Time.

6.3 The Run-Time Speeds

Question : Does our pre-processing effect the time of Delta Debugging and that of which is felt by the user?

With the unresolveds minimised we hypothesis that the time taken by Delta Debugging should reduce. However, as our pre-processing is linear, based on lines of code in the program, an overall run-time increase is also expect. This meant we split our evaluation into two measurements; time taken for Delta Debugging (precluding pre-processing) and overall run-time (including pre-processing).

In Figure 6.3 we show the outcome of just the run-time of Delta Debugging in seconds. As in Section 6.2 we have again modified the figure so that we can see the data more clearly dropping off the most extreme results of Gramarye19 at {60,1295}, {80,1482}, {64,4201},and {63,4299} .

On average Gramarye19 took 285 seconds to run the Delta Debugging algorithm, 219 seconds more than Elucidate20 at 66 seconds showing a clear link between total unresolveds received and the time taken to locate a type error. However, when running a debugger the user experiences the entire process not

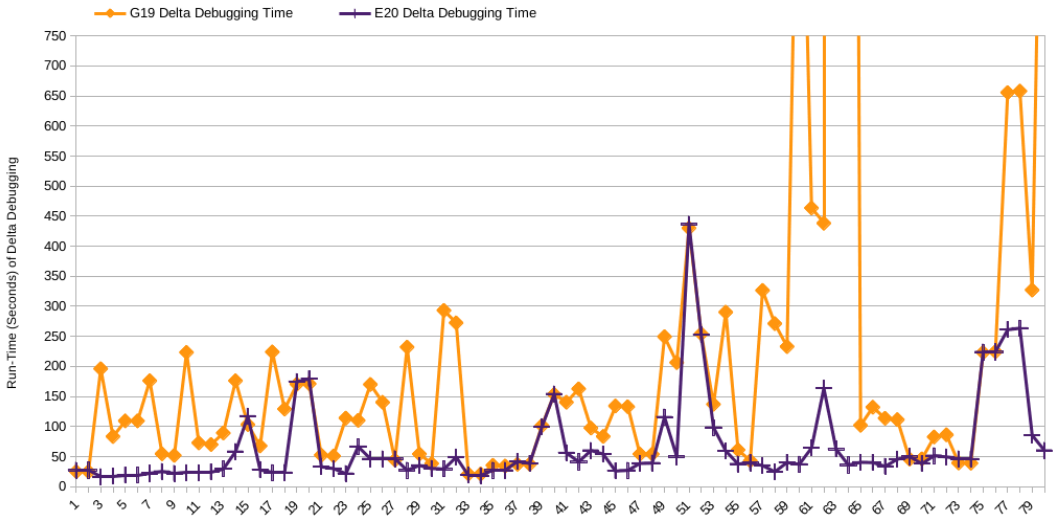

Fig. 3. Delta Debugging Run-time

just the algorithm doing the locating and so we also need to take into consideration the total run-time cost of which we can see in Figure 6.3.

Between the two solutions, Gramarye19 with its lack of Moiety algorithm takes on average 303 seconds compared to Elucidate at 419 seconds, however in some cases, namely benchmarks 63 and 64, we can see that pre-processing does improve overall debugging time. These fluctuations are due to the chances of receiving unresolveds, the higher the chance the more beneficial it is to run Elucidate20 over Gramarye19. Unfortunately, we currently do not have a way of knowing this information prior to running Delta Debugging and so it would be the case of running one version at least once to decide the optimum debugger.

7 Related Work

Type Error Debugging research has a long and fruitful history starting in the eighties[21]. It spans many solutions in a variety of categories each specialising on their own core ideas [13, 6, 4, 20, 17, 14, 3, 9, 1, 11, 26, 18]. However, these solutions do not attempt to directly aim at Real-World programs with type errors instead evaluating success on small programs typically of the size that first-time programmers would produce. One general method of debugging that has been applied to a large 178,000 lines Real-World program is Delta Debugging. Defined by Zeller in 1999 it comes in two forms Simplifying and Isolating of which he applied to a general debugging domain avoiding specifying in certain areas of errors[5, 23–25]. In our previous paper we applied Zellers work specifically to type errors in functional languages employing the compiler as a blackbox [15]. A Blackbox Compiler is different to other Blackbox solutions mentioned in prior

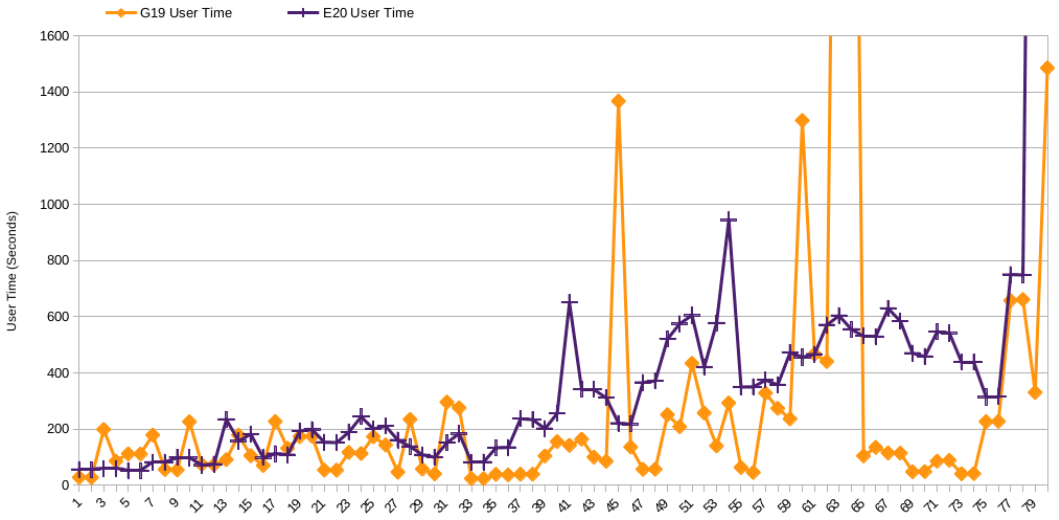

Fig. 4. User Time

literature(Blackbox Type Checkers, Blackbox Type Inference) as it treats the entire compiler as an external entity rather than a component of it [13, 19, 8]. This method of only taking external cues, such as whether a program is ill or well-typed, avoids users having to patch or download a specific compiler to explicitly improve type error discovery. Though combining a Blackbox Compiler and Isolating Delta Debugging to the domain of type errors returned positive results reducing Unresolveds was seen to be beneficial future work and one option for doing so was the modification of the Delta Debugging configuration. Generating Configurations to avoid invalid inputs for Delta Debugging is not new [10, 12]. The closest to our work observes that modifying lines of source code can and will generate broken code that will still need to be sent to the test function causing debugging times to increase [7]. In *Binary Reduction of Dependency Graphs* the authors aim to reduce these invalid inputs by using dependency graphs to map the smallest set of elements that are invalid without each other, reference's to other classes, in Java bytecode. Their method is solely based on the simplifying delta debugging in contrast to isolating of which we feel along with the blackbox compiler gives us a unique aspect in type error debugging.

8 Conclusion and Future Work

We presented a method of combining Isolating Delta Debugging and a blackbox compiler to locate type errors in real-world programs. Real-world programs when applying Delta Debugging to locate type errors are susceptible to parse errors, particularly in functional programming languages 'parse errors on input'. We introduce an algorithm that pre-processes an ill-typed program to eliminate these

parse errors. Our pre-processing algorithm, Moiety, groups lines of source code that cannot be separated without causing a parse error. These moieties are then used as a configuration for Delta Debugging to reduce the unresolveds caused by parse errors which in turn is linked to the time taken in debugging real-world programs.

To test the success of our solution on locating type errors in real-world programs we introduced a new set of 80 benchmarks and a framework based on Data Science standards. One metric within the framework, recall, is the most commonly used in our domain and showed a positive result for Elucidate20 with a 21 percentage points increase in locating a type error compared to Gramarye19. However, when the entire framework is applied it shows that the difference between Elucidate20 and Gramarye19 drops to just 1 percentage point. This significant difference in results shows that just applying the traditional recall metric is not satisfactory for evaluations in this field and the application of the framework on future type error debugging solutions is needed to be able to report clearer results, and comparisons between solutions. To measure our methods effect on the reduction of unresolveds we record the amount returned by each debugger. Elucidate20 on average returned 72 fewer unresolveds per benchmark reducing the time taken for Delta Debugging to run by an average of 216 seconds, however, the overall time the user experiences did increase by 100 seconds compared to Gramarye19.

For future work an increase of the categories of parse errors we treat with the pre-processing along with adding other errors such as *Variables not in Scope* is a concrete direction; as the Moeity algorithm already works though individual lines adding these will not increase the overheads and has the possibly of reducing the time Delta Debugging takes further down. It is also clear that though pre-processing speeds up Delta Debugging it also slows the overall run-time of the debugger. Reducing the time it takes to generate a list of moieties would be extremely beneficial. Lastly, though we applied our method to Haskell programs, our debugger is nearly language agnostic. Delta Debugging and the Moiety algorithm are not specific for the programming language, allowing for a reasonable modification towards an agnostic debugger in the future.

References

1. Bernstein, K.L., Stark, E.W.: Debugging type errors. Tech. rep. (November 1995)
2. Chen, S., Erwig, M.: Counter-factual typing for debugging type errors. In: POPL 2014. pp. 583–594. ACM (2014)
3. Chen, S., Erwig, M.: Guided type debugging. In: Functional and Logic Programming - 12th International Symposium. pp. 35–51 (2014)
4. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: ICFP 2001. pp. 193–204 (2001)
5. Cleve, H., Zeller, A.: Locating causes of program failures. In: 27th International Conference on Software Engineering. pp. 342–351 (2005)
6. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. Sci. Comput. Program. **50**(1-3), 189–224 (2004)

7. Kalhauge, C.G., Palsberg, J.: Binary reduction of dependency graphs. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 556–566. ACM (2019)
8. Lerner, B.S., Grossman, D., Chambers, C.: Seminal: searching for ML type-error messages. In: Proceedings of the ACM Workshop on ML. pp. 63–73 (2006)
9. McAdam, B.J.: On the unification of substitutions in type inference. Lecture notes in computer science **1595**, 137–152 (1999)
10. Misherghi, G., Su, Z.: Hdd: Hierarchical delta debugging. In: ICSE '06. pp. 142–151. ACM (2006)
11. Rahli, V., Wells, J.B., Pirie, J., Kamareddine, F.: Skalpel: A type error slicer for standard ML. Electr. Notes Theor. Comput. Sci. **312**, 197–213 (2015)
12. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI 2012. pp. 335–346. ACM (2012)
13. Schilling, T.: Constraint-free type error slicing. In: Trends in Functional Programming, 12th International Symposium. pp. 1–16 (2011)
14. Seidel, E.L., Jhala, R., Weimer, W.: Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In: ICFP 2016. pp. 228–242. ACM (2016)
15. Sharrad, J., Chitil, O., Wang, M.: Delta debugging type errors with a blackbox compiler. In: IFL 2018. pp. 13–24. ACM (2018)
16. Shung, K.P.: Accuracy, precision, recall or f1? (November 2019), https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9
17. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 72–83 (2003)
18. Tip, F., Dinesh, T.B.: A slicing-based approach for locating type errors. ACM Trans. Softw. Eng. Methodol. **10**(1), 5–55 (2001)
19. Tsushima, K., Asai, K.: An embedded type debugger. In: IFL 2012. pp. 190–206 (2012)
20. Tsushima, K., Chitil, O.: Enumerating counter-factual type error messages with an existing type checker. In: PPL2014 (2014)
21. Wand, M.: Finding the source of type errors. In: POPL 1986. pp. 38–43. ACM (1986)
22. Witten, I., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005)
23. Zeller, A.: Yesterday, my program worked. today, it does not. why? In: Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference. pp. 253–267 (1999)
24. Zeller, A.: Why Programs Fail Guide to Systematic Debugging, 2nd Edition. Academic Press (2009)
25. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. **28**(2), 183–200 (2002)
26. Zhang, D., Myers, A.C., Vytiniotis, D., Peyton Jones, S.L.: Diagnosing type errors with class. In: PLDI 2015. pp. 12–21. ACM (2015)