UNIVERSITY OF KENT

DEBUGGING TYPE ERRORS WITH A BLACKBOX COMPILER

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of phd.

By

Joanna Sharrad

3rd November 2021

Abstract

Type error debugging can be a laborious yet necessary process for programmers of statically typed functional programming languages. Often a compiler compounds this by inaccurately reporting the location of a type error, a problem that has been a subject of research for over thirty years. However, despite its long history, the solutions proposed are often reliant on direct modifications to the compiler, often distributed in the form of patches. These patches append another level of arduous activity to the task of debugging, keeping them modernised to the ever-changing programming language they support.

This thesis investigates an additional option; the blackbox compiler. Split into three central parts, it shows the individual solutions involved in using a blackbox compiler to debug type errors in functional programming languages. First is a demonstration of how the combination of a blackbox compiler and a generic debugging algorithm can successfully locate type errors. Next tackled is a side-effect of this new combination, the introduction of extra errors, combated with a new speed boosted algorithm, evaluated with a proposed framework based on Data Science techniques to quantify the quality of a type error debugger. Lastly, the algorithms employed throughout this thesis, along with the blackbox compiler, have agnostic properties, they do not need language-specific knowledge. Thus, the final part presents utilising the agnostic abilities for an agnostic debugger to locate type errors.

Contents

Chapter 1

Introduction

Debugging has a long and rich past. From the engineering era of Thomas Edison[1], the first computer "bug" made famous by Grace Hopper, to the many modern-day debuggers using tracing, interactivity, causality, holistic and more that programmers today use to track down troublesome errors (Agrawal, DeMillo and Spafford 1993; Booth and Jones 1997; Albertsson 2006; Magoun and Israel 2013; Smithsonian-Institution 2019). However, even with this vast history, errors can still be tricky to locate (Lauesen 1979; McCauley et al. 2008). In one specific domain, type error debugging in functional programming languages, this issue still has no widely adopted solution even though it has been under the spotlight for over thirty years (Wand 1986; Johnson and Walz 1986).

Type errors in statically typed functional languages such as Haskell, ML and OCaml are challenging to understand and repair. The type error message of a compiler gives the location of a type error in an ill-typed program. However, the message produced can be lengthy, confusing, misleading and often, the location presented is far from the defect that needs repairing, causing programmers hours of frustration.

The cause of inaccurate type error locations can be traced 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 type annotations; this is where type inference fills in the gaps. Type inference works by generating constraints for the type of every expression in the program and solving

[1]Edison is the first to coin the word bug when describing a flaw in designs (Spectrum 2021)

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

Consider the following Haskell program from Stuckey et al. (Stuckey, Sulzmann and Wazny 2004):

```
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 type error location 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 the 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 $\alpha \rightarrow$ `[`β`]` $\rightarrow \alpha$. 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 $\alpha \rightarrow$ `[`α`]` $\rightarrow \alpha$. 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 `[`α`]` and type α, a contradiction reported as type error.

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

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

GHC identifies the type error location correctly:

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

However, one study showed that type signatures are often wrong, causing 30% of all type errors (Wu and Chen 2017)! 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, the order the type checker solves the type constraints determines the reported type error location. There is no fixed order to obtain the right type error location, and requiring type annotations in the program does not help.

Determining the correct type error location is a problem in which researchers have proposed many sophisticated solutions. However, hardly any made it into practice; scaling these solutions to full programming languages such as Haskell and maintaining them with every change to the compiler is complex. This thesis proposes two objectives, *separation* and *scaling*, to tackle this complexity of type error debugging.

The *separation objective* is the detachment of the debugging solution, a type error debugger, from the compiler. This separation is a new concept introduced in this thesis as

current solutions come as either as modifications to an existing compiler or as an introduction of a new one (Schilling 2011; Heeren, Leijen and van IJzendoorn 2003). However, in this thesis, the type error debugger must work by being agnostic to all compiler aspects to ease implementation and maintenance.

Thus, this thesis aims to investigate a type error debugger that avoids the shortcomings and implements the *separation objective* by applying a well-known debugging algorithm, *Delta Debugging* united with a *blackbox compiler*, to the type error debugging domain.

Delta Debugging automates the way programmers systematically debug errors, for example adding and removing lines of code, and is successfully applied to various types of debugging (Zeller 1999). *Delta Debugging's* strength lies in its independence from both the compiler and programming language it is applied too. As a brief example of this strength, *Delta Debugging* is applied to locate a type error in the previous example:

Result of the type error debugger

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

As already noted *Delta Debugging* does not have any knowledge of the underlying programming language or compiler. Instead, *Delta Debugging* takes the results of the *blackbox compiler* to group the lines that do and do not cause the type error. Here bold confirms a line that contains a type error. In the case of the example, this is line one, and whichever lines fall in this group are included in the final result that *Delta Debugging* returns, showing the correct location of the type error. Italics are lines of the program that do not contain the type error; however, they are needed for the type error to occur. The example shows that line two is needed to be combined with line one for the type error to appear. Lines in this group are not reported in the final result. Lastly, fading text represents the final group, those lines that even if that line is removed from the program, it will make no difference to finding the type error; this program is ill-typed even without it. When applying *Delta Debugging* to the ill-typed example, the final result will show line one as the cause of the type error, successfully returning the correct location.

The type error debugger presented in this thesis also works directly on the program source code rather than the abstract syntax tree (AST), making modifications that generate many variants of the ill-typed program. Working directly on source code rather than the AST, though first suggested in 2004, has never been implemented in a type error debugging solution (Braßel 2004). This thesis argues that this is an oversight. Applying changes directly to the source code not only allows for the implementation of at least two of the rules from the 'Manifesto of Good Type Error Reporting'[2], "no renaming of variables" and "keep it source-based", it also compliments a *blackbox compiler* by removing the need for compiler modification. Having a *blackbox compiler* means the type error debugger does not duplicate compiler work such as parsing or have syntactic knowledge. In the case of the *Delta Debugging* algorithm, it uses the *blackbox compiler* as a testing function, gathering only the results of trying to compile a variant of the ill-typed program. The results guide the choices of the *Delta Debugging* algorithm, which terminates when it has found the minimal subset of lines that contain a type error. However, the combination of *Delta Debugging*, source code modification and a *blackbox compiler* does not just fulfil the *separation objective*.

Zeller evaluated *Delta Debugging* on large programs, one such being 178,000 lines long, showing that the algorithm has scalability (Zeller 1999). The *scaling objective* is using this ability to scale to reduce type error debugging complexity.

The type error debugger must work on more than toy languages and small programs used for evaluations. Most solutions do not attempt to evaluate large programs with type errors directly, instead aiming for success on small programs, typically of the size that first-time programmers produce. For example, in *'Learning to blame: localising novice type errors with data-driven diagnosis'*, the authors state; "We acknowledge, of course, that students are not industrial programmers and our results may not translate to large-scale software development..."(Seidel et al. 2017) and in a well-known type error debugging paper *'Counter-Factual Typing for Debugging Type Errors'* the authors say "...the numbers do not tell much about how the systems would perform in everyday practice."(Chen and Erwig 2014b). The *scaling objective* is met by not solely applying *Delta Debugging*. Two more agnostic algorithms were created for this thesis and combined with the type error debugger to improve scaling. The

[2](Yang et al. 2000) Seen in more detail in chapter 2.

type error debugger is evaluated on a new data set of large programs and against an introduced framework for quantifying the quality of type error debuggers to test the scalability.

1.1 Main Contributions

- **A type error debugger**, that applies the *Delta Debugging* algorithm to generate variants of ill-typed programs by adding and removing lines of code. *Delta Debugging* is combined with a *blackbox compiler* to check each of the variants for type errors (Chapter 3). A set of heuristics is introduced to reduce unwanted results from the *blackbox* compiler that cause run-time increases (Chapter 4).

- **A framework**, based on Data Science, for quantifying the quality of type error debuggers is presented to argue that a new way of evaluating type error debuggers is needed (Chapter 5).

- **A new algorithm**, named *Moiety*, which uses the *blackbox compiler* to generate a set of lines based on their ability to cause unwanted results. Each set, called *Moieties*, is pre-processed before being used by the *Delta Debugging* algorithm when adding and removing lines of code. The *Moieties* allow the *Delta Debugging* algorithm to avoid removing lines that cause the *blackbox compiler* to return an unwanted result from an ill-typed variant, thus stopping the slowing down of run-time and increasing scalability (Chapter 6).

- **A second new algorithm**, called *Good-Omens*, increases the ability to scale further by removing the limitations of the pre-processing algorithm *Moiety* by becoming "on-request" (Chapter 7).

- **An agnostic type error debugger** that embraces the agnostic behaviour of all of the algorithms and the *blackbox compiler* by removing any knowledge of the underlying programming language from the type error debugger itself. Initialising the idea that future type error debuggers could be agnostic (Chapter 8).

1.2 Publications

In this thesis, previously published papers form chapters; here are those publications and their associated chapters.

- Sharrad, J., Chitil, O., and Wang, M. (2018), Delta debugging type errors with a blackbox compiler. In Implementation and Application of Functional Languages 2018, ACM, pp. 13-24. This paper introduces the Delta Debugging algorithm combined with a Blackbox Compiler within the Type Error debugging domain. Integrated into Chapter 3 and featuring heavily in Chapters 4-7 as the core of the thesis research and further investigations.

- Tsushima K., Chitil O., Sharrad J. (2019), Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker. In Implementation and Application of Functional Languages 2019, ACM. This paper is a side project undertaken on type error debugging and is discussed in detail in the Related Works in Chapter 9.

- Sharrad J., Chitil O. (2020), Scaling Up Delta Debugging of Type Errors. In Trends in Functional Programming 2020. This paper shows the introduction of the algorithm Moiety. This new algorithm generates input for the Delta Debugging algorithm to reduce unwanted parse errors. The paper also introduces a scalability data-set and a framework to aid in evaluating type error debuggers. Chapters 4, 5, and 6 contain the core of this paper.

- Sharrad J., Chitil O. (2021), Refining the Delta Debugging of Type Error. In Implementation and Application of Functional Languages 2021. This paper is currently under review for publication. The paper presents replacing the pre-processing algorithm Moiety with a new 'on-request' algorithm named Good-Omens. The paper also introduces the agnostic type error debugger. Chapters 7 and 8 contain the core of this paper.

1.3 Thesis Structure

The thesis statement and contributions are addressed in the following chapters:

- Chapter 2: provides the background, which includes material on key concepts.
- Chapter 3: shows how to apply the *Delta Debugging* algorithm to type errors using the compiler as a *blackbox*.
- Chapter 4: covers an investigation into reducing *blackbox* results that negatively impact overall debugging performance using heuristics, specifically looking at the effects of programming language syntax.
- Chapter 5: introduces a new framework for evaluating type error debuggers that, when adopted, will allow for ease of comparison to previous type error debuggers.
- Chapter 6: provides further analysis of the blackbox results and introduces a new algorithm, *Moiety*, in response to unwanted results.
- Chapter 7: introduces a new algorithm, *Good-Omens*, that reduces the number of calls to the *blackbox compiler* to decrease the debugging run-time.
- Chapter 8: implements all of the positive observations from previous chapters into a type error debugger that is agnostic to programming languages.
- Chapter 9: provides the related works, which is split into key topic areas and in roughly chronological order.
- Chapter 10 concludes the thesis with future work.

Chapter 2

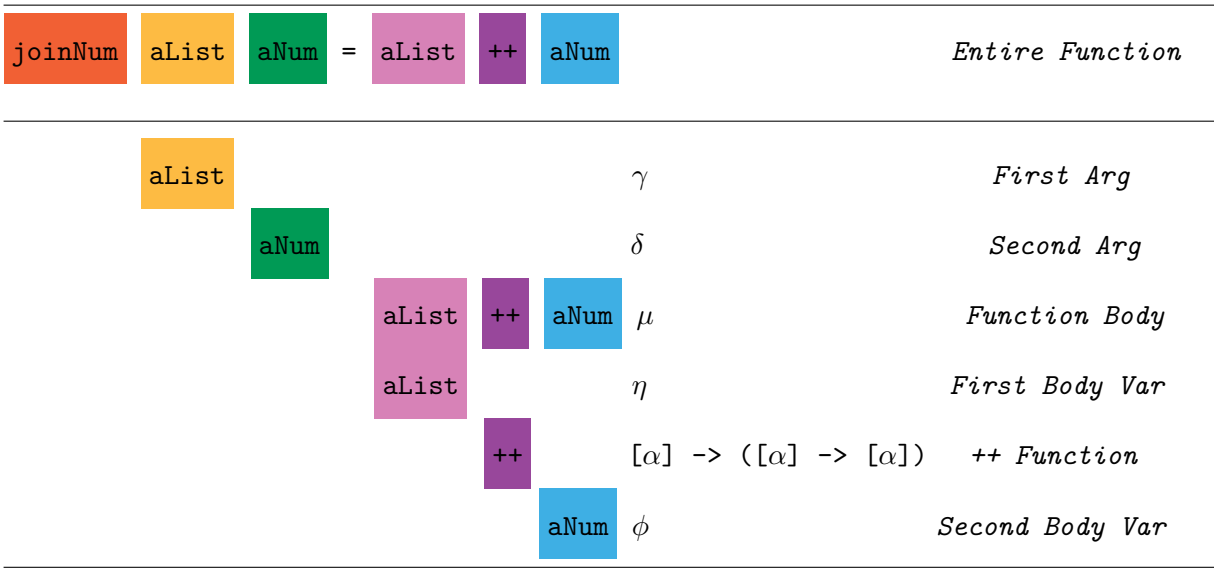
Background

This chapter covers the core fundamentals needed for the subsequent chapters. In section 2.1 an example explains the connection between type inference and the problem of locating type errors. Then in sections 2.2 and 2.3 key terminology and how they work, covering *Delta Debugging* and agnostic behaviour using a *blackbox compiler*, is presented.

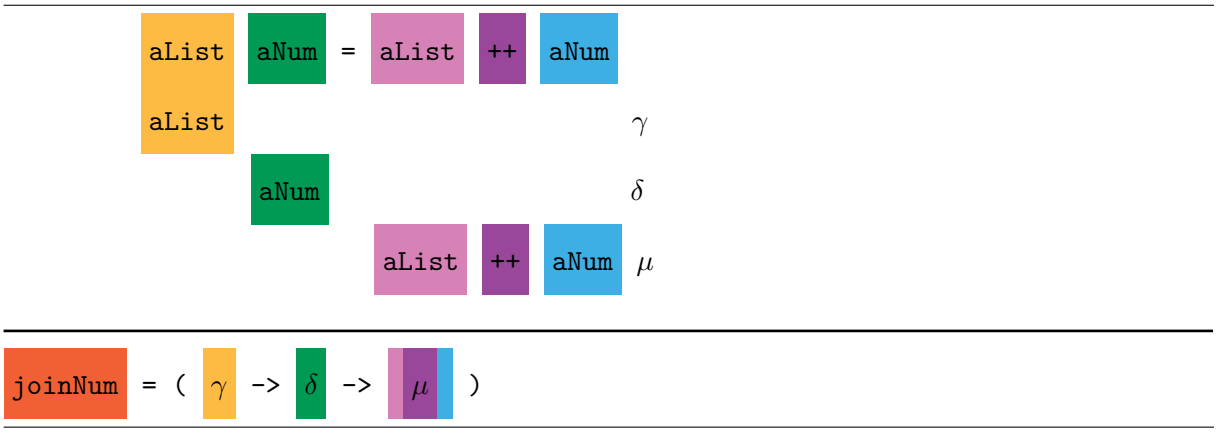
2.1 Type Inference and the Type Error Problem

This section shows an example of how a generic type inference algorithm works and how this process causes reports of type errors far from the cause. Note that there is a separation of the generating and solving of constraints in the example. However, typically older type inference algorithms complete constraints simultaneously.

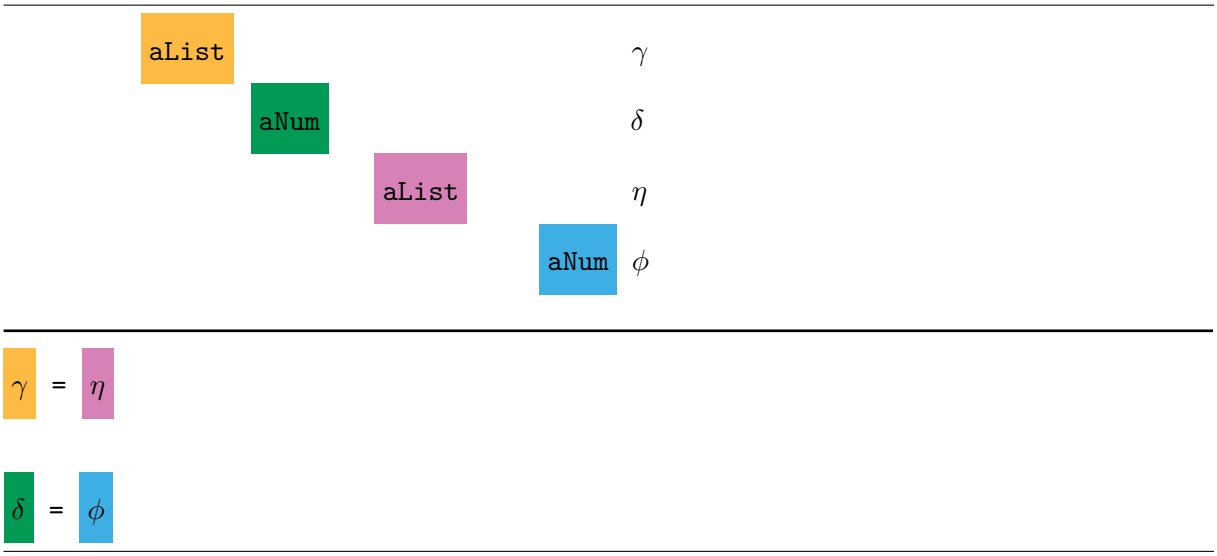
The example program, a one-line function, can be seen on the first line of the example with each construct, which has been colour coded for readability, placed on the lines that follow. The initial stage of the algorithm is to assign generic type variables, represented as a letter of the Greek alphabet, to each construct.



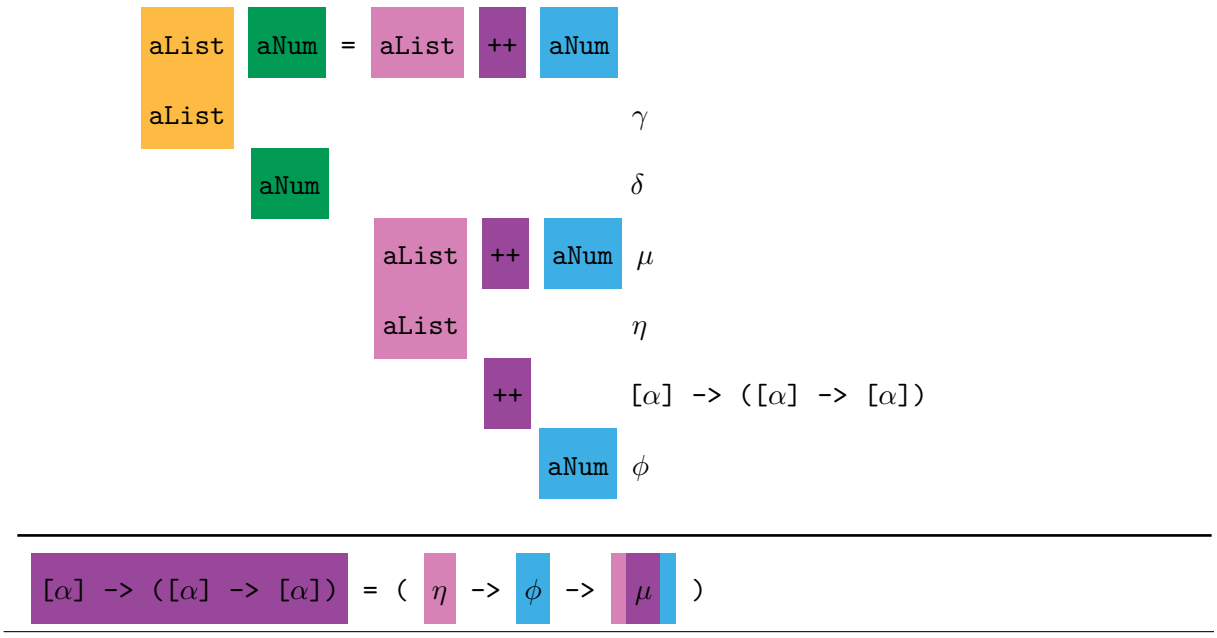
The type inference algorithm now has to work though the generic type variables to generate constraints that will in turn allow it to infer the correct types. As the parameters have type γ and δ and the function body has type μ, it must have constraints that say that the type of the function must equal the type of its parameters and body; to do so parameters are compared to the body:



Next, the parameters must equate to the variables used in the body of the function. Represented in the parameters γ and δ that have the types η and ϕ; the algorithm identifies these together, and since aList appears in both γ and η and aNum appears in both δ and ϕ their types must be the same.



Lastly, the (++) function has to be dealt with; it needs to have the same type as the parameters it is applied too. The function ++ has type $[\alpha]\rightarrow([\alpha]\rightarrow[\alpha])$ and the parameters have types η and ϕ:



The constraints have now been generated and now have to be solved. Here is where both Hindley and Milner used the Robinson Unification algorithm to solve constraints by substitution. Unifying and substituting substitutes the entire set of constraints by replacing the constraints one by one until they are all solved or a type error is received. A solution of

constraint solving using the example would mean that if all of the generic variables appear only on the left hand side of an equation and not on the right; with a fully solved set of constraints would look like:

```
joinNum = [α] -> [α] -> [α]
γ = [α]
δ = [α]
η = [α]
ϕ = [α]
```

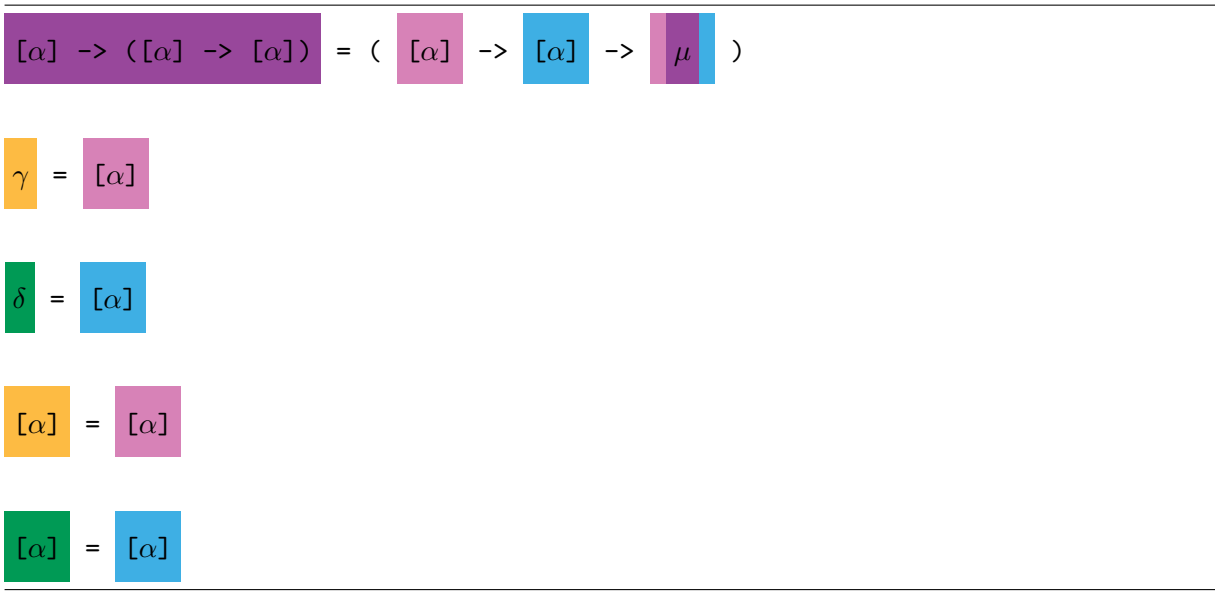
Returning to the current set of unsolved constraints; summarised as;

```
Join parameters with the body;
joinNum = ( γ -> δ -> μ )
```

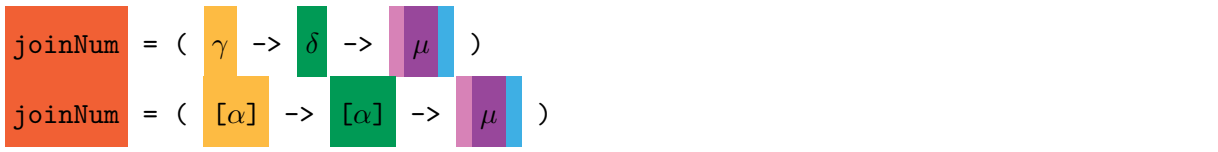
```
Join parameters with the variables body;
γ = η

δ = ϕ
```

```
Join body with the (++) function;
[α] -> ([α] -> [α]) = ( η -> ϕ -> μ )
```

When starting unification and substitution, the procedure starts from the bottom of the constraints. η and ϕ must have the type $([\alpha] \rightarrow [\alpha])$ as the type of the ++ function body also has this type. Substitution by the type $[\alpha]$ happens to each of them. However, the algorithm knows that the functions parameters and variables in the body also should have the same type; these can also solve and substitute again with type $[\alpha]$;



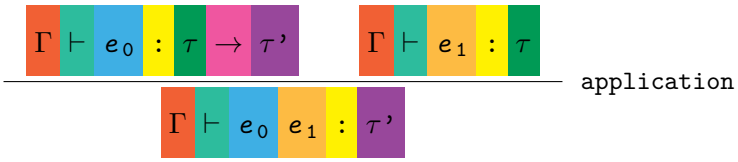
The original function, *joinNum*, still has the generic type variables but this can now be changed as at this point it is known that they have to be equal to the above.



μ is the body of the function and has the same type as the parameters. A final substitution can know be done which means that the joinNum function, has the type;

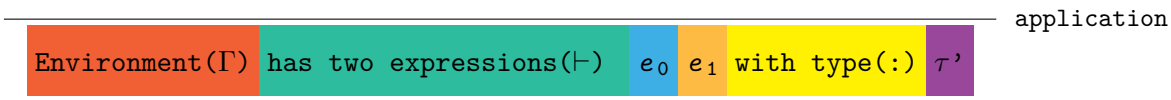
The above explains that the type inference algorithm works by applying constraints. Several rules make up the constraints. However, here, to show in more depth how type errors occur, the discussion will be of only the 'application' rule using an example with colour to highlight matching elements[1].

[1]More in-depth discussion on type inference and constraints appears in Chapter 10 of Advanced Topics in Types and Programming Languages (Pierce 2005)



Above the line of the rule shows the two premises; these rules define the relationship of the type; if the premises hold, so does the conclusion.

The first premise on the left states that if e_0 has type τ it will have type τ', and the right premise states that e_1 has type τ. The conclusion of the type rule, when both premises hold, states that if applied e_1 to e_0 (application), a conclusion can be formed that both expressions must have type τ'.



However, if e_1 and e_0 have previously been assigned types, and these types differ, the *application* rule cannot conclude, the constraint cannot be solved, the program is ill-typed, and the type inference algorithm stops.

Type errors are the compilers way of telling the programmer that the type inference algorithm can no longer solve the constraints. The compiler generates the type errors with details known at the point in time when the type inference algorithm stopped, including information such as the location of the type error and cause. Unfortunately, the function application described above is particularly susceptible to failure to solve constraints and the compiler returning an incorrect location in the type error message. The problem with the application rule is that if it fails to be solved, the compiler has no way of telling whether it is e_1 that has the correct type or e_0. In most cases the compiler will take e_0 as the correct type and report a type error blaming e_1 as the cause, this is known as left-to-right bias and is covered further in Chapter 9. This incorrect reporting by the compiler can show a type error

location quite far from where the actual cause occurred, and this thesis aims to produce a solution that reduces this issue. To do so, it will employ three aspects, Delta Debugging, blackbox compiler, and agnostic behaviour. To understand the three aspects used in the subsequent chapters, the background of each is provided next.

2.2 Delta Debugging

When programmers debug, they first use the error message to conjecture a possible cause in the program (hypothesis), then modify the program and recompile (test), and lastly use the outcome of the recompilation (result) to either repeat with a new, improved hypothesis, or terminate with the proven hypothesis. Zeller recognised the scientific workflow of *Hypothesis-Test-Result* in debugging and automated it, giving it the name Delta Debugging (Zeller 1999, 2002, 2009; Zeller and Hildebrandt 2002; Cleve and Zeller 2005).

Unlike the solution in this thesis, which works with static errors, Zeller does not only consider locating a cause in a defective program but alternatively locating a cause in an input to a program that causes a failure at run-time or locating a cause in the run-time state of a program execution. He abstracts program/input/state by talking about configurations and differences between configurations. Generating variants of the configuration to compare requires removing pieces of the configuration; for example, if the configuration is a program, 'pieces' could be lines of code or characters within a line of code. Essential for checking the differences between configuration variants is the existence of a testing function that places a configuration into one of the following three categories:

1. Fail ($\times$). The configuration variant contains the bug.
2. Pass ($\surd$). The configuration variant does not contain the bug.
3. Unresolved (?). All other results. Commonly, a different bug to the bug the initial debugging is trying to find.

Zeller presents two delta debugging algorithms to implement the generation and testing of configurations for locating errors. He refers to them as Simplifying and Isolating (Zeller 2009).

2.2.1 Simplifying Delta Debugging

The Simplifying algorithm is a greedy algorithm that determines a smaller configuration variant of the given faulty, Fail (×) configuration. The result is minimal in that removing any other pieces makes the configuration variant Pass (√). The algorithm works by removing pieces of a faulty configuration until it no longer returns a Fail (×). It divides the configuration into two halves and tests each one. This division is seen in the example below. At first, a faulty Haskell program provides the initial configuration, and secondly, the generation of a configuration variant that has had the opening division.

Initial Faulty Configuration

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

First generation of a configuration variant

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

If one half receives the Fail (×) category, the algorithm works recursively for that half. Suppose neither half has the Fail (×) category. In that case, it receives an Unresolved (?) result, and it divides the configuration into four pieces and tests each one, this division is named *Granularity* and is presented in more detail in Section 2.2.2.1. When the configurations division cannot go further, the algorithm stops with the last configuration variant that received a Fail (×) as a result. The minimality allows for surmising that all pieces of the configuration left must contribute to the error.

The Simplifying Delta Debugging algorithm has the disadvantage that it often reports a rather large configuration as containing the bug; a failing configuration still contains many passing pieces. To counter this issue, Zeller created the second Delta Debugging algorithm, *Isolating*, which aims to reduce the reported configuration size further.

2.2.2 Isolating Delta Debugging

Algorithm 1 shows a python style pseudocode based on the original *Isolating Delta Debugging* algorithm implemented by Zeller (Zeller 2021). Isolating Delta Debugging employs the Simplifying algorithm to generate a minimal faulty configuration. At the same time, the algorithm also produces a maximal passing configuration. The maximal program is created by taking a passing configuration, for example, an empty program, and adding pieces from the faulty configuration until the configuration fails. If any tested configuration yields a passing, Pass ($\surd$), outcome, it can become the new passing configuration; if any tested configuration yields a failing, Fail ($\times$), outcome, it can become the new failing configuration; then, the algorithm calls itself recursively with a new pair of configurations. Suppose all of the tested configurations yield unresolved outcomes. In that case, *Granularity* is used to divide the configurations into four, eight, et cetera, pieces, similar to the simplifying delta debugging algorithm, until it eventually finds a passing or failing configuration; if no further division is possible, the algorithm terminates.

The algorithm does not specify how to divide the pieces of the difference between the two configurations, and there may be several passing and failing outcomes. Thus, the algorithm is non-deterministic; however, like any other implementation, the solution in this thesis makes a choice and is deterministic, as no study on the effect of different approaches has yet been done. In every recursive call, the passing configuration is a subconfiguration of the failing configuration (both subconfigurations of the original ill-typed program). Thus, every recursive call reduces the difference between the two configurations until the difference cannot reduce any further.

The final result of *delta debugging* is a pair of configurations. The first configuration is a passing subconfiguration of the second failing configuration, such that there is no passing or failing configuration between the two configurations. The difference between the minimal failing and the maximal passing configuration is then considered a cause of the fault. This difference is substantially smaller than a minimal faulty configuration generated by *Simplifying Delta Debugging*. Furthermore, Zeller states that the Isolation algorithm is much more efficient in practice than the Simplification algorithm (Zeller 2009). For the reasons just

Algorithm 1: Delta Debugging of a type error

```
1  define dd (cPass, cFail, cont)
2      n ← 2
3      loop
4          delta ← cMinus(cFail, cPass)
5          if n > len(delta) then
6              return (cPass, cFail)
7          deltas ← cSplit(delta, n)
8          unres ← True
9          j ← 0
10         while j < n do
11             nextCPass = cPlus(cPass, deltas[j])
12             nextCFail = cMinus(cFail, deltas[j])
13             resNextCFail ← test(nextCFail, cont)
14             resNextCPass ← test(nextCPass, cont)
15             if resNextCFail == PASS then
16                 cPass ← nextCFail
17                 n ← 2; unres ← False; break
18             else if resNextCPass == FAIL then
19                 cFail ← nextCPass
20                 n ← 2; unres ← False; break
21             else if resNextCFail == FAIL then
22                 cFail ← nextCFail
23                 n ← max(n − 1, 2); unres ← False; break
24             else if resNextCPass == PASS then
25                 cPass ← nextCPass
26                 n ← max(n − 1, 2); unres ← False; break
27             else Try next part of delta
28                 j ← j + 1
29         end while
30         if unres then all deltas give unresolved
31             if n >= len(delta) then
32                 return (cPass, cFail)
33             else increase granularity
34                 n ← min(n ∗ 2, len(delta))
35     end loop
36 end define
```

stated, the solution presented in the rest of this thesis will use Isolating Delta Debugging. As such, any reference to *Delta Debugging* will refer to that version.

2.2.2.1 Granularity

The Isolating Delta Debugging algorithm consists of two parts: the generation of configuration variants, described in the previous section, and Granularity. Within the algorithm, a granularity parameter determines how many pieces move between the configurations. For example, a granularity of 2 means that the algorithm resembles a binary chop algorithm; it repeatedly divides the faulty program in half. The Delta Debugging algorithm starts with Granularity set to 2. However, Granularity can grow and shrink depending on the testing function results, only stopping when there is a one-line difference between two configurations.

To understand Granularity, below is an example showing the algorithm receiving an eight-line program with an error on line 8. Here the pieces that isolating delta debugging move are lines of code, and the program is the initial configuration for the algorithm. Only when the testing function returns 'unresolved', the Granularity may increase. Hence, assume that lines 1 to 3 and lines 4 and 5 of the program belong together: any program that contains only some lines of these two sets yields 'unresolved'.

At step 1 the passing and respectively failing configurations have the following lines, displayed as sets of lines within braces:

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

Because granularity is 2 and the two programs differ by 8 lines, the algorithm moves the first $8/2 = 4$ lines from the failing program to the passing one and then test both modified programs:

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

Both programs are unresolved. The algorithm cannot move any other four lines (only adjacent lines move). Hence Granularity is doubled from 2 to 4. The algorithm moves $8/4 = 2$ lines from the step 1 failing program to the passing program. The algorithm first tries

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

Both are unresolved, so the algorithm tries the next two lines:

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

Again both are unresolved, so the algorithm continues with another two lines:

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

Still unresolved, so the algorithm tries:

`{7,8}`$_\times$ `{1,2,3,4,5,6}`$_\surd$

Finally, the test function gives a different results. The algorithm resets granularity to 2. The algorithm selects the passing program for Step 2; thus currently there are the following passing and failing program:

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

The two programs differ by 2 lines. The algorithm moves $2/2 = 1$ line from the failing program to the passing program:

`{1,2,3,4,5,6,7}`$_\surd$ `{1,2,3,4,5,6,8}`$_\times$

The first program is passing, the second failing. Granularity stays at 2. The algorithm select the failing program for Step 3:

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

Because the two programs differ only by one line, the isolating delta debugging algorithm stops. The algorithm has identified the single line difference, line 8, as the cause of the error.

2.2.2.2 Working with lines of source code

In the previous section, Delta Debugging moves lines of source code between configurations to find an error in a program. The strategy of moving lines of source code and returning a

line as the location of a type error is used throughout this thesis. To support the line-by-line approach, the solutions in this thesis work directly on the source code, instead of working on the Abstract Syntax Tree (AST), which other solutions that are examined later in Chapter 9 do. Working directly with the AST benefits from not being dependent on the source code layout, which the line-by-line approach does. Formatting styles, such as declaring a type signature over several lines compared to a singular line, could cause different outcomes in a line-by-line approach, impacting precision and efficiency. As such, a separate investigation into the impact of programming styles when directly manipulating source code is a direction for future work. Nevertheless, as discussed in detail in Chapter 3, working directly with the source code on a line-by-line approach also has benefits, such as the ability to allow for both the use of blackboxes and agnostic systems, which are described next.

2.3 Blackboxes and Agnostic Systems

In this thesis, the term *blackbox* means the testing function for *delta debugging*. For the *blackbox* testing function, the debuggers in this thesis use the entire compiler. Using the whole compiler as a *blackbox* testing function is different from previous solutions, in the type error debugging domain, as they use the term *blackbox* to mean just the type checker. Recall how *Delta Debugging* is implemented. Every time a configuration variant is generated, it needs to be tested. To test configurations, calls are made to a testing function, and this testing function is the *blackbox compiler*. The *blackbox compiler* can be any compiler, however, for the majority of this thesis, it will be the Glasgow Haskell Compiler, and the examples will all be in the Haskell programming language.

As *delta debugging* runs it gathers the information that a configuration has one of three categories. To place the configurations into categories the *blackbox compiler* is called and using the Standard Input/Output/Error system that all software uses a result is returned. As an example, calling the *blackbox compiler* on two different configurations, one that is a Fail ($\times$) and the other a Pass ($\surd$) will return the following information:

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

Listing 2.1: Standard Error - Category: Fail (×)

```
0
```

Listing 2.2: Standard Output - Category: Pass (√)

The results allow for no knowledge of the compilers inner workings. Any solution implementing *delta debugging* can differentiate between them by using a standard set of terms, an idea investigated in more detail in Chapter 3. As the exploitation of GHC, or any other compiler, to gather information is done without altering the compiler itself, the type error debugger is kept separate from the compiler. Not modifying the compiler has many benefits; the compiler developers' changes will not affect how the proposed debugger works. Users of the debugger can avoid downloading a specialist compiler and do not have the hassle of patching an existing one. Avoiding modification of the compiler means that though the initial investigation in this thesis employs Haskell and GHC, the method is not restricted to this language, giving scope to expand to other functional languages such as OCaml, an idea described in detail in Chapter 8.

As with *blackbox*, agnostic is a word that appears periodically. In computer science, an agnostic solution contains no knowledge of the syntax of the source code, meaning it can work with many different programming languages. In the case of this thesis, it takes on a slightly different meaning. The type error debugger still does not know the programming language, with all algorithms being agnostic, such as *Delta Debugging* and its testing function; however, it can require some information from external sources.

2.4 A Manifesto of Good Type Error Reporting

This background will close with a subset of rules proposed by Yang, Michaelson, Trinder, and Wells (Yang et al. 2000). The original seven rules suggest the properties a good type

error message should have. For instance, **Succinct** refers to the length of an error message, not too short or long, and **Amechanical** states that the message only contains recognisable source code by not introducing external information or renaming variables. This thesis does not analyse type error messages; however, I believe that a subset of the rules, five of the seven, can be interpreted regarding the type error debugging solution itself and not just its message. Here, listed next are those five rules which the type error debugging solutions in this thesis aim to incorporate along with the discussed objectives in Chapter 1.

- **Correct.** The solutions will strive to return the correct location of the type error with all locations reported being connected to the fault.
- **Source-based.** The solutions will use only the source code of the ill-typed program and will have no knowledge of the compiler's internals.
- **Precise.** The minimal amount of incorrect source code should be found. The optimal result is a singular line of source code; however, this property can be overruled by the following rules.
- **Unbiased** and **Comprehensive.** If more than one location is found to cause the type error; there will be no bias in which location to report; all sites that contributed to the error will be reported.

Chapter 3

Delta Debugging with a Blackbox Compiler

In the Introduction, an example provides an overview of correctly locating type errors and a solution in the form of applying *Delta Debugging*. This chapter will expand on the initial part of the solution, a type error debugger named Gramarye that combines the *Delta Debugging* algorithm with a *Blackbox compiler*. Let us briefly recollect *Delta Debugging* as discussed in Section 2.2 and how it applies to type error debugging.

Recall from Section 2.2 that a tested program is called a *configuration* and from that configuration variants are generated. Configuration variants are a subconfiguration in that they are a division of pieces of the original configuration. Different options are available for the division. However, the type error debugger in this chapter uses the same choice of configurations as many other implementations of *Delta Debugging*: Gramarye always chooses to remove whole lines from the configurations[1]. A major benefit of using whole lines is that Gramarye can avoid undesirable changes in the layout of the original program by keeping empty lines. Therefore, abides by rule 5 of the "A Manifesto of Good Type Error Reporting": keep it source-based. However, all complexity measures of type error debugging are concerning the number of lines of the ill-typed program. There is an exponential number of configurations for a given ill-typed program, and already finding a failing configuration of minimal size is known to be NP-complete (Misherghi and Su 2006).

[1]Removing single characters is another choice presented by Zeller (Zeller 2009).

A configuration variant is the original Fail (×), now also referred to as ill-typed, configuration with some lines replaced by empty lines and the Pass (√) configuration, now also known as well-typed, which has lines added. A configuration being a subconfiguration of another configuration is a natural partial order on configurations. The empty configuration consists of many empty lines being the minimum and the original ill-typed configuration being the maximum. Recall that this thesis uses the *isolating delta debugging* algorithm for type error debugging due to a minimal ill-typed configuration being often still extensive because every function or type it uses has to include its definition, which is usually well-typed. Gramarye here excludes these well-typed definitions to isolate a cause of the type error.

In the implementation of Gramarye, the algorithm starts with the empty configuration as a passing configuration and the ill-typed program as a failing configuration. Recall from chapter 2 the pseudocode of the *delta debugging* algorithm, when applied to the type error debugging domain `nextCFail` is our ill-typed program and `nextCPass` is our well-typed program. The algorithm divides the difference between the two configurations into two pieces, and for Gramarye, the split lines are kept in consecutive order. It tests the passing configuration with each of these pieces added and the failing configuration with the pieces removed. Remember from Section 2.2 that testing a configuration yields one of three outcomes: Fail (×), Pass (√) or Unresolved (?). Here, the categories are applied differently with: Fail (×) as the configuration variant that contains a type error, Pass (√) as when the variant compiles and Unresolved (?) as any other error such as parse error or unbound identifier. In Gramarye, the testing is done by a *blackbox compiler*. Once the initial division and configuration categories are known, Gramarye continues to follow the choices of the *delta debugging* algorithm until it terminates. The final pair of configurations is the result of the *delta debugging* algorithm. The difference between the two configurations, which may be neither a passing nor a failing configuration, isolates a failure cause. This difference is the result of Gramarye and is reported as a line number.

Figure 1 gives a brief informal overview of the flow taken by Gramarye, labelled as steps 1 to 4, as described above, to locate type errors in an ill-typed program. The initial step is to input the raw source code into Gramarye. Next, this source code is giving to the *Delta Debugging* algorithm in step 2. The algorithm at step 2 now has a choice if it has finished,

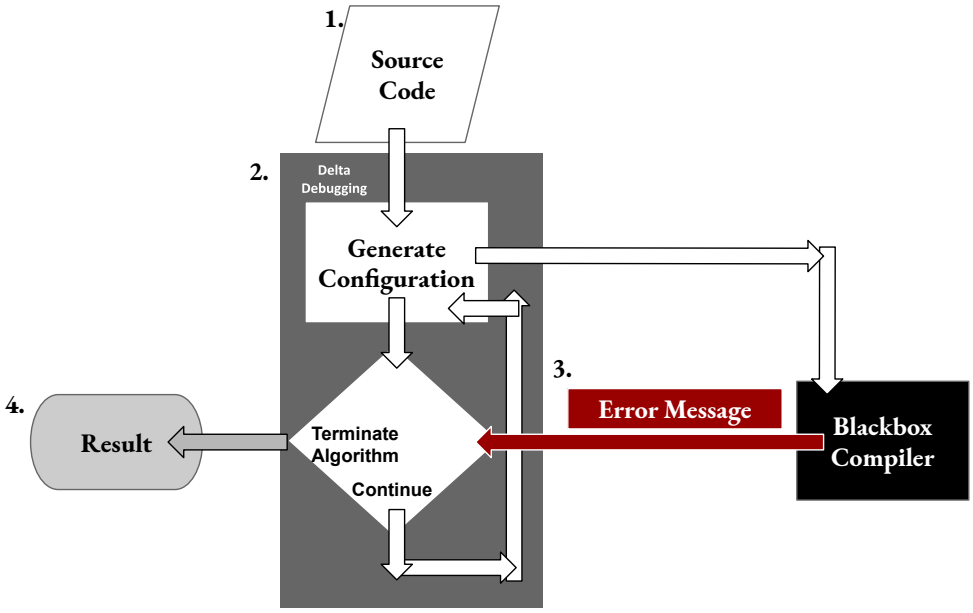


Figure 1: The informal overview flow of Gramarye

it can terminate, or if not, it will generate configurations. Step 3 is the configurations sent to the *blackbox compiler*, the compiler then sends the *Delta Debugging* algorithm the results from trying to compile the configuration. This back and forth between steps 2 and 3 continues recursively until it can longer generate more configurations. Thus, the algorithm stops and, at step 4, returns a result of where the type error occurs.

3.1 An Illustration of the proposed debugger

Next, this section shows in more depth how the brief overview of the flow works in practice. Step 1 consists of starting with a single ill-typed program. From this program, the *Delta Debugging* algorithm creates two configuration variants. One is the ill-typed configuration, from which the algorithm removes lines that are irrelevant for the type error. The other is the empty, well-typed configuration:

Step 1: well-typed configuration

```
1
2
3
```

Step 1: ill-typed configuration

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

Moving onto step 2, the *Delta Debugging* algorithm starts generating configuration variants and line 3 is transferred from the ill-typed configuration to the well-typed configuration:

Step 2: modified well-typed configuration

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

Step 2: modified ill-typed configuration

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

The two configurations are then sent back to the *blackbox compiler* for category checking; entering step 3 of the Gramarye's flow in Figure 1:

- Step 3: modified well-typed configuration: unresolved.
- Step 3: modified ill-typed configuration: ill-typed.

The modified well-typed configuration is not a syntactically valid Haskell program; the compiler yields a parse error. As described in Section 2.2 and the previous section, the testing function results can be three things: Pass, Fail, and Unresolved. However, Gramarye cannot

use an Unresolved configuration when locating a type error, but each of the other two possible results is useful. The modified ill-typed configuration is smaller than the original ill-typed configuration. The modified variant is ill-typed too, so replaces the ill-typed configuration for the next step:

New well-typed configuration

```
1
2
3
```

New ill-typed configuration

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

Gramarye is back at step 2 again. The algorithm now repeats: Again, moving a single line from the ill-typed configuration to the well-typed configuration. Picking line 2:

Step 2: modified well-typed configuration

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

Step 2: modified ill-typed configuration

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

Again two configurations are checked for categories at step 3 by the *blackbox compiler*:

- Step 3: modified well-typed configuration: well-typed.

- Step 3: modified ill-typed configuration: well-typed.

Because both variants are well-typed and larger than the previous well-typed configuration, either of them can be the new well-typed configuration. The modified well-typed configuration is picked and thus obtain;

New well-typed configuration

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

New ill-typed configuration

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

Gramarye returns to step 2, the *delta debugging* algorithm, and as the well-typed and ill-typed configurations differ by only a single line, the algorithm terminates, and step 4 presents the results, which are described in more detail in Section 3.2.3.

The isolating delta debugging algorithm is non-deterministic. Often different choices lead to the same final result, but not always. Zeller argues that this non-determinism does not matter and that one result provides insightful debugging information to the programmer (Zeller 2009). Hence his algorithm is deterministic, making arbitrary choices. The implementation follows his algorithm and, for the example, makes the choices described here.

The algorithm uses an ordering of configurations, where a configuration is just a sequence of strings. A configuration P_1 is less or equal to configuration P_2 if they have the same number of lines, and for every line, the line content is either the same for both configurations, or the line is empty in P_1. All configurations considered are variants between the initial well-typed and ill-typed configurations. The final well-typed and ill-typed configurations have minimal distance; that is, they either differ by just one line, or configurations between them are unresolved; that is, they are not syntactically valid programs.

In this example, only a single line moves from the ill-typed to the well-typed configuration in each step. For programs with hundreds of lines, this simple approach would be expensive in time due to the number of configurations needing consideration. Hence, the complete *Delta Debugging* algorithm starts with moving either the first or second half of the configuration from the ill-typed to the well-typed configuration. If both modified configurations are unresolved, the algorithm increases the granularity of modifications from moving half the configuration to moving a quarter of the configuration. In general, every time both modified configurations are unresolved, the modifications are halved in size. This increase in granularity can continue until only a single line is modified.

Zeller analysed the complexity of the *Delta Debugging* algorithm. In this case, complexity is the number of calls to the blackbox compiler in relation to the number of lines of the original program. In the worst case, if most calls yield unresolved, the number of calls is quadratic. In the best case, when no call yields unresolved, the number of calls is logarithmic (Zeller 2009).

3.2 Implementation

As illustrated in Figure 1 and the description above, the solution has four components: Delta Debugging, described in both the previous Section and in Section 2.2, the Blackbox Compiler, Source Code Modification, and Result Processing. The combination of these four components is the *Gramarye* Type Error Debugging Tool. A description of each of the last three components will now follow.

3.2.1 A Blackbox Compiler

As already described in Section 2.3, compilers naturally lend themselves to this usage, taking an input (source code) and returning an output; a successfully compiled program or error. Anything that happens within the blackbox remains a mystery. The compiler chosen to be a blackbox for this part of the research is the Glasgow Haskell Compiler (GHC), which the Haskell community widely use. During each iteration of the *Delta Debugging* algorithm, the blackbox compiler is called to determine the status from the modified ill-typed and

well-typed configurations described in section 3.1. When using the *blackbox compiler*, the tool receives the same output a programmer would when they are using GHC. Though compiling with GHC gives a message that includes many details, the only interest is whether the configurations are well-typed, using this information to categorise. Depending on the categories returned, there are modifications in different ways of the configurations' source code, which is then again sent to the *blackbox compiler* for further categorisation.

Recall the examples from Chapter 2 where it is noted that special terms known by the debugger are the only knowledge needed:

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

Listing 3.1: Standard Error - Category: Fail (×)

```
0
```

Listing 3.2: Standard Output - Category: Pass (√)

For example, for Gramarye to categorise the results from GHC, only two terms are required. One recognises 0 as a Pass, and another detects the term "type" in the error output to apply Fail. If neither of these terms match, Gramarye recognises the output as an Unresolved result. Unfortunately, this type of categorising is compiler-specific. However, Chapter 8 begins an investigation into removing this barrier.

3.2.2 Source Code Manipulation

When programmers manually debug, they edit their source code directly, looking at where the error message suggests the occurrence and making changes in the surrounding area. The Gramarye debugging tool is also directly manipulating the source code, modifying configurations using line numbers determined by the *Isolating Delta Debugging* algorithm on a line-by-line based approach. As observed in Section 3.1, modifying is by adding and removing lines of source code until completion of the algorithm where two final configurations are left. One configuration has all ill-typed source code removed, and the other only contains

well-typed code. Gramarye has directly modified the source code to achieve these two final configurations; the tool can use them to find the line number where the type error appears by calculating the difference between them.

Gramarye does not work on the Abstract Syntax Tree (AST). Thus, it does not need to parse the source code with each modification, allowing the avoidance of changes to an existing compiler or creating a parser. Not editing the AST also means Gramarye can stay true to the programmer's original program, keeping personal preferences in layout intact by using empty lines as placeholders, thus allowing error messages that refer to the original program. Overall, direct source code manipulation allows Gramarye to be easy to maintain, something that prior solutions struggle with, often becoming obsolete quickly, whilst giving it the possibility of being separate from a specific programming language, a hypothesis covered in Chapter 8.

3.2.3 Processing the Results

The idea is that if one configuration is well-typed and the other ill-typed, then the source of the type error lies within the difference of the two; the relevant difference (Zeller 2009). Thus, after the *Delta Debugging* algorithm has terminated, two configurations are left. Recall the example from the beginning of this chapter. The *Delta Debugging* algorithm had stopped with the following two configurations:

Configuration One

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

Configuration Two

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

The Gramarye debugging tool takes these two configurations and looks for where they intersect. In this case, both configurations have line 2 intact and line 3 removed, leaving the result of the intersection between the two configurations as being: {2,3}. However, to discover the type error, the debugger needs to report the relevant difference and as we already know that lines {2,3} are in both configurations that only leaves, in this example, line 1, which is removed in configuration one and still intact in configuration two. The final outcome of the Gramarye debugging tool is the following:

Gramarye Tool result

```
The lines that contain the type error: {1}
```

With the {1} representing the line number of which the type error is found. Reporting whole lines in this way is also beneficial in two ways; firstly, when evaluating how successful the tool is in locating type errors. If the line number matches the one known to be of cause, when using an oracle, explained next, then the debugging is a success, a method used in prior type error debugging literature[2]. Secondly, it cooperates with the intention of being disengaged from the programming language. By representing the location of the type error by line number only, no syntactic knowledge is necessary.

3.2.4 Multiple Type Errors

Multiple errors can naturally occur when programming. In this thesis, multiple errors in an ill-typed program represent more than one type error in a single program. By default, *Delta Debugging* will return one error, and as such, so does the *Gramarye* tool; neither have specific provisions for multiple errors. Note that the returned single error does not mean that the error is in the numerical order of line numbers. For example, two type errors could occur in a 10 line long ill-typed program, one on line 3 and the other on line 9. Depending on the categories of the configurations *Delta Debugging* could return line 9 as the error located. How well Gramarye and Delta Debugging will handle these types of programs is investigated in the evaluation next.

[2]See Related Works in Chapter 9 for more details.

3.3 Evaluation

This evaluation sets Gramarye against a benchmark of programs specially engineered to contain type errors. Chen and Erwig collated the programs to evaluate their Counter-Factual approach to type error debugging (Chen and Erwig 2014a). In all, there are 121 programs in the CE benchmark, but not all had what Chen and Erwig called the 'oracle', the knowledge of where the type error lay. Those programs that did not have the 'oracle' to state the correct location of the type error were removed due to being unusable as it would not be apparent if the result from the debugger is successful. Also removed were programs that produced similar type errors, reducing the test programs' set to thirty. The evaluation will also be used to see whether Gramarye could report multiple type errors. Therefore, each pair of different programs from the 30 test programs are taken and the two programs are joined together, thus generating a further 870 programs for the evaluation.

This evaluation answers the following questions;

1. Gramarye is applied to Haskell programs that each contain a single type error. Is there an improvement in locating the errors compared to the Glasgow Haskell Compiler? (Section 3.3.1)

2. Gramarye is applied to Haskell programs that each contain two type errors. Is there an improvement in locating these errors compared to the Glasgow Haskell Compiler? (Section 3.3.2)

3. Does the Gramarye debugging tool return a smaller set of type error locations compared to the Glasgow Haskell Compiler? (Section 3.3.3)

The evaluation uses GHC 8.2.2 as a comparison and also as the *blackbox compiler* within the tool Gramarye. However, as its sole use is to categorise, Gramarye does not use the line numbers that it reports, and thus these line numbers have no interference with this evaluation. GHC and Gramarye take the CE benchmarks and categorise each one; this results in a set of line numbers suggested as a cause of the type error. The evaluation uses the same criterion as Wand to judge the success of locating the type error(Wand 1986). Wand states that even with the return of multiple locations, Gramarye is classed as a success if the

type error's exact location is within these. As both Gramarye and GHC can report multiple line numbers for one type error, Wand's criterion considers all line numbers returned and not just the first.

3.3.1 Singular Type Error Evaluation

(1) Gramarye is applied to Haskell programs that each contain a single type error. Is there an improvement in locating the errors compared to the Glasgow Haskell Compiler?

As stated in Section 3.3, each program of the first set contains one single type error; if the line number reported matches the 'oracle' response, the result returned is accurate, identical to how the test programs are used by the original curators (Chen and Erwig 2014a). Recall that *delta debugging* works on two configurations; however, it is the relative difference between the two the debugger uses as the final location of the type error and as such is the number reported and compared in the evaluation against the 'oracle'. The graph in Figure 2 shows for all 30 ill-typed programs whether Gramarye and GHC correctly discover the position of the type error. The results are positive. Out of the 30 ill-typed programs, Gramarye accurately locates 23 (77%) of the type errors, compared to 15 (50%) for GHC.

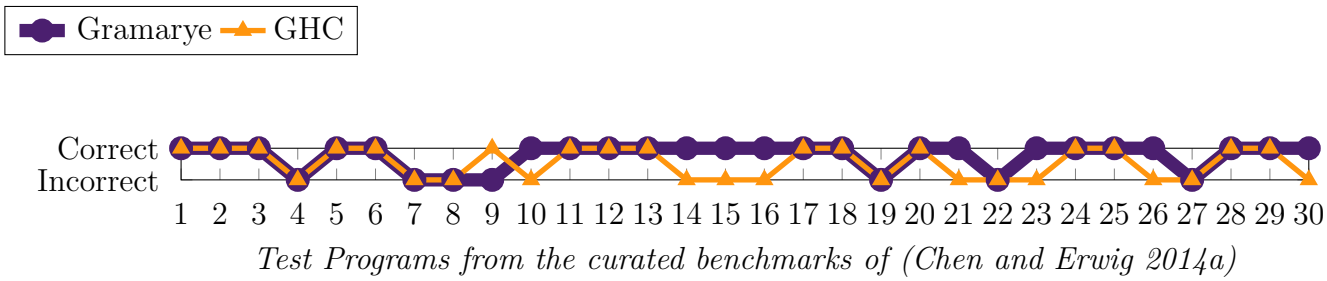


Figure 2: Gramarye vs GHC - Single Type Errors

In some cases, multiple line numbers were returned. The primary cause of multiple line numbers is large expressions, especially those consisting of several keywords, such as If-Then-Else or Let-In expressions.

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

Listing 3.3: Expression over two lines

In this example, Gramarye identifies both lines as the cause of the type error, and GHC wrongly suggests the first line as causing the error. Having a middle ground of reducing the return of many line numbers from these significant expressions and having accurate location results is necessary and entangled within the rest of this thesis's ensuing chapters. The evaluation also drew attention to two individual programs from this set of tests, Program 9 and 24, that needed more investigating and are discussed, along with other discoveries, further in section 3.4.

To summarise, in singular discovery, Gramarye has a 27 percentage point success rate over GHC when locating type errors in the Haskell source.

3.3.1.1 Comparison with the CE benchmark origin

As noted in section 3.3, the test programs that are used to evaluate the debugger in this chapter are a subset of benchmarks collated by Chen and Erwig to evaluate their CF typing strategy in both of their 2014 papers; examined further in the Chapter 9 (Chen and Erwig 2014a,b). Similar to the evaluations in this thesis, part of Chen and Erwigs evaluation checks if the locations returned by their debugger are identical to where the 'oracle' states the type error appears. However, unlike the evaluation in this thesis, Chen and Erwig give their debugger several chances to find the type error by changing the number of suggestions allowed to be taken into consideration. Figure 3 shows their results, as percentages, for 86 test programs using an 'oracle'. 'Never' represent the debugger failing to find a correct location, with the numbers 1 through to 4 showing the number of suggestions allowed to be checked for the result.

Though the evaluation in section 3.3.1 of this thesis only contains 30 test programs compared to the 86 in the figure above, a comparison between the two debuggers is conceivable. Firstly, when looking at the 'never' category, CF Typing fails to find 8.1% of the ill-typed

	86 examples with Oracle				
	1	2	3	≥ 4	never
CF typing	67.4	80.2	88.4	91.9	8.1

Figure 3: Results from evaluating CF Typing (Chen and Erwig 2014a)

programs compared to Gramarye at 23%. However, this smaller result for CF Typing is only available with their maximum number of suggestions. Gramarye, as previously noted, only returns one location each time the debugger runs, and as such, it is fairer to compare it with CF Typing restricted to a singular suggestion. When only allowing one suggestion, Gramarye's has a 77% success rate, a ten percentage point gain, over CF Typing with 67.4%.

Overall, though Gramarye has an advantage over CF Typing when locating type errors, CF Typing's ability to produce more than one suggested location may help when it comes to locating multiple errors. Section 3.4 explores if this ability can be used in Gramarye to locate more than one type error.

3.3.2 Multiple Type Errors Evaluation

(2) Gramarye is applied to Haskell programs that each contain two type errors. Is there an improvement in locating these errors compared to the Glasgow Haskell Compiler?

The evaluation merges singular programs, from the Chen and Erwig benchmark (Chen and Erwig 2014a), for locating multiple type errors, giving programs that each had two self-contained type errors within two separate functions that do not interact. Both functions contain a single type error. The example in Listing 3.4 shows the first function has an error on line 2 and the second function on line 6, but neither type error affects the other;

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

Listing 3.4: Multiple Type Error Example

Listing 3.4 is just one of the programs generated that contain multiple type errors. These were created by merging the CE benchmark programs (Chen and Erwig 2014a). Each set of programs includes the earlier source code with another CE program attached to the bottom, generating 870 new ill-typed programs for testing.

The success criteria for reporting an accurate discovery of the position of a type error in an ill-typed program that contains multiple errors are similar to the criteria for singular errors. The only difference being, that though each program has two type errors, only one error needs to be reported for success.

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

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

In total, Gramarye finds seven percentage points fewer type errors in the multi-error programs than GHC; as previously mentioned, the *Delta Debugging* algorithm restricts Gramarye to always locate just one type error, the first it comes across. Once this error is found, the algorithm assumes the job is complete and does not check any further. However, GHC reports many type error messages giving GHC an advantage over Gramarye. The more error messages reported, the more chance GHC has to report a correct line. Further investigation of locating multiple type errors is found in Section 3.4.

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

Table 1:
Singular program - two type errors

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

Table 2:
Overall testing of two type errors.

3.3.3 Precise Type Error Evaluation

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

Though the success criteria allowed the check of multiple returned line numbers for the correct type error position, reporting many lines to the programmer is not ideal. To return just a singular line number as the cause of the type error was the aim. So an additional evaluation criterion was allowed to pinpoint how specific the Gramarye tool is compared to GHC. All of the programs tested had a single type error on a distinct line; the new rule introduced for multiple errors specified that if either Gramarye or GHC returned a single accurate location, they have a 'precise success'.

The example program below contains one such error on line 4:

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

When GHC compiles this program it returns three locations as being why it is ill-typed[3]:

```
ExampleProgram.hs:1:12: error: ...
1 | intList = [12, 3]

ExampleProgram.hs:2:8: error: ...
2 | zero = 0.0

ExampleProgram.hs:4:33: error: ...
4 | totalOfReciprocals = foldl zero addReciprocals intList
```

Though the last part of the error message does correctly locate the cause, this result is not precise. However, Gramarye, using the same program, returns the more precise, singular, location:

[3]For ease of reading the error messages are contracted

```
The lines that contain the type error: {4}
```

Table 3 shows all the programs with a single type error; a tick denotes if either Gramarye or GHC accurately report a single line number as 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.

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

When evaluating programs that included multiple self-contained type errors, there is a slightly different criteria, judging 'precise success' under the following rules using an example program;

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

```
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
```

- 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
```

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

Table 3: 'precise success' - one type error.

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

Table 4: 'precise success' - two type errors.

- 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' criterion of discovering type errors. Table 4 shows the 'precise success' for both GHC and Gramarye with the original program's name and the percentage of type error locations deemed a 'precise success'.

Analysing Table 4 shows that Gramarye's method is again successful in reporting the correct type error location using just one line number with 41% accuracy compared to GHC with 16%. GHC tends to report as many line numbers as it feels are associated with the type error, much like slicing, a method explained in more detail in Chapter 9.

3.3.4 Efficiency Evaluation

The Gramarye debugger can successfully locate type errors in the Haskell source code; however, it also needs to be efficient. This thesis measures efficiency against run-time; the quicker the debugger takes to locate a type error, the better. Efficiency is an important aspect for any programmer wanting to use the tool, so it needed evaluation against the following questions:

1. How many calls to the compiler are needed?
2. How long does Gramarye take to find the type error?

The evaluation ran on a computer containing an AMD Phenom X4 965, 32GB RAM and a Samsung 850 Solid State Drive whilst running Ubuntu Linux 16.04 LTS. Table 5 shows the evaluated programs, how many lines of code each contains, the 'clock-time' that the programmer experiences, the number of calls the Gramarye debugger makes to GHC, and

the 'clock-time' time for GHC to return the result. On average, Gramarye took 3.359 seconds to provide a location, which meant calling GHC as a blackbox 11 times.

Most 'clock-time' is caused by calling GHC to check the categories for configurations with single type errors. Reducing the number of calls to the *blackbox compiler* would increase the efficiency of the debugger. As Gramayre is working on a line by line basis, there is a risk of producing many unresolved configurations that all need to be compiled. Table 5 shows the category placement of the configurations compiled. In total, the configuration categorisations are 97 well-typed, 85 ill-typed, and 81 containing unresolved errors, such as a parse failure.

Table 6 shows the results have been condensed into averages for each group of programs, denoting the groups by the program number used to generate the programs. The evaluation of programs with multiple type errors has a similar outcome to the single type errors, with GHC calls tightly associated with the debugger 'clock-time'. For example, the worst result received debugging a program with multiple errors took 28.672 seconds and called GHC 110 times. However, on average, the debugger took 4 seconds and 14 calls to return a type error location.

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	5	2.3	6	0.3	3	2	1
2	8	4.7	14	0.3	2	1	6
3	8	3.0	8	0.3	3	4	1
4	8	2.7	8	0.2	4	2	2
5	5	2.7	8	0.2	2	2	2
6	8	3.2	10	0.3	3	2	3
7	6	2.6	8	0.3	5	3	0
8	5	2.2	6	0.2	3	2	1
9	8	2.9	8	0.3	5	3	0
10	10	5.6	16	0.3	5	4	4
11	5	2.7	8	0.3	2	2	2
12	6	2.5	6	0.2	2	3	1
13	6	2.2	6	0.3	3	3	0
14	8	4.3	18	0.2	2	4	8
15	10	4.3	20	0.2	2	4	8
16	8	4.3	14	0.2	4	3	4
17	5	2.4	6	0.3	4	2	0
18	7	2.1	6	0.2	3	2	1
19	20	9.4	44	0.3	3	6	11
20	8	2.6	8	0.3	4	2	2
21	7	2.8	8	0.3	4	3	1
22	12	2.8	8	0.3	3	3	2
23	8	4.0	14	0.2	4	3	4
24	6	2.7	8	0.2	3	4	1
25	7	4.1	14	0.3	3	2	5
26	7	6.8	24	0.3	4	4	9
27	5	2.2	6	0.3	4	2	0
28	6	2.0	6	0.3	3	3	0
29	5	2.5	8	0.2	2	2	2
30	5	2.2	6	0.3	3	3	0
	7(211)	3.4(101)	11(330)	0.3(7.9)	3.2(97)	2.8(85)	2.7(81)

Table 5: Efficiency on programs with one type error. Average and (Totals).

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	11	3.4	11	0.3	3	4	2
2	14	5.3	18	0.3	4	5	6
3	14	3.7	12	0.4	3	5	3
4	14	4.0	14	0.3	3	4	4
5	11	3.5	12	0.3	3	4	3
6	14	4.1	15	0.3	3	4	4
7	12	3.3	11	0.3	3	4	2
8	11	3.1	11	0.3	3	4	2
9	14	4.0	14	0.3	3	4	4
10	16	4.4	16	0.3	3	4	5
11	11	3.2	11	0.3	3	4	2
12	11	3.4	11	0.3	3	4	2
13	12	3.3	11	0.3	3	4	2
14	12	3.9	13	0.5	2	4	4
15	15	4.9	17	0.5	26	4	6
16	14	5.2	19	0.3	4	4	6
17	11	3.4	11	0.3	3	4	2
18	13	3.1	11	0.3	3	4	2
19	26	9.4	34	0.4	3	6	14
20	14	3.6	12	0.3	3	4	3
21	13	3.4	11	0.3	4	4	2
22	18	4.3	15	0.3	4	5	4
23	14	5.2	18	0.3	4	4	6
24	12	3.1	10	0.3	3	4	2
25	13	3.3	11	0.3	3	4	3
26	13	3.2	11	0.3	3	4	3
27	11	3.3	11	0.3	3	4	2
28	12	3.3	11	0.3	3	4	2
29	11	3.7	12	0.3	3	4	3
30	11	3.4	11	0.3	3	4	2
Average	14(455)	3.4(118)	13(405)	0.4(9.6)	3(117)	4(125)	4(111)

Table 6: Efficiency of programs with two type errors. Average and (Totals).

3.4 Multiple Type Error Discovery

The investigation in this chapter drew attention in the evaluation to two different problems with the Gramarye debugger. Firstly two individual programs needed more investigating, and secondly, the 81 calls to the blackbox compiler caused by Unresolved configurations in Table 5. The latter is addressed in Chapter 4, while the former is next for discussion.

Initially, Gramarye failed to discover a type error in program 24, yet GHC did it successfully. On inspection, program 24 contained a mistake in the original benchmark programs. The program contained two type errors and not the singular expected error. Removal of one of the type errors returned the results expected, with both the debugger and GHC successfully locating the error. Due to this anomaly, a modified Gramarye checked all programs for multiple type errors. Program 9 also contained two errors. Unfortunately, once fixed, the debugger was no longer successful in finding the type error due to Program 9 containing an unnecessary call to a variable bound to foldl.

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

Listing 3.5: Program 9

Here, Gramarye returns line 1 as faulty, yet the type error is on line 5 where the variables 'zero', and 'addReciprocals' should swap. Instead, if foldl is used directly rather than via a variable, the debugger then finds the correct broken line number.

The fact that an unknown type error made such a stark difference in the results caused a rethink in the earlier discussion in this chapter about Gramarye's restriction of locating one type error at a time (Section 3.3.2). Though this thesis argues that programmers should fix a single error before moving on to the next, which allows for a preference of an accurate location over a broad suggestion, there was an awareness that this thesis needs to investigate this option to see if the argument was not unfounded. Furthermore, this chapter needed

to show if Gramarye could be extended to report multiple type error locations, as ill-typed programs commonly have several.

3.4.1 A Multiple Type Error Solution

The solution initially proposed in this chapter discovers a singular type error in an ill-typed program. However, the evaluation did include multiple errors as the Glasgow Haskell Compiler (GHC) reports more than one error at a time. The outcome of locating multiple errors found that Gramarye had on average 67% location discovery compared to GHC at 74%. The evaluation also discovered that when applying Gramarye to a benchmark known as 'Program 9', presented in Listing 3.5, and 'Program 24' that contained multiple unknown errors, there is a significant difference in results as discussed in Section 3.4. Due to these findings, a brief sub-investigation commenced.

As described in Section 2.2, Delta Debugging works by splitting configurations into pieces and uses the results from a testing function to decide the pieces' subsequent split. If the algorithm receives a Fail, it will continue to split those pieces further until it can no longer do so. Therefore, following a singular Fail's pathway is not an issue when using the Simplifying version. However, as Isolating Delta Debugging works on two configurations in certain circumstances, as with multiple errors, both configurations can return a Fail from the testing function. If two Fail results occur, the original Isolating Delta Debugging algorithm will ignore one of the available pathways causing the location of any other error to be lost and not appear in the end results. Therefore, the algorithm needs modification to add additional choices to enable the secondary Fail result to be accepted and support the returning of multiple error locations. Recall the presentation of Isolating Delta Debugging, algorithm 1, in Section 2.2.2, algorithm 2 shows changes to the inner `while` loop that represents the 'choices' delta debugging can make on lines 15-18 and 22-25. Also recollect that in the type error debugging domain `nextCFail` is the ill-typed program and `nextCPass` is the well-typed program.

Here added are two extra choices. Each deals with results from the testing function that are identical. First is when the result is two Passes, and the second when receiving two Fails. A short example now follows to justify these new choices.

Algorithm 2: New choices within Isolating Delta Debugging

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

3.4.1.1 Small Choices Example

Let us start *isolating delta debugging* with the passing configuration {} and the failing configuration {`1,2,3,4,5,6,7,8,9,10,11,12`}. Our passing configuration, is a subset of the failing. The algorithm divides the difference between the two configurations by two and hence test the configurations {`1,2,3,4,5,6`} and {`7,8,9,10,11,12`}. The first configuration gives the outcome Fail and the second Unresolved. The algorithm follows the Fail outcome and again divides by two giving the configurations {`1,2,3`} and {`4,5,6`}. This time both Fail. When the choice calls to branch the search it calls another instance of delta debugging and both configurations continue to divide separately as if calling the debugger on a different program. The first produces {`1,2`} and {`3`} with the outcome of Pass and Fail, whilst the second gives {`4,5`} and {`6`} with the same results. As neither of the branches can continue, each algorithm terminates. The debugger then collates the results and gives them to the programmer showing that there are two different errors, one on line three, {`3`}, and the other on line six, {`6`}.

3.4.1.2 Mini-Evaluation

Table 7 shows the results of a brief evaluation of the changes just described. The first two columns are a copy of Table 2 placed here for ease of comparison. The third column shows the results of the modified debugger, named Gramarye.v2 (G.v2). Here it is seen that Gramarye.v2 locates correctly 95% of the type errors compared to GHC and the original debugger, which found 67% of the type errors in the benchmarks. However, unfortunately, these results have a problem; more calls to the testing function occur to locate multiple type errors. Gramarye.v2 now takes on average 12 seconds to locate type errors compared to the original debugger at 4 seconds. This rise in time is due to the increased calls to the testing function, with the blackbox compiler receiving an average of 18 more requests than previously. Due to the evaluation outcome here and in Section 3.3, it is clear that a reduction of calls is necessary. Therefore, the following chapter, Chapter 4, covers this issue, leading onto the rest of the thesis, which concentrates on the type error debuggers ability to scale and its speed, which both rely on decreased compiler calls.

Program	Gramarye	GHC	G.v2
1	69%	100%	100%
2	62%	100%	93%
3	72%	97%	90%
4	72%	52%	93%
5	66%	100%	100%
6	72%	100%	97%
7	62%	52%	97%
8	66%	52%	100%
9	72%	55%	93%
10	62%	52%	86%
11	62%	100%	100%
12	66%	100%	100%
13	62%	100%	100%
14	76%	52%	100%
15	90%	48%	79%
16	62%	52%	90%
17	69%	100%	100%
18	69%	100%	97%
19	79%	21%	79%
20	66%	100%	97%
21	79%	45%	97%
22	38%	52%	86%
23	66%	52%	90%
24	59%	100%	100%
25	62%	100%	97%
26	69%	55%	100%
27	62%	52%	100%
28	69%	100%	100%
29	62%	100%	100%
30	62%	52%	100%
Average	67%	74%	95%

Table 7: Overall testing of programs with two type errors.

Chapter 4

The Pragmatics of Type Error Debugging

Two clear research paths formed from evaluating the Gramarye type error debugger in Chapter 3. The first path is in section 3.4 and discussed the issue with a program containing multiple type errors and a solution to fix it. The modification was a success; however, further improvements make it a topic for future work. The second path concerns itself with the time taken for the debugger. As the debugger works with a blackbox compiler, compilation speed restricts it; the more calls, the more time is taken.

In general, the run-time of delta debugging is proportional to the number of tests made[1]; this applies to type error debugging as well, with nearly all run-time spent by the compiler returning test results. As Delta Debugging is agnostic, it does not have any syntactic knowledge of the program. Due to this lack of knowledge, all lines in an ill-typed program are affected by the algorithm, producing two issues. The first issue is redundant calls to the compiler. These calls involve lines of code that will not change the outcome of the compilers results. Adding and removing a line containing a comment will increase the compiler calls; however, it will cause no change to the algorithm's next choice. The second issue is the unnecessary generation of unresolved results. This generation happens when adding and removing lines that contain an essential construct needed for the program to work. These constructs never contain the type error and so allowing them to remain intact will reduce

[1]This assumes similar run-time for every test, which may not be the case.

unresolved outcomes. Thus the hypothesis is that removing these categories of calls to the blackbox compiler will reduce the time taken to locate type errors. This chapter will discuss unresolved results first, introducing the need for benchmark programs with more source code, compared to the previous chapters smaller examples, and then redundant calls afterwards. Followed by an evaluation, using a new set of modular benchmarks, to show that removing these calls is less expensive in time than keeping them.

4.1 Unresolved Results

As can be seen from the description of delta debugging in Section 2.2, 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 et al. parts and make more tests. If every configuration is unresolved, the algorithm starts to generate configurations that contain a single line until all lines of the program have been checked.[2] The *isolating delta debugging* algorithm has logarithmic time complexity if no outcome is unresolved and becomes less efficient, up to quadratic time complexity, with many unresolved outcomes. Therefore any successful application of delta debugging makes some effort to avoid unresolved outcomes.

The issue with many unresolved outcomes can be shown more clearly within the studies earlier results found in Chapter 3 Section 3.3. These 900 programs, which includes the original 30, were generated by concatenating pairs of some of the original small CE benchmark programs. For ease of reading, the ordering of the 900 programs are by the number of lines and placed into four groups: the shortest 225 in the first group, the next 225 in the second group, etcetera. Table 8, along with the graphical representation, shows the average outcomes. It indicates that the number of unresolved results grows faster than the linear number of lines; the larger the program, the worse the issue is with unresolved results.

However, these ill-typed programs are short. The longest program in the CE benchmark suite (Chen and Erwig 2014a) of 121 programs has just 23 lines. Such programs are suitable

[2] A proof is available in Appendix A.1 of 'Why Programs Fail' (Zeller 2009)

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

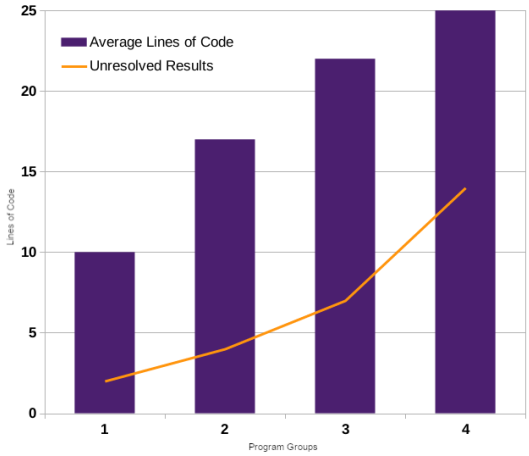


Table 8: Average number of unresolved outcomes compared to number of lines of code.

for studying how a type error debugger works, and many of these programs are representative of the first programs written by novices learning a functional programming language. However, they do not show us how a type error debugger will scale as not just novices need help with type error debugging but also more experienced functional programmers who build useful, real-world programs.

4.2 Modular Programs and Unresolved Results

In the previous section, short, non-modular - those that do not import other modules, ill-typed programs show an increase in run-time the more unresolved results received. However, in October 2019 I measured the top 100 Haskell programs on the popular public repository GitHub[3]. On average, each program has 31872 lines of code, 138 modules, and 229 lines of code per module, far from the 23 lines mentioned. During the same GitHub investigation, I also found that the programs were all modular. To support these "real world" modular programs, the type error debugger has to also work with Cabal[4], which packages and builds them. To gain this functionality, a flag to call the build tool Cabal instead of the Glasgow Haskell compiler is necessary. Note that the user has to state the target program instead of the ill-typed module when using Cabal. The type error debugger in this chapter assumes

[3] `https://github.com/search?l=Haskell&q=Haskell&s=stars&type=Repositories`
[4] `https://www.haskell.org/cabal/`

that the first module identified by the compiler as ill-typed does contain the type error location; the type error debugger works solely on that module. If a module causes the first compiler type error, all modules directly or indirectly imported are well-typed. The location produced will be from within that first module; there is no support for mutually recursive modules. An identifier defined in an imported module may have a type that contradicts with the identifiers usage 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 the treatment of modules is consistent with the general treatment of definition vs. use.

It is syntax like that which is used to import modules that can cause more unresolved calls. The syntax causes the generation of unresolved results when removed from Haskell programs. Here the constructs are split into two categories: single lines and multiple lines. An example of the syntax for single-line `import` comprises of:

```
16 import Control.Exception (throw)
```

whereas the multiple line category contains multi-line comments, syntax that is split across more than one line, denoted in Haskell between {- -},

```
2 {- |
3   Module     : Text.Pandoc.Filter.Lua
4   Copyright  : Copyright (C) 2006-2019 John MacFarlane
5   License    : GNU GPL, version 2 or above
6
7   Maintainer : John MacFarlane <jgm@berkeley@edu>
8   Stability  : alpha
9   Portability : portable
10
11 Apply Lua filters to modify a pandoc documents programmatically.
12 -}
```

and syntax from the single line category that due to programmer style preferance may span multiple lines:

```
21 import Text.Pandoc.Lua (Global (..), LuaException (..),
22                         runLua, runFilterFile, setGlobals)
```

A configuration generated with either a singular line or part of a multiple-line removed will cause an unresolved result and, as such, will increase the time taken by the debugger.

For example, Gramarye will now be applied to some modules of the program Pandoc, the most popular program on GitHub, to test if this causes issues with larger programs. As already stated, the Gramarye debugger works with the Glasgow Haskell Compiler as the blackbox compiler, and as such, examples outlined next will be Haskell specific.

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 module in 150 modules. Here two of the modules, DokuWiki and Lua, are made ill-typed to show the consequences of large numbers of unresolved results on large programs.

First, DokuWiki. This module has 534 lines of codes and the following type error, replacing the initial String with a Boolean, is placed on line 220:

```
220 return $ "<HTML><ul></HTML>\n" ++ vcat contents ++ "<HTML></ul></HTML>\n"
```

```
220 return $ True ++ vcat contents ++ "<HTML></ul></HTML>\n"
```

Applying this ill-typed program to Gramarye returns the following results:

Module	Lines of Code	Time (Minutes:Seconds)	Pass	Fail	Unresolved	Located
DokuWiki	534	117m30s	0	158	4862	✓

The table shows that the Gramarye debugger took almost two hours to discover the type error successfully. It also shows an astonishing 4862 unresolved results, much higher than the average of 14 from the small programs in Table 8.

The second module, Lua, is much smaller than DocuWiki at 67 lines of code long. An extra choice in the case statement, at line 35, makes it ill-typed.

```
34 let format = case args of
35                (x:_) -> x
36                _     -> error "Format not supplied for Lua filter"
```

```
34 let format = case args of
35                (x:_) -> True
36                (x:_) -> x
37                _     -> error "Format not supplied for Lua filter"
```

Again after running Gramarye on this ill-typed program the result returned are:

Module	Lines of Code	Time (Minutes:Seconds)	Pass	Fail	Unresolved	Located
Lua	67	7m16s	0	22	318	✓

With 318 unresolved results for just 67 lines of code, the debugger, on average, is generating almost five unresolved results per line. The number of unresolved results, which collate to compiler calls, also clearly impacts the length of time, taking an unsatisfactory 7 minutes and 16 seconds to return the location of the type error. However, what these two investigative modules show is a viable solution to reducing unresolved results. Nearly half of the Lua modules 67 lines of code cannot contain a type error, yet their removal almost always causes unresolved results.

4.3 Redundant Results

As stated at the beginning of this chapter, it is not only Unresolved results that cause extra calls to the compiler. Redundant calls are configurations that, when sent to the blackbox compiler, do not cause a change in the outcome. Two such syntactic examples are lines that contain only comments or are empty. In Haskell, two dashes, `--` denote a comment, and an empty line is just any line that contains no data at all. Still using the Lua module from Pandoc, both an empty line on 24 and a line containing a comment, on line 25, are seen in the excerpt:

```
24
25 -- | Run the Lua filter in @filterPath@ for a transformation to the
```

Generating a configuration with just these two lines added and sending it to the compiler would be redundant. The outcome would not change. A redundant call could contain only this syntax or the removal of such lines. Such as:

```
23 import Text.Pandoc.Options (ReaderOptions)
24
25 -- | Run the Lua filter in @filterPath@ for a transformation to the
```

and the removal of line 25

```
23 import Text.Pandoc.Options (ReaderOptions)
24
25
```

will return the same result.

4.4 A Solution

The solution works on both the unresolved and redundant results identically by providing a checklist of syntax to the *Delta Debugging* algorithm. The checklist stops the algorithm from removing any required syntax when creating a configuration variant. To show an example of how applying the syntax checklist works, below is a sub-section of a module from Pandoc called "slides". The example has a type error on line 29; `True` should be a `Bool` and not a `Char`:

```
15 import Prelude
16 import Text.Pandoc.Definition
17
18 -- | Find level of header that starts slides (defined as the least header
19 -- level that occurs before a non-header/non-hrule in the blocks).
```

```
20 getSlideLevel :: [Block] -> Int
21 getSlideLevel = go 6
22   where go least (Header n _ _ : x : xs)
23                  | n < least && nonHOrHR x = go n xs
24                  | otherwise               = go least (x:xs)
25         go least (_ : xs) = go least xs
26         go least [] = least
27         nonHOrHR Header{}     = False
28         nonHOrHR HorizontalRule = False
29         nonHOrHR _            = "True"
```

Recall that this initial version of the ill-typed program becomes our 'Failing' configuration, and from this the debugger generates a 'Passing' configuration. In Chapter 3 that 'Passing' configuration would be empty, however, now with the syntax checklist it generates the following:

```
15 import Prelude
16 import Text.Pandoc.Definition
17
18 -- | Find level of header that starts slides (defined as the least header
19 -- level that occurs before a non-header/non-hrule in the blocks).
20
21
22
23
24
25
26
27
28
29
```

Here when generating the 'Passing' configuration, only lines that do not match the syntax

checklist are removed. Identical to Chapter 3, parsing is still not duplicated. The debugger only checks each line for key terms such as *import*. For syntax that flows over multiple lines, the solution matches two key terms, the beginning and the end. All lines between the two key terms are part of the multi-line syntax; thus, they are also not removed. However, though the lines, both singular and multiple, are not removed from the configurations, the algorithm still works as previously. It divides configurations as if nothing has been modified, and as such, two aspects of this solution should be noted. First, the solution does not change the correctness of the result; it only improves the efficiency of the run-time. Second, the debugger never returns; as a result, any of the lines marked as matching the syntax checklist, meaning no superfluous lines are presented to the user.

For example the result, from the debugger, of the above example would look like this:

```
The lines that contain the type error: {29}
```

Embedding the solution into Gramarye led to the following changes to the Lua modules results:

Module	Lines of Code	Time (Minutes:Seconds)	Pass	Fail	Unresolved	Located
Lua	67	3m50s	6	22	130	✓

Here, the unresolved results decreased significantly from 318 to 130, a reduction of 188 blackbox compiler calls, with the time taken to debug dropping by over four minutes, from 7 minutes and 16 seconds to 3 minutes and 50 seconds. This brief test gave excellent results, and so a further evaluation using 20 Pandoc modules follows.

4.5 Evaluation

This evaluation asks one question: Does the introduction of heuristics decrease the unresolved results and, thus, the time taken to locate type errors? Firstly, to answer this question, two versions of Gramarye now exist. The first without the heuristics, identical to the version introduced in chapter 3, and now called Gramarye-Without (G-W/O), and the second with heuristics, called Gramarye-With (G-W). To test using Pandoc, both versions of Gramarye

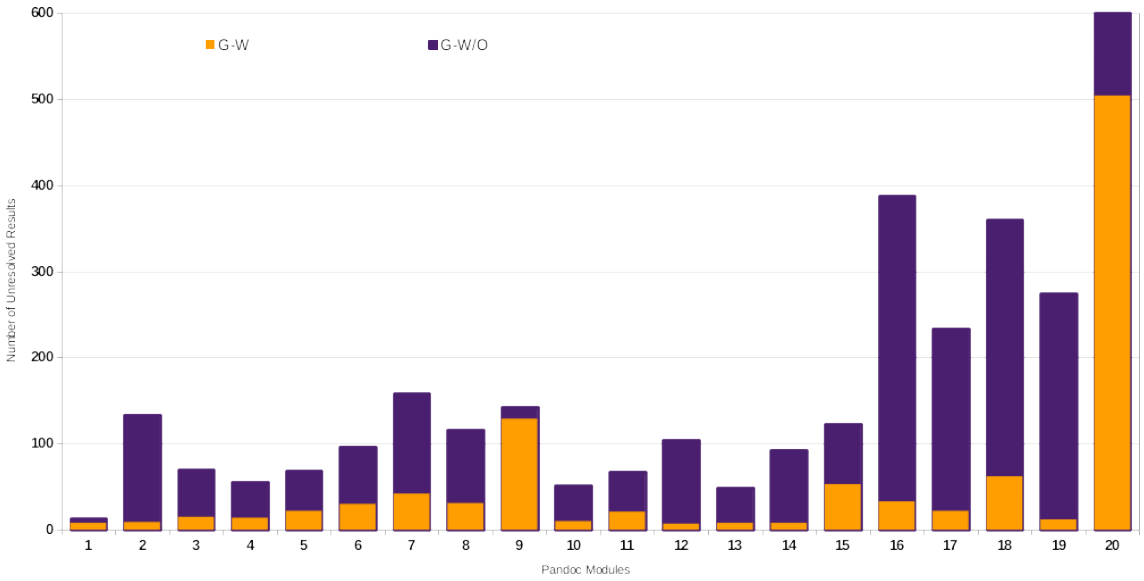


Figure 4: G-W and G-W/O - Unresolved Results

now support the build tool Cabal. More information about this change is found in chapter 6. Lastly, the hardware used for evaluation is the same as described in section 3.3.4 of the previous chapter.

The 20 Pandoc modules are between 32 and 2306 lines long. Each module contains a singular type error manually inserted into the source code, which both debuggers endeavour to locate. Figure 4 shows the total number of unresolved results from the evaluation. Along the x-axis is the module tested, listed in length order; module 1 has 32 lines up to module 20 with 2306 lines of code. The y-axis represents the number of unresolved results. For ease of reading the graph, there is a capping of the y-axis at 600 unresolved results. However, note that module 20 with the Gramarye-Without debugger surpasses this with a total of 5100 unresolved results and appears in more detail in Figure 5.

Overall it is clear to see from figure 4 that applying the heuristics dramatically decreases the number of unresolved outcomes on every module. On average, the modification removed 333 unresolved results per module and, in total, went from 7700 unresolved to 1040. As already stated, module 20 has the largest number of results with 5100 unresolved when using the Gramarye-Without; however, it also had the largest decrease with 4596 calls to the compiler not occurring. The most negligible difference happened with module 1; our most minor

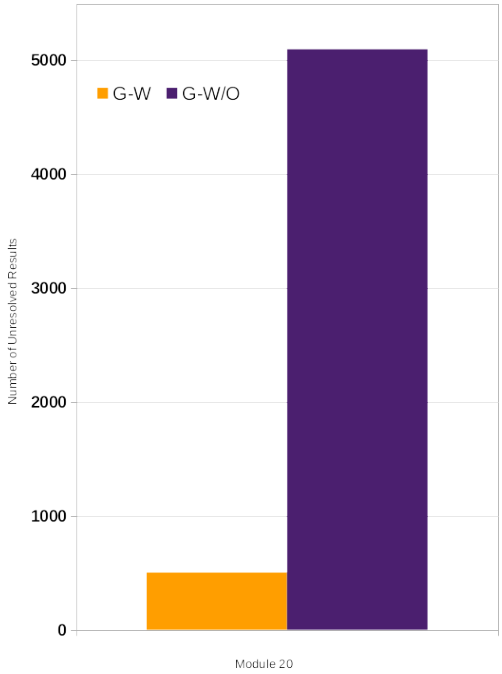


Figure 5: Module 20 - Unresolved

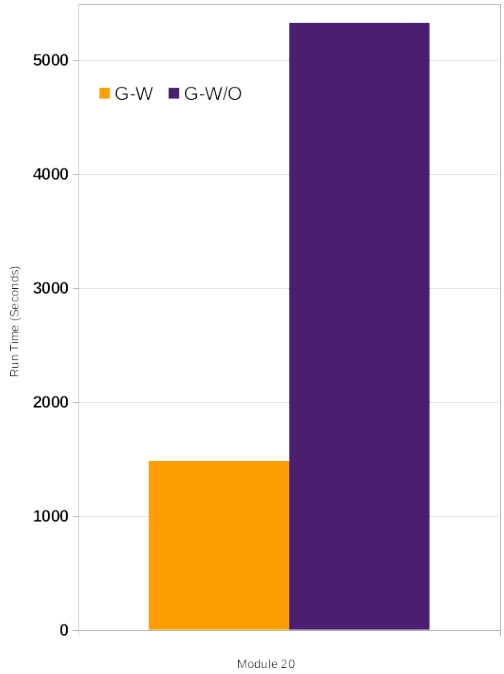


Figure 6: Module 20 - Run-Time

program, with only five unresolved results differences, went from 13 with the Gramarye-Without and 8 with Gramarye-With. Nevertheless, even with these lesser differences, an improvement in the time taken appeared.

In figure 7 the results of the length of time both debuggers took to locate the type error. Again, the x-axis represents the modules in length order; however, the y-axis now presents the time in seconds. Like the previous figure, there is a cap, placed at 2000 seconds, and module 20 exceeds this by over 3000 seconds at 5331 seconds; a closer look at this module appears in the separate figure 6.

Similarly to the unresolved results, the run-time decreases significantly, on average, by 4 minutes 16 seconds (256 seconds). Again the more minor and more impressive differences are held by modules 1, the smallest, and module 20, the largest. The heuristics with Module 1 reduced the time by only 3 seconds; however, with module 20, they removed over an hour to locate the type error.

In all, this evaluation shows that the solution can significantly improve the run-time of the debugger. This argument is not new to this thesis, pointed out in the chapter of related work, others already stated that making Delta Debugging non-agnostic can improve results

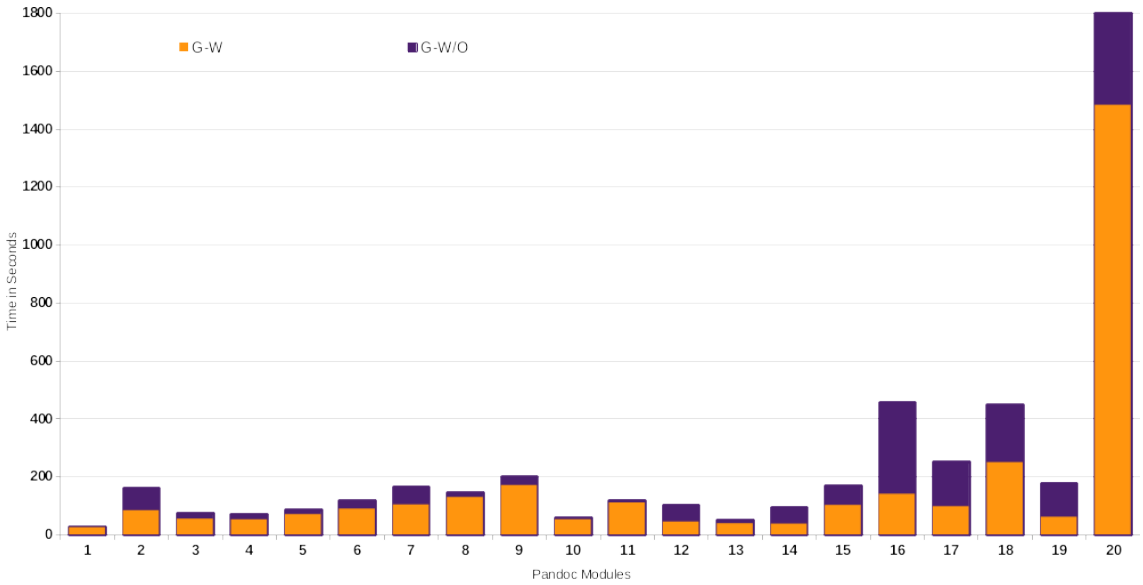


Figure 7: G-W and G-W/O - Run-Time Results

(Stepanov, Akhin and Belyaev 2019). However, in chapter 8 this thesis will show a solution to having both heuristics and agnostic features combined, and so the improvements made in this chapter will continue forward into future debuggers that use delta debugging and a blackbox compiler in this work. Though the solution in this chapter gave a positive result, it is clear that unresolved results still persist; in total, Gramarye-With still returned 159 unresolved results. Noted in this investigation is that breaks in groups of lines, multi-line syntax such as comments will always cause an unresolved result. The solution covered those multi-lines that contained needed syntax only, such as import statements, ignoring the rest. Chapter 6 looks at these uncategorised unresolved results, investigating the leading cause and introducing one viable way to fix it. However, first, the next chapter investigates a new framework for evaluating type error debuggers.

Chapter 5

An Evaluation Framework for Type Error Debugging

Within type error debugging evaluations, only the metric *Recall* - whether a type error has been located correctly or not, run-time and the author's personal goals are deemed important (Seidel, Jhala and Weimer 2016; Lerner, Grossman and Chambers 2006). However, I cannot entirely agree with using only the *Recall* metric. On its own *Recall* is an unsatisfactory measure that can give biased results and unintentionally hide the quality of a debugger, and in later works, authors seem to agree (Seidel et al. 2017; Zhang et al. 2015b). Nevertheless, though researchers are slowly seeing other metrics joining *Recall* in type error debugging evaluations, they are not representing the same formulas. I propose the following as a framework for future evaluations to allow for ease of solution comparison, and apply it to the rest of the evaluations in this thesis.

5.1 The Metrics

In data science, using model metrics such as Accuracy, Precision, and Recall are an accepted standard (Witten and Frank 2005; Shung 2019). Data science uses the terms True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) to define the model metrics. Table 9 shows an overview of the data science terms in the context of type error debugging with new terminology given to allow for distinction between the two.

Our Terms	Meaning	Data Science Equivalent
R_E	Reported Errors (correct errors)	TP
U_L	Unreported Lines (correct unreported lines)	TN
I_E	Reported Errors (incorrect errors)	FP
I_L	Unreported Lines (incorrect unreported lines)	FN
R_L	Reported Lines (lines returned)	TP + FP
L	Lines of code (total source code)	TN+TP+FN+FP
E	Errors (errors in the code)	TP+FN

Table 9: Terminology

As already stated, the terms define the model metrics, Accuracy, Precision, and Recall. Application of the model metrics happens on the type error debugger results, typically during evaluation, along with information, such as the number of lines of code, from the initial ill-typed program; below is a description of this process for each metric, explicitly concentrating on type error debuggers in this thesis.

Accuracy tells us the typical distance from a measure to the optimum value. For the type error debugging domain, this represents the number of lines correctly excluded plus correctly reported lines containing a type error. When applied to type error debuggers in this thesis, *Accuracy* first takes the total lines of source code from the initial ill-typed program (L). Next, it checks the results of the solution, the relative difference between the passing and failing configurations, for which lines are returned as the result. As the evaluations in this thesis have an *oracle*, the position of the type error is known prior; the metric can distinguish between correctly reported lines (R_E) and correctly unreported lines, i.e. all lines that have not been reported (U_L). However, this is problematic as results may represent many True Negative answers, the number of correct lines not reported. So this is generally ignored in the type error debugging domain in favour of *Recall*.

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} = \frac{U_L + R_E}{L} \tag{1}$$

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

$$Recall = \frac{TP}{TP + FN} = \frac{R_E}{E} \tag{2}$$

For type errors, this measures the number of errors that are reported correctly (R_E), compared to the number of errors within the source code (E). As already noted, this metric is most used in type error debugging evaluations. It shows if a debugger can successfully discover the correct number of type errors within an ill-typed program. Again, to apply this metric, certain information is required. Necessary is the knowledge, or *oracle*, of how many type errors the source code contains and on what line they are positioned. The result from running a type error debugger from this thesis is a list of lines represented by (R_L). As the *oracle* knows where the type errors lay and how many exist, it is possible to get False Negative results (I_R), a type error is not reported at all, as well as True Positive (R_E). However, like *Accuracy*, it is not without fault, as the following example will show.

Let us assume having an ill-typed program containing eight lines (L=8) and 1 type error (E=1). Running a debugger returns all eight lines of code as containing the type error (R_L=8) and returns the correct line location within this (R_E=1). Most type error debugging evaluations do not mention the number of lines returned, only if their debugger located the line correctly. Using *Recall* as the only metric in evaluations, I can state that this example shows the debugger is 100% correct.

$$Recall = \frac{R_E}{E} = \frac{1}{1} = 100\% \tag{3}$$

This result is incorrect, yet the metric proves it to be true. To counter this issue, Data Science employs another metric.

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

$$Precision = \frac{TP}{TP + FP} = \frac{R_E}{R_L} \tag{4}$$

Mapping to the type errors domain, this is the number of correct lines of code reported by the debugger (R_E), compared to the total number of lines returned (R_L). Precision allows us to see if the debugger has returned the correct location as one line versus a correct location within several lines.

Applying Precision to the ongoing example, the results are:

$$Precision = \frac{R_E}{R_L} = \frac{1}{8} = 12.5\% \tag{5}$$

As can be seen, this is a significant difference from the results of *Recall*. However, it is also not practical to use Precision as a singular metric due to its reliance on False Positives (I_E), meaning some of the lines returned does not contain a type error; this is where the Data Science domain employs the F_1 Score.

F_1 Score is calculated from the harmonic mean of the two metrics *Recall* and *Precision*. The F_1 Score produces an accuracy measure that accounts for the imbalance of data within type error debugging, meaning the F_1 Score is crucial in showing the real results of evaluations.

$$F_1 = 2\frac{Precision \cdot Recall}{Precision + Recall} = 2\frac{R_E}{E + R_L} \tag{6}$$

Now with the example, meaningful feedback for evaluation appears.

$$F_1 = 2\frac{R_E}{E + R_L} = 2\frac{1}{1 + 8} = 22\% \tag{7}$$

The framework laid out in this chapter provides a new, more thorough way of evaluating type error debuggers. The framework primarily concentrates on making evaluations fairer by expanding the metrics to cover more than Recall, which an example shows can cause confusing and bias results. In the future, comparisons with previous evaluation frameworks would be advantageous. However, here, as evidence that this framework can generate easily comparable evaluations for future work in the type error debugging domain, the rest of the evaluations in this thesis apply it. The first application of the framework comes in the next chapter, which introduces a new algorithm to remove the excess unresolved results discussed in Chapter 4.

Chapter 6

The Moiety Algorithm

In chapter 4 the unresolved results are successfully reduced. However, in the discussion, it is noted that there were still many unresolved results left. This chapter investigates these unresolved results, showing what produces them, and introduces an algorithm to remove the cause.

6.1 Previous Results

In an earlier chapter, 3, a type error debugger, Gramarye, is presented and evaluated. Recall that this debugger implements the *isolating delta debugging* algorithm (Zeller 2009) to locate the defective line in an ill-typed program. Gramarye works solely on a line-based principle, directly adding and removing the lines of the source code to generate configurations. These configurations, that is, variants of the ill-typed program, are tested by calling the compiler. Gramarye does not duplicate compiler work such as parsing; instead, it uses minimal information from the outcome of the compiler call. In particular, the only information Gramarye uses is whether compilation succeeded (passed), failed with a type error (fail), or failed with some other error (unresolved). As a consequence, such a debugger is mostly programming language agnostic.

As shown previously, Gramarye yields good locations in a reasonable time for a benchmark sample of 121 ill-typed programs. However, unlike delta debugging of run-time failures,

which Zeller evaluated with large programs, successfully finding a fault in a 178,000 line program, all the programs in the benchmarks are short; the longest has 23 lines. So for many type error debugging methods proposed in the literature that use this and other benchmarks, including Gramarye in chapter 3, it is unknown from their evaluations whether they scale for larger programs. Unfortunately, the larger the program, the more unresolved results, an issue that increases debugging time. To counter this, chapter 4 introduced a set of heuristics and then evaluated them using the large program, Pandoc. The modules tested from Pandoc are between 32 and 2306 lines of code; thus, the solution could scale for the 20 modules evaluated and successfully reduced unresolved results. However, many unnecessary unresolved results remain, still slowing the debugging run-time, and they are due to applying delta debugging to lines of code.

6.2 Brief example of the line-based problem

As Gramarye is line-based, it is affected by where the *isolating delta debugging* algorithm chooses to split the source code. The *isolating delta debugging* algorithm tests a logarithmic[1] number of configurations if no outcome is unresolved. For example, an ill-typed program containing just one line will immediately locate the fault on that line from the first configuration. In contrast, an ill-typed program containing six lines of code can take three configurations to locate the type error. However, as previously said, the type error debuggers in this thesis do not duplicate the compiler's parser. Every line combination can be a possible configuration; this has the detrimental effect of causing many ill-formed configuration variants, producing a significant number of unresolved results as they do not parse.

Take as a brief example this Haskell program from Stuckey et al. (Stuckey, Sulzmann and Wazny 2004) that is used in chapter 3:

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

[1] Concerning the number of lines of the original ill-typed program.

The program is ill-typed. The first line is incorrect; the x should be a list containing the single element x. The Glasgow Haskell compiler[2] gives us line 2 as the incorrect line, whereas Gramarye correctly points out line 1. However, even in this three-line program, the *isolating delta debugging* algorithm still produces unresolved results. For example, the following configuration returns a parse error:

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

The more outcomes are unresolved, the less efficient *isolating delta debugging* 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" (Zeller 2009).

6.3 Initial investigation

As already stated, 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 this suspicion, the most popular software from the investigation of Github results, Pandoc, is chosen again, to initiate the studies *scalability* benchmarks (Sharrad 2021b). As an initial test, I introduce a single type error in a single module. The ill-typed module has 87 lines, and the debugger returns 126 unresolved outcomes:

Pass	Fail	Unresolved
5	11	126

[2] `https://www.haskell.org/ghc`, version 8.4.3

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 10: Number of error messages giving unresolved outcome.

Table 10 details each category, grouped by the error message of the Glasgow Haskell Compiler, an unresolved result is placed into. The root cause of most unresolved outcomes is parsing errors, and "parse error on input" is by far the most frequent one.

Building some kind of parser for the debugger would contradict the objectives of this thesis. Hence, I present a new algorithm that calls the compiler as a blackbox. The algorithm produces a set of all configurations of the original program that consists of consecutive lines that should not be split. Recall that the split is of entire lines from each other and not a splitting of a single line itself. Splitting any of these configurations will produce a parse error. In summary, the information guides the *isolating delta debugging* algorithm to reduce unresolved test outcomes and thus reduce the time taken for the algorithm to run.

6.4 The Moiety Algorithm and Delta Debugging

Thus far, the type error debuggers in this thesis always obtain a configuration that does not parse if it splits the original ill-typed program at certain consecutive lines. Given its dominance, this investigation solely focuses on the "parse error on input" error message. These indicate that parsing failed at the beginning of a line in the configuration, whereas, for example, "parse error (incorrect indentation or mismatched brackets)" indicates that parsing fails at the end of the configuration. Concentrating on the former means the new algorithm can distinguish between the two. So the algorithm uses this information first to

determine which lines should never be separated as they will cause a "parse error on input" and then applies the delta debugging algorithm such that it never splits in these places.

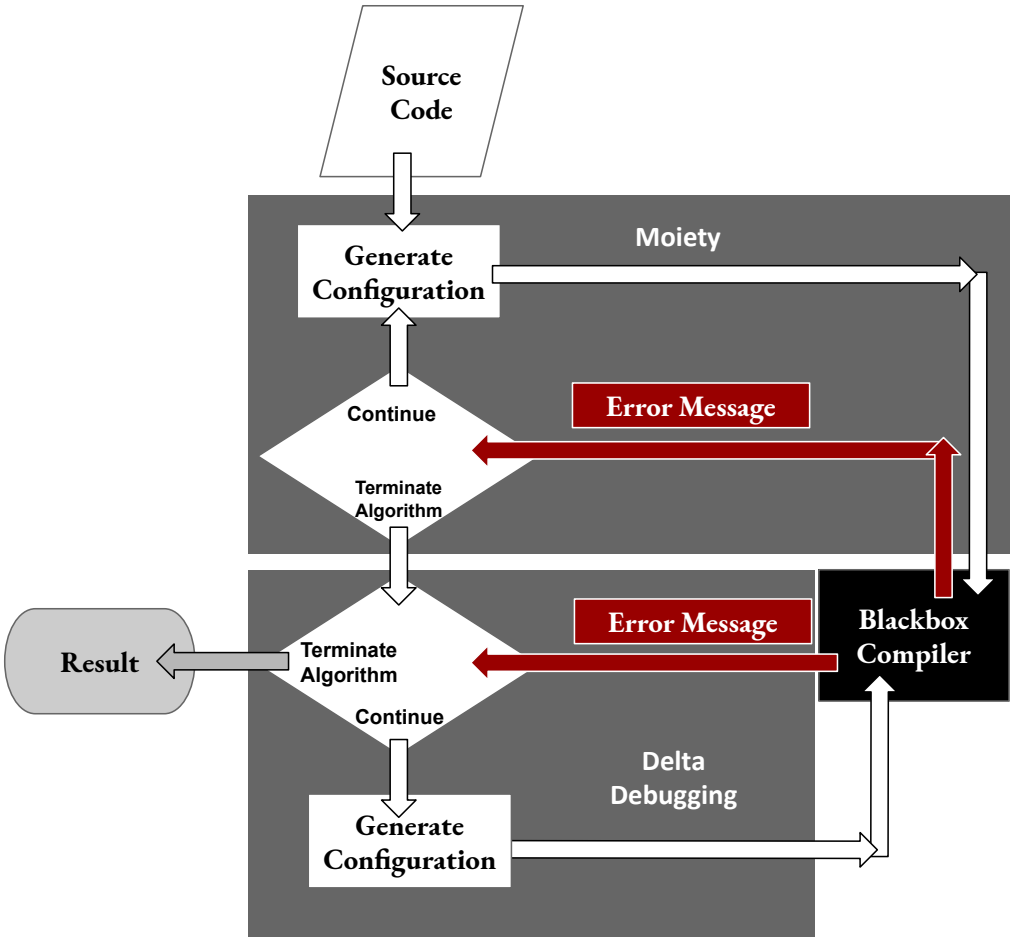


Figure 8: The informal overview flow of the Elucidate debugger

I named the pre-processing algorithm *moiety*; which according to the Merriam-Webster dictionary, a moiety is "one of the portions into which something is divided".[3] Moiety divides the ill-typed program into moieties, places in the source code where the lines can split. Figure 8 shows the informal flow of the new debugger, named Elucidate, with the moiety algorithm, which differs slightly from the previous debugger Gramarye, whose flow

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

appears in chapter 3, section 3.1. The only difference between the two figures is the moiety algorithm situated before the *delta debugging* algorithm. The moiety algorithm works on the source code that previously would go directly to the *delta debugging* algorithm, generating a configuration variant that contains a singular line of code. Each configuration variant is sent to the blackbox compiler, which returns whether or not a "parse error on input" is present, which in turn is used to determine the moieties. I designed the Moiety algorithm to return a list of moieties in the shortest time possible, which is linear in the number of lines of the ill-typed program. *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, the overall time complexity of type error debugging is close to linear. The moiety algorithm terminates once the total number of lines of the ill-typed program have all received a result from the blackbox compiler. The delta debugging algorithm then starts, with only a slight modification; however, it now does not generate any configuration variants that cause unresolved results due to "parse error on input".

6.4.1 Illustration of the Algorithm

The Moiety algorithm is designed to reduce Unresolved, "parse error on input", results from large programs. However, to present how Moiety works concisely we have to consider the following small 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 run-time, the algorithm may only traverse the program once from beginning

to end to produce its set of moieties. Each moiety within the set of moieties is a singular line number that does not cause "parse error on input" when split from the line before it. The Moiety algorithm calls the compiler to test a configuration variant of the program to see whether a line yields a "parse error on input" or not. This section presents the tested configuration on the left, with the test outcome and the resulting moiety set on the right. Note that line 1 never yields "parse error on input," so the example starts with line 2.

```
1
2       0 -> [0]                    "parse error on input"
3
4                                   moieties:{}
5
6
7
8
9
```

As line 2 produces a "parse error on input" it cannot be the starting line for a split; and so the algorithm continues with line 3:

```
1
2                                   "parse error on input"
3   1 -> 1
4                                   moieties:{}
5
6
7
8
9
```

Like line 2, line 3 also cannot be a new moiety; the algorithm continues with line 4:

```
1
2                                   not "parse error on input"
3
4 plus :: Int -> Int -> Int         moieties:{4}
5
6
7
8
9
```

Line 4 is not a "parse error on input", so the algorithm can create a new moiety. The delta debugging algorithm can successfully split line 4 from line 3, as the lines are always split from the one above it; the moieties set only needs to contain line 4. Next line 5:

```
1
2                                   not "parse error on input"
3
4                                   moieties:{4,5}
5 plus = (+)
6
7
8
9
```

Likewise, line 5 is a new moiety as it can be split from line 4. The algorithm moves on to line 6:

```
1
2                                   not “parse error on input”
3
4                                   moieties:{4,5,6}
5
6 fib x = case x of
7
8
9
```

So line 6 is a new moiety too. The moiety algorithm continues with line 7:

```
1
2                                   “parse error on input”
3
4                                   moieties:{4,5,6}
5
6
7       0 -> f x
8
9
```

At this point, it is hopefully evident that lines 8 and 9 each also gives the outcome “parse error on input” and so the algorithm finishes with the moieties {4,5,6}.

Working through the example shows how simple the Moiety algorithm, seen in algorithm 3, is. The algorithm tests every single line of the original ill-typed program whether it yields “parse error on input” or not. The line cannot be split from the preceding lines in the former so that no generation of moiety can happen. Otherwise, it does start a new moiety. The result is an ordered set of moieties, lines of code that can be successfully split.

Algorithm 3: The Moiety Algorithm

```
1  define moiety(cont)
2      moieties ← initialMoieties(cont)
3      moiety ← mkMoiety(l)
4      l ← 2
5      while l < len(cont) do
6          if test (cLine (cont,l) == NOPARSE then
7              addLine(l, moiety)
8              break
9          else
10             moieties ← updateMoiety(moiety, moieties)
11             moiety ← mkMoiety(l)
12         l ← l + 1
13     end while
14     return updateMoiety (moiety, moieties)
15 end define
```

6.4.2 Example of Isolating Delta Debugging with Moieties

Algorithm 4 shows the changes needed to allow the *delta debugging* algorithm to use the set of moieties the *moiety* algorithm generated. However, this section covers an example of how delta debugging and moiety work together using the moieties {4,5,6}.

Elucidate starts *isolating delta debugging* with the passing configuration {} and the failing configuration {[1,2,3],[4],[5],[6,7,8,9]}. The failing configuration, of source code line numbers, is now split using the moieties unlike previously where every line is an acceptable splitting point which is represented as {[1],[2],[3],[4],[5],[6],[7],[8],[9]}. The Delta Debugging algorithm divides the difference between the two configurations by two and hence tests the configurations {[1,2,3],[4]} and {[5],[6,7,8,9]}. Both configurations give the outcome unresolved. Hence delta debugging has to divide the difference between the passing and failing configuration by four {[1,2,3]}, {[4]}, {[5]}, {[6,7,8,9]} and test 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]}. The implementation happens to test {[5]} first, and the test gives an outcome of a pass.

Next, *isolating delta debugging* calls itself recursively with the new passing configuration

Algorithm 4: Changes to Delta Debugging Algorithm for Moiety Support

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

{[5]} and the failing configuration {[1,2,3],[4],[5],[6,7,8,9]}. It divides 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]}. It divides the difference by two and hence tests 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 these two configurations as a result. The new moiety type debugger, Elucidate, 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 Elucidate can return at best a single moiety.

This is no different to our non-moiety configuration due to merging our second moiety with our first and our third moiety with our fourth. We again receive a double Unresolved, and we grow our Granularity to 4. This time we cannot split lines 1 and 2 from line 3 as we know they have reliance on each other so we divide as follows:

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

This division is odd-looking because we are dividing our Moiety list of 4 moieties by the Granularity of 4; we are now having to testing the moieties individually and as we always move our lines from right to left moiety 3 is the first to be tested. The result is a Pass and an Unresolved. As we received a Pass our Granularity is reset to 2 and we keep our moiety 3, line 5, in the left-hand side. We split again halving the right-hand side.

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

We now have Fail and an Unresolved. We stay at Granularity 2, and the left-hand side is now the base program for the right.

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

The results are a Fail and a Pass. The algorithm terminates, Delta Debugging returns

the difference between our two sides as the cause of the type error; lines 1,2, and 3.

6.5 Evaluating the Elucidate

In chapter 4, the investigation found that Pandoc is a good source of programs to test for real-world scalability, so I now expand the number of the test programs from 20, naming the set the *scalability* benchmarks. Here, I place within Pandoc 80 individual type errors into 40 of its modules (using each module twice), of which each contains between 32 and 2305 lines of code (Table 11).

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 11: Lines of Code per Module with Associated Errors

The modules chosen were the first 39 in size order that contained code that could be made ill-typed. The last module was the largest module Pandoc contained at 2305 lines. A random number generator decided upon the placement of the error. If the line suggested was unsuitable for type error placement, the generator was re-run. The type errors were inserted manually with no prior planning on the category of type error. The categories listed by the individual error message presented by GHC can be seen in Table 12. To note, all of the type errors inserted are Equality Errors as according to *TcErrors*[4].

The evaluation compares the new debugger, Elucidate20, with Gramarye19. Gramarye19 is a modified version of the previous debugger Gramarye, found in Chapter 3; and like

[4]TcErrors is part of the Glasgow Haskell Compiler and states that type errors fall into one of 4 groups; more information about this appears in https://github.com/JoannaSharrad/ghcErrorsDoc/blob/master/RoughGuidetoGHCTcErrors.pdf

Category	Errors Total
Couldn't match...	**79**
Rigid type variable bound by the type signature ...	5
In the ... field of a record ... In the expression ...	3
...In the expression: ...	**22**
In an equation ...	7
In a stmt of a 'do' block ...	3
In a case alternative ...	7
In the expression: ...	5
...In the ... argument of ...	**20**
In the expression ... In an equation for ...	7
In a stmt of a 'do' block ...	11
In the ... argument of ...	2
...In the pattern: ...	**3**
In a case alternative ... In the expression ...	2
In equation ...	1
...is applied to...arguments ...	**26**
Possible cause ... is applied to too many arguments ...	3
Probable cause ... is applied to too few arguments ...	11
The function ... is applied to ... argument/s ...	12
Couldn't deduce...	**1**
Arising from a use of ... from the context ...	1

Table 12: Type error categories

Elucidate20 now supports Modular Programs and a Build tool as described in Chapter 4. However, only Elucidate20 can apply the moiety pre-processing.

For this evaluation, the benchmarks ran on an AMD Phenom X4, 32GB RAM, Samsung SSD 850, PC running Ubuntu 18.04LTS to answer the following questions:

1. Does the Moiety algorithm reduce the number of unresolved, "parse error on input", results?

2. Does the pre-processing reduce the time taken by *Isolating Delta Debugging*?

3. Does Elucidate produce quality results when applied to the framework?

6.5.1 Reduction of Unresolved results

Question: Does the Moiety algorithm reduce the number of unresolved, "parse error on input", results?

The Moiety algorithm produces a set of splitting locations in the source code. The *scalability* benchmark contained a total of 16264 lines of code, of which 16184 were places that the *Isolating Delta Debugging* algorithm was allowed to split. Pre-processing the source code using the Moiety algorithm sees that 7953 (68%) were places that the configuration could be split without causing a "parse error on input". On average, 39% of a single benchmark caused "parse error on input" when splitting.

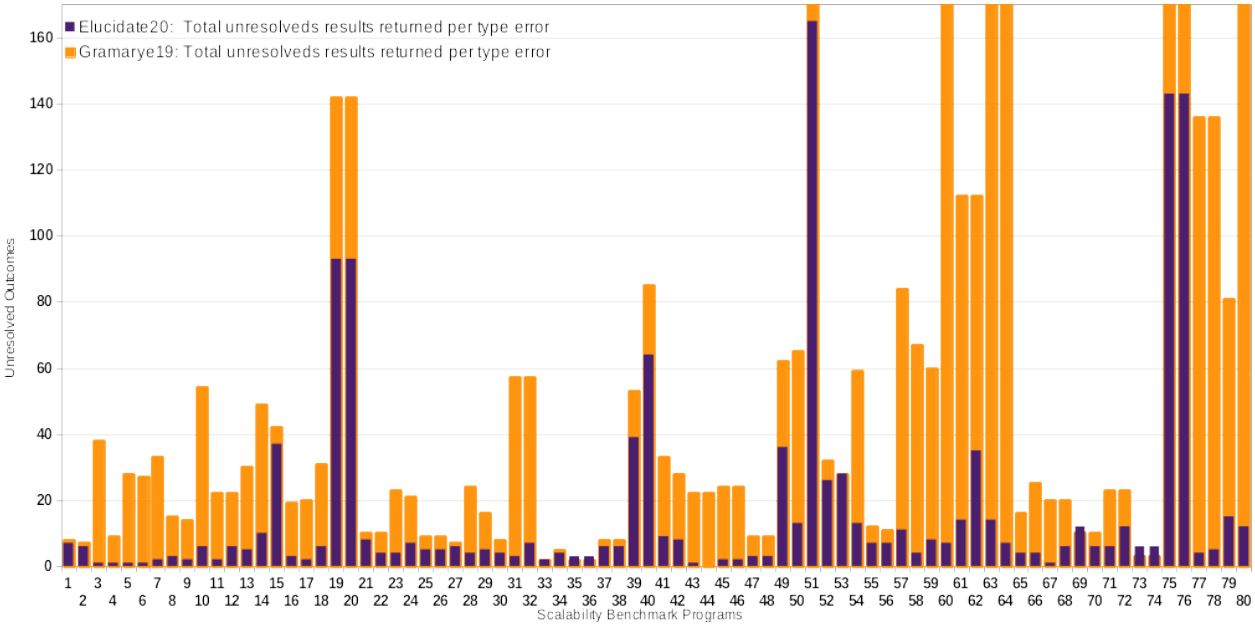


Figure 9: Unresolved results per introduced type error

Figure 9 shows the number of unresolved outcomes, on the y axis, for each of the 80 type errors in the *scalability* benchmark listed on the x-axis. For the desired outcome, each bar should be as close to zero as possible. For ease of reading, capping of Figure 9 at a maximum of 170 unresolved results has happened; however, it is worth noting that Gramarye19 returned seven results higher than this, with modules 51, 60, 63, 64, 75, 76 and 80 returning 265, 395, 1436, 1436, 221, 221, and 504 unresolved results respectively, which can be seen in figure 10. The highest outcome of Unresolved from Elucidate20 was 165, with

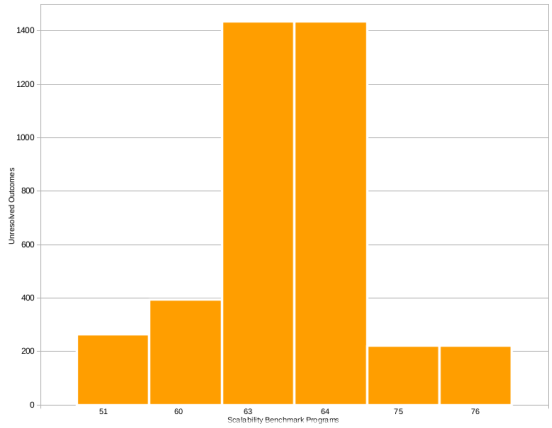


Figure 10: Unresolved: 51,60,63,64,75,76

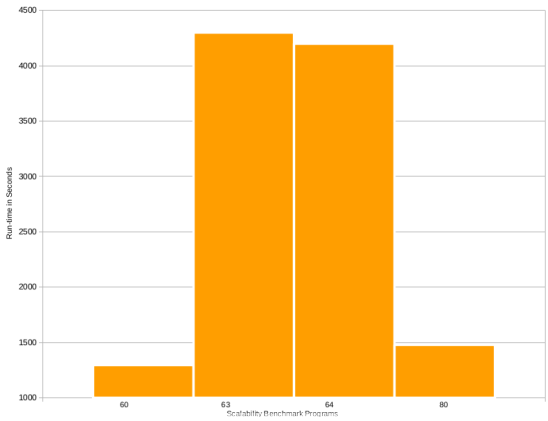


Figure 11: Run-time: 60,63,64,80

its lowest being 0 compared to Gramarye19 with 2.

There are 16 unresolved outcomes per type error from Elucidate20 compared to Gramarye19 at 88, meaning, on average, a reduction of 72 calls to the blackbox compiler. The investigation shows the importance of reducing calls in benchmark 64, a module with 240 lines of code. Here, Gramarye19 has 1436 unresolved outcomes and takes just over an hour to run the *Isolating Delta Debugging* algorithm, whereas Elucidate20 receives only seven unresolved results and the time taken drops to just 36 seconds, a difference of around 52 minutes.

Elucidate20 has a significant impact, totalling a removal of 5743 Unresolved outcomes from the entire benchmark over Gramarye19. However, though the Delta Debugging Run-Time can decrease with some benchmarks, like benchmark 64, does the Moiety algorithm reduce all of the benchmarks?

6.5.2 The Run-Time Speeds

Question: Does the pre-processing reduce the time taken by Isolating Delta Debugging?

With the unresolved results minimised, I hypothesise that the time taken by Delta Debugging should reduce. Figure 12 shows the outcome of the run-time of Delta Debugging, excluding (Gramarye19) and including pre-processing (Elucidate20), in seconds on the y

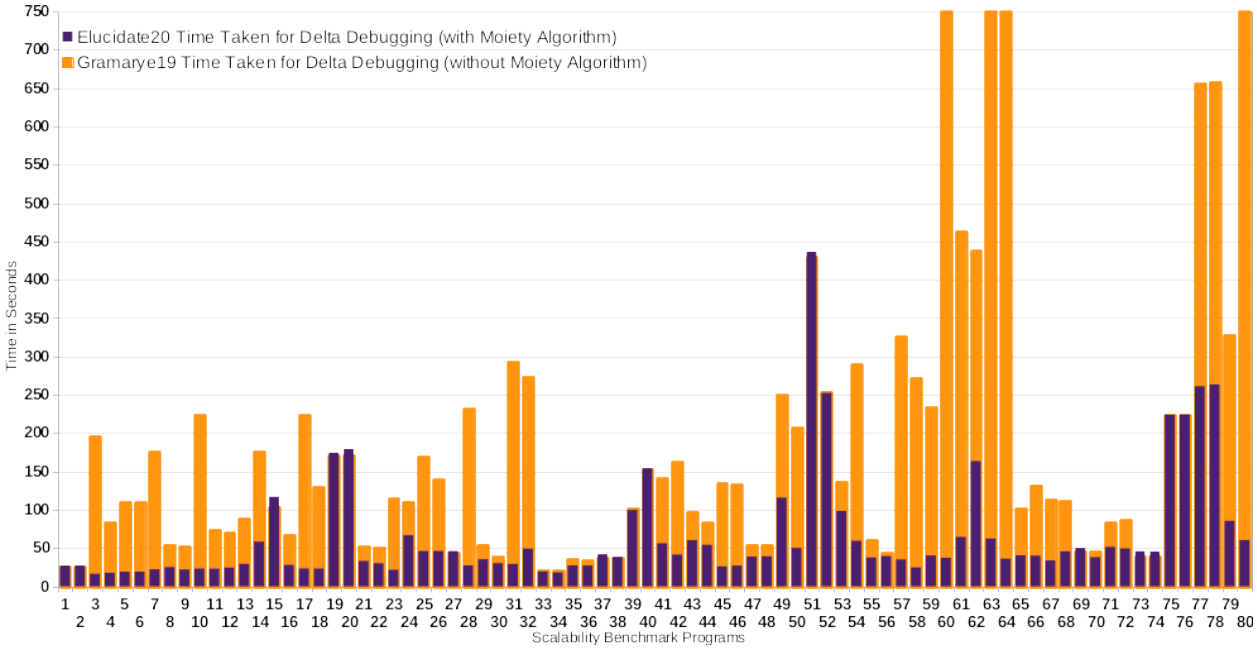


Figure 12: Delta Debugging Run-time

axis, and again each type error on the x-axis. As in Section 6.5.1, the figure is again modified so that the data is more clear by dropping off the most extreme results of Gramarye19 in type errors 60, 63, 64, and 80 who returned run-time results of 1295s (21m35s), 4299s (1h11m39s), 4201s (1h10m1s), and 1482s (24m42s) each, and placed them in figure 11. The highest result from Elucidate20 is 436s (7m16s), with the lowest recorded at 16s compared to Gramarye19 at 21s.

On average, Gramarye19 took 285 seconds (4m45s) to run the *Isolating Delta Debugging* algorithm, 219 seconds (3m39s) more than Elucidate20 at 66 seconds (1m6s), showing a clear link between total unresolved outcomes received and the time taken to locate a type error. In total, Elucidate20 reduced the time taken by *Isolating Delta Debugging* algorithm for the entire benchmark by 4h52m8s.

However, when running a debugger, the user experiences the entire process, not just the algorithm locating the type errors. Elucidate's pre-processing is linear, based on lines of code in the program, and the length equals the number of calls the algorithm needs to make to the blackbox compiler. Gramarye19, with its lack of Moiety algorithm, takes on average 303 seconds (5m3s) compared to Elucidate20 at 419 seconds (6m59s). It is clear that when using the Moiety algorithm, the results are around a minute slower than the previous

debugger, Gramarye. This issue with pre-processing is down to the calling of the compiler as a blackbox. In the case of the scalability benchmark, the debuggers are calling the build tool Cabal. As an example, if we take the worst-case result, benchmark 79, reduction of the run-time of the *Isolating Delta Debugging* algorithm goes from 327s (5m27s) to 85s (1m25s); however, the user-time increases from 330s (5m30) to the awful 4888s (1hr21m28s). Looking closer at this benchmark, it is 2306 lines of code, and every call to Cabal takes around 2 seconds. If applying 2 seconds to every line of code, the result is 4612s (1hr16m52s), close to the worst-case benchmark. However, the pre-processing method has occasional successes improving debugging time, with Elucidate20 reducing the user-time for some benchmarks by over an hour. This mixed result means that one viable solution is a heuristic that decides between using the moiety algorithm or not depending on data, such as the number of lines.

6.5.3 The quality of Elucidate

In Figure 13 and Table 13 the thesis presents the data from applying the framework from Chapter 5 to the evaluation results. The thesis displays the outcome of Recall in more depth to mimic other type error debugging evaluations. The two graphs show all 80 modules on the x-axis and if the type error they contained were either correctly located (100%) or not (0%) or the y axis.

The framework results table shows the average outcome for all four of the framework metrics. The higher the percentage, the more desirable.

Metric	Gramarye19	Elucidate20
Accuracy	94%	88%
Recall	38%	59%
Precision	16%	14%
F_1 Score	20%	19%

Table 13: Framework Results - Average

Question: Does Elucidate produce quality results when applied to the framework?

Recall shows us if Elucidate has returned the correct type error specified. As there is only a single type error per benchmark, the result is binary. Elucidate20 correctly locates 59%

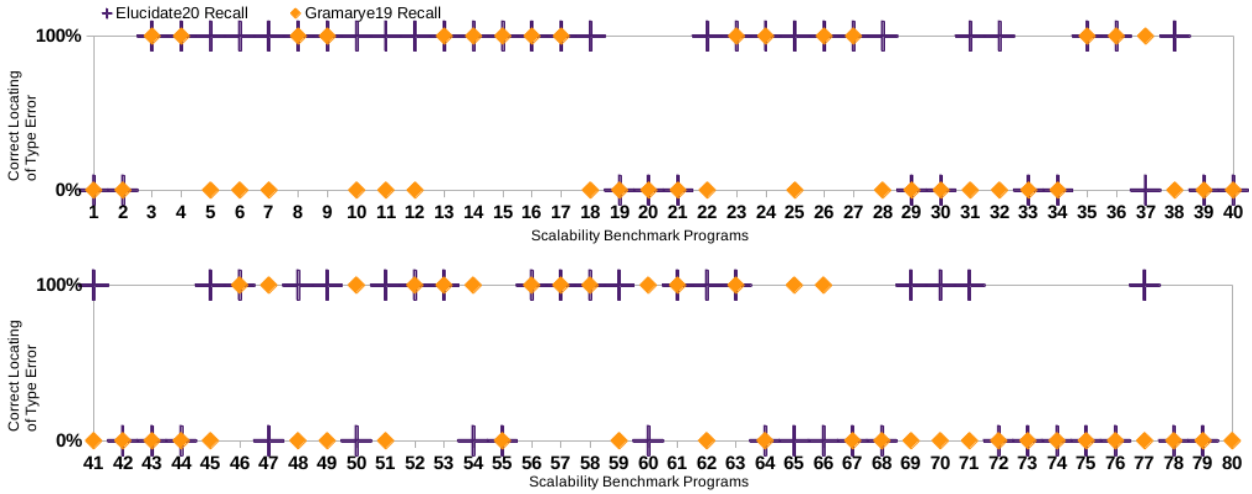


Figure 13: Recall

(47/80) of the type errors compared to Gramarye19, which returns fewer correct type errors at 38% (30/80). This rise in incorrect results comes from Elucidate20's direct link to the pre-processing of the source code. Firstly, as Elucidate passes a new configuration to Delta Debugging, setting out how to split the lines, there is the chance to generate an alternative pathway of modifications; this leads to different results from the Blackbox compiler. As Elucidate's path relies on these outcomes, an alternative result can happen. Secondly, as Elucidate does not allow the splitting of lines outside the moieties, the results gain the bias of returning a greater set of locations and so increasing the chances of success. As described in Chapter 5, this bias can allow us to return 100 results as suggested locations with an ill-typed program of only 100 lines; this would not make a suitable solution and so is countered with the precision metric.

In Table 13 the results show that indeed Gramarye19 is more precise than Elucidate20; however, overall, this only accounts for a difference of 2 percentage points, meaning the evaluation needs to invoke the F_1 score for an accurate reading.

The F_1 score blends the metric results, Recall and Precision, to form an accurate overview of the results; as already mentioned, this is the harmonic mean of the two metrics. With this set of benchmarks, the results receive a 1% difference between the presented debugger Elucidate20 and the previous Gramarye19, with the latter providing a higher F_1 score. This outcome is not surprising; the Moiety algorithm hampers the precision of Elucidate20.

However, I do not see this as a negative; it aimed to avoid causing Unresolved results. This outcome was also positive evidence showing the importance of using more than one metric when evaluating debugging solutions and works well to indicate that many metrics are needed to present the true quality of a type error debugger.

6.5.4 Summary

Applying the Moiety algorithm reduced the number of unresolved outcomes significantly; this, in turn, reduced the time taken for the *Isolating Delta Debugging* algorithm to run by an average of around 3 minutes. However, for the actual time the user experiences, the evaluation must include the pre-processing that moiety provides. In investigating these new evaluation results, I found that calling the blackbox compiler caused the overall run-time to increase, giving unsatisfying results that show an alternative to pre-processing each line of code is needed to reduce the time taken by the debugger. When applying the framework, I found that using the de facto *Recall* metric did show improved results for Elucidate20. However, when adding the metrics precision and F_1 score from the framework, a more accurate picture is presented, with Elucidate20's results being slightly lower than Gramarye19. Nevertheless, Elucidate did improve the time taken by *Isolating Delta Debugging*.

6.6 Quitting the Compiler

While investigating reducing unresolved outcomes, a hypothesis formed that the build tool feature may significantly slow down the debuggers run-time when specific outcomes happen. A brief investigation and evaluation of this hypothesis are next.

In this chapter, the type error debugger, Elucidate, uses benchmarks for evaluation. The benchmarks use Pandoc, an extensive multi-module program built using a build tool. Build tools allow for compiling programs with many modules distributed within a complex file structure. It compiles the modules in the necessary order to support import statements between modules individually. Recall that the previous debuggers in this thesis have to compile a configuration variant of the initial program, using a *blackbox compiler* to receive one of a possible four results. However, when applying Elucidate to a program that is built

using a build tool, the debugger cannot simply call the *blackbox compiler* directly. It has to call the build tool, which in turn calls the compiler. One such build tool is Cabal which is the build tool for Pandoc and of which Elucidate has the ability to use. When using a build tool such as Cabal, the whole program compiles only under certain circumstances. If a module is ill-typed, Cabal will stop and return a type error; however, if all the modules are well-typed, the entire program and its modules get compiled. It is the latter aspect that causes the generation of a hypothesis. If the build tool compiles the entire program every time the program is well-typed, then every Pass configuration also builds the entire program. Therefore, a Pass configuration takes more run-time to check than a Fail or Unresolved result.

As described previously, Elucidate creates moiety sets. When creating them, the algorithm is only interested in if each line causes a "Parse Error on Input" or not. As the algorithm does not care about the other results, this means they are discardable. As all errors, including "Parse Error on Input", return their status before compiling the entire program, the hypothesis is that they are quicker to return their result. One solution to the hypothesis is to exploit this fact by forcing all other results to return as an error, thus never allowing the generation of a well-typed configuration. The compile never fully compiles the configurations when creating the moiety sets.

For the exploitation to work, a keyword is added to the last line of each generated configuration during pre-processing. This keyword needs to generate a different parse error to a "parse error on input". For example, when debugging type errors in Haskell source code, this can be a string of characters.

```
insert x [] = [x]
insert x (y:ys) | x > y = y : insert x ys
                | otherwise = x : y : ys
algsjnslnmlw9834up0wfacdmlc;l'C;XQ39AFX;AFNOV84UQCCD
```

When the debugger then sends these configurations to the blackbox compiler, the results are either a "Parse Error on Input":

```
parse error on input,          | otherwise = x : y : ys
```

or a generic parse error:

```
Parse error:
module header, import declaration, or top-level declaration expected.
```

The first will add the line of code to the moiety set, whilst the second ignores it.

6.6.1 Mini Evaluation

There are now two versions of the Elucidate debugger. The first is introduced during chapter 6 and a second version that implements the previous section's solution. Using the scalability data, this evaluation can compare both versions of the debugger to see if the solution is viable and a speed increase occurs.

Question: Does applying the new solution reduce the time taken by the moiety algorithm?

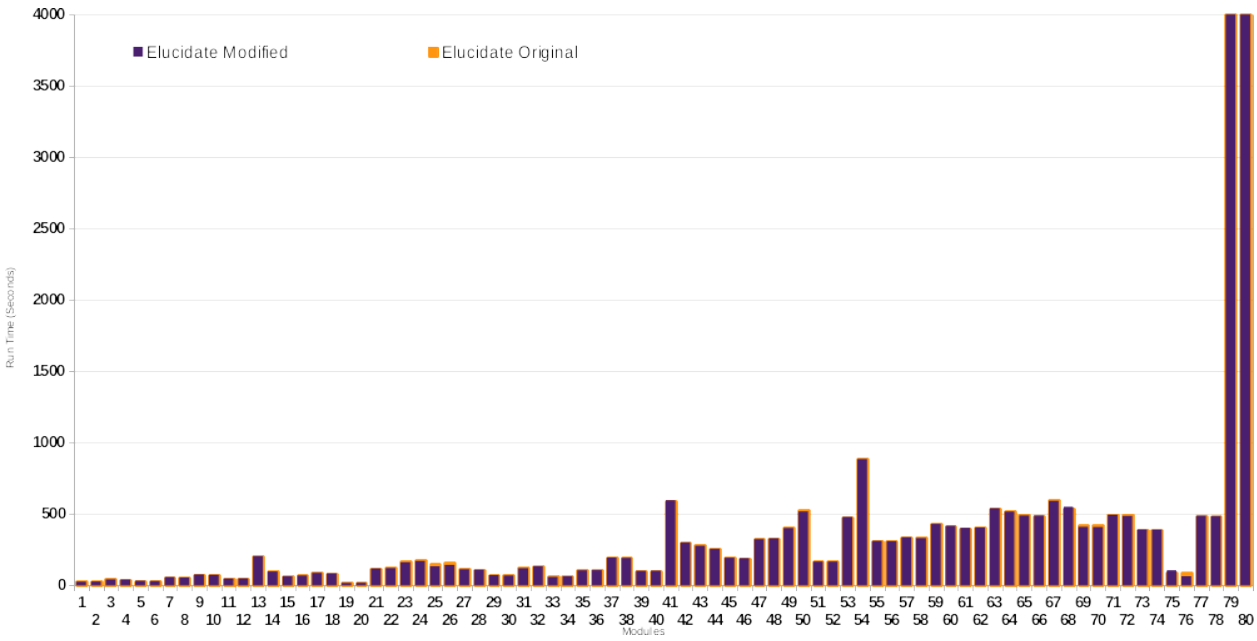


Figure 14: Moeity Set Creation Time

Figure 14 shows the results of the time it took to create the moiety sets. On the x-axis, the modules tested are in size order. Shown on the y-axis is the run-time of the moiety algorithm. Unfortunately, from the data, there is little difference between each version of the debugger. On average, the original Elucidate debugger took 5 minutes and 53 seconds

to run the moiety algorithm compared to the modified version, which took 0.29 of a second less time.

Table 14 shows the modules with errors and the total of well-typed configurations, those that received a Pass result, generated by each. Sadly, the table shows no relationship between the number of well-typed configurations and the time taken, seen in more detail with modules 32 and 71. Module 32 contains eleven well-typed configurations when pre-processing, whilst module 71 contains 50. However, when running the moiety algorithm within the modified debugger, module 71 takes 0.4 seconds less time than module 32, which took 1 second more. Unfortunately, this evaluation shows that the solution did not improve the moiety algorithms run-time. However, it did show an interesting pattern; the modules are in size order, yet, the growth is not smooth; the time taken by the Moiety algorithm does not simply grow with the number of lines.

Errors	WT-C	Errors	WT-C	Errors	WT-C	Errors	WT-C
{1,2}	9	{21,22}	20	{41,42}	19	{61,62}	58
{3,4}	2	{23,24}	5	{43,44}	1	{63,64}	22
{5,6}	3	{25,26}	3	{45,46}	7	{65,66}	13
{7,8}	4	{27,28}	11	{47,48}	43	{67,68}	0
{9,10}	4	{29,30}	6	{49,50}	62	{69,70}	57
{11,12}	4	{31,32}	11	{51,52}	0	{71,72}	50
{13,14}	10	{33,34}	10	{53,54}	44	{73,74}	39
{15,16}	15	{35,36}	8	{55,56}	39	{75,76}	13
{17,18}	4	{37,38}	29	{57,58}	18	{77,78}	82
{19,20}	1	{39,40}	29	{59,60}	21	{79,80}	323

Table 14: Well-typed configurations per Module with Associated Errors

This chapter has left the investigation with two directions. Either design a ‘fallback’ heuristic that reverts to non-moiety debugging under certain circumstances or provide an alternative to the pre-processing of source code. The latter is under examination in the next chapter.

Chapter 7

The speeding up of type error debugging

Elucidate, the current debugger, successfully reduces Delta Debugging time; nevertheless, it is still too slow in practice, as seen in the evaluation in the last chapter. Recall that the Moiety algorithm sends each line of the original ill-typed program separately to the blackbox compiler. For a typical module of 400 lines, that can take around 13 minutes. My theory is that by combining *isolating delta debugging* and moiety algorithms, making moiety on-request rather than pre-processing, there should be a reduction in the time taken to locate type errors. Suppose a split leads to a configuration yielding a "parse error on input". In that case, the new algorithm uses the idea of Moiety to find valid and invalid splits to avoid them in future iterations. The following section will illustrate this summary with an example.

7.1 Illustrating the solution by an example

The debugger, named Eclectic, will accept only an ill-typed program as input; so let us consider another program given by Chen and Erwig in their benchmark suite (Chen and Erwig 2014a). This program has a singular type error on line 2:

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

Recall that the first step of the *isolating delta debugging* algorithm is to generate the initial configurations. The first configuration, on the left, is the ill-typed program; this is the initial program started with, and the algorithm removes, minimising, lines that do not cause the type error. The second configuration, on the right, is the well-typed program; this configuration starts empty, and the algorithm adds lines from the ill-typed program, maximising so that there are only well-typed lines:

Step 1: ill-typed configuration

```
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)
```

Step 1: well-typed configuration

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

Next, the *isolating delta debugging* algorithm starts by splitting the program in half. The algorithm removes the second half of the program from the ill-typed configuration and adds it to the well-typed configuration:

Step 2: modified ill-typed configuration

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

Step 2: modified well-typed configuration

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

Eclectic sends both configurations to a blackbox compiler. Remember that the debugger uses only the message returned by the compiler, which tells us whether a configuration has a type error (*fails*), compiles successfully (*passes*) or causes any other error (*is unresolved*). The modified ill-typed configuration on the left *fails* and the modified well-typed configuration on the right is *unresolved*. Therefore the modified ill-typed configuration becomes the new, smaller, ill-typed configuration, whereas the (empty) well-typed configuration remains unchanged. The next iteration of *delta debugging* again creates two modified configurations:

Step 3: modified ill-typed configuration

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

Step 3: modified well-typed configuration

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

The left configuration *fails* and the right one *passes*. *Isolating delta debugging* prioritises *passing*, and the modified well-typed configuration becomes the new, bigger, well-typed

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

Step 4: ill-typed configuration

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

Step 4: well-typed configuration

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

This time Eclectic gets a 'Parse Error on Input' for the right configuration and calls the good-omens algorithm with both configurations and the current moieties. The good-omens algorithm adds a single line that precedes the parse error to the configuration with the 'Parse Error on Input', in this case, the well-typed configuration and line 2:

Step 5: good-omens

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

Step 6: good-omens

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

Eclectic sends this to the blackbox compiler and again receives a 'Parse Error on Input'.

So the algorithm adds line 1 back and then receives a fail result from the blackbox compiler. The good-omens algorithm finishes and returns the following lines: {1},{2},{3}. Lines 1 to 3 form a moiety $\{1, 2, 3\}$ and are no longer valid splitting points. Eclectic sends the new list of moieties back to the *Isolating delta debugging* algorithm, lines {4},{5} can be split but not lines $\{1, 2, 3\}$ so the new moieties look like: $\{1, 2, 3\}$,{4},{5}. As lines 4 and 5 have already had a pass result, and Isolating Delta Debugging cannot divide lines 1,2, and 3 any further, Eclectic terminates with the result that the type error location is within the lines $\{1, 2, 3\}$.

7.2 Eclectic

Eclectic is a modified version of the previous debugger in chapter 6; however, unlike its predecessor, the new debugger implements two core elements:

1. The modified *Isolating Delta Debugging* algorithm
2. The *Good-Omens* algorithm

Figure 15 shows how the *isolating delta debugging* and good-omens algorithms informally flow together with a basic overview. Next is a description of each aspect and how they relate in more detail.

7.2.1 Delta Debugging

The Delta Debugging algorithm has been discussed in detail in Chapter 2. However, as a quick refresh, delta debugging forms the backbone of all the debuggers in this thesis due to its ability to mimic how programmers naturally debug with just compiler output. The process goes as such: discover a bug, modify the source code, and recompile to see if it has achieved the desired outcome of bug-free code.

In the case of this thesis, the pass configuration, cPass, contains a well-typed version of the program, an empty program, and the fail configuration, cFail, contains the ill-typed program. The algorithm minimises the fail configuration by removing lines and adds lines back to the empty pass configuration. Each configuration gets sent to the blackbox compiler

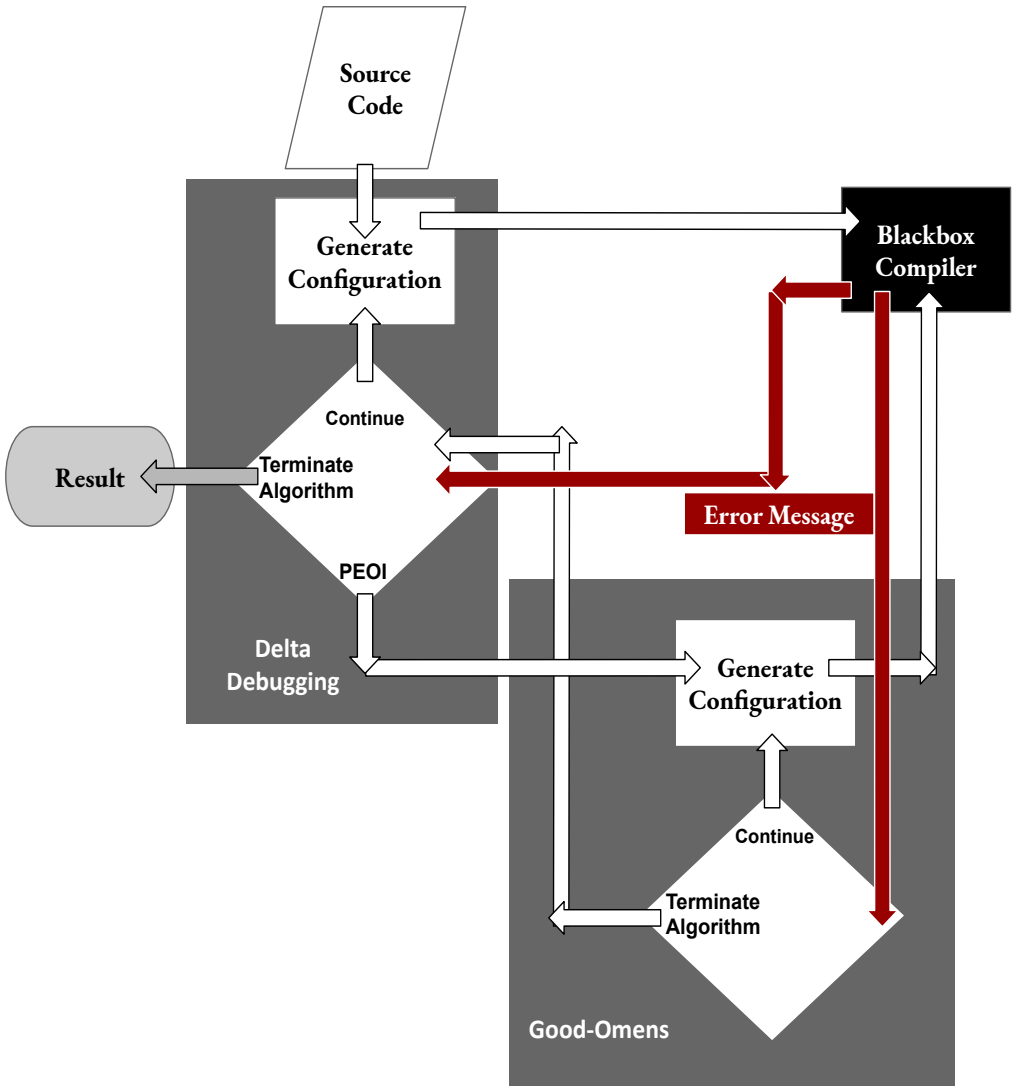


Figure 15: The informal flow of the Eclectic debugger

for a result. The compiler lets us know if the results are a Fail (×), the configuration variant that contains a type error, a Pass (√) the configuration compiles and Unresolved (?) as any other error such as parse error or unbound identifier.

If the *Isolating delta debugging* algorithm receives a Fail (×) or Pass (√) it will continue to call itself recursively. Otherwise, an *Any other error* (Unresolved) result will again increase the granularity until it is no longer viable. Once the algorithm terminates, two configurations are left, one that contains only failing lines and the other only passing.

It is the intersection of these two configurations that generates the result of which location

Only Failing Lines

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

Only Passing Lines

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

Figure 16: The two final configurations once the algorithm has terminated.

is ill-typed. For example, Figure 16 lines 1, 2, and 3 appear in the failing configuration but not in the passing. The intersection of the two configurations here is lines 1 to 3.

7.2.2 Modified Isolating Delta Debugging

Delta Debugging is easily applied to the domain of type error debugging. However, to integrate the good-omens algorithm, some modifications had to be applied. Algorithm 5 shows an outline of the new modified *Isolating delta debugging* algorithm, the changes are within lines 15 to 23.

Here added is an additional clause that detects for 'parse error on input' results. This means that the mapped results from the blackbox compiler now look like the following: Fail ($\times$), Pass ($\surd$), Unresolved (?), and Parse error on input *(ParseInput)*.

These new results are applied as follows; first, Eclectic sends the ill-typed program to the *Isolating delta debugging* algorithm. At this stage, it works as previously described, creating two configurations, one that is empty and one that is the entire ill-typed program. It recursively modifies each configuration with either more or fewer lines, requesting the results of the changes from the blackbox compiler until the delta debugging algorithm receives a 'ParseInput' result. Eclectic then calls the good-omens algorithm with the current configurations. These are the failing and passing configurations in their current variant. Once the good-omens algorithm terminates, it returns a set of valid and invalid splitting points, or

Algorithm 5: Changes to Delta Debugging for Good-Omens

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

moieties, for the ill-typed programs source code and the *Isolating delta debugging* algorithm starts again from its last position. However, this time when dividing the configurations, it uses the moieties for guidance. The *Isolating delta debugging* algorithm then recursively calls itself again until it either terminates or receives another 'ParseInput' result.

7.2.3 Good-Omens Algorithm

In the previous chapter, the thesis introduced the algorithm Moiety. The algorithm works by pre-processing each line of an ill-typed program before reaching the *Isolating delta debugging* algorithm. The design of this pre-processing is to eliminate all 'Parse Errors on Input' by generating moieties, also described as sets of line numbers that are valid splitting points in the program. For the debugger, Eclectic, I modified the moiety algorithm to become the on-request good-omen algorithm seen in algorithm 6. The debugger no longer needed to check each line for a 'Parse Error on Input'; however still needs to generate sets of moieties, this time representing both valid and invalid splitting points.

Algorithm 6: The Good-Omens Algorithm

```
1 define go (l, moieties, cont)
2     moiety ← mkMoiety(l)
3     l ← l − 1
4     while l > 0 and test (cLine (cont,l) == NOPARSE do
5         addLine(l, moiety)
6         l ← l − 1
7     end while
8     return updateMoiety (moiety, moieties)
9 end define
```

Section 7.1 ran though a full example of the Eclectic tool. Here an example will just show how the good-omens algorithm works. Recollect that the example program caused *Isolating delta debugging* to call the good-omens algorithm at this point. The ill-typed configuration is on the left and the well-typed on the right:

Step 1: ill-typed configuration

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

Step 1: well-typed configuration

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

The cause of the 'Parse Error on Input' on the right configuration is currently the invalid split between lines 3 and 2. The *isolating delta debugging* provides the good-omens algorithm with both of the above configurations. It also provides the current set of moieties. In the case of the example, this is: {1},{2},{3},{4},{5},{6},{7},{8},{9}, there are no invalid splitting points.

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

Step 2: ill-typed configuration

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

Step 2: well-typed configuration

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

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

Step 3: ill-typed configuration

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

Step 3: well-typed configuration

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

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

{1,2,3},{4},{5},{6},{7},{8},{9}. The algorithm now returns the moieties list to the *Isolating delta debugging* algorithm.

7.3 Evaluation

In Section 7.1, the new solution is presented, making the pre-processing algorithm work on a request only basis. Presented in a previous chapter are benchmarks based on the real-world program Pandoc. Recall that the 'scalability benchmarks' contains 80 modules of Pandoc, each with a manually inserted singular type error. The modules range in size from 32 to 2305 lines of code, giving a good overview of how the debugger affects programs of different sizes. The evaluation compares the results against the previous debugger, Elucidate, whose results have also been re-captured on a PC running Ubuntu Linux 20.04 with an AMD Ryzen 7 3800X, 32GB RAM and a Samsung 850 SSD.

7.3.1 Reduction of time

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

Let us look at Figure 17. Along the x-axis are the 80 modules from the scalability benchmarks, and the y-axis represents the time taken in seconds. To make the graph easier to read, the graphs have omitted two tests and have placed them in the separate Figure 18, for Elucidate only, tests 79 at 2532 seconds (42 minutes 12 seconds) and test 80 at 2496 seconds (41 minutes 36 seconds).

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

Unfortunately, not all of the tests successfully reduced the time taken. One such example is module 38, shown in more detail in Figure 19, which had the worse time increase at 482 seconds (8 minutes 2 seconds) over Elucidate. These increases on only some of the results are

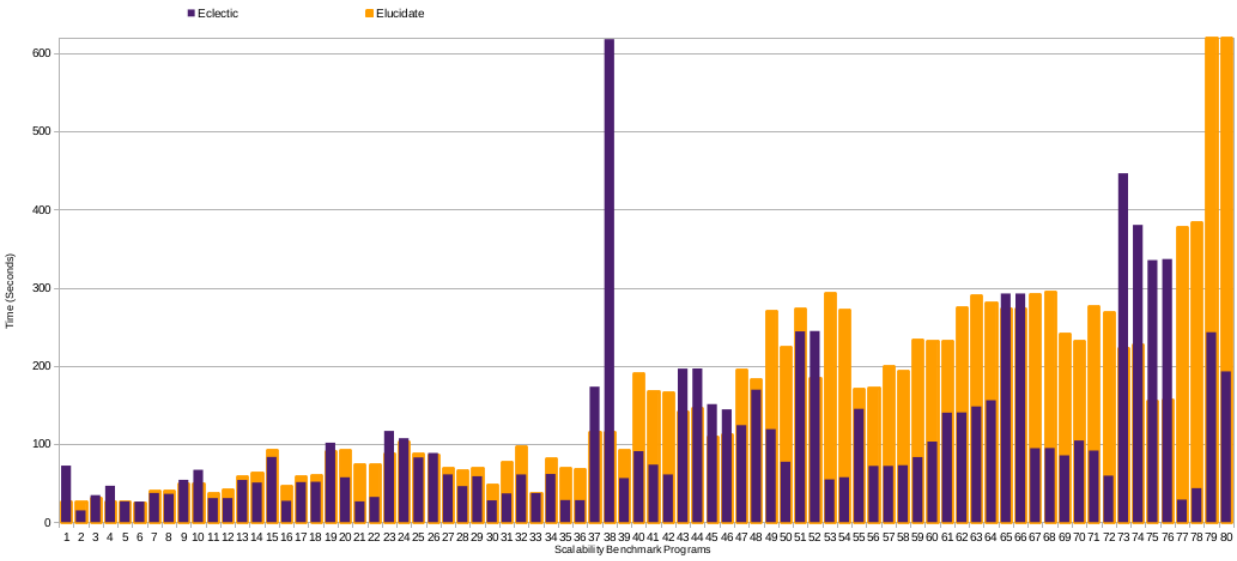


Figure 17: Elucidate and Eclectic - Run-Time

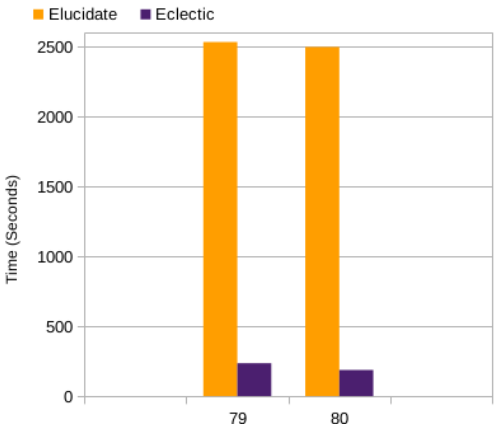


Figure 18: Programs 79-80

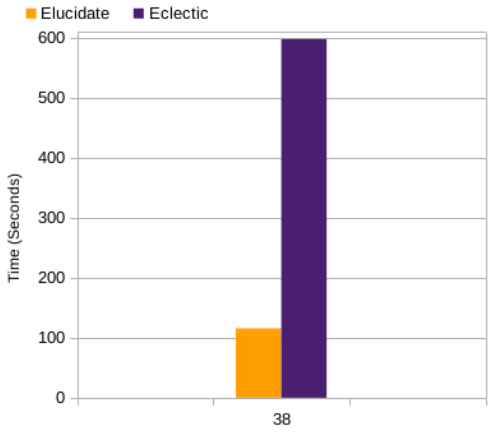


Figure 19: Program 38

understandable. It is easy to assume that a reduction of time occurs because the debugger is no longer pre-processing entire programs linearly. However, this assumption excludes that applying the pre-processing algorithm, Moiety, compared to the 'on request' algorithm, Good-Omens, can cause *isolating delta debugging* to generate the configurations differently. Having the *isolating delta debugging* algorithm traversing different paths can increase the overall number of the results, particularly Unresolved outcomes. Each extra result is an additional call to the blackbox compiler, which raises the run-time. Figure 20 shows this increase in run-time and calls to the blackbox compiler, the category of the compiler results is on the x-axis, and the number of times each result is received on the y-axis. Here, Eclectic, on all result categories, has increase calls. This increase in compiler calls corresponds to all 21 out of 80 modules, which increased this evaluation's run-time. As mentioned, the evaluation shows that the addition of Unresolved and 'Parse Error on Input' outcomes increases the time taken for the debugger. Currently, there is no way of predicting those outcomes before the debugger runs, especially agnostically.

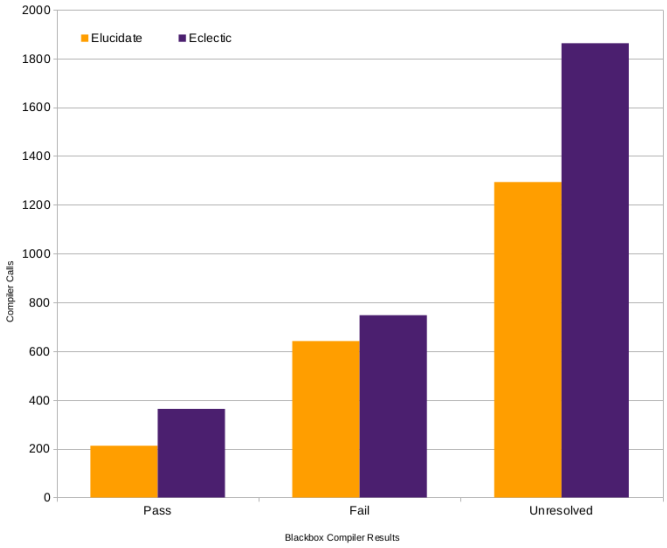


Figure 20: An increase of compiler calls leads to an increase of run-time

7.3.2 The quality of the debugger

The evaluation shows that Eclectic successfully reduced the time to locate type errors in the previous section. However, it is essential to show that a type error debugging tool has overall

quality. Chapter 5 introduced a framework to quantify the quality of a debugger, and here the evaluation applies that framework to the debugger Eclectic. Recall that the framework consists of four sections Accuracy, Recall, Precision and the F_1 Score. Commonly in type error debugging evaluations use recall only, the number of successful tests. However, it is also helpful to apply the other three sections to give a more rounded evaluation. Accuracy shows us the number of type error locations correctly returned compared to those incorrectly returned. Precision tells us how many of the lines returned are correct, and the F_1 Score is the harmonic mean between recall and precision.

Metric	Gramarye	Elucidate	Eclectic
Accuracy	94%	88%	83%
Recall	38%	59%	79%
Precision	16%	14%	11%
F_1 Score	20%	19%	18%

Table 15: Framework Results - Average for the reduction of time evaluation

Table 15 shows the results of applying the framework to Eclectic. Here, along with Figure 21, the recall metric shows that the debugger increases from 38% to 59% to 79% on the number of correct locations; Eclectic located 63 out of 80 errors compared to Elucidate at 47, and Gramarye at 30. If the evaluation just used this metric, the new debugger would look significantly better on results and time reduction.

However, I want to provide a more authentic depiction of the debuggers. Unfortunately, that does not put Eclectic in a good light. Accuracy, precision, and F_1 Score are lower than both the previous debuggers. The lower than expected results are due to an increase in the returned line locations in 35 out of the 80 tests that contained a larger number of incorrect results. Module 70 contains an example of this. Elucidate returns the correct answer with one reported line, while Eclectic does this in four lines. The reason for this discrepancy is the implementation of the good-omens algorithm. Currently, when calling the algorithm, an additional branch is generated. This branch is useful as it allows for more than one type error to be discovered, as seen in module 70's results. Elucidate returns only line 265, which in itself is a one-line function. However, Eclectic returns a three-line function {`49,50,51`} and the single line function at {`265`}. The need to discover more than one type error is

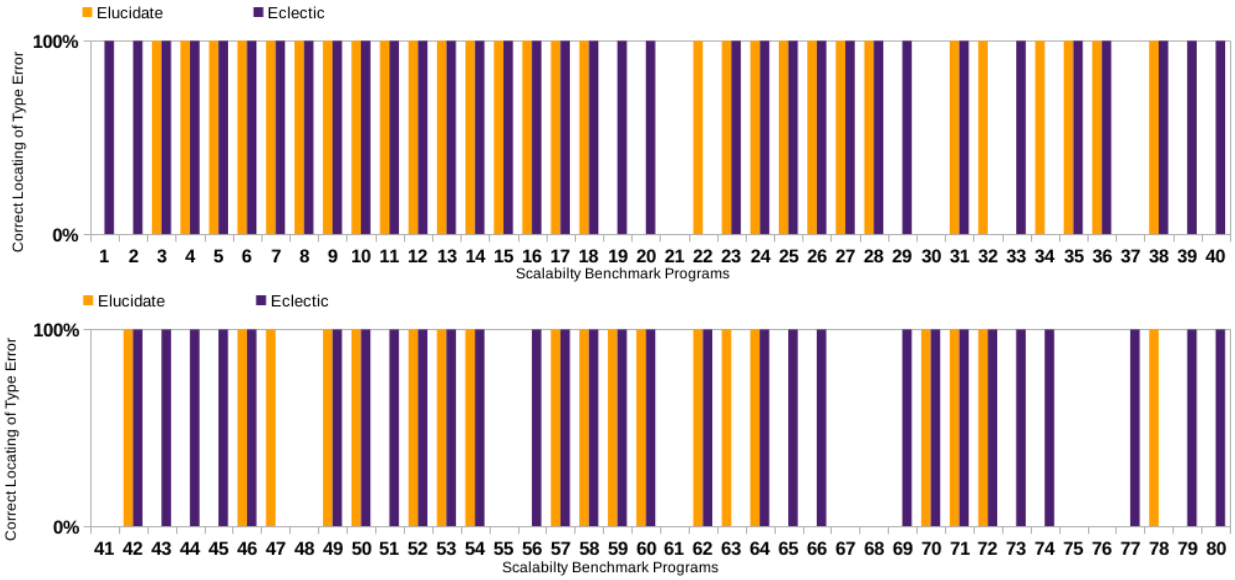


Figure 21: Recall data shows that the new debugger, Eclectic, locates 16 more type errors correctly, than the previous debugger Elucidate

subject to opinion, and future work will see if this feature is more of a hindrance than a help. However, in some cases, the evaluation does get the opposite effect. When looking at module 77, Elucidate returns 28 results, each a different line number, and all 28 are incorrect. On the same module, Eclectic returns fewer results at two line numbers and gets the correct location of the type error.

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

7.3.3 Summary

The evaluation proved that the new algorithm successfully locates type errors within the debugger in a timely fashion on average. In the most favourable result, Eclectic reduced the time taken by over 38 minutes. The debugger also discovered more correct locations of type errors than with Elucidate and Gramarye, as seen with the recall metric. However, when the evaluation gathered a more detailed look at the debugger, it was clear to see that it struggled with a lower F_1 Score. Altogether, Eclectic has succeeded in reducing the time taken on average to locate type errors.

This chapter presented the solution of combining a modified *isolating delta debugging* algorithm, the moiety algorithm, a blackbox compiler, and the good-omens algorithm. Though successful in locating type errors, the previous debugger from chapter 6 had too slow run-times. The new debugger, Eclectic, addresses this problem of speed. Previously the self-contained moiety algorithm acted as a pre-processor for *isolating delta debugging*. Moiety generated 'Parse Error on Input'-free configurations for the *isolating delta debugging* algorithm. These configurations allowed *isolating delta debugging* to know valid splitting points, locations that are available to be split without introducing an error. However, pre-processing came with a price: each line had to be type-checked against the compiler, leading to linear run-time.

In contrast, Eclectic allows the *isolating delta debugging* algorithm to request invalid splitting points only when it observes a 'Parse Error on Input'. This change gives us an average reduction in run-time of 1 minute 37 seconds.

Chapter 8

An Agnostic Type Error Debugger

Throughout this thesis, it is highlighted that all of the algorithms thus far are agnostic; they have no knowledge or understanding of the internal components of a specific compiler or programming language. However, the implementation of the algorithms in the type error debuggers featured in this thesis thus far relies on a specific programming language Haskell and explicitly supports the Glasgow Haskell Compiler (GHC) and Cabal as the blackbox compiler. In 2019, it was argued that when it comes to debugging, at least with delta debugging, language-specific versions of the algorithms consistently outperform pure agnostic algorithms (Stepanov, Akhin and Belyaev 2019). Nevertheless, I hypothesise that if all the algorithms are already agnostic, then the entire debugger that contains these algorithms can also have this functionality without affecting the metrics: recall, accuracy and precision in the evaluation framework. In this chapter, I investigate the possibility of producing an agnostic type error debugger comprising these aspects.

8.1 Programming Language-Specific Terminology

I propose that an agnostic type error debugger has the following trait: no awareness of language-specific details or terminology within its source code. However, remember from chapter 4 that there are certain aspects that the debugger might need to know when debugging to help reduces unresolved results and redundant calls, especially when working on large programs, such as import declarations and comments. Fortunately, in the context of this

thesis, an agnostic type error debugger can require some information from external sources.

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

```
putStrLn langHelp
```

Listing 8.1: Software using language translation

```
langHelp = "Help"
```

Listing 8.2: External Language File - English

```
langHelp = "Hilfe"
```

Listing 8.3: External Language File - German

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

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

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

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

Now that the type error debugger knows what "settings file", and thus what programming language it will use, the substitution can happen. In figures 22 and 23 the full "settings files" for both GHC and OCaml are presented.

```
###ALGORITHM### - Which debugger algorithm do you want to use default is: ddm
ddm

###FILE_TYPE### - File type the compiler uses: for example hs for Haskell
hs

###TYPE_ERRORS### - Terms used in the compiler message to show type errors
type, type , type:, type-variable

###TYPE_IGNORE### - Terms that conflict with above that should be ignored
parse error, type signature, type constructor

###PARSE_ERRORS### - Terms used in the compiler message to show parse errors
parse error on input

###PARSE_IGNORE### - Terms that conflict with above that should be ignored

###EXCEPTIONS### - These lines will not be removed
--,import

###MULTI_EXCEPTIONS### - Lines between these will not be removed
({-;-})
```

Figure 22: GHC Settings

As already stated, there are several exceptions that the type error debugger needs to not

```
###ALGORITHM### - Which debugger algorithm do you want to use default is: ddm
ddm

###FILE_TYPE### - File type the compiler uses: for example hs for Haskell
ml

###TYPE_ERRORS### - Terms used in the compiler message to show type errors
type, type , type:, type-variable

###TYPE_IGNORE### - Terms that conflict with above that should be ignored
parse error, type signature, type constructor

###PARSE_ERRORS### - Terms used in the compiler message to show parse errors
parse error on input

###PARSE_IGNORE### - Terms that conflict with above that should be ignored

###EXCEPTIONS### - These lines will not be removed
{*,*}

###MULTI_EXCEPTIONS### - Lines between these will not be removed
({*;*})
```

Figure 23: OCaml Settings

remove from the generated configurations to reduce blackbox compiler calls and to allow for substitution; these exceptions need to be listed in the "settings file". In figures 22 and 23, there are two sections, one for singular lines, in section `###EXCEPTIONS###`, and one for multi-line, in section `###MULTI_EXCEPTIONS###`, with commas separate both sets. Those with multi-lines are placed within braces, so the type error debugger knows when these begin and end. However, exceptions are not the only pieces of information an agnostic solution needs to recognise. Recall that the blackbox compiler returns a result that is categorised by the type error debugger. Two of those categories are Fail, the configuration contains a type error, and ParseInput, the configuration contains a 'Parse Error on Input'. The previous type error debuggers used key terms from the output of the compilers error message to categorise the configurations correctly. Thus, knowledge of if the error contains the terms 'type error' or 'parse error' is necessary. Unfortunately, there is no way to discover the substitutions for a programming language, nor are they the same for all statically typed function languages, so

again, these need to appear in the "settings file" as seen in figures 22 and 23 under sections `###TYPE_ERRORS###` and `###PARSE_ERRORS###`.

8.2 Evaluation

The substitution of the placeholders is completed only once when the debugger first runs. From that point, the agnostic type error debugger acts identically to its non-agnostic predecessor in the last chapter. An evaluation took place comparing both agnostic and non-agnostic versions using Haskell and GHC. As expected, the results were identical and, as such, will not be duplicated again here. However, for the type error debugger to be agnostic, there needs to be evidence that its agnostic behaviour works. Here, the solution is evaluated on an additional statically typed language, OCaml.

To evaluate, 11 ill-typed Haskell programs from the benchmarks collated by Chen and Erwig, mentioned throughout this thesis, were converted to the OCaml programming language (Sharrad 2021a). Eleven programs are a small subset of the initial 121 benchmarks; however, the conversions needed to be as identical as possible, including length and structure, so that those aspects did not interfere with the results.

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

Table 16: Framework Results - Average for the agnostic evaluation

Table 16 shows the results of the agnostic type error debugger on both the Haskell and OCaml versions. The number of times the agnostic type error debugger correctly reported the line the error occurs on, known as *recall*, shows that the debugger has identical results for 8 out of the 11 benchmarks. However, this is where the similarities stop. Accuracy, Precision, and F_1 Score show that the OCaml language's results are more beneficial than Haskell's. One reason for this outcome could be the debugger not calling the Good-Omens algorithm due to OCamlc's lack of an equivalent to Haskells 'Parse Error on Input'. Unfortunately, as shown

in chapter 6, the absence of an algorithm to remove these errors stunts the debugger's ability to scale to more extensive programs. Thus, more research is needed to see if OCamlc's lack of a 'Parse Error on Input' category will hinder the agnostic debuggers scalability when applied to it as well as other programming languages. However, though the results give a new direction for an in-depth investigation, it is clear that the results do not affect the overall positive outcome that the type error debugger can support many languages and, as such, is agnostic.

Chapter 9

Related Works

There have been thirty years of research in the type error debugging sphere covering many categories. Some of the papers reviewed below are only connected to this thesis due to being in the same type error debugging field. However, I felt that it is essential to cover all the categories involved to understand the subject better. Thus in this thesis, the categories form the backbone, an approach employed by Heeren, who thoroughly covered each in the literature review section of his PhD Thesis (Heeren 2005). This chapter first describes the brief history of type inference that leads to an overview of where type error debugging started in section 9.1. Then, section 9.2 discusses publications that form a core basis for the type error debugging field, and lastly, section 9.3 covers Delta Debugging.

9.1 A Brief History

Recall that type inference algorithms are a core aspect of functional programming languages today and, unfortunately, can cause inaccurate locating of type errors. However, the initial type inference algorithm extended previous work in Combinatory Logic (Curry and Feys 1958), not functional languages, and was to prove that polymorphic types, also referred to as *principal type schemes* - the most general type, can be deduced with a type inference algorithm (Hindley 1969). Programming languages use types to declare what a value will hold. For example, Haskell has basic types, such as `Bool` to represent true and false, and the ability for a programmer to declare their own, making the number of types available

infinite. This capacity to have polymorphic types means the programming language supports Polymorphism, the ability for one part of the program, such as a function, to take on many different types. Many different forms of Polymorphism now exist; however, the research in Combinatory Logic was solely working with Parametric Polymorphism.

Hindley's extension did prove that in Combinatory Logic, a type inference algorithm can discover polymorphic types. Nevertheless, the paper received no citations until 1978 when, without prior knowledge of the previous work, Milner published a similar method (Milner 1978). Though the same approaches for the type inference algorithm occurred, their application is different. One applies, as already mentioned, to Combinatory Logic and the other to the functional programming paradigm, specifically the language ML. The ML-based type inference was named algorithm W and in the 1980s was extended by Damas (Damas 1984). Thus, the algorithm became known as Hindley-Milner-Damas type inference and today is the base type system in many functional programming languages. The Hindley-Milner-Damas type system was revolutionary with its inclusion of type inference, being adopted by many programming languages such as ML, OCaml and Haskell, and inspiring other type systems. However, one aspect of the type system, its way of handling conflicts and the resulting type errors, became subject to many research papers to this day, including this thesis which would not exist without it. The next section of this related works is an overview of those papers.

9.2 Type Error Debugging

The field of type error debugging in functional programming languages appeared in the 1980s. Though the Hindley-Milner-Damas type system was revolutionary with its inclusion of type inference, in January 1986, two papers presented at the Thirteenth Annual ACM Symposium on Principles of Programming Languages pointed out the flaws.

Johnson and Walz complained that in ML, the type error was usually far from the cause (Johnson and Walz 1986). An argument that still underlines type error research today, as shown in this thesis with the example in chapter 1. Laying the blame on Robinson's Unification algorithm, the paper introduces a 'maximum flow' algorithm combined with an

editor to produce improved error messages. Presented were two principles that the authors believe are important when producing error messages:

1. *"The user's attention should be drawn to what appear to be the anomalies that are responsible for errors."*

2. *"Error indications should be complete but parsimonious; the user should see highlighted on the screen everything that contributed directly to an error, but nothing more."*

Both of which are the foundations of *Slicing* a core type error debugging category and a similar method to the adding and removing lines found in the solutions of this thesis.

The second 1986 paper also claimed that there was an issue of type error locations being far from the source, claiming that it was not helpful to the programmer if they kept receiving the wrong reported line number (Wand 1986). Summarising the problem, when the type checker can no longer solve constraints, it stops and returns that conflict as the cause. However, the issue causing the conflict could occur somewhere else, so the wrong location is reported. The solution was to modify the algorithm to record the expression it was working on and use this information to pinpoint why the error happened. Unlike Johnson and Walz, this paper provides a detailed evaluation against nine program files in which the algorithm reports the correct error location every time. Wand's view on a successful evaluation result is that if an algorithm returns at least one of the error sites as the correct location. A metric still used today for testing if type error debugging solutions are successful, including this thesis. Both 1986 papers argue that type inference algorithms have problems stating the correct location of a type error.

9.2.1 Inference Modification

The two papers described in Section 9.2 are in the category *inference modification*; laying the blame for the type error problem with the inference algorithm; the only cure is to modify or replace it. The solutions in this thesis do neither; however, it is essential to understand this significant part of type error debugging history.

Almost ten years after these papers were published, readers have an introduction to another inference modification solution (Bernstein and Stark 1995). As an extension to

Algorithm W in the ML programming language, the solution supported open expressions, those with free variables, instead of just the standard closed expressions, those without free variables. A prototype implementation worked with 'breakpoints' placed in the source code, this is an unbound variable, to force type information from the compiler at specific locations. However, an evaluation of the usefulness of this method was not provided.

Another paper that also applies its method to ML was published one year later and introduced the idea that the decisions the type inference algorithm made should come with explanations (Duggan and Bent 1996). The method was embedded into the Robinson unification algorithm of ML and recorded the steps taken before instantiating a type variable. However, though it is seen as a framework for other programming languages, it went no further, leaving space for more solutions in this domain.

Another promising solution was to replace the type inference algorithm entirely and use one that had been around as folklore; Algorithm M (Lee and Yi 1998). Never widely used due to not having formal proofs to back it, the type inference algorithm M was only used in some compilers. Lee and Yi's aim was three-fold: show that the algorithm was sound and complete, that it stops earlier than algorithm W, and implement a method to automatically allow programmers or the compiler to swap between the two algorithms. Their view was that the two algorithms had specific strengths and should change between them depending on the situation. They fulfilled all three of these aims. Evidence for the first aim came from presenting the formal proofs needed. Algorithm M did indeed stop earlier due to visiting fewer nodes in the AST; lastly, the ability to swap algorithms was completed with positive results.

As the 90s ended and the 2000s began more papers on inference modifications were seen (Yang 1999; McAdam 1999b; Lee and Yi 2000; Choppella and Haynes 2002). One of these papers, by McAdam, expanded into his PhD Thesis and argued that Algorithm M did not eliminate one of the most critical issues that Algorithm W suffers from; left-to-right bias (McAdam 2001, 2002). Left-to-right bias lies in the inference algorithm always type-checking the left subexpression before it checks the right. This bias towards checking the left-hand subexpression first means that the wrong side of the expression, the right side, sometimes gets blamed. One solution is to show that the issue happens because the

function application's left and right sides do not match. McAdam's did this by introducing a new algorithm *Us*, a modified algorithm W, to check both sides of the subexpression before using Robinson's unification algorithm (McAdam 1999b). Left-to-right bias still prevails in the Hindley-Milner-Damas type system to this day, and due to this, many varied solutions have been suggested (Heeren, Hage and Swierstra 2002; Jun, Michaelson and Trinder 2002; Kustanto and Kameyama 2010; Charguéraud 2014). However, before moving on to the next category, there is one more notable paper in this domain.

The papers discussed have implemented their solutions in standard compilers that target functional languages like ML and Haskell. On the other hand, Helium is a specialist compiler for a subset of Haskell aimed at providing excellent error messages for novice programmers (Heeren and Hage 2002; Heeren, Leijen and van IJzendoorn 2003; Hage and Keeken 2006; Burgers 2019). In *Parametric type inferencing for Helium* the reader is introduced to the type inferencer Helium uses, whose implementation is flexible enough to mimic both algorithm M, W and others for evaluation purposes. The critical aspect of their type inferencer was solving constraints, a core element of type inference and discussed in chapter 2, globally by employing a type graph to avoid the left-to-right bias. Type graphs have been mentioned before in the type error domain, with McAdam summarising their application previously (McAdam 1999a). However, Heliums 'type graphs' represent all of the constraints before being solved. The edges are the equality between the types, and vertices represent types themselves with information stored that represents a path that can increase the chances of which constraint is to blame (Figure 24).

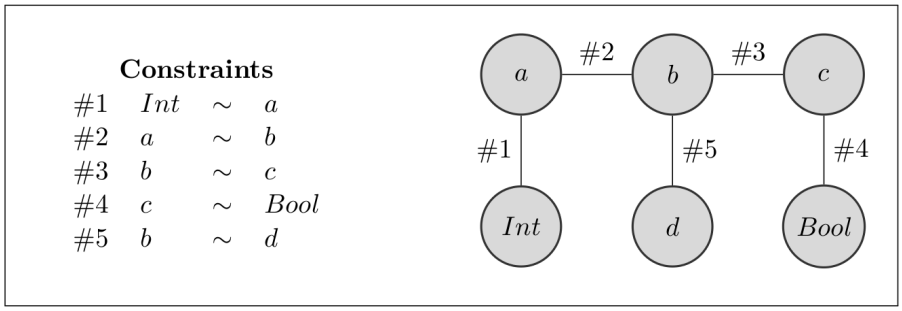


Figure 24: A simple type graph (Burgers 2019)

Though the paper describes Helium's type inference method overall, its main contribution

is how its algorithm handles constraints. In which order constraints are solved is a category reviewed in Section 9.2.3. Nonetheless, this paper's placing under the inference modification section shows how interlinked the type error debugging domain is and how choosing where to place papers is not always easy.

As seen, there have been many inference modification solutions; however, during the later years, a new way of improving type error debugging started to flourish.

9.2.2 Interactivity

The research so far concentrates on locating the type error while ignoring the critical concept of context from the programmer. Context in the realm of this thesis means information from sources that explain what a program does. In functional languages that use type inference, signatures and annotations are the core context a programmer can provide in their programs. However, they are an optional feature. If the programmer correctly produces them, broken type signatures or annotations will still produce the wrong location in type error messages. So the idea of gathering extra context from the programmer was born.

The first steps into *Interactivity* were 'context-driven', meaning that the discovering of the intended types were solely formed by interacting with the programmer. In 1993 a draft paper (Rittri 1993) pointed out how pinpointing an actual error is challenging without the programmer's intentions and implemented Wand's idea of providing multiple points of errors in a program; a method the debuggers in this thesis use. However, to supplement this, an interactive interface, one that asks programmers questions about the types of these points of error, was suggested (Wand 1986). Unfortunately, as far as is known, the draft never made full completion. The next mention of Interactivity in type error debugging was not seen again until 1993 when Beaven and Stansifer introduced a method to ask *Why* and *How* type errors appear in a program (Beaven and Stansifer 1993). However, they did not implement any interactive behaviour, only surmising what it would be like if their approach had that functionality. Eight years later, in 2001, their idea emerged when the type error debugging field had its first introduction to Algorithmic Debugging.

In 1982 Algorithmic Debugging was introduced to compare the program's workings to

what the programmer thinks they have written (Shapiro 1982). From those humble beginnings, it inspired countless researchers in non type error debugging fields (Naish and Barbour 1996; Silva 2006, 2011; Caballero, Riesco and Silva 2017). Nevertheless, it was not until 2001 that Chitil introduced Algorithmic Debugging to type errors in functional programming [1]. His fusing of debugging domains involved merging algorithmic debugging with a compositional graph; each explanation of why an expression had a specific type had to be small and have meaning on its own. (Chitil 2001).

```
Type error in: (last xs) : (init xs)

last :: [[a]]->a
Is intended type an instance? (y/n)   n

head :: [a]->a
Is intended type an instance? (y/n)   y

reverse :: [[a]]->[a]
Is intended type an instance? (y/n)   n

(++) :: [a] -> [a] -> [a]
Is intended type an instance? (y/n)   y
```

Figure 25: A small snippet of algoritmic debugging interaction (Chitil 2001)

A compositional graph is a tree-like structure where the types of the child nodes decide the types of the nodes. A new inference algorithm, similar to Algorithm W, structures the tree. Chitil's type inference algorithm generates the tree by copying the entire inference tree to every place where the polymorphic value appears. The graph is then traversed with Algorithmic Debugging using an Oracle; in this case, the oracle is the programmer themselves answering yes or no to a set of questions about the types they expected expression and variables to have. The algorithm starts "breadth-first", going deeper after each answer from the programmer until it pinpoints the type error position. In an implementation, it is stated that the method lacked efficiency. However, Chitil continued to improve on algorithmic debugging, both in and out of the type error domain, over the next two decades (Chitil 2004; Silva and Chitil 2006; Chitil and Davie 2008; Tsushima and Chitil 2014, 2018) with other authors also applying it to type error debugging tools (Stuckey, Sulzmann and Wazny 2003a) as well as type error debugging in OCaml (Tsushima and Asai 2011) and Scala

[1] An earlier paper did apply algorithmic debugging to type errors; however, this was in a logic programming domain (Naish 2000)

(Plociniczak 2013).

Along with context-driven Interactivity, visual-driven Interactivity helps the programmer discover why the type error exists using visualisation to aid them. One such solution, a visualisation of Polymorphic Type Checking, gave us a taste of a different way of looking at types. Its core aim is to provide a visual representation that allows programmers to understand types more efficiently; however, its methodology also lends itself to debugging type errors as well (Jung and Michaelson 2000). The graphical representation is in the style of visual programming systems that typically use boxes or icons; in this case, using icons would cause clutter, so rectangles containing different colours and patterns were applied. The use of the rectangles meant that it should be quick to see if two types do not match; for example, if there were two rectangles with different patterns and colours, it would be easy to tell the difference. To evaluate, they had programmers with ML experience, and the text-based approach tests what type a function was expecting to return when given a specific argument. Unfortunately, they found that this programming style for type checking was no better or worse than using pure text and that those who used it could not agree on its usefulness in type error debugging. There is currently no evidence to back the use of a visual style, so this thesis's solutions stick with text only, showing the results as just the line numbers from the source code.

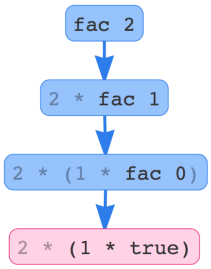


Figure 26: Snippet of a visual-driven method (Seidel, Jhala and Weimer 2016)

However, in 2016 another form of visual-driven debugging was presented, allowing the programmer to step through a program graphically (Seidel, Jhala and Weimer 2016). Again, type error debugging was not the paper's core idea; however, Dynamic Witnessing, which takes a programmer's source code and produces a graphical representation, allows programmers to browse through each step of a program run. This witnessing of how the program

appears at each stage allows the programmer to hypothesise why the type error occurred; a method, also seen in other visual-driven systems; for example, Haskeu, which provides an editor for programming that can turn GHC error messages into visual icons (Alam and Bush 2016). As previously mentioned, these solutions overlay heavily with the type error debugging tools discussed in Section 9.2.5.

9.2.3 Constraints

If a solution can find the minimal set of constraints that cause a type error, it can report its location and why it occurred much more efficiently. One such way to gather these subsets is slicing, discussed in Section 9.2.4, with another relying on changing the order in which the constraints are solved. 'Correcting Type Errors in the Curry System' showed an early application of managing constraints for type error debugging. The solution produces a constraint set that was the maximum that could be declared as containing an error (Gandhe, Venkatesh and Sanyal 1996). However, though they have positive results for the Curry type system, one that uses extrinsic types - similar to an untyped language, they state that the output was unknown for Hindley-Milner, which uses intrinsic types - the types are part of the semantics. Thus, it was not until six years later that readers saw constraint solutions applied to functional programs.

Heeren et al. generated type constraint graphs in a new type inferencer (Heeren et al. 2002). This method aims to eliminate the issue behind left-to-right bias by making unification and substitution symmetric and to do so uses a heuristic combined with separating constraint generation and solving (Heeren, Hage and Swierstra 2002). The separation of these two aspects meant that a delay in solving the constraints would allow the unification algorithm to overview all the constraints meaning it can try and solve the entire set of constraints with heuristics first (Heeren, Hage and Swierstra 2003b,a; Hage and Heeren 2005, 2006, 2009)[2].

Though Heeren et al. provided many heuristics for constraint solving, other ideas also bloomed, incremental constraint solvers, constraint handling rules with justifications, dualise, and advanced data mining techniques, all leading to the generation of minimal unsatisfiable constraints.(de la Banda, Stuckey and Wazny 2003; Stuckey, Sulzmann and Wazny 2003c,

[2]In Heeren's PhD Thesis a complete discussion of this work can be found (Heeren 2005)

2004; Bailey and Stuckey 2005; Wazny 2006; Stuckey, Sulzmann and Wazny 2006; Sagonas, Silva and Tamarit 2013). MYCROFT, a type inference debugger, is one such solution (Loncaric et al. 2016). Using two arguments, one the type constraints generator and the other the type solver, MYCROFT generates sets of typing constraints testing them against the solver. If the solver returns that the constraints have failed, MYCROFT then uses an SMT solver to narrow down the constraints repeatedly until successful results appear. This is similar to the adding and removing of lines and applying the variants to a testing function in the debuggers in this thesis. Another similar aspect is that the solution is also agnostic. Recall that in this thesis, agnostic means a solution can work with several programming languages. MYCROFT works with both OCaml and a sub-set of Javascript without knowing anything about their constraints or the layout of their source code and thus is agnostic.

However, another important trend was joining constraint solving. SHErrLoc (Zhang et al. 2015b, 2017) and Skalpel (Rahli, Wells and Kamareddine 2010; Rahli et al. 2016) both take similar approaches and use Slicing. To highlight its slices, the former concentrates on the Bayesian principles applied to a graph of unsatisfiable constraints, first suggested in 2015 (Zhang et al. 2015a). The latter uses a strict ordered constraint system to provide Slicing, which colour-codes the source code to direct the programmer to the cause of their type error.

```
val rec f = fn x => (x (fn z => z), x (fn () => ()))
val rec g = fn y => y true
val u = f g
```

Figure 27: Skalpel's Colour-Coding (Rahli et al. 2016)

Highlighting source code to improve type error understanding is a common practice in type error debugging tools, discussed in Section 9.2.5; however, next, a look at the method used by Skalpel and SHErrLoc, Slicing.

9.2.4 Slicing

In 1982 *Programmers Use Slices When Debugging* was published (Weiser 1982), showing that programmers natural slice when debugging their source code. The programmers strip out all unnecessary aspects of the code to narrow down the error leaving only the elements that do

cause it (Kamkar 1995; Binkley and Gallagher 1996). Slicing was initially proposed for run-time errors, however, in Wands 1986 paper (Wand 1986), he describes an unnamed method that maps directly to Slicing, and following its introduction into the type error debugging domain, many solutions used this technique (Zhang, Gupta and Gupta 2006; Rahli, Wells and Kamareddine 2009; Rahli et al. 2015; Tsushima and Asai 2013). Slicing discovers all the points in an ill-typed program that contribute to a type error. Unlike when unification fails, with only one location reported, Slicing reports all areas that are to blame, with a slice said to be complete if all the locations reported are involved in causing the type error. The debuggers in this thesis also report all sources of the type error by returning a result of all line numbers that may contain the type error. However, unlike Slicing, which returns a slice of an ill-typed program, this thesis's debuggers report the difference between an ill-typed and a well-typed program.

The first to use the name *program slices* in type error debugging were Choppella and Haynes in a technical report; however, the first published paper to mention the term was produced by Dinesh and Tip two years later (Choppella and Haynes 1995; Dinesh and Tip 1997). Nevertheless, in 2003, it was Haack and Wells's most noticeable contribution. Their two papers apply program slicing with constraint handling to type errors and aim to supply a minimal explanation. (Haack and Wells 2003, 2004).

```
(.. y => (.. y + (..) .. (..)::y ..) ..)
```

Figure 28: A minimal program slice (Haack and Wells 2004)

In program slicing, a minimal explanation shows that removing extra *slices*, a part of the ill-typed program - a line or character, for example, will make the type error disappear. The slices returned are those who contribute to the error and nothing more. The minimising here is identical to the simplifying Delta Debugging algorithm discussed in Section 2.2. Haack and Wells's stated that before minimising the program slices, they must first be stored. In this case, program slices are information attached to type constraints with a function to keep track, similar to Wand's suggestion of storing expressions. It is this association that is essential for locating type errors. Two algorithms, named minimisation and enumeration, run several times on the initial constraint sets to find the minimal points. First, the enumeration

algorithm finds the almost minimal points, and then the minimisation algorithm is repeatedly called to find if each slice does or does not contain an error. Once the minimal points are stored, slices are generated, and two user-friendly ways of presenting them were introduced: to highlight the slices or remove any surrounding code from the slices (For the latter, see Figure 28).

Haack and Wells's method and others such as Skalpel worked over constraints; however, in this section, the last paper is constraint-free (Schilling 2011). Applied to the Glasgow Haskell Compiler (GHC), the constraints-free technique works by applying slices directly to the Abstract Syntax Tree via a GHC API, meaning no modifications to the compiler is necessary. This technique is the first time a solution uses a compiler that does not need to be modified, just like the debuggers in this thesis; however, unlike Schillings, the debuggers in this thesis do not work on the Abstract Syntax Tree. Once submitted, the GHC type checker is used as a blackbox, compared to the solutions within this thesis that use the entire compiler to tell if slices are compliant with the type errors existence; if it is gone, that slice must be part of the error. This constraint-free method is based on SEMINAL, a type error debugging tool that uses many solutions mentioned in the prior sections and will be discussed further in Section 9.2.5.

9.2.5 Debugging Tools for Type Errors

Many type error debugging papers mentioned in the previous sections do not provide an implementation for programmers. Those that do are categorised here as having a debugging tool. A tool in the interpretation of this thesis is either a physical entity such as a debugging environment or a patch for an existing compiler that provides additional descriptive sources without modifying the underlying type system. Tools can also work on run-time errors or type errors. However, for this related works, the bulk of the tools described locate only type errors. All the solutions in this thesis fit this definition. Tools for functional programs that also fit this description start appearing as far back as the 1980s. However, it was not until the following decade that debugging tools became more frequent in publications. Introduced in 1990, one of the early functional debugging environments uses traces displayed as trees that provide interaction (Kamin 1990). The idea is that using a point and click interface the

programmer can investigate the route their program has taken during the bug's appearance. The author's prototype, unlike former debugging tools, supports functional programmings biggest feature, lazy typing. However, it does not concentrate on type errors similarly to other debugging tools aimed at functional programming (Lapalme and Latendresse 1992; Hazan and Morgan 1993; Sparud 1995; Chitil, Runciman and Wallace 2000; Marlow et al. 2007).

In 1997, Whittle, Bundy, and Lowe stated that when programming in Standard ML *"The most common errors were syntax errors and type errors."* and *"The students found it particularly difficult to pinpoint the source of a type error."* (Whittle, Bundy and Lowe 1997). A view cemented in 2015 with an in-depth student observation, and again in 2017 with a study of 55,000 programs written by novices (Tirronen, Uusi-Makela and Isomottonen 2015; Wu and Chen 2017). The 1997 paper's authors implemented an editor, the first functional programming tool that addresses type errors specifically. However, instead of improving the locating and reporting of type errors in the language itself, the authors use the tool, CyNTHIA, to force the users to declare all type signatures. With the type signature already stated, the tool checks the function body with its, not the compilers, type checker. The tool automatically fixes any errors in the function body, meaning any program written in the tool is already well-typed, preempting any type errors. This method is far from the solutions in this thesis which, as they have no syntactic knowledge, do not get affected by type signatures. Two years later, Keane provided a different view on type errors which concentrated on providing improved feedback (Keane 1999).

Unlike the authors of CyNTHIA, Keane focuses on advancing the reporting of type errors to the user. Ignoring any improvements to locating, different from previous solutions in this Chapter and the solutions of the thesis, the *point and click* approach replaces manual debugging activities on source code. One such activity is allowing the user to highlight expressions in the source code to retrieve type information which is also seen in other solutions (Simon, Chitil and Huch 2000; Neubauer and Thiemann 2004). Along with automating specific tasks, Keane concentrates on the debuggers ability to provide error messages with enhanced typing details, an area others also target (McKenzie and Wyber 1999; Chen, Erwig et al. 2013; Chen, Erwig and Smeltzer 2017).

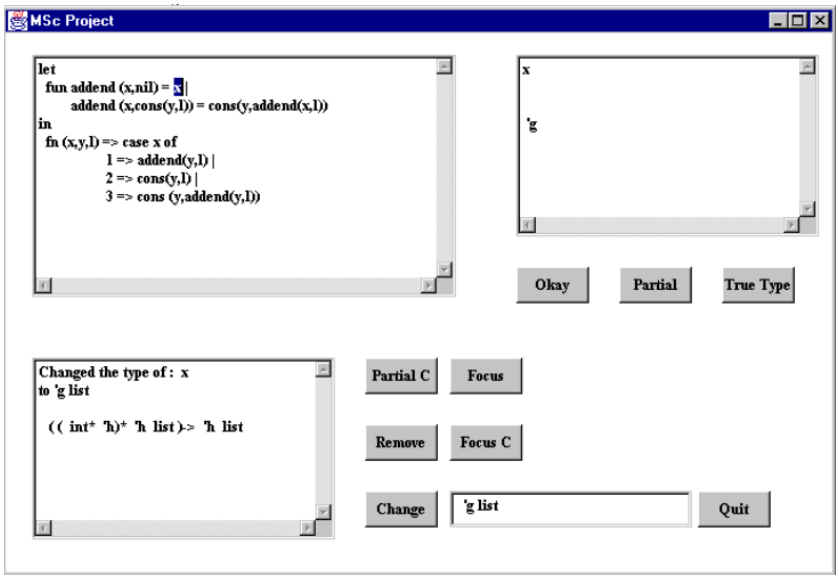


Figure 29: *point and click* approach to type debugging (Keane 1999)

One such tool aimed at improving type error messages is the Chameleon Type Debugger (Stuckey, Sulzmann and Wazny 2003b,a) and in contrast to Keane's *point and click*, the Chameleon Type Debuggers interface is text-based. As previously mentioned, slicing solutions, such as Skalpel, Chameleon, highlights minimal causes of type errors. Chameleon does this by underlining the places in the source code that cause the error. In addition to this, the debugger also allows the user to investigate the highlighted sections further. To query the source code, Chameleon has a range of commands to apply. One such command allows the user to receive more in-depth type information about the overall function that has the type error. Another allows the user to delve deeper by specifying which variables and expressions to check. The ways that Chameleon dealt with type errors became common in this domain, as will be reported in the rest of this section.

Nevertheless, just one year later, TypeHope is introduced, which designs its tool around how programmers naturally debug (Braßel 2004). The author's of TypeHopes opinion is that programmers naturally debug by adding and removing the faulty program lines, in the same way the solutions in this thesis work. However, instead of following the idea, TypeHope takes a slightly different path and adds and removes dummy expressions, *an expression of an arbitrary type*, into the program via the Abstract Syntax Tree (AST). With the newly added dummy expressions, each program's variant is type-checked, returning what the type

checker states are the correct type. Once all variants have had their results returned, a heuristic is then employed to weigh the replaced dummy expressions and report them to the programmer using the same style as Haack and Wells and the previous solution, Chameleon, highlighting the slices that need considering. Unfortunately, for this to work, a polymorphic type signature is always added, an addition that is now known to not always work as intended (Wu and Chen 2017). TypeHope is the first approach to suggest using the type checker as a blackbox[3], relying on the type checker as an oracle instead of the programmer as algorithmic debugging does.

In 2006, another tool appeared, that used the type checker as a blackbox, named SEMINAL (Lerner, Grossman and Chambers 2006; Lerner et al. 2007). Unlike previous solutions and the ones in this thesis, SEMINAL was not an external tool and came in a patch for the OCaml compiler, sitting between parsing and type-checking. SEMINAL contains several previous methods approaches: Slicing, blackbox type checker, weighted heuristics, and dummy types. SEMINAL works by inputting wildcards, similar to TypeHopes dummy types; however, instead of modifying the source code, the wildcards are placed directly into the Abstract Syntax Tree (AST). These wildcards can either change the expression's order, move two arguments, for example, or replace a sub-expression completely. After each change, the tool uses the type checker, not the whole compiler, as a blackbox to see if the changes have been successful and have removed the type error. Like TypeHope, once the tool had collected the minimal changes, they were ranked and presented along with a suggested change report to the programmer.

SEMINAL has four parts: Search, Enumerator, Type Checker, and Ranker. The searcher tells the Enumerator where to place the changes. Starting at the root of the AST, it replaces the expression with a wildcard, then type-checks. A record of the change happens when the AST modification is well-typed. The tool then continues to do this with all sub-expressions until they have a collection of well-typed modifications or changes when including a wildcard. Finally, the Enumerator suggests the syntax changes using a set of replacement rules to implement when it comes across certain expressions. The authors describe the Enumerator as *a giant case expression that matches the node to a list of modifications.* One example

[3]However, the author does not use that term to refer to the idea.

shows that if the issue is with the function arguments, the Enumerator should try moving the arguments around. The authors separate these changes into two categories; tree-based mods, such as rotating arguments, and language-specific, such as curried arguments. The authors state that the language-specific category is non-exhaustive; however, they claim it is easy to add new modifications into the debugger. The type checker is the third part; it is unmodified and, as such, used to tell if a program is well-typed or not. Using the type checker as a blackbox is similar but not the same as the blackbox compilers used in this thesis. Lastly, the Ranker ranks the modifications and produces the suggestions for the programmer, as seen in Figure 9.1. The authors admit they do not know the programmers intent, so they use a heuristics set to decide what to suggest. The first is the preference for smaller changes to the AST, like a single node rather than a whole sub-tree. Next, the Enumerator gives preference to specific rules. Then the hiding of intermediate modifications from the programmer, and finally, three metrics measure the distance of the AST modifications.

```
File "/home/jo/CE5-Ocaml.ml", line 4, characters 17-20:
This expression has type string but is here used with type 'a list
Relevant code: "a"
------------------------------------------------------------------
File "/home/jo/CE5-Ocaml.ml", line 4, characters 17-27:
Try replacing
    (addList "a" ["b"])
with
    (addList ["b"] "a")
of type
    string list
within context
    let v5 = (addList ["b"] "a") ;;
```

Listing 9.1: Shortened SEMINAL Result

After the initial introduction of SEMINAL, the authors released a secondary paper covering two extra implemented features. The first is the adaption to context, the ability to

distinguish between a function that contains a type error from part of a program that might be well-typed individually, and the second is locating multiple type errors in one program. Unfortunately, in private correspondence with the first author of SEMINAL for this thesis, it was noted that the tool no longer works with the latest OCaml compiler. However, there was no reason why it could not be updated to do so.

Five years passed before other type error debugging tools appeared. One aimed, like SEMINAL, at OCaml uses the algorithmic debugging solution from Chitil, as mentioned in Section 9.2.2, and pairs it with a blackbox type inferencer (Tsushima and Asai 2012). Another from the same year concentrates on Scala using a *point and click* solution similar to Keane; however, it differs by incorporating interactive visualisations of the type-checking process (Plociniczak and Odersky 2012). A few years later is the introduction of both Skalpel and, shortly later, SHErrloc. As already mentioned in Section 9.2.3, both are based on constraint solving paired with Slicing. Both Skalpel and SHErrloc and several other solutions mentioned in this related works section came under scrutiny in 2018 by Chen and Erwig, who discussed their positive and negative attributes (Chen and Erwig 2018). To respond to the issues raised, Chen and Erwig extended their paper on counter-factual typing, first introduced in 2014 (Chen and Erwig 2014a). When receiving a type error, if the expected type is different from the actual type, this is counter-factual. *Counter-factual typing* finds all the changes that can be made to a type that will remove a type error. Once all of the changes are collected, three rules are applied. First, the changes are presented to the programmer in size order with the minor changes, a change to a single type, primary and the others hidden unless the programmer says none of the initial set works. Second, suggestions are made that do not rely on the semantics of the program. Finally, the suggestions are ranked by a set of heuristics. An evaluation against Seminal, GHC, and Helium showed that the method successfully outperformed all three. In 'Guided Type Debugging' Chen and Erwig combine counter-factual typing with algorithmic debugging (Chen and Erwig 2014b). Here they extend the counter-factual idea by placing the changes found by *counter-factual typing* into a graph. According to the programmer's input, the graph is then traversed, which states their intentions for the program to narrow down the solution's changes.

In 2019, a type error debugger, in the paper *Type Debugging with Counter-Factual Type*

Error Messages Using an Existing Type Checker, also combined counter-factual typing with algorithmic debugging; however its authors also applied a blackbox type checker (Tsushima, Chitil and Sharrad 2019). As an example, take this ill-typed OCaml program:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

The one-line program defines a function `f` that takes two arguments; using the first argument `n` in a function that is mapped to the second argument `lst`. However, there is a type error and OCaml gives the following error message that does provide a possible fix:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0

Error: This expression has type float
  but it should be an expression of type string
```

This error message is counter-factual; the expected type, string, is different from the actual type, float. As already noted, the suggested fix is viable; however, the provided solution is only one of the options that will make the program well-typed. Another viable suggestion is that the string concatenation, `^`, is incorrect as the programmer intends to apply the function to a list of floats, meaning replacing the incorrect code with `**`. With this alternative intent in mind, a better error message would have been:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0

Error: This expression has type string -> string -> string
  but it should be an expression of type 'a -> float -> 'b
```

However, the compiler does not know the programmer's intent. One solution to this issue is to highlight all areas in the program that cause the faulty positions. With the example given this would give four faulty positions:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

This highlighting of all positions is called counter-factual typing, recall that this was introduced first by Chen and Erwig in 2014 (Chen and Erwig 2014a, 2018). The solution in the paper gathers a list of these counter-factual typings by replacing leaves in the Abstract Syntax Tree (AST), basically any potential type error location, with holes. These variants of the AST are then sent to a blackbox type checker to report what each hole should have as its type. Once the expected and actual types are gathered, the programmer receives a list of the reported counter-factual typing. However, looking at the example program, which is only one line long, four counter-factual positions are found. More extensive programs could produce more counter-factual typings, so asking the programmer to select a position from a long list is not viable; this is where algorithmic debugging is applied to counter this issue. The solution uses algorithmic debugging, described in Section 9.2.2 as a way to ask for the programmers intent, to get the programmer to re-fill the holes. Shown below using the previous example:

```
Choose your intended type for this expression.

let f = (fun n -> (fun lst -> List.map (fun x -> x ^ n) lst)) in f 2.0

A: float
B: string
Your choice (C: another type):
```

The two type choices listed under A and B are the gathered types from the combination of counter-factual typing and the blackbox type checker. The first is the Actual type, and the second is the Expected type, with a third option allowing for an unknown type. The programmer then chooses the closest match to their intent. In this example, they choose A, a float, to be the type desired. This type then replaces the hole that the question is referring too and then the program is given back to the blackbox type checker.

```
let f = (fun n -> (fun lst -> List.map (fun x -> x ^ (n:float) lst))in f 2.0
```

Again the blackbox type checker either returns with if the new modified AST is well or ill-typed. If it is the latter, the algorithm generates another question from a different randomly chosen counter-factual typing and again calls the blackbox type checker. If receiving the former, the algorithm stops and returns the location of the type error. With this example, after the blackbox type checkers result of the program still being ill-typed, the program now only contains a singular counter-factual typing. Due to having the prior knowledge of the programmers intent to use a float, the only viable option for the type errors location is in the position of ^. So the algorithm terminates with the following error message:

```
let f = (fun n -> (fun lst -> List.map (fun x -> x ^ n) lst)) in f 2.0

Error: This expression has type string -> string -> string
  but it should be an expression of type 'a -> float -> 'b
```

This solution takes the number of locations for the programmer to work on from four to one and in an evaluation shows an improvement from the unmodified OCaml compiler of locating 80% more correct locations as well as asking 30% fewer questions than a rival solution, seen in Figure 30.

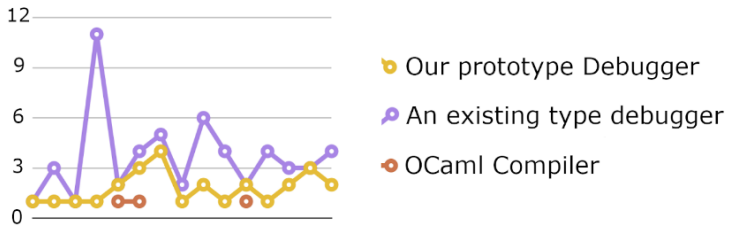


Figure 30: Evaluation results (Tsushima, Chitil and Sharrad 2019).

However, this is now not the only way the term *blackbox* is used in the type error debugging domain. Another usage of a blackbox uses the entire compiler and, unlike blackbox type checkers, does not need modifications to the underlying compilers code-base. This version of *blackbox* is unique to this thesis.

This section concludes the related works on type-error debugging. Nonetheless, other papers from the type error debugging domain must be briefly noted here as they did not fit neatly into the categories above. These include: adopting data flow for explanations (Gast 2004), applying lazy-typing (Chen, Erwig and Smeltzer 2014), using SMT solvers (Pavlinovic, King and Wies 2014; Pavlinovic 2014; Pavlinovic, King and Wies 2015), application on domain specific languages (Hage 2014a,b; Serrano and Hage 2016; Serrano Mena and Hage 2016), a data-driven approach (Seidel 2017; Seidel et al. 2017), and the connection between gradual typing and type error debugging (Chen and Campora 2019).

9.3 Delta Debugging - Usage and Development

Recall in Chapter 2 the description of Zellers Delta Debugging algorithms; here, a few unique non-type error debugging solutions, either categorised as including Delta Debugging or trying to improve upon the algorithm, are discussed. Note that only one solution is applied to the functional programming domain using the *simplifying* algorithm, whereas the solutions in this thesis use the *isolating* version. In 2005, the first solution to incorporate Delta Debugging appeared (Gupta et al. 2005). The authors slicing tool combines forward and backwards slicing with isolating delta debugging. The tool operates by using delta debugging to provide a minimal input for the slicing algorithms to use. The tool then combines this reduced input with the programs faulty outputs, with the slicing algorithm working forwards on the minimal input and backwards on the faulty outputs. The slicing results are called failure-inducing chops and are smaller than just using slicing as the only sections checked are those that delta debugging returned as causing the error. Once the faulty chops are collected, the algorithm terminates, and a comparison between chops shows which line of the faulty program contains the error. The author's evaluation is positive, with evidence that the combination of slicing and delta debugging provides more minor results than either solution individually. However, other authors argue that delta debugging is valid as a standalone solution if tailored for specific environments with evidence provided by solutions covering areas such as web services (Hammoudi et al. 2015), micro-services (Zhou et al. 2018), and big data (Gulzar 2018). Three years after the first hybrid solution combined

delta debugging with slicing came a solution that combines delta debugging with static and dynamic analysis techniques (Zhang et al. 2008). Like the slicing solution, this solution first calls upon the analysis techniques before using delta debugging to minimise the results further. However, unlike the slicing solution, the analysis-hybrid uses delta debugging as a three-phase algorithm, changing the granularity in three individual stages whilst applying it to a call graph hierarchically rather than when receiving an unresolved result.

With *Hierarchical Delta Debugging (HDD)* (Misherghi and Su 2006, 2007) which also works hierarchically, the environment is programming languages, for instance, XML, that produce tree-like structures, an environment also targeted by other delta debugging solutions (Hashimoto, Mori and Izumida 2018; De Bleser, Di Nucci and De Roover 2020). The algorithm, HDD, works with the simplifying version of delta debugging, calling it on each level of a tree-like structure, for example, an Abstract Syntax Tree (AST) as seen in Figure 31.

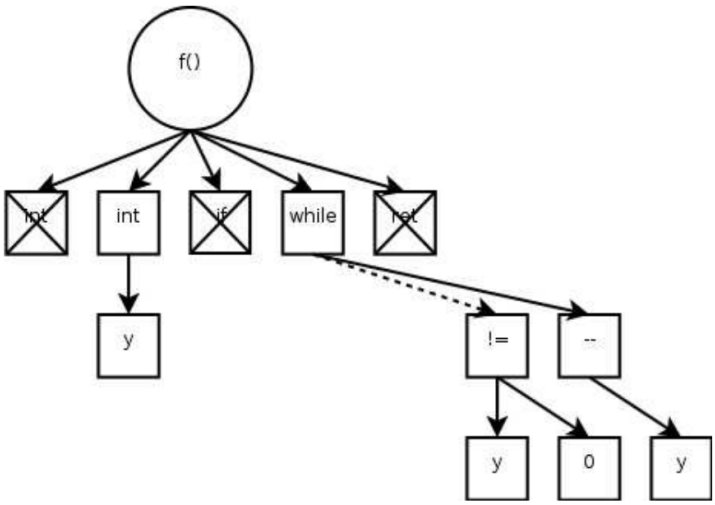


Figure 31: HDD after one application on an AST (Misherghi and Su 2006).

The algorithms application starts from the top of the tree structure, recursively applying delta debugging on each layer until it reaches the bottom. For each layer, delta debugging reports back the minimal causes of the error; the algorithm then minimises these results further, using a heuristic to remove results that are not relevant. The evaluation shows that the number of locations reported is smaller than from delta debugging alone; however, as

the algorithm does not backtrack up the tree structure, there is no guarantee of returning a singular location. Unfortunately, additional research also found issues with HDD, with one remarking that HDD's outcomes are not efficient or accurate, and others suggesting improvements before the algorithm becomes practical (Yu et al. 2012b; Hodován and Kiss 2016a; Kiss, Hodován and Gyimóthy 2018; Christi 2019). However, this did not stop the incorporation of delta debugging in other solutions.

In 2011, *Iterative Delta Debugging (IDD)* extended delta debugging by allowing it to work on source code from the past that contains previously fixed bugs (Artho 2011). As already discussed in Section 2.2 *isolating* delta debugging works with two configurations, one that works and one that contains a bug. The author of IDD points out that many programs contain a history of working configurations within repositories; however, they note that this historic source code could contain legacy bugs that had already been discovered and fixed, confusing the delta debugging algorithm. IDD aims to remove this confusion.

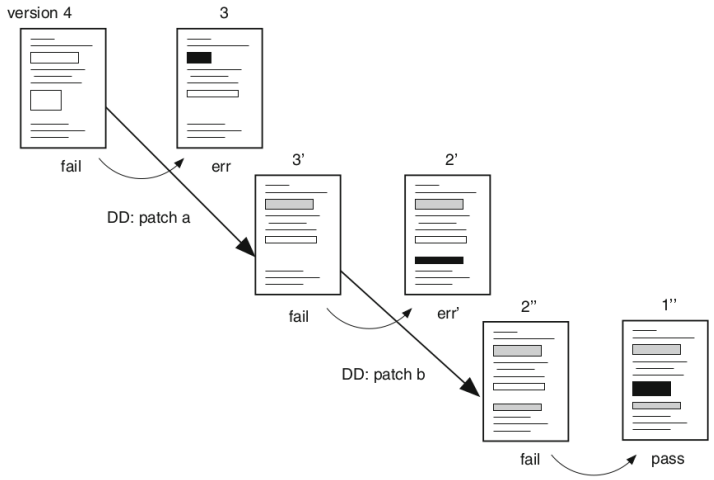


Figure 32: An example of IDD (Artho 2011).

The algorithm, IDD, does not start with a working configuration, only a failing one. To discover a working configuration, the algorithm repeatedly requests an older version of the source code from the program's repository. When the older version no longer shows the bug, it becomes the working configuration. Delta debugging is then applied, and the test function produces one of the following outcomes: fail, pass, and err. Err means that the bug is still present, and the source code differs. With an Err outcome, the source code

from the failing configuration replaces the source code in the working configuration that the test function highlighted, and delta debugging is applied again. The algorithm runs until both the working and non-working configurations are identical apart from the failing locations. The *Iterative Delta Debugging* algorithm successfully extended isolating delta debugging's configurations feature. However, other researchers still felt that the underlying delta debugging algorithm had faults, for example, scalability, that restrict its usage in the real world (Yu et al. 2012a; Elyasov et al. 2014). One solution, Picire, aimed directly at delta debugging scalability by decreasing the algorithm's time to run via parallelisation (Hodován and Kiss 2016b). The authors recognise sections of the delta debugging algorithm that lend themselves to parallelisation, replacing such sections with parallel loops. With the change to parallelisation, the solution significantly improves the running time of delta debugging, reducing the time taken by 7580%. Two years later, another paper that had a reduction of 65% on average, in time taken to delta debug, concentrates on revisiting configurations sections (Gharachorlu and Sumner 2018). Delta debugging revisits the configuration sections after it reduces; sending these same sections to the testing function several times leads to large overheads. The solution introduced stops the revisits. Calling their solution *One Pass Delta Debugging (OPDD)* the algorithm only allows repeat calls to the testing function with the same variants once per granularity change. As already stated, the modification reports significant positive results and works with the isolating delta debugging algorithm, unlike the last selection of papers in this section.

Similar to other solutions in the last few years, the following paper concentrates on the simplifying delta debugging algorithm (Christi et al. 2018; Gopinath et al. 2020; Zheng et al. 2020; Kiss 2020). In *Binary Reduction of Dependency Graphs*, the authors aim to reduce invalid inputs in Java bytecode by using dependency graphs (Kalhauge and Palsberg 2019). In the case of the paper, they note that languages that have code dependent on each other cause a longer run-time of delta debugging. The idea is similar to the Moiety algorithm presented in this thesis; however, instead of having issues with 'Parse Errors on Input', *Binary Reduction* finds references between classes as the cause. Another difference already mentioned is that Moiety works directly on source code, unlike *Binary Reduction* that works

on dependency graphs. A dependency graph comprises nodes representing classes in a program and edges that are the references between them. Similar to Moiety, which does not allow the removing of lines that cause 'Parse Errors on Input', when running their modified delta debugging algorithm, named Verify, all the classes referenced by the non-working program cannot be removed. Unfortunately, like the evaluation of Moiety found in chapter 6, Verify also led to several classes, unlike lines in Moiety, to be returned. Nevertheless, their extension to the simplifying delta debugging algorithm achieves a 12x speed increase over vanilla delta debugging; however, it does not support the strength of delta debugging, its agnosticity, which it leaves for future work.

The final paper of this section applies the Simplifying Delta Debugging algorithm to a specific environment: Liquid Haskell. Unlike all the previous solutions in this section, Simplifying Delta Debugging is applied to the functional programming domain (Tondwalkar et al. 2016; Tondwalkar 2016). Liquid Haskell is a type checker for the Haskell language that allows programmers to annotate using the Liquid Haskell notation to verify programs. The author's idea is to use delta debugging to improve the type checking. Unlike the solutions in this thesis that work on source code, Tondwalker took Delta Debugging and used it to minimise the constraint set. Allowing for the return of more than one minimal unsatisfiable set, something the original Delta Debugging could not perform, using a type checker, not the compiler, as a blackbox to check if the set was truly minimal. The solutions evaluation showed improvement in locating twice as many bugs as the unmodified Liquid Haskell type checker.

Chapter 10

Future Work and Conclusions

Several investigations appeared during the production of the thesis. First, a closer look at the non-determinism of the solution is needed: is one choice better than another? Which appears as an issue when looking at multiple type errors. Could a different pathway lead to more accurate results? Next, using both the *Moiety* and *Good-Omens* algorithms showed they had strengths in different sized programs. *Moiety* mainly works best for larger programs, with those at the shorter end not benefiting in reducing time. An investigation into heuristics to decide if to call the pre-processing or 'on request' algorithm would benefit. One such solution would resort back to the pre-processing Moiety algorithm if the program's source code is under a specific size. I want to increase the categories of parse errors being treated by the algorithms, adding other errors such as '*Variables not in Scope*'. I would also want to increase the scalability benchmarks to include more than one core program. Doing so will remove any bias away from how a programmer may precisely layout out their source code. Though the agnostic version of the debugger returned promising results, another direction could be exploiting the features of different compilers by adding their specific differences into the "settings files". Allowing these differences to be used, though removing the agnostic behaviour, might improve the accuracy and precision of the framework results.

Lastly, I would like to implement an empirical study using real programmers. The study will provide insight into the debuggers' ability to locate type errors and the solutions agnostic features.

In conclusion, the primary reason for non-adoption is the effort required to implement proposed solutions for full programming languages and maintain 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. I believe that a slight improvement that requires little implementation and maintenance effort is much better than a big improvement that requires substantial effort. Hence, this thesis aims to develop a type error debugger that uses the compiler as a true blackbox; it calls the compiler as an external program, not duplicating any parsing or type checking, providing an easy to implement, scalable type error debugger.

The first solution, in chapter 3 implemented Zeller's *isolating delta debugging* algorithm for type error debugging. The algorithm isolates a fault caused by working on configurations. For the implementation, a configuration is any subset of the lines of the original ill-typed program. The algorithm computes two configurations; one is a well-typed configuration that is a subset of the other ill-typed configuration. The difference between the two configurations is a cause of the type error. The algorithm shrinks the difference between the two configurations starting with the empty, trivially well-typed configuration and the original ill-typed program's configuration. To categorise these configurations into Pass, Fail or Unresolved, to provide the pathway for the algorithm, *delta debugging* needs a test function; here, the blackbox compiler provides this necessary aspect. The combination of type error debugging, delta debugging, and a blackbox compiler allowed for the implementation of six out of the seven properties of the "Manifesto of Good Type Error Reporting", with only *Succinct* not being able to be provided due to discrete error messages not being part of the debugger.

The debugger, Gramarye, received favourable results during its evaluation which reported a 27 percentage points improvement over GHC when locating type errors. Gramarye also gave favourable results when comparing to the Counter-Factual Typing debugger whose paper introduces the benchmarks used in the evaluations throughout this thesis (Chen and Erwig 2014a,b). Gramarye returns a 10 percentage point improvement over the Counter-Factual Typing debugger. However, the Counter-Factual Typing debugger can locate more than one error compared to Gramarye, which locates only one. As such, a small investigation into multi-error locations shows that Gramarye can locate more than one error successfully.

However, besides well-typed and ill-typed, the blackbox compiler may also report a different error for a configuration, e.g. a parse error or an error for using an unknown identifier. In all these cases, delta debugging calls the configuration *unresolved*. In chapter 4 it was shown that the more unresolved configurations the algorithm encounters, the slower it becomes. I found that parse errors cause most unresolved configurations, and hence in chapter 6 developed an algorithm termed *Moiety* that creates only configurations that do not cause parse errors. The algorithm pre-processes an ill-typed program to eliminate 'parse errors on input' by avoiding splitting the source code in a place known to cause parse errors. To evaluate the new algorithm, I created a framework based on Data Science, presented in chapter 5. This new framework is designed to quantify the quality of type error debuggers that traditional use the *recall* metric, which is not satisfactory for evaluations in this field.

Although the evaluation shows that the *Moiety* algorithm speeds up locating a type error substantially, it is still too slow in practice. The Moiety algorithm sends each line of the original ill-typed program separately to the blackbox compiler and for a typical module of 400 lines that can take around 13 minutes. Thus, a new algorithm is introduced in chapter 7. The new debugger, Eclectic, that incorporates this new algorithm, *Good-Omens*, no longer pre-processes the source code. Instead, Eclectic allows the *delta debugging* algorithm to request valid splitting points only when it observes a 'parse error on input'. This change alone gave an average reduction in the run-time of 1 minute 37 seconds.

Lastly, in chapter 8, a short investigation and evaluation occurred to see if a solution for type error debugging could have agnostic behaviour. The evaluation showed that the solution and its algorithms could be applied to more than one programming language.

Bibliography

Agrawal, H., DeMillo, R. A. and Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Softw, Pract Exper*, 23(6), pp. 589–616.

Alam, A. and Bush, V. (2016). Haskeu: An editor to support visual and textual programming in tandem. In *SAI Computing Conference (SAI), 2016*, IEEE, pp. 805–814.

Albertsson, L. (2006). Holistic debugging – enabling instruction set simulation for software quality assurance. In *14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2006), 11-14 September 2006, Monterey, California, USA*, pp. 96–103.

Artho, C. (2011). Iterative delta debugging. *STTT - Software Tools for Technology Transfer*, 13(3), pp. 223–246.

Bailey, J. and Stuckey, P. J. (2005). Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings*, pp. 174–186.

Beaven, M. and Stansifer, R. (1993). Explaining type errors in polimorphic languages. *LOPLAS*, 2(1-4), pp. 17–30.

Bernstein, K. L. and Stark, E. W. (1995). Debugging type errors (full version). Tech. rep., State University of New York at Stony Brook, Stony Brook, NY 11794-4400 USA.

Binkley, D. and Gallagher, K. B. (1996). Program slicing. *Advances in Computers*, 43, pp. 1–50.

Booth, S. P. and Jones, S. B. (1997). Walk backwards to happiness - debugging by time travel. In *AADEBUG*, pp. 171–183.

Braßel, B. (2004). Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*, pp. –.

Burgers, J. (2019). *Type error diagnosis for OutsideIn (X) in Helium*. Master's thesis, Utrecht University.

Caballero, R., Riesco, A. and Silva, J. (2017). A survey of algorithmic debugging. *ACM Comput Surv*, 50(4), pp. 60:1–60:35.

Charguéraud, A. (2014). Improving type error messages in ocaml. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, pp. 80–97.

Chen, S. and Campora, J. P. (2019). Blame Tracking and Type Error Debugging. In B. S. Lerner, R. Bodík and S. Krishnamurthi, eds., *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 136, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 2:1–2:14.

Chen, S. and Erwig, M. (2014a). 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*, pp. 583–594.

Chen, S. and Erwig, M. (2014b). Guided type debugging. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pp. 35–51.

Chen, S. and Erwig, M. (2018). Systematic identification and communication of type errors. *Journal of Functional Programming*, 28.

Chen, S., Erwig, M. and Smeltzer, K. (2014). Let's hear both sides: On combining type-error reporting tools. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pp. 145–152.

Chen, S., Erwig, M. and Smeltzer, K. (2017). Exploiting diversity in type checkers for better error messages. *J Vis Lang Comput*, 39, pp. 10–21.

Chen, S., Erwig, M. et al. (2013). Better type-error messages through lazy typing. Tech. rep., Oregon State University. School of Electrical Engineering and Computer Science.

Chitil, O. (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.*, pp. 193–204.

Chitil, O. (2004). Source-based trace exploration. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, pp. 126–141.

Chitil, O. and Davie, T. (2008). Comprehending finite maps for algorithmic debugging of higher-order functional programs. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pp. 205–216.

Chitil, O., Runciman, C. and Wallace, M. (2000). Freja, hat and hood - A comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, pp. 176–193.

Choppella, V. and Haynes, C. T. (1995). Diagnosis of ill-typed programs. Tech. rep., Indiana University, USA.

Choppella, V. and Haynes, C. T. (2002). *Unification source-tracking with application to diagnosis of type inference*. Indiana University.

Christi, A. et al. (2018). Reduce before you localize: Delta-debugging and spectrum-based fault localization. *ISSRE Workshops*, pp. 184–191.

Christi, A. M. (2019). Building self adaptive software systems via test-based modifications. -.

Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pp. 342–351.

Curry, H. and Feys, R. (1958). *Combinatory Logic*. No. v. 1 in Combinatory Logic, North-Holland Publishing Company.

Damas, L. M. M. (1984). *Type Assignment in Programming Languages*. Ph.D. thesis, University of Edinburgh.

De Bleser, J., Di Nucci, D. and De Roover, C. (2020). A delta-debugging approach to assessing the resilience of actor programs through run-time test perturbations. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test (AST2020)*, p. 21–30.

de la Banda, M. J. G., Stuckey, P. J. and Wazny, J. (2003). Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pp. 32–43.

Dinesh, T. B. and Tip, F. (1997). A slicing-based approach for locating type errors. In *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997*, p. 5–55.

Duggan, D. and Bent, F. (1996). Explaining type inference. *Sci Comput Program*, 27(1), pp. 37–83.

Elyasov, A. et al. (2014). Reduce first, debug later. In *Proceedings of the 9th International Workshop on Automation of Software Test*, New York, NY, USA: ACM, AST 2014, pp. 57–63.

Gandhe, M., Venkatesh, G. and Sanyal, A. (1996). Correcting errors in the curry system. In *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, pp. 347–358.

Gast, H. (2004). Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, pp. 72–89.

Gharachorlu, G. and Sumner, N. (2018). Avoiding the familiar to speed up test case reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 426–437.

Gopinath, R. et al. (2020). Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, ISSTA 2020, p. 237–248.

Gulzar, M. A. (2018). Interactive and automated debugging for big data analytics. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 509–511.

Gupta, N. et al. (2005). Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: Association for Computing Machinery, ASE '05, p. 263–272.

Haack, C. and Wells, J. B. (2003). Type error slicing in implicitly typed higher-order languages. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pp. 284–301.

Haack, C. and Wells, J. B. (2004). Type error slicing in implicitly typed higher-order languages. *Sci Comput Program*, 50(1-3), pp. 189–224.

Hage, J. (2014a). Domain specific type error diagnosis (domsted).

Hage, J. (2014b). Domain specific type error diagnosis (the domsted project paper). Tech. rep., Utrecht University.

Hage, J. and Heeren, B. (2005). Ordering type constraints: A structured approach.

Hage, J. and Heeren, B. (2006). Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, pp. 199–216.

Hage, J. and Heeren, B. (2009). Strategies for solving constraints in type and effect systems. *Electr Notes Theor Comput Sci*, 236, pp. 163–183.

Hage, J. and Keeken, P. (2006). Mining for helium.

Hammoudi, M. et al. (2015). On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: ACM, ESEC/FSE 2015, pp. 333–344.

Hashimoto, M., Mori, A. and Izumida, T. (2018). Automated patch extraction via syntax- and semantics-aware delta debugging on source code changes. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA: ACM, ESEC/FSE 2018, pp. 598–609.

Hazan, J. E. and Morgan, R. G. (1993). The location of errors in functional programs. In *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93, Linköping, Sweden, May 3-5, 1993, Proceedings*, pp. 135–152.

Heeren, B. (2005). *Top quality type error Messages*. Ph.D. thesis, Utrecht University, Netherlands.

Heeren, B. and Hage, J. (2002). Parametric type inferencing for helium.

Heeren, B., Hage, J. and Swierstra, S. D. (2002). Generalizing hindley-milner type inference algorithms.

Heeren, B., Hage, J. and Swierstra, S. D. (2003a). Constraint based type inferencing in helium. In *In Workshop Proceedings of*, p. 57.

Heeren, B., Hage, J. and Swierstra, S. D. (2003b). Scripting the type inference process. *SIGPLAN Notices*, 38(9), pp. 3–13.

Heeren, B., Leijen, D. and van IJzendoorn, A. (2003). Helium, for learning haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pp. 62–71.

Heeren, B. et al. (2002). Improving type-error messages in functional languages. Tech. rep., Utrecht University.

Hindley, R. (1969). The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146, pp. 29–60.

Hodován, R. and Kiss, Á. (2016a). Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, pp. 31–37.

Hodován, R. and Kiss, Á. (2016b). Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pp. 241–248.

Johnson, G. F. and Walz, J. A. (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*, pp. 44–57.

Jun, Y., Michaelson, G. and Trinder, P. W. (2002). Explaining polymorphic types. *Comput J*, 45(4), pp. 436–452.

Jung, Y. and Michaelson, G. (2000). A visualisation of polymorphic type checking. *J Funct Program*, 10(1), pp. 57–75.

Kalhauge, C. G. and Palsberg, J. (2019). Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA: Association for Computing Machinery, ESEC/FSE 2019, p. 556–566.

Kamin, S. (1990). *A debugging environment for functional programming in Centaur*. Ph.D. thesis, INRIA.

Kamkar, M. (1995). An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3), pp. 197 – 214.

Keane, A. (1999). *A tool for investigating type errors in ML program*. Master's thesis, University of Edinburgh.

Kiss, A. (2020). Generalizing the split factor of the minimizing delta debugging algorithm. *IEEE Access*, 8, pp. 219837–219846.

Kiss, A., Hodován, R. and Gyimóthy, T. (2018). Hddr: A recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, New York, NY, USA: ACM, A-TEST 2018, pp. 16–22.

Kustanto, C. and Kameyama, Y. (2010). Improving error messages in type system. *Information and Media Technologies*, 5(4), pp. 1241–1254.

Lapalme, G. and Latendresse, M. (1992). A debugging environment for lazy functional languages. *Lisp and Symbolic Computation*, 5(3), pp. 271–287.

Lauesen, S. (1979). Debugging techniques. *Softw, Pract Exper*, 9(1), pp. 51–63.

Lee, O. and Yi, K. (1998). Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans Program Lang Syst*, 20(4), pp. 707–723.

Lee, O. and Yi, K. (2000). A generalized let-polymorphic type inference algorithm. In *Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology*, pp. –.

Lerner, B. S., Grossman, D. and Chambers, C. (2006). Seminal: searching for ML type-error messages. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pp. 63–73.

Lerner, B. S. et al. (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*, pp. 425–434.

Loncaric, C. et al. (2016). A practical framework for type inference error explanation. In *OOPSLA*, p. 781–799.

Magoun, A. B. and Israel, P. (2013). Did you know? edison coined the term "bug" – ieee history.

Marlow, S. et al. (2007). A lightweight interactive debugger for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pp. 13–24.

McAdam, B. J. (1999a). Generalising techniques for type debugging. In *Selected papers from the 1st Scottish Functional Programming Workshop (SFP99), University of Stirling, Bridge of Allan, Scotland, August 29th to September 1st, 1999*, pp. 50–58.

McAdam, B. J. (1999b). On the unification of substitutions in type inference. *Lecture notes in computer science*, 1595, pp. 137–152.

McAdam, B. J. (2001). How to repair type errors automatically. In *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), University of Stirling, Bridge of Allan, Scotland, August 22nd to 24th, 2001*, pp. 87–98.

McAdam, B. J. (2002). *Repairing type errors in functional programs*. Ph.D. thesis, University of Edinburgh, UK.

McCauley, R. et al. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), pp. 67–92.

McKenzie, B. and Wyber, B. J. (1999). Type debugging in functional languages.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3), pp. 348–375.

Misherghi, G. and Su, Z. (2006). Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, New York, NY, USA: ACM, ICSE '06, pp. 142–151.

Misherghi, G. S. and Su, Z. (2007). *Hierarchical delta debugging*. Ph.D. thesis, University of California, Davis.

Naish, L. (2000). A three-valued declarative debugging scheme. In *23rd Australasian Computer Science Conference (ACSC 2000), 31 January - 3 February 2000, Canberra, Australia*, pp. 166–173.

Naish, L. and Barbour, T. (1996). Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18, pp. 401–408.

Neubauer, M. and Thiemann, P. (2004). Haskell type browser. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pp. 92–93.

Pavlinovic, Z. (2014). General type error diagnostics using maxsmt.

Pavlinovic, Z., King, T. and Wies, T. (2014). Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pp. 525–542.

Pavlinovic, Z., King, T. and Wies, T. (2015). Practical smt-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pp. 412–423.

Pierce, B. C. (2005). Advanced Topics in Types and Programming Languages. *The Computer Journal*, 49(1), pp. 130–131, `https://academic.oup.com/comjnl/article-pdf/49/1/130/1161818/bxh138.pdf`.

Plociniczak, H. (2013). Scalad: An interactive type-level debugger. In *Proceedings of the 4th Workshop on Scala*, New York, NY, USA: ACM, SCALA '13, pp. 8:1–8:4.

Plociniczak, H. and Odersky, M. (2012). Implementing a type debugger for scala. In *Asia-Pacific Programming Languages and Compilers Workshop*, EPFL-CONF-179877, pp. –.

Rahli, V., Wells, J. and Kamareddine, F. (2009). Challenges of a type error slicer for the sml language. Tech. rep., Technical Report HW-MACSTR-0071, Heriot-Watt University, School of Mathematics & Computer Science.

Rahli, V., Wells, J. and Kamareddine, F. (2010). A constraint system for a sml type error slicer. Tech. rep., Heriot-Watt University, MACS, ULTRA group.

Rahli, V. et al. (2015). Skalpel: A type error slicer for standard ML. *Electr Notes Theor Comput Sci*, 312, pp. 197–213.

Rahli, V. et al. (2016). Skalpel: A constraint-based type error slicer for standard ML. *J Symb Comput*, 80, pp. 164–208.

Rittri, M. (1993). Finding the source of type errors interactively.

Sagonas, K. F., Silva, J. and Tamarit, S. (2013). Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, pp. 33–42.

Schilling, T. (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*, pp. 1–16.

Seidel, E. L. (2017). *Data-Driven Techniques for Type Error Diagnosis*. University of California, San Diego.

Seidel, E. L., Jhala, R. and Weimer, W. (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*, pp. 228–242.

Seidel, E. L. et al. (2017). Learning to blame: Localizing novice type errors with data-driven diagnosis. *CoRR*, abs/1708.07583, `1708.07583`.

Serrano, A. and Hage, J. (2016). Type error diagnosis for embedded dsls by two-stage specialized type rules. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 672–698.

Serrano Mena, A. and Hage, J. (2016). Context-dependent type error diagnosis for functional languages.

Shapiro, E. Y. (1982). Algorithmic program diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA: ACM, POPL '82, pp. 299–308.

Sharrad, J. (2021a). 11 programs converted for agnostic evaluation. Https://github.com/JoannaSharrad/TypeErrorDebuggingScalabilityDataSet.

Sharrad, J. (2021b). Pandoc for evaluation of type error debuggers. Https://github.com/JoannaSharrad/TypeErrorDebuggingScalabilityDataSet.

Shung, K. P. (2019). Accuracy, precision, recall or f1? Https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9.

Silva, J. (2006). A classification of algorithmic debugging. Tech. rep., UNIVERSIDAD POLITÉCNICA DE VALENCIA.

Silva, J. (2011). A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11), pp. 976–991.

Silva, J. and Chitil, O. (2006). Combining algorithmic debugging and program slicing. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pp. 157–166.

Simon, A., Chitil, O. and Huch, F. (2000). Typeview: A tool for understanding type errors. In M. Mohnen and P. Koopman, eds., *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, Aachen, Germany, pp. 63–69.

Smithsonian-Institution (2019). Log book with computer bug.

Sparud, J. (1995). Towards a haskell debugger. In *Functional Programming Languages and Computer Architecture*, pp. –.

Spectrum, I. (2021). Did you know? edison coined the term "bug".

Stepanov, D., Akhin, M. and Belyaev, M. (2019). Reduktor: How we stopped worrying about bugs in kotlin compiler. *arXiv preprint arXiv:190907331*.

Stuckey, P. J., Sulzmann, M. and Wazny, J. (2003a). The chameleon type debugger (tool demonstration).

Stuckey, P. J., Sulzmann, M. and Wazny, J. (2003b). Interactive type debugging in haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pp. 72–83.

Stuckey, P. J., Sulzmann, M. and Wazny, J. (2003c). Type debugging in the hindley/milner system with overloading.

Stuckey, P. J., Sulzmann, M. and Wazny, J. (2004). Improving type error diagnosis. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pp. 80–91.

Stuckey, P. J., Sulzmann, M. and Wazny, J. (2006). Type processing by constraint reasoning. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, pp. 1–25.

Tirronen, V., Uusi-Makela, S. and Isomottonen, V. (2015). Understanding beginners' mistakes with haskell. *Journal of Functional Programming*, 25.

Tondwalkar, A. (2016). *Finding and Fixing Bugs in Liquid Haskell*. Master's thesis, University of Virginia.

Tondwalkar, A. et al. (2016). Finding bugs in liquid haskell, -.

Tsushima, K. and Asai, K. (2011). Report on an ocaml type debugger. In *ACM SIGPLAN Workshop on ML*, vol. 3, pp. –.

Tsushima, K. and Asai, K. (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*, pp. 190–206.

Tsushima, K. and Asai, K. (2013). A weighted type error slicer. *Implementation and Application of Functional Languages, Lecture Notes in Computer Science*, 8241, pp. 190–206.

Tsushima, K. and Chitil, O. (2014). Enumerating counter-factual type error messages with an existing type checker. In *16th Workshop on Programming and Programming Languages, PPL2014*, pp. –.

Tsushima, K. and Chitil, O. (2018). A common framework using expected types for several type debugging approaches. In *FLOPS 2018: Fourteenth International Symposium on Functional and Logic Programming*, Springer, pp. 230–246.

Tsushima, K., Chitil, O. and Sharrad, J. (2019). Type debugging with counter-factual type error messages using an existing type checker. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, New York, NY, USA: Association for Computing Machinery, IFL '19.

Wand, M. (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*, pp. 38–43.

Wazny, J. (2006). *Type inference and type error diagnosis for Hindley/Milner with extensions.* Ph.D. thesis, Computer Science and Software Engineering The University of Melbourne Parkville, Melbourne.

Weiser, M. (1982). Programmers use slices when debugging. *Commun ACM*, 25(7), pp. 446–452.

Whittle, J., Bundy, A. and Lowe, H. (1997). An editor for helping novices to learn standard ML. In *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, pp. 389–405.

Witten, I. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

Wu, B. and Chen, S. (2017). How type errors were fixed and what students did? *Proc ACM Program Lang*, 1(OOPSLA).

Yang, J. (1999). Explaining type errors by finding the source of a type conflict. In *Selected papers from the 1st Scottish Functional Programming Workshop (SFP99), University of Stirling, Bridge of Allan, Scotland, August 29th to September 1st, 1999*, pp. 59–67.

Yang, J. et al. (2000). Improving polymorphic type error reporting.

Yu, K. et al. (2012a). Practical isolation of failure-inducing changes for debugging regression faults. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: ACM, ASE 2012, pp. 20–29.

Yu, K. et al. (2012b). Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software*, 85(10), pp. 2305–2317.

Zeller, A. (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*, pp. 253–267.

Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pp. 1–10.

Zeller, A. (2009). *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press.

Zeller, A. (2021). The debugging book tools and techniques for automated software debugging. Https://www.debuggingbook.org/.

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans Software Eng*, 28(2), pp. 183–200.

Zhang, D. et al. (2015a). Diagnosing haskell type errors. In -, pp. –.

Zhang, D. et al. (2015b). Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 12–21.

Zhang, D. et al. (2017). Sherrloc: A static holistic error locator. *ACM Trans Program Lang Syst*, 39(4), pp. 18:1–18:47.

Zhang, S. et al. (2008). Effective identification of failure-inducing changes: A hybrid approach. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, New York, NY, USA: ACM, PASTE '08, pp. 77–83.

Zhang, X., Gupta, N. and Gupta, R. (2006). Pruning dynamic slices with confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, PLDI '06, pp. 169–180.

Zheng, Y. et al. (2020). Probing model signal-awareness via prediction-preserving input minimization. `2011.14934`.

Zhou, X. et al. (2018). Delta debugging microservice systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, pp. 802–807.