Delta Debugging Type Errors with a Blackbox Compiler

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

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

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

ABSTRACT

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

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Compilers for Haskell, OCaml and many other statically typed functional programming languages produce type error messages that can be lengthy, confusing and misleading, causing the programmer hours of frustration during debugging. One role of these messages is to tell the programmer the location of a type error within the ill-typed program. Although there has been over thirty years of research [8, 22] on how to improve the way we locate type conflicts and present them to the programmer, type error messages can be misleading. We can trace the cause of inaccurate type error location to an advanced feature of functional languages: type inference.

A typical Haskell or OCaml program contains only little type information: definitions of data types, some type signatures for top-level functions and possibly a few more type annotations. Type inference works by generating constraints for the type of every expression in the program and solving these constraints. An ill-typed program is just a program with type constraints that have no solution. Because the type checker cannot know which program parts and thus constraints are correct, that is, agree with the programmer's intentions, it may start solving incorrect constraints and therefore assume wrong types early on. Eventually, the type checker notes a type conflict when considering a constraint (generated by an expression of the program) that is correct.

1.1 Variations of an Ill-Typed Programs

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

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

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

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

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

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

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

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

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

GHC identifies the type error location correctly:

IFL'18, August 2019, Lowell, MA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/0000001.0000001

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

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

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

GHC wrongly locates the cause in line 2 again:

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

In summary we see that the order in which type constraints are solved determine the reported type error location. There is no fixed order to always obtain the right type error location and requiring type annotations in the program does not help.

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

1.2 Our Method

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

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

Listing 1: Result of our type error location method

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

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

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

Once implemented in our tool Gramarye, we can apply our method to any ill-typed program, no matter how many type errors it contains, to locate one type error. Once our approach has the correct location, the programmer can fix it and reuse the tool to find further type errors.

Our tool Gramarye works on Haskell programs and uses the Glasgow Haskell Compiler as a black box. We evaluated Gramarye against the Glasgow Haskell Compiler using thirty programs containing single type errors and eight hundred and seventy programs generated to include two type errors.

Our paper makes the following contributions:

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

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

2 AN ILLUSTRATION OF OUR METHOD

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

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

From this program, we obtain two programs that the *Isolating Delta Debugging* algorithm will work with. One is a lower bound, and the other one is an upper bound with respect to the type error. The empty program is definitely a lower bound and the ill-typed program itself an upper bound:

Listing 2: lower bound program, step 1

```
1
2
3
```

Listing 3: upper bound program, step 1

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

Now we move a single line from the upper bound program to the lower bound program. We can pick any line, for example, line 3:

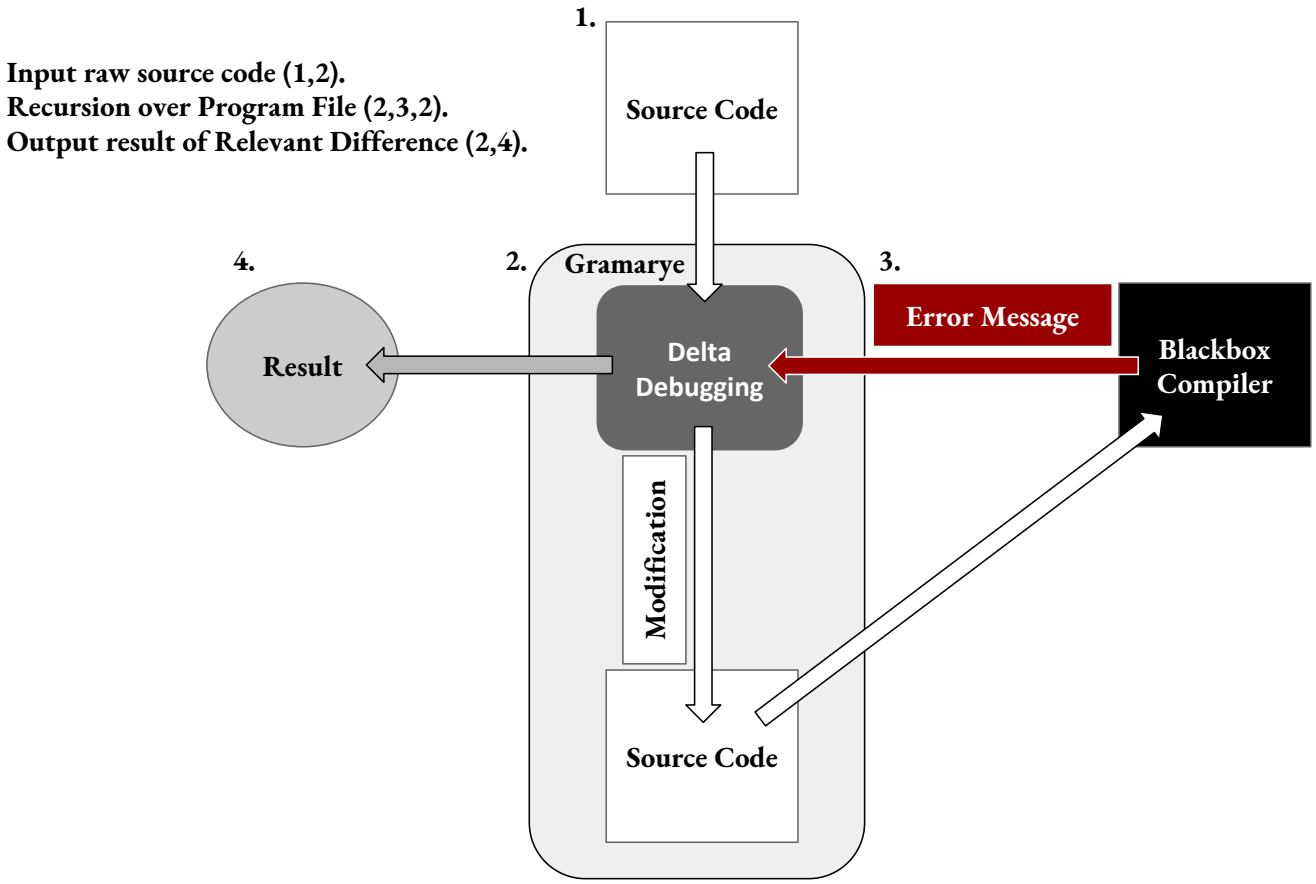

Figure 1: The Gramarye Framework

Listing 4: modified lower bound program, step 1

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

Listing 5: modified upper bound program, step 1

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

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

- Modified lower bound program, step 1: non-type error.
- Modified upper bound program, step 1: ill-typed.

The lower bound program is not a syntactically valid Haskell program; the compiler yields a parse error. So note that our black box compiler yields one of three possible results:

(1) non-type error
(2) ill-typed
(3) well-typed; compilation was successful

A compilation result *non-type error* is not useful for locating a type error, but each of the other two possible results are. Our modified upper bound program is smaller than our original upper bound program. We now know that the modified variant is ill-typed too, so we can replace our upper bound for the next step:

Listing 6: lower bound program, step 2

```
1
2
3
```

Listing 7: upper bound program, step 2

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

The algorithm now repeats: Again we move a single line from the upper bound program to the lower bound program. Let us pick line 2:

Listing 8: modified lower bound program, step 2

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

Listing 9: modified upper bound program, step 2

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

Again we send these two programs to the black box compiler for type checking:

- Modified lower bound program, step 2: well-typed.
- Modified upper bound program, step 2: well-typed.

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

Listing 10: lower bound program, step 3

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

Listing 11: upper bound program, step 3

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

The upper and lower bound differ by only a single line, and hence our algorithm terminates.

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

Our method is non-deterministic. Often different choices lead to the same final result, but not always. Zeller argues that the non-determinism still does not matter and that one result provides insightful debugging information to the programmer [25].

The algorithm is based on an ordering of programs, where a program is just a sequence of strings. A program P_1 is less or equal a program P_2 if they have the same number of lines and for every line, the line content is either the same for both programs, or the line is empty in P_1. All programs that we consider are between the lower and upper bound programs that we start with. The final upper and lower bound have minimal distance, that is, they either differ by just one line or programs between them yield a non-type error at compilation (and thus are not syntactically valid programs).

In this example, in each step, we moved only a single line from upper bound to lower bound. For programs with hundreds of lines, this simple approach would be expensive in time due to, too many programs needing consideration. Hence we use the full Isolating Delta Debugging algorithm which starts with moving either the first or second half of the program from upper to lower bound. If both modified programs yield a non-type error, then we change the granularity of modifications from moving half the program to moving a quarter of the program. In general, every time both modified programs yield a non-type error, we half the size of our modifications. This change of granularity can continue until only a single line is modified.

3 IMPLEMENTATION

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

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

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

3.1 Delta Debugging

The first component of our tool is *Delta Debugging*, a method formalised by Zeller [6, 24–26], that can be described as a systematic replication of the scientific approach of *Hypothesis-Test-Result* [25]. When programmers debug they first use the error message to narrow the cause (Hypothesis), then modify the source code and recompile (Test), and lastly use the outcome of the recompilation (Result) to see if the modifications were successful. To implement the scientific approach Zeller splits his Delta Debugging method into two algorithms he refers to as Simplifying and Isolating [25].

3.1.1 *Simplifying Delta Debugging.*

Simplifying Delta Debugging has similarities with program slicing [7, 16]. The algorithm tries to assemble a minimal set of source code, returning this set to the programmer as the smallest working version of their program. To complete the generation of the minimal set the algorithm removes sections of a broken program until it no longer contains an error. To make sure the set is truly minimal a secondary working program is necessary. The secondary program can either be empty(containing no source code), or a previous working version of the initial program. A minimal set can be declared when the broken program is as close to the working program as possible without the removal of the error. The minimal set of source code allows us to surmise that the parts of the program left must be the cause of our error. However, the Simplifying Delta Debugging algorithm has the same flaws as program slicing and, can return large minimal sets. The second Delta Debugging algorithm, *Isolating*, aims to reduce the size of the sets even further.

3.1.2 *Isolating Delta Debugging.*

Isolating Delta Debugging incorporates the Simplifying algorithm to generate a minimal set of source code that contains an error. As well as employing the use of the simplifying algorithm, the isolating algorithm produces its own minimal set of source code; one that does not hold an error. The isolating minimal set is created by taking the working program and, adding sections until the program reports an error. The aspect of having two minimal sets, one that

contains the error and one that does not, is our reason for choosing the latter algorithm over the former. Focusing on the output of both minimal sets, we should receive a smaller result.

The Isolating Delta Debugging algorithm is composed of two parts; granularity and, 'program replacement'. Granularity has the task of supplying which lines of the source code to add and remove from our programs to generate the two minimal sets. Initially, granularity is set at two and, applied in combination with the length of the program. The initial setting of granularity means it resembles a binary chop algorithm and when applied divides our program in half. After the initial application granularity is increased, decreased or remains static as the Delta Debugging algorithm iterates. We present a brief demonstration of granularity works using a generic four-line program that contains a type error below;

We shall show our ill-typed program as a set of line numbers gathered from the original broken program and converted into a list format;

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

Our granularity currently equals 2. The initial divide splits our list in half;

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

Isolating Delta Debugging checks the leading half first. These are the line numbers we shall modify in our program.

```
[1,2]
```

The program is type checked with a blackbox compiler. The result is that there is no type error so, we check the second half;

```
[3,4]
```

Again, type checking returns a well-typed result. We split the granularity and set it to 1, dividing our initial list into to chunks of one;

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

Using the blackbox compiler we type check each chunk starting from the leading list. We locate the type errors position in line 4 and, return;

```
   Line [4] contains a type error.
```

The increasing and decreasing of the granularity depends on the result category. We return a category when checking the success of the program modifications against our blackbox compiler; which we explain in more detail in section 3.2. Zeller does not use a blackbox compiler and as such assumes the use of a 'testing function' [24] to place the results into the following categories;

- The test succeeds (PASS, ✓)
- The test has an error (FAIL, ×)
- The test is undetermined (UNRESOLVED, ?)

Our tool, on the other hand, categorises them slightly differently. Restricting the categories further due to the nature of only wanting to discover the position of type errors;

- The test succeeds ('Well-Typed', ✓)
- The test returns a *type* error ('Ill-Typed', ×)
- The test returns *any other* error ('Unclassified', ?)

Program replacement also using these categories to determine the path the algorithm takes after each iteration. The program files that the Isolating Delta Debugging algorithm use are fluid. In our case, they start with the 'Ill' and 'Well-Typed' programs of which we convert to our upper and lower bound programs. As we iterate over the algorithm, the upper and lower bound programs replace our initial programs depending on the result of the modifications.

ALGORITHM 1: Granularity and 'Program Replacement'

if *'upper bound program' == Ill-Typed(×) && granularity == 2* **then**
  Replace 'Ill-Typed' result program with current 'upper bound program'.
  Granularity == 2.
else if *'upper bound program' == Well-Typed(✓)* **then**
  Replace 'Well-Typed' result program with current 'upper bound program'.
  Granularity == 2.
else if *'lower bound program' == Ill-Typed(×)* **then**
  Replace 'Ill-Typed' result program with current 'lower bound program'.
  Granularity == 2.
else if *'upper bound program' == Ill-Typed(×)* **then**
  Replace 'Ill-Typed' result program with current 'upper bound program'.
  Granularity == max (granularity - 1) or 2.
else if *'lower bound program' == Well-Typed(✓)* **then**
  Replace 'Well-Typed' result program with current 'lower bound program'.
  Granularity == max (granularity - 1) or 2.
else
  Try other half.

Keeping our terminology from the example program (section 2) the changes seen in algorithm 1 are completed depending on the result of type checking with the blackbox compiler.

3.2 A Blackbox Compiler

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

We employ our blackbox compiler by using it as a type checker. During each iteration of the Isolating Delta Debugging algorithm, we determine the status of our upper and lower bound programs as described in section 2. When using the blackbox compiler, our tool receives the same output a programmer would when they are using GHC. Though the result of compiling with GHC gives a message that includes many factors, we are only interested in if our programs are well-typed, using this information to attach the categories we discuss in section 3.1. Using our example from section 2, type checking both programs would give us the following messages and applied categories;

Listing 12: Ill-Typed Initial Error Message

```
Occurs check: cannot construct the infinite type: a ~ [a]
....
Category: FAIL
```

Listing 13: Well-Typed Initial Error Message

```
0
Category: PASS
```

The Isolating Delta Debugging algorithm receives the attached categories and uses them to determine which path to apply as presented in algorithm 1. Depending on the route taken, we modify the source code of our programs in different ways, and again send them to the blackbox compiler for further type checking before reiterating over the whole method again. Where our programs source code is modified is automated by the Isolating Delta Debugging algorithm but, the idea of directly changing the raw code is solely inspired by how programmers manually debug.

3.3 Source Code Manipulation

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

Our overall concern is the inaccurate reporting of the line number a type error occurred on, and as such, our tool works on a line-by-line based approach. As observed in section 2, we do this by adding and removing lines of source code and, on completion of the algorithm we are left with two programs. One program has all ill-typed source code removed and, the other only contains well-typed code. As we have directly modified the source code to achieve these two programs we can use them to find the line number of where our type error appears by calculating the difference between the two.

3.4 Processing the Results

The idea is if one program is well-typed and the other ill-typed the source of the type error lays within the variation of the two; the relevant difference [25]. Processing the result is inherently uncomplicated. After the Isolating Delta Debugging algorithm has completed, two programs are left. The two final programs are used to create two lists generated by adding the line number of each empty line in the program. If a line number does not make an appearance in both of these generated lists we report it as a relevant difference. Working on whole lines of code in which we can report line numbers, also means we can easily evaluate how successfully we are in locating type errors.

4 EVALUATION

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

We chose to evaluate our method against a benchmark of programs specially engineered to contain type errors. The programs collated by Chen and Erwig [3] were used to assess their Counter-Factual approach to type error debugging. In all, there are one hundred and twenty-one programs in the CE benchmark, but not all had what the Chen and Erwig called the 'oracle', the foresight of where the type error lay. As we needed to know the correct location of where the type error occurred to evaluate accurately, we cut all programs that did not specify the exact cause. To make our evaluation more compact, programs that were ill-typed in similar ways were also removed, reducing our set of test programs to thirty. However, as we also wanted to see if our method could report multiple type errors we took these thirty test programs and generated a further eight hundred and seventy programs to use in evaluation.

Our evaluation answers the following questions;

(1) When applying our method to Haskell source code that contains a single type error; Do we show improvement in locating the errors compared to the Glasgow Haskell Compiler? (Section 4.1)
(2) If we add multiple(two) type errors in our Haskell source code; Do we show improvement in locating these errors compared to the Glasgow Haskell Compiler? (Section 4.2)
(3) Does our method return a smaller set of type error locations? Specifically, a single precise line number of where the type error occurred. (Section 4.3)

Answering these questions involved creating a series of tests. To evaluate these tests we chose to compare our approach against GHC 8.2.2. We are using GHC as a blackbox compiler within our own tool, but as we use it solely as a type checker we do not have any knowledge of the line numbers it reports and, thus it has no interference with our evaluation. GHC and our tool take the CE benchmarks, and type checks each one; this results in a set of suggested line numbers where the cause of the type error could occur. To judge the success of locating the type error in the tests we have chosen to use the same criteria as Wand [22]. Wand states that even if we get multiple locations returned, the method is

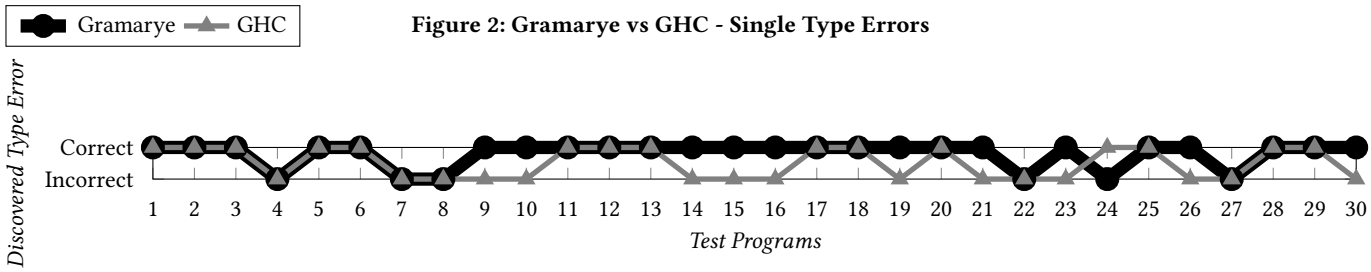

Figure 2: Gramarye vs GHC - Single Type Errors

classed as a success if the exact location of the type error is within these. As, both our tool and GHC can report multiple line numbers for one type error; we use Wands criteria to allow us to take into consideration all line numbers returned and, not just the first.

4.1 Singular Type Error Evaluation

(1) When applying our method to Haskell source code that contains a single type error; Do we show improvement in locating the errors compared to the Glasgow Haskell Compiler?

The first set of test programs contain one single type error; if the line number reported matched the 'oracle' response, then our result was accurate. In figure 2, we can see an overview of the outcome. The graph shows all thirty ill-typed programs and whether Gramarye and GHC correctly discovered the position of the type error. The results of our approach were positive. Out of the thirty ill-typed programs we accurately located 24 (80%) of the type errors, compared to 15(50%) from GHC.

In some cases, multiple line numbers were returned but still contained the correct errored line. We found the primary cause of multiple line numbers was due to statements that relied on each other or line breaks. Examples of this are If-Else or Let-In statements or lines that wrap around; the latter of which we present below;

Listing 14: Layout over two rows

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

In this example, our tool correctly identifies the line number even though we are returned two to choose from but, this was not the same for GHC, who suggests the first line in the above program as causing the issue. These issues caused by the programmer's layout decisions are one direction for future work.

In all our method using the initial evaluation criteria, has a 31% success rate over locating type errors in Haskell source code than GHC, but, when programming we can often end up with multiple errors in our programs. A second evaluation of programs containing more than one type error would be advantageous.

4.2 Multiple Type Errors Evaluation

(2) If we add multiple(two) type errors in our Haskell source code; Do we show improvement in locating these errors compared to the Glasgow Haskell Compiler?

In the context of our evaluation, testing multiple type errors is represented by having more than one self-contained error within an ill-typed program. Self-contained type errors within a program mean we have two separate functions that do not interact with each other, with both functions contain a single type error. In listing 15, the first function has an error on line 2 and the second function on line 6, but neither type error affects the other;

Listing 15: Multiple Type Error Example

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

Listing 15, is just one of the programs we generated that contains multiple type errors. We created these by merging the CE benchmark programs. Each set of programs includes the original source code with the addition of another CE program attached to the bottom. In all, we generated eight hundred and seventy new ill-typed programs to test. The success criteria for reporting an accurate discovery of the position of a type error in an ill-typed program that contains multiple errors is similar to what we used for singular errors. The only difference being, that though we have two errors per program we only need one error to be reported to deem a success.

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

With this particular combination of CE benchmark programs, we can see that Gramarye finds 50% more type error positions than GHC. However, this is not always the case. Table 2, provides the total results for all of our combination of programs. Column one lists the base program, and the last two columns show the percentage of how accurate our tool and GHC were at locating type errors.

In total, we can see that Gramarye finds 3% fewer type errors in our multiple programs than GHC, this is not surprising. The Isolating Delta Debugging algorithm restricts Gramarye to always locating just one type error, the first it has come across. Once it

has found this error, the algorithm assumes the job is complete and does not check any further. Currently, the programmer has to repeatedly use the tool after each implemented fix, working on each type error separately. We feel the removal of this limitation, would close the gap between Gramarye and GHC considerably, however, being restricted to working on one error at a time could also prove to be beneficial. Our evaluation of allowing a return of only a precise line shows this is the case.

Table 1: Testing a program with two type error.

Original Program	Merged Program	Gramarye	GHC
15	1	✓	✓
15	2	✓	✓
15	3	✓	×
15	4	✓	×
15	5	✓	✓
15	6	✓	✓
15	7	✓	×
15	8	×	×
15	9	✓	×
15	10	✓	×
15	11	✓	✓
15	12	✓	✓
15	13	✓	✓
15	14	✓	×
15	16	✓	×
15	17	✓	✓
15	18	✓	✓
15	19	✓	×
15	20	✓	✓
15	21	✓	×
15	22	×	×
15	23	✓	×
15	24	✓	✓
15	25	✓	×
15	26	✓	×
15	27	×	×
15	28	✓	✓
15	29	✓	✓
15	30	✓	×
	Total	89.66%	44.83%

4.3 Precise Type Error Evaluation

(3) Does our method return a smaller set of type error locations? Specifically, a single precise line number of where the type error occurred.

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

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

Program	Gramarye	GHC
1	72.41%	96.55%
2	65.52%	96.55%
3	68.97%	96.55%
4	75.86%	51.72%
5	68.97%	96.55%
6	72.41%	93.19%
7	65.52%	48.28%
8	62.07%	48.28%
9	75.86%	55.17%
10	58.62%	93.10%
11	65.52%	93.10%
12	68.97%	96.55%
13	68.97%	96.55%
14	75.86%	00.00%
15	89.66%	44.83%
16	65.52%	51.72%
17	68.97%	96.55%
18	65.52%	93.10%
19	82.76%	51.72%
20	68.97%	96.55%
21	82.76%	44.83%
22	31.03%	48.28%
23	72.41%	51.72%
24	55.17%	89.66%
25	68.97%	96.55%
26	72.41%	55.17%
27	65.52%	51.72%
28	65.52%	89.66%
29	65.52%	96.55%
30	65.52%	51.72%
Total	68.39%	71.03%

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

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

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

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

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

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

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

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

Program	Gramarye	GHC
1	✓	✓
2	×	✓
3	✓	×
4	×	×
5	×	✓
6	×	✓
7	×	×
8	×	×
9	✓	×
10	✓	×
11	×	✓
12	✓	✓
13	✓	✓
14	×	×
15	×	×
16	✓	×
17	✓	×
18	✓	✓
19	×	×
20	✓	✓
21	✓	×
22	×	×
23	✓	×
24	×	×
25	✓	✓
26	✓	×
27	×	×
28	✓	✓
29	×	✓
30	✓	×
Total	53.33%	40.00%

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

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

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

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

All other results, even those that include the correct location, are recorded as failing the "precise success" criteria of discovering type errors. Table 4 represents the test programs that contained two type errors. The name of the original program along with the percentage of type error locations deemed to be a "precise success" are shown.

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

Program	Gramarye	GHC
1	48.28%	37.93%
2	44.83%	34.48%
3	48.28%	6.90%
4	51.72%	00.00%
5	44.83%	41.38%
6	37.93%	34.48%
7	44.83%	00.00%
8	41.38%	00.00%
9	51.72%	00.00%
10	48.28%	00.00%
11	48.28%	37.93%
12	48.28%	48.28%
13	48.28%	37.93%
14	34.48%	00.00%
15	13.79%	00.00%
16	44.83%	00.00%
17	51.72%	00.00%
18	41.38%	31.03%
19	10.34%	00.00%
20	44.83%	34.48%
21	72.41%	00.00%
22	20.69%	00.00%
23	41.38%	00.00%
24	37.93%	00.00%
25	48.28%	34.48%
26	48.28%	00.00%
27	44.83%	3.45%
28	41.38%	34.48%
29	37.93%	34.48%
30	48.28%	00.00%
Total	42.99%	15.06%

Analysing Table 4 we can see that our method is again successful in reporting the correct type error location using just one line number with 43% accuracy compared to GHC at 15%. GHC tends to report as many line numbers it feels are associated with the type error, very much like slicing. However, our evaluation shows that it may be more useful and accurate for the programmer to receive only one location at a time.

Overall, our evaluation has proven positive towards our method of type error debugging. From the testing, our strength lies in the reporting of singular type errors, be that one per program or the reporting of one instance of type error amongst many. Our results compared to GHC when testing more than one type error in a program suggests an algorithm that improves upon locating multiple types errors at a time could be beneficial. However, we believe that several locations for one error is an unnecessary burden

on the programmer and, a preference of accurate location over broad suggestion is preferential.

5 RELATED WORK

Type error debugging has taken many forms over the past thirty years so; we will not be able to cover them all of them. Some core categories within type error debugging include; Slicing [7, 14, 18], Interaction [4, 5, 15, 16, 21], Type Inference Modification [1, 11] and, working with Constraints [13, 27]. However, these solutions are complicated to implement. Some expect reliance on the compiler developers to accept the changes, for the programmer to patch their version or to use a particular compiler. Others, do not provide an implementation to use and in the cases where there is an implementation, it is not maintained to work with the latest version of the programming language [9]. We, however, counter these by providing our approach within a tool, used separately from the compiler that employs the Delta Debugging algorithm to locate the type errors.

Delta Debugging, the name for two algorithms, one that simplifies and another that isolates, sparked our interest due to it's closeness to debugging techniques that programmers use[6, 24–26]. Demonstrating the application of the Simplifying Delta Debugging algorithm with the Liquid Haskell type checker [19]. Their approach differs from ours in that we concentrate on the Isolating Delta Debugging technique. Combining the algorithm with direct modification of the programs source code and, with an unmodified Glasgow Haskell Compiler, using its type checker as a blackbox.

Prior works that mention using the idea of a black box include; using the compiler's type inferencer as a black box to construct a type tree to use to debug the program [20] and, having an SMT solver as a blackbox to return the satisfiable set of constraints to show type errored expressions[12]. SEMINAL, a tool which uses the type checker as a black box is the closest to our approach [9, 10]. Unlike our method though, SEMINAL along with previous solutions of using a blackbox compiler, either make modifications to an existing compiler or present an entirely new one. In our approach, we do neither, only passing it source code and gathering the results without any interference from us. Though SEMINAL is also passing information, a patch is required for it to work with the OCaml compiler.

Another difference between our tool, Gramarye, and SEMINAL is that SEMINAL modifies the Abstract Syntax Tree (AST), unlike our strategy of working directly on the source code itself. Other approaches that talk about altering source code are a constraint-free tool inspired by SEMINAL, but though it refers to source code modification they to work with the AST [14]. Another tool TypeHope also discusses changing the source code of a program to stay true to how a programmer debugs. However, again, the solution edits the AST [2]. At this point, as far as the authors know, modifying source code directly is a new approach in the type error debugging field.

6 CONCLUSION AND FUTURE WORK

Our method combines the Isolating Delta Debugging algorithm, a black box compiler and direct source code modification to locate type errors. Our tool Gramarye implements the method for Haskell using the Glasgow Haskell compiler as a black box. From our evaluation, we have gathered positive results that support our method for type error debugging. For single type errors our tool gives a 31% improvement over GHC. However, for two separate type errors in a single program GHC was 3% more successful. When applied to our aim of returning only a single line number for type errors, our method proved positive with 53% for locating singular type errors and, 43% when applied to a program that contained two type errors. A significant practical advantage of our method is that our tool Gramarye has only a small GHC-specific component and thus can easily be modified for other programming languages and compilers.

In the future we will be looking at were Gramarye did well and what its points of failure were. We will then use the outcome of the investigation to improve our algorithm for type error debugging. We will study closer the non-determinism of our method: can we sometimes determine whether one choice is better than another? After we have improved our method to determine the correct line number, we can easily increase the granularity of the tool further to eventually modify programs by single characters instead of lines, thus identifying subexpressions that cause type errors. On the theoretical side, there is clearly a close link between our method and methods described in the literature that perform type error slicing based on minimal unsolvable constraint sets. We want to formalise that link.

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

REFERENCES

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

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

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

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

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

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

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

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

[9] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Proceedings of the ACM SIGPLAN 2007*

Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. 425–434. https://doi.org/10.1145/1250734.1250783

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Received May 2018