Towards Graphically Representing Skalpel Type Error Slices

Final Year Dissertation

Christian Gregg *

Supervised by: J. B. Wells

Submitted in partial fulfilment of the requirements for the
Honours Degree of Bachelor of Science (Computer Science)
at Heriot-Watt University

April 22, 2018

I, Christian Gregg, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.[1]

Signed:

Date: 22-APR-2018

[1]This statement is based on a sample declaration provided in Appendix C.1 of [13].

Abstract

Traditional *single-node* type-error reporting systems for Standard ML — such as those implemented in the Standard ML of New Jersey, MLton, or PolyML compilers — often generate confusing output and are liable to place the blame for a type error on a program point far away from the *real error location*. *Multi-node* type error reporting systems, such as Skalpel, do not attempt to report an exact error location. Instead, they report the location of a type error as a set of program points that must be present for the type error to occur. Whilst multi-node systems improve the description of type errors considerably, the output they generate can often become cumbersome and unwieldy for large, complex programs. This project begins to implement a new system for creating graphical representations of type-error slices generated by Skalpel. It enhances the information which Skalpel outputs and creates a new visualisation front-end that ingests this output and displays it. The purpose of these graphical representations is to make Skalpel's output more usable and intuitive; this is achieved by allowing the programmer to *see* how different identifiers in their program are interacting and conflicting.

Acknowledgements

Joe Wells. I'd like to thank Joe for taking me on as a project student. For the hours he spent in meetings with me; for sharing his knowledge; for giving valuable advice and feedback; for helping to make this project a success right through to the very last week; and for his patience as I learned the very basics of a field he is an expert in.

Verena Rieser. I'd like to thank Verena for accepting to be the second reader for this dissertation. It is undoubtedly not an easy nor a quick task to read through and evaluate this work. I would like to show my appreciation for taking the time to do so.

Lawrie Porter. I want to thank Lawrie for making sure I got up in the morning and made my way into university to work; for making sure I didn't stay too long; and for the many hours we've spent together being productive and not-so-productive.

"The Wizard Skwad". I'd like to thank all of you for the never-ending and very welcome distractions that you offer from University — or is it University that offers distraction from you? For the good vibes and *all* the shenanigans we've had the chance to get up to.

Contents

1 Introduction

Functional programming languages with type inference algorithms such as Standard ML (SML), Haskell, OCaml, and F# often do not require the programmer to provide type annotations for bindings of names to expressions (such as function name declarations, variable bindings, and parameter bindings). Instead, type inference algorithms guarantee strong static typing by analysing how expressions are used within a program and infer the type of these expressions based on this. As described by Heeren [15]: "[type checking] analysis guarantees that functions (or methods) are never applied to incompatible arguments".

Any expression within a program for which the algorithm is not able to assign a sensible type is deemed as being malformed and is rejected before evaluation can occur. It is in this case that a type error is generated and that a type error report must be presented to the user.

Type error reports in existing implementations of SML will very often return confusing error messages. The functional core of SML uses Milner's W algorithm [10] to perform type checking. This algorithm is a *single-node* algorithm — it is only able to blame a single node of the abstract syntax tree (AST) for any type error.

Rahli et al. [36] point out single-node type error reporting systems often produce confusing error messages that blame a program point that can be far away from the actual location of the program error (the *real error location*). They make the following comment on attempts to resolve these issues so far, saying that they "(1) fail to include the multiple program points which make up the type error; (2) report tree fragments which do not correspond to any place in the user program; and (3) give incorrect type information/diagnosis which can be highly confusing" [36]. This leads to increased debugging times and increases the learning curve required to become proficient in the language for beginners [32].

Other algorithms have been developed (W′ [26], UAE [45], M [23], and others) in attempts to try and improve this situation. However these systems are all still single-node, and still suffer from the same shortcomings. §2.6.1 discusses single-node approaches in more depth.

On a conceptual level, reporting only one program point as the cause of an error is not ideal. Type errors occur due to the way in which multiple program points with non-compatible types have been erroneously forced to interact with each other by the programmer. Blaming only one of these will never be able to fully describe how or why the error has occurred.

A promising way to improve this situation is to develop *multi-node* type error reporting

systems. These are systems that describe type errors as a set of program points that contribute to the generation of an error. Being able to identify multiple program points should allow for more complete descriptions of type errors and lead to an improvement in type error reports. Multi-node approaches are discussed in §2.6.2.

Skalpel — designed and developed at Heriot-Watt University by Christian Haack, Vincent Rahli, John Pirie, Joe Wells, the ULTRA group, and a number of project students — is a multi-node type error reporting system that produces complete and minimal sets of program nodes contributing to a type error. These sets are known as *slices*. Skalpel is integral to this project and is discussed in detail within §2.7.

One criticism of multi-node error report system that identify *complete* type error slices is that the size of these slices — even when minimal — can often become unwieldy for more complex type errors [39, 47, 48].

The aim of this project is to improve the manner in which type error reports generated by Skalpel are presented to the end-user; in order to make them more easily understandable, thereby increasing understanding and reducing debugging times. This is achieved by modifying the output generated by Skalpel and by creating a new program capable of ingesting this new output format and displaying it graphically. The changes made to Skalpel (referenced in this document as the "back-end") are described and discussed in §3.1.

The new interface implements all of the features currently available from the Skalpel command-line interface (text highlighting, error message). It also attempts to augment it by drawing lines between declarations and uses of identifiers that are relevant to the error (i.e. that are present within the type-error slice). Chitil, Huch, and Simon [8] note that more difficult type errors involve variables that are used in several places. In order for a programmer to deduce why a certain variable has a certain type (or multiple conflicting types) they must examine the surrounding expressions for every occurrence of that variable identifier. By explicitly drawing lines between these identifiers, the hope is to help the programmer in finding the cause of a type error in what Chitil, Huch, and Simon [8] label as being the "natural" way in which this is done. Reducing the amount of time the programmer has to search for relevant uses of an identifier should reduce the overall amount of time required to correctly identify and correct a type error. The reader is referred to §2.8 for a summary of some of the literature on visualisation — specifically of programming languages, and of their associated types and type systems. See Section §3.2 for discussion related to the visualisation implemented as part of this project.

This project requires a substantial amount of background knowledge of type theory, Standard ML (and its type system), and currently available type error reporting systems (specifically Skalpel). Section 2 introduces all of these core components. Section 3 details the work carried out over the course of this project. It enumerates and discusses the requirements of this project and the degree to which they were or were not achieved. It discusses the decisions made and the rationales behind them; as well as describing the actual implementations. Section 4 describes the methods used to evaluate the project and presents the results of these evaluations. Section 5 concludes this document, summarising the achievements and limitations of this work; while also suggesting and briefly discussing potential future work.

2 Background and Literature Review

This section provides an overview of the current literature on type error reporting and the visualisation of type errors. The review focuses heavily on methods and research relevant specifically to Skalpel. This is because this project will be extending Skalpel itself.

2.1 Types & Type Inference

Bird and Wadler [6] introduce the notion of types by saying that the set of all values can be partitioned into organised collections called types. Each of these collections have operations that are not meaningful — that cannot be applied — to other types. For example, multiplication is an operation that is meaningful for numeric types such as the integers or reals. However, it does not make sense multiply two functions together; as this operation is not meaningful for function types.

Type inference is the process by which types are discovered and assigned for all expressions and identifiers within a program.

For example, given the multiplication operator it is possible to infer that the expressions on either side of the operator must be of a numeric type. This is because the multiplication operator is defined as, or typed as, a function that takes as input two numeric values. If this multiplication was part of a larger expression, then it could be inferred that this particular part of the expression will be typed as being numeric, as the multiplication operator is typed as returning a numeric value. This process continues on until every expression in a program has been assigned a type.

2.2 Functional Languages, Standard ML, Polymorphism

Standard ML (SML) is derived from the Meta-Language (ML) that was originally designed to be used in the Logic for Computable Functions verification [12] system developed at the University of Edinburgh and Stanford University.

SML allows for programming in the functional paradigm. What this means is that SML allows functions to be first-class.

SML supports higher-order function (functions that take other functions as arguments), let-style polymorphism, pattern matching, and type inference, amongst other advanced language

features[2].

Let-style parametric polymorphism is one of the most powerful features that SML implements. A `let` expression is of the form `let <declarations> in <expression> end`, where `<declarations>` are a number of `val` bindings, and `<expression>` is an SML expression. The declarations made in the `let` expression can be referenced within the `<expression>`. Every reference to a binding with polymorphic type has its type variable(s) instantiated independently of any previous reference during the type inference process. This is what allows for the `id` function in Figure 1 to be applied to parameters of both `int` type and `bool` type. (It also this mechanism that causes the exponential growth of constraints referred to when discussing the multi-node approaches of Haack and Wells [14] and Rahli et al. [36] in §2.6.2.) An example SML program using the `let` construct is shown in Figure 1.

```
let
    val id = (fn x => x)
in
    (id 1, id true)
end
```

Figure 1: Example of let-style polymorphism

The program shown in Figure 1 will compute the tuple `(1,true)` of type `int * bool`. The polymorphic type of the `id` function (`` `a -> `a ``)[3,4] allowed the first application of the function to be instantiated to the type `int -> int`, and the second to the type `bool -> bool`. This is only possible using the `let` construct. Knowing that SML allows higher-order functions the reader may be inclined to think that the program shown in Figure 2 should also be valid.

```
(fn f => (f 1, f true)) (fn id => id)
```

Figure 2: Invalid higher-order function

[2]See Stansifer [40] for a good introduction to the key features of ML in general with examples in SML. For a more in-depth view of ML see Paulson [31].

[3]What this really means is $\forall\alpha.\alpha \longrightarrow \alpha$, where α is some type that will be instantiated on use.

[4]SML uses primed letters (`` `a, `b, `c, ... ``) to denote polymorphic type variables. That is to say that `` `a `` is free to be instantiated to any type. SML also uses double-primed letters (e.g. ``` ``a ```) to denote a polymorphic equality type — a type for which the equality operator, =, is defined.

Unfortunately the program in Figure 2 results in a type constructor clash between `int` and `bool`. This occurs because the domain of `f` becomes constrained to be of type `int` after its first use[5]. This means that when it comes to be applied to `true` (which is of type `bool`) a type constructor clash occurs, constituting a type error.

2.3 Type Errors

Type errors can occur for a multitude of reasons during the process of assigning types to expressions. Some of the more common errors are known as type constructor clashes. One way in which these errors can occur is when during the process of type inference an expression can be deduced to be of at least two incompatible types.

```
let
    val app = (fn f => fn x => f x)
in
    app () (fn x => x)
end
```

Figure 3: Untypable SML code

Taking the expression `app () (fn x => x)` from Figure 3 as an example — which is applying `app` to `()` and then applying the result of that to `(fn x => x)`. The types of these three sub-expressions are as follows: `()` is of type `unit`[6], `fn x => x` (the identity function) is of type $\forall\alpha.\alpha \longrightarrow \alpha$, that is to say it is a function where the input and output are of any type as long as those types are equal, and `app` which is of type $\forall\beta.\forall\gamma.(\beta \longrightarrow \gamma) \longrightarrow \beta \longrightarrow \gamma$ which can be seen as a function that takes two arguments[7] — one of which is a function of type $\beta \longrightarrow \gamma$ (where β and γ are chosen to be the same as the other occurrences of β and γ) and the other is of type β — and returns something of type γ. (By looking at the code we can see

[5]Which type the domain of `f` is first constrained to depends on the order in which the type inference algorithm evaluates elements of a tuple. Some implementations evaluate tuples left-to-right (which would result in it becoming constrained to type `int` first), others right-to-left (constraining to `bool` first). Some type-inference algorithms (those that are 'unbiased') evaluate elements in an order-independent manner, meaning that there is no constraint that is 'first'; nevertheless a type constructor clash will still occur.

[6]The `unit` type is the simplest basic data type in ML, it consists of exactly one element: `()`, referred to as the *unit element* or *unity*. It is also syntactic sugar for `{}`, the empty record type [31, 40].

[7] This is not entirely accurate. Functions in SML can only ever be applied to a single argument. What actually occurs is that the applying a function to multiple arguments will apply the function to the first argument, then apply the result of that application to the second argument, and so on. This is known as currying. This can give the illusion of functions that can 'take' multiple arguments.

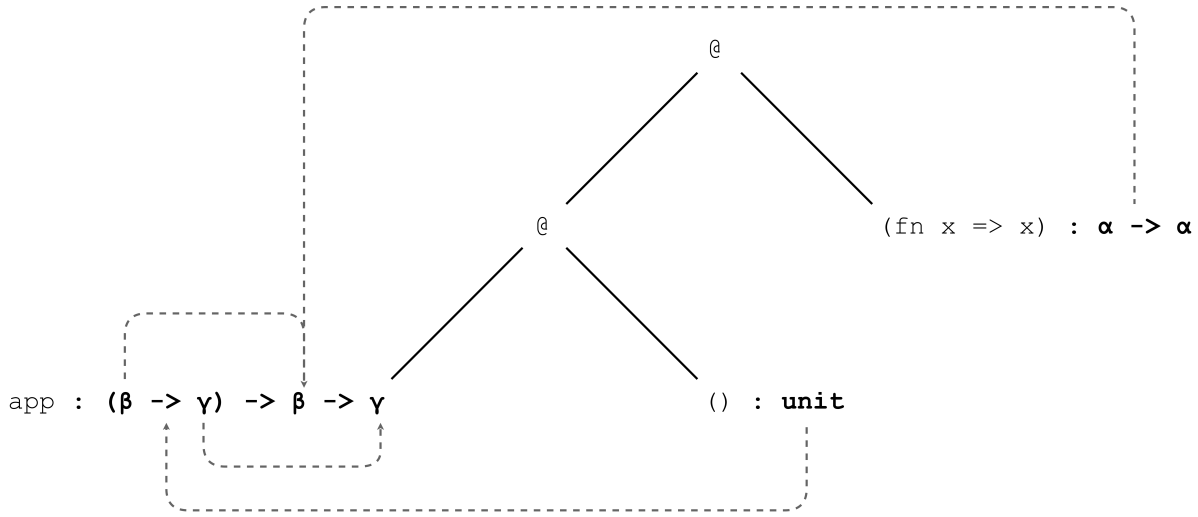

Figure 4: A simple constraint graph for `app () (fn x => x)`[8]

that what `app` does is take a function and an argument and return the result of applying that function to the given argument.)

What can be derived from these types is that the first argument to `app` must be of function type. The second argument must be of the same type as the first argument to the function that is the first argument of `app`. The reason for the piece of code in Figure 3 not being typable is the clash between the `unit` type and the $\beta \longrightarrow \gamma$ type. The definition of `app` restricts its first argument to be of a function type, which `()` is not. The restriction that requires two expressions to be of compatible types is known as a type constraint. We can say that the first argument to `app` is constrained to be of type $\beta \longrightarrow \gamma$. This constraint cannot be satisfied, or solved. When this occurs the type inference algorithm will fail, and return a type error.

2.4 Type Constraints

The literature relating to type inference algorithms often differentiates some approaches as being *constraint-based*. This differentiation is somewhat misleading. In truth all type inference algorithms must use type constraints to some degree in order to decide whether a program can be typed. The distinction between a system being constraint-based, or not, is related to how eagerly after generation constraints are solved. Systems that solve constraints immediately are not regarded as being constraint-based due to the fact that constraints are generated and solved within the same step. As such the constraints never really 'exist'. Other systems split the task of generating constraints and solving them. This allows for the solving of more complex

[8]The `@` symbol in this diagram represents an application, that is to say the expression to the left is applied to the expression on the right.

constraints, enabling the use of more complex type systems. Only simple constraints (such as equality constraints) can be solved immediately after generation. This restricts methods that attempt to eagerly solve constraints to more simplistic type systems. For example, constraints generated by type systems with subtyping will generally not be able to be solved eagerly and require constraint solving at a later stage.

A type annotation in the program adds a restriction on the types that an expression or identifier can take on, for example an expression or identifier can be constrained to being of type `int`, `bool`, `real`, and so on. Identifiers may also be constrained to take on polymorphic types such as `` `a -> `a ``, meaning that it must be a function where the input and output types can be of any monomorphic type upon instantiation, as long as they are same.

Often the constraints generated by a type inference algorithm will be modelled as a graph, where each node represents an expression, and constraints are drawn as edges between nodes. For these graphs to be solvable — and the program to be typed — the nodes at each end of every edge must be compatible types (either equal or in a sub-type relation). It is the task of the type inference algorithm to go through this graph and attempt to assign types to each node so that this condition is satisfied.

Figure 4 shows what part of a constraint graph for the program described in Figure 3 might look like.

2.5 Type Error Reporting

Type error reporting is the process of informing an end user that an error occurred during the type inference process. Type inference algorithms are generally designed with efficiency of analysis in mind, and have efficient error reporting only as a secondary goal. This has led to the development of multiple type inference algorithms with reporting mechanisms that leave a lot to be desired (See §2.6.1 for discussion on criticisms of reported error location in traditional type inference algorithms). The following section summarises some of the work that has been done in investigating type error reporting systems, different approaches that have been taken, and ways that they may be improved.

2.5.1 Yang et al. [46]

Yang et al. [46] define a manifesto comprising of seven properties that constitute a good type error report (See Figure 5 on page 22 for these properties). The purpose of this manifesto is to

improve the quality of type error reports generated by reporting systems.

This manifesto is a good starting point for beginning a discussion of what constitutes a good type error report — and offers points of consideration for anyone designing/improving reporting systems. Unfortunately, there has not been much critical discussion regarding this manifesto and the validity of the criteria which it sets out. It may be beneficial to renew the manifesto through an analysis of work on type error reporting in recent years, evaluating what the common themes across these works are and consolidating them. Given the time constraints and scope of this project, this is left for future work.

Their work goes on to evaluate existing type inferences algorithms (such as algorithms M and W), error explanation systems, and error reporting systems against their manifesto.

The authors also introduce two new type inference algorithms (UAE and IEI), and also evaluate them against their manifesto. (The claim of the UAE algorithm being unbiased has since been found to be untrue. The UAE algorithm retains a left-to-right bias when handling let-bindings [36].)

2.5.2 Heeren [15]

Heeren [15] differentiates between multiple different approaches that have been taken in efforts to improve the quality of type error reports. These approaches are: *Order of Unification*, *Explanation Systems*, *Reparation Systems*, *Program Slicing*, and *Interactive Systems*. A summary of these approaches (as presented by Heeren [15]) is given in the sections to follow.

Order of Unification

By changing the order in which types are unified the detection of conflicting types can be delayed or sped up. Type errors are typically reported to occur at the location at which unification fails while traversing the AST of a program. By changing the order in which this traversal occurs, the reported error/location can also be changed. The majority of traditional type inference algorithms (including M, and W) traverse the AST of a program in a predetermined order (i.e. always traversing the left hand sub-expression before the right one, or vice versa). This leads to a well documented bias where algorithms tend to report errors towards the start, or the end of a program depending on their preferred traversal method. This can lead to reported errors being far away from actual errors. This problem is discussed in §2.6.

Heeren also notes that this issue is particularly interesting as compilers currently in use are

'hybrid'; some types are pushed downwards in the AST, while others flow upwards. This can make the type checking process much less transparent to the programmer, as it becomes unclear which parts of the program have already been considered by the compiler, and which have not.

Heeren suggests that the most promising way to improve this situation is to develop and use algorithms with symmetrical treatment of subexpressions in the AST, although one drawback of this approach is that the reported error sites are further up in the AST, and therefore less precise.

Explanation Systems

These systems approach the problem of error reporting by trying to explain *why* specific types have been inferred. This is often done by collecting why certain inferences were made during the type inference process. Upon failure this collection can be used to create an explanation for failure.

One major challenge with explanation systems is choosing the level of detail for the explanation. The ideal level of detail depends on the level of expertise of the person reading the report, as well as the complexity of the problem. Too much detail is likely to overwhelm the user and lead to the explanation being disregarded. Too little detail may leave out crucial information.

Further to this, explanations are often heavily guided by the underlying type checking/inference algorithm. This means that knowledge of the algorithm will be needed to fully understand the reported explanation. This becomes particularly obvious when dealing with polymorphic functions, and the type variables that they introduce. If these variables show up in a type error explanation, the programmer will need to know how these variables are introduced by the type checking algorithm to make use of that information in debugging.

Lastly, type explanation systems naturally do not scale well to larger and more complex programs. As the size and complexity increases in a program, so will the explanations required to adequately describe errors occurring in the program.

Reparation Systems

Reparation systems try to report a single location that is the most likely cause of a type error. Many heuristics, of varying degrees of complexity, can be used to make this selection from a list of possible locations — these systems are synonymous to what are describe as *single-node* systems in this document, the reader is referred to §2.6 (§2.6.1 in particular) for more detailed

discussion. Some more advanced reparation systems may also suggest potential modifications to the program that would fix the reported error.

Program Slicing

Slicing systems report a set of program points (a *slice*) that contribute to a failure in type inference. The parts of the program not in the slice can be disregarded by the user. Standard textual output cannot be used to effectively report these slices, and most systems use underlining, highlighting, or a combination of both for reporting. Program slicing is referred to as a *multi-node* system in this document and is further discussed in §2.6.2.

Interactive Systems

Interactive systems allow the programmer to work through their program in an interactive manner. This is done in order to help them understand the different assumptions they and the type checker have made, and to help them uncover where they disagree. For example, many languages allow for the programmer to define type declaration for expressions. An interactive system may ask the programmer to explicitly type some parts of their program, and through this process it may become apparent where the assumptions/intentions of the programmer are in conflict with the type checking algorithm, leading to a fix.

A problem with this approach is that for long complex programs and errors these interactive sessions can become too long and tedious to be effective (for example asking the programmer to manually type a complicated function).

Other approaches have been taken to try and improve understanding type error reports, type, and type checking through visualisations. These are discussed in §2.8

2.6 Type Error Location

Finding the *real error location* — the location of the actual programming error, which if fixed will cause the program be typable and to carry out the intended computation — of a type error is an inherently challenging problem as it requires the type error reporting algorithm to make assumptions regarding the programmer's intentions. There is a substantial body of work that has been undertaken to try and overcome this challenge, with a variety of different approaches being taken. The aim of this section is to give an overview of these different approaches, before focusing on the approaches that are relevant to Skalpel and in turn this project.

When attempting to find the location at which a type error has occurred, a reporting system can either return only one node of the parsed abstract syntax tree (AST) or a set of nodes. This is one of the key distinctions in many of the approaches that have been undertaken so far. These will be referred to as *single-node* and *multi-node*. Type error reporting systems in implicitly typed languages have been widely criticised [14, 16, 32, 36, 37, 42]. Much of the criticism is directed at the location that is reported as the source of an error. Single-node systems are notoriously known for being prone to blame sections of programs that are far away from the real error location. Multi-node approaches are better able to identify the location of type errors and as such can avoid causing the frustration that single-node systems often induce. This being said, multi-node systems bring with them their own set of challenges (discussed in §2.6.2).

2.6.1 Single-Node Approaches

The original type checking algorithm for the functional core of SML implements Milner's W algorithm [10], which reports only a single program node to describe type errors. The W algorithm works by traversing the AST of a program and upon failure to solve a type equality constraint will blame the node currently being visited. This node is often far away from the real error location, causing the generation of an unsatisfactory error message that often leads the programmer to look in the wrong place for programming errors.

A number of alternative algorithms, all based on Milner's W algorithm, have been developed in attempts to improve the reported error location, including the W' [26], M [23], and UAE [45] algorithms.

All three of these algorithms claim to improve upon W, either by reporting a location that is closer to the real error location or by returning a location earlier on in the computation. While this may be true, all three of the algorithms still suffer from the flaws inherent to single-node systems.

Single-node based type checking algorithm inherently make it more difficult for programmers to understand type errors by only reporting a single program node as the source of an error. A single program node cannot fully describe any type error. This is because type errors occur due to the way in which multiple program nodes are interacting. Reporting only a single node forces the programmer to mentally search for all program points that are interacting with the reported program point and to manually decide which ones could be responsible for the type error. This search is made more painful as the reported error location is often far from the real

error location. This is an issue that has been recognised for over three decades [16, 42].

While these shortcoming are generally recognised and accepted by experienced programmers, beginners are much more affected by these as they are prone to focus their attention on the error location reported and not the error message itself [15, 17]. To compound the confusion, implementations of these algorithms often produce error message containing the internal representation of the AST which will often be significantly mutated from the source code. They also often reference internal representations of partially constructed types — adding another layer of complexity for the programmer to identify the expression being blamed (which, as mentioned earlier, may not even be at fault).

Improving this situation, and type error messages in general, would improve the experience for many programmers and would aid in increasing the adoption of more advanced type systems [24].

2.6.2 Multi-Node Approaches

Multi-node approaches attempt to improve on the problem of error location by providing the programmer with a set of program points that contribute to a particular error — this set is commonly referred to as a *slice*.

The notion of *slicing* in programming was first described by Weiser [43]. Weiser describes program slicing as a method used by experienced programmers to reduce a subset of a program's behaviour to the minimal program that retains this behaviour. This notion has since been utilised to reduce ill-typed programs to contain only those program points pertinent to the error. Pinpointing the exact real error location for a type error is difficult as it requires the type checking and reporting system to make assumptions about what the programmer intended a program to do [39].

This section gives an overview of work that in one way or another tries to tackle the problem of locating and describing type error through a multi-node approach.

Dinesh and Tip [41]

Dinesh and Tip [41] build on the notion of program slices to begin work on improving the error location reported by type-checking algorithms. They do this by representing a type error as a subset of a program that contains exactly those program constructs within the program that cause the error. The methods they introduce assume that the language's type checking system

can be represented as a series of rewrite rules and are only applicable to languages which are more explicitly typed than functional languages like SML. Dinesh and Tip attempted to apply this method to Mini-ML, a simple typed λ-calculus. However, due to the lack of a minimisation algorithm their attempt tended to compute slices that were larger than necessary.

Haack and Wells [14]

Hack and Wells [14] develop a slicing technique applicable to languages with type inference. Their technique represents type errors through slices that are both minimal and complete.

While it might be possible to represent the type system of implicitly-typed languages as a series of rewrite rules, Haack and Wells do not believe that a direct application of the methods described by Dinesh and Tip would be able to identify accurate locations for type errors in *any* language with significant reliance on type inference.

Instead Haack and Wells base their work on the type inference system developed by Damas [9], which they name "Damas' System T" (as it uses Damas' algorithm T). Their system is based on the analysis of an unsolvable constraint set (§2.4 gives a brief introduction to type constraints). It gives minimal type error slices by removing all constraints within a slice that are not responsible for a type error, such that the removal of any additional constraint would render the constraint set solvable.

By removing all constraints that do not contribute to a type error what is left is the smallest set where every constraint is necessary for a type error to occur. These constraints are related back to program points so that this minimal set of constraints can then be presented to the programmer in a more intuitive manner.

The method proposed by Haack and Wells is computationally expensive. They themselves acknowledge that solving complete minimal sets of constraints "will be difficult because the worst-case time complexity for enumerating all such sets is intractable". There is a combinatorial explosion that occurs while generating the initial constraint set that is due to the constraint typing rules that govern handling of let-bound polymorphic variables that result in the copying of constraints. In order to overcome the first combinatorial explosion their method will enumerate only some minimal unsolvable subsets of constraints before being terminated by a time limit.

Rahli et al. [36]

Rahli et al. [36] build on the work of Haack and Wells, most notably by solving one of the problems of exponential growth in the algorithm. They do this by using variants of polymorphic let-constraints and type scheme instantiations introduced by Pottier and Rémy [35]. This enables building constraints with linear space, improving the efficiency of Haack and Wells' method significantly and making it feasible for use. Their method makes the duplication of constraints a core design concern. One of their design principles addresses this issue directly by stating that duplication of constraints should be unnecessary. A key contributor to achieving this design principle, and reducing the size of constraint sets, is the development of a new representation of constraints as "hybrid constraint/environment forms". Leveraging this new representation means that less duplication is required. Their method also eagerly simplifies any constraints and calculates types related to these constraints as much as possible before making any duplications. They implement their method in a program called Skalpel, which is discussed in §2.7.

Neubauer and Thiemann [29]

Neubauer and Thiemann [29] use "discriminative sum types" with flow set annotations to find type error slices. The authors use flow-analysis to determine the sources and sinks of all values in a slice — those are the expressions that introduce and consume types. This they claim allows them to determine the direction of the data flow within a program, leading to more precise error locations being reported. The authors provide the theoretical framework for a new type system and carry out preliminary experimentation using a prototype implementation that does not support any full language so far. One of the main advantages of this approach is that it is able to identify multiple type errors in a single pass, this is possible through the use of *multivocal types*. Multivocal types are able to represent multiple types that are not necessarily compatible. By using these types more type information can be encoded after a single pass, increasing the efficiency of constraint generation and constraint solving.

Pavlinovic et al. [32]

Pavlinovic et al. [32] locate type errors by reducing the problem of type error localisation to a maximum satisfiability modulo theory (MaxSMT) problem. Their approach builds on previous constraint based approaches by assigning weights to generated type constraints and allowing

a weighted MaxSMT solver to compute satisfiable subsets of constraint that have maximal cumulative weights. This approach is able to support more complex typing systems, as long as the MaxSMT solver is given adequate reasoning theory. Furthermore, this method allows for compiler specific ranking criteria of type error locations — not all reported error locations are equally useful for solving type errors. For example, a compiler may wish to disregard locations that involve standard library function, as it is unlikely that the error stems from implementation of the standard library.

Seidel et al. [39]

Seidel et al. [39] use machine learning to rank the most likely error location within a program. Their tool, *NATE*, is able to find the real error location in a set of 5,000 ill-typed OCaml programs[9] with 91% accuracy if the top three locations that reported by *NATE* are considered. They find that the "most important contextual signal is whether or not the expression occurs in a minimal type error slice" to rank the likelihood of an expression being at fault for an error. In fact, they find that this contextual signal is so important that it is beneficial to discard parts of the program that were not part of a slice and to only train the model on those parts of the program that are in a slice (instead of letting the model learn to do so itself). They attribute this domain-specific insight as being crucial to producing a classifier that is capable of significantly outperforming the state-of-the-art.

2.7 Skalpel

Skalpel outputs type error reports that conform to the seven criteria set out by Yang et al. [46] under their "Manifesto for Good Type Error Reporting" (See Figure 5) [33]. It does this by building upon the slicing method developed by Haack and Wells discussed in §2.6.2.

Example output generated from SML/NJ and Skalpel on the piece of code shown in Figure 6 demonstrates this improved type error reporting. Figure 8 shows the output given by Skalpel, while Figure 7 shows that given by SML/NJ.

Already with a small example program we can see the advantages of being able to return

[9]Seidel and Jahla [38] compiled a set of 5,000 pairs of programs containing both the ill-typed and type-correct version of each program by extracting both the versions of the from *interaction traces*. They modified the OCaml compiler so that it would record for each programmer every version of the program submitted to the compiler and record whether it was deemed type-correct. This recorded data, which they call the programmer's 'interaction trace', was then an analysed so that every ill-typed program would be paired with the first subsequent program in the programmers trace that is type-correct.

1. **Correct.** This entails both correct detection: errors are reported exactly when the program is not legal, and correct reporting: all the reported sites contribute to the type conflict.

2. **Precise.** Each conflicting site should be located in the smallest useful amount of source text. Moreover, there should be a simple relationship between the conflicting type and the site from which it was inferred.

3. **Succinct.** Error reports should maximise useful information and minimise non-useful information. Long and verbose explanations are tedious to read. Short and terse explanations are hard to understand.

4. **Amechanical.** An error report should not reproduce large amounts of counter-intuitive mechanical inference. This includes reporting with artificially-introduced type variables.

5. **Source-based.** The user need not know anything of the compiler internals to understand the error message. For example error reports should not use "core" syntax generated by the compiler from the original source syntax. The inference algorithms can infer and report on source syntax. However the implementation may be not source-based.

6. **Unbiased.** There should [be] no inherent left-to-right (or similar) bias in the choice of where to report an error when multiple sites contribute to the error.

7. **Comprehensive.** The algorithm should be able to report all sites that contribute to the reported type conflict. The user can reason about the error from the reported sites and does not need to look at other parts of the program to locate the error.

Figure 5: Yang et al.'s "Manifesto for Good Type Error Reporting" (Taken from [46])

```
val f = fn x => fn y => let val w = y + 1 in w::y end
```

Figure 6: Untypable SML code (from Haack and Wells [14])

```
example.sml:1.60-1.64 Error: operator and operand don't agree [literal]
  operator domain: int * int list
  operand:         int * int
  in expression:
    w :: y
```

Figure 7: Output generated by SML/NJ (v110.72)

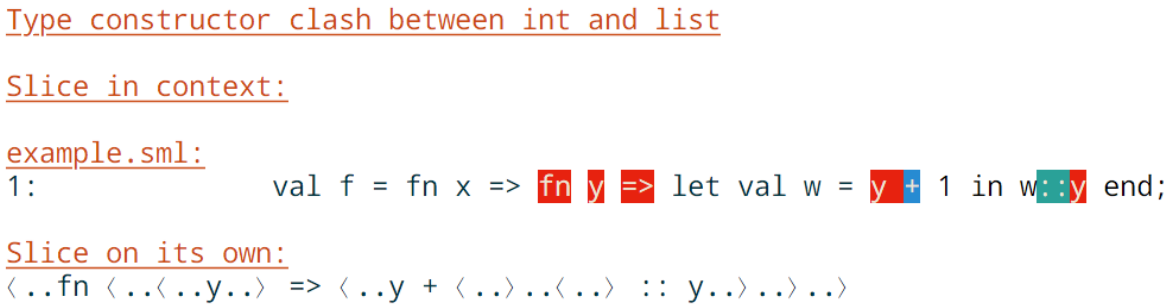

Figure 8: Excerpt of output generated by Skalpel (at git commit `8e61b0d`)

a set of syntax tree nodes. While SML/NJ is only able to point us to the expression `w::y`; Skalpel is able to identify all parts of the program involved in this type error, making it easier for the programmer to diagnose and resolve the error.

While the type error reports outputted by Skalpel are complete[10] and minimal[11], error reports for non trivial programs can be extremely verbose due to Skalpel returning complete sets of program nodes contributing to a type error. This can mean that the error reports can become considerably large, sometimes producing output not much smaller than the input [39], making them hard to understand [48].

Figure 9 presents an informal overview of how the core parts of Skalpel interact with each other (boxes within the diagram represent algorithms, while ovals represent data). Skalpel's core is split into multiple distinct stages, each responsible for specific tasks, all contributing to the generation of type error slices. These stages are: labelling, constraint generation, constraint solving, minimisation, enumeration, and slicing.

[10] Skalpel type error slices are complete, meaning that they contain every part of the program that contributes to a given type error.

[11] Skalpel type error slices are minimal, meaning that they contain only those constraints, and their related program points, that contribute to the type error.

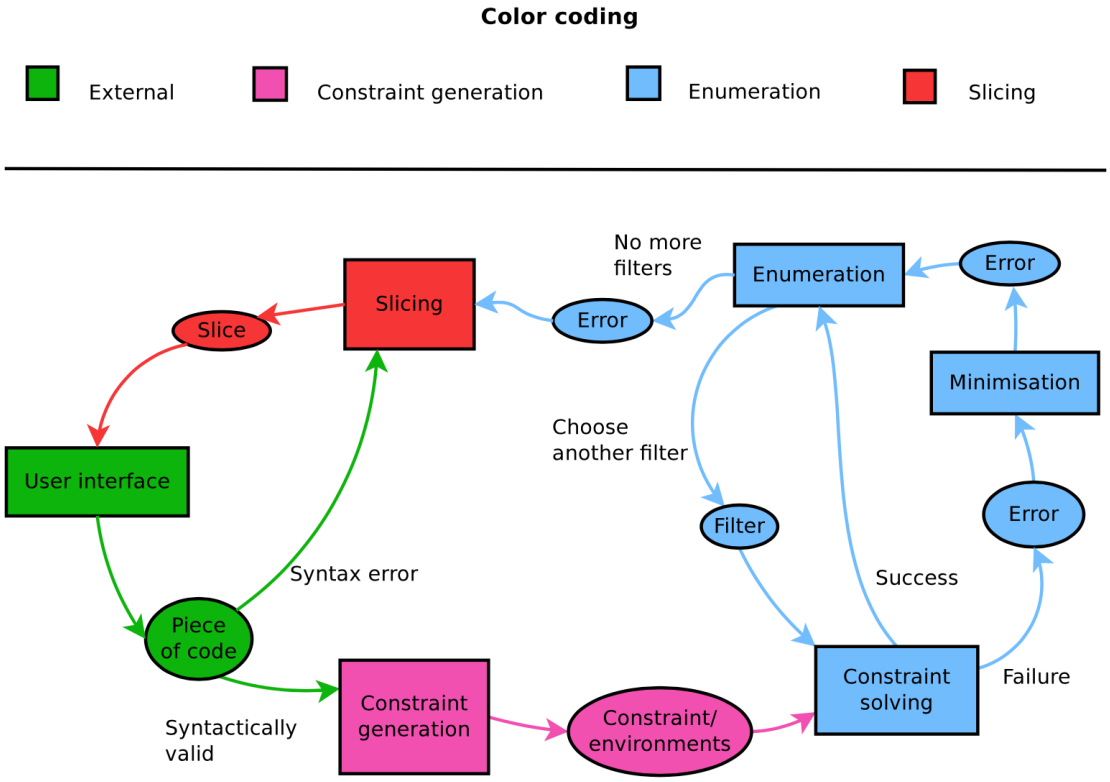

Figure 9: Overview of the Skalpel back end (from Rahli et al. [36])

2.7.1 Program Labelling

One of the reasons that many SML compilers produce bad type error reports is due to their approach to program labelling. Program labelling is the process of assigning a label to every expression in a given program. This is done so that expressions can easily be referenced and differentiated.

Some SML compilers will simplify the inputted program to use only core SML language features before assigning labels to every expression. This simplification is done destructively, resulting in the error reporting system having no access to the original input. This approach is detrimental to the quality of type error messages that can be generated because when a type error occurs the labels that describe the type error relate to the simplified source code — which does not necessarily match the code originally written by the programmer. Outputting this internal representation of the program makes it more difficult for the programmer to find the location being reported as it may not resemble the code contained in the source file.

Skalpel does not destroy any of the original expressions given to it, meaning that the program labels within a type error slice can be related back to the original input. This is a major

contributor to conforming to points (4) and (5) of the Manifesto for Good Type Error Reporting [46] (see Figure 5). The ability to relate these labels back to the original input allows for the creation of error messages that a programmer can easily relate back to the source code, decreasing debugging time and increasing the understanding of the reported error.

2.7.2 Constraint Generation

Skalpel's constraint generator[12] is based on the earlier work of Haack and Wells [14] (discussed in §2.6.2). Skalpel is able to tame the exponential "explosion" of constraints the Haack and Wells method suffers from so that the constraint set increases only linearly in relation to the size of the input program (see discussion of Rahli et al. [36] in §2.6.2 for more detail of how this is achieved). The constraint generation phase creates a hybrid constraint/environment from the labelled source program to be solved by the constraint solver.

2.7.3 Constraint Solving

Constraint solving is the process of attempting to infer a type for each expression in a program such that all constraints are met. The constraint solver takes as input the constraint/environment generated by the constraint generator and terminates in either a success or failure state. If constraint solving succeeds then the input program is typable and no error is generated. If constraint solving fails an error is returned by the constraint solving algorithm. All type errors found during constraint solving are then enumerated and minimised.

2.7.4 Enumeration and Minimisation

In the event that constraint solving fails there may be multiple type errors present in the provided code. It is the task of the enumerator go through all of the errors reported by the constraint solver and to use the minimiser to find minimal sets of program points that represent type errors.

Every error reported by the constraint solver contains within it a set of program labels. Every label within this set does not necessarily contribute to the reported type error. The enumerator passes this set on to the minimiser to reduce this label set into a minimal set of program labels that still produces a reported type error.

[12]The constraint generator is sometimes referred to as the *initial* constraint generator to differentiate it from the constraint solver which can create more constraints.

```
⟨..fn ⟨..⟨..y..⟩ => ⟨..y + ⟨..⟩..⟨..⟩ :: y..⟩..⟩..⟩
```

Figure 10: Type error slice with dot terms

```
val f = fn x => fn y => let val w = y + 1 in w::y end;
```

Figure 11: Type error slice in context (with highlighting)

A set of program labels that produce a type error is minimal when the removal of any label within the set results in a solvable set of constraint associated with those labels (i.e. when all non-contributing labels and their associated constraints have been removed). This minimisation process uses the constraint solver to determine whether the label set is minimal (as shown by the blue loop in Figure 9).

After each minimal error is found the enumerator passes the error containing now minimal sets of program points to the slicer.

2.7.5 Slicing

After enumeration has terminated it is the job of the slicer to inform the user of the type errors that have been found in the source program. The slicer takes the minimal set of program labels for each type error and creates from it a type error slice to present to the user (Figure 8 shows an excerpt of the output generated by Skalpel). The type error slice generated by Skalpel is represented both on its own using *dot-terms* to replace parts of the input program that are not included in the slice (see Figure 10) and in context of the whole input where the program points contained within the type error slice are highlighted (see Figure 11).

In-context highlighting allows a user to see at a glance within the context of the original source code exactly which parts contribute to the generated type error (in this case a type constructor clash between `int` and `list`).

2.8 Visualisation

Lee & Vickers [22] state that the aim of data visualisation is to display information in a manner which allows accurate and effortless human comprehension; data visualisations should establish a "communication channel" between the data and the human trying to understand it.

They make the two fundamental assumption about data visualisation. Firstly, they assume

that information becomes more meaningful to humans when presented in a manner that is compatible with their mental representation of the data. Secondly, they assume that information is more accurately conveyed if presented in a way which is compatible with human visual processes.

Using data visualisation in order to improve clarity and understanding is not a novel idea. In 1879, Frege [11] described a graphical representation of logic expressions. Since then, there have been instances of graphics and data visualisation being used to ease understanding of programming concepts. Visual Programming is a term used to describe the idea of programs being created, edited, and/or executed in a graphical/visual form [30].

Visual Programming has become an increasingly popular teaching tool used to introduce core programming concepts to students. Tools such as MIT's Scratch [4] and Android App Inventor [3], and Microsoft's MakeCode [2] make programming more interactive and easily understandable by 'hijacking' the power of our visual perception. Work has also been carried out in creating similar tools for higher-order functional programming languages and their associated type systems.

The following sections summarise some of the work carried out in visualising functional programming languages.

2.8.1 Pagan [30]

Pagan [30] describes one of the first attempts at creating a visual/graphical functional programming language. The language implemented is "the purely applicative, variable-free functional programming...". Pagan defines six functional combining forms, each with their own graphical representations. The language also comes with a number of predefined primitive functions — such as arithmetic, comparison, logical, and list manipulating functions. Pagan also describes the editing environment that may be used to create, edit, and execute these graphical programs in an interactive manner.

The language implements a simple type error system, where whenever a function is applied to an inappropriate argument a special symbol, ?? called 'bottom', is returned. In addition to this, all functions in this language map ?? to itself.

Figure 12 depicts the six functional combining forms of the graphical functional programming language introduced by Pagan [30] (the language is not named in the paper, but will henceforth be referred to as 'GFP' in this document). The six functionals of GFP are (in clockwise order

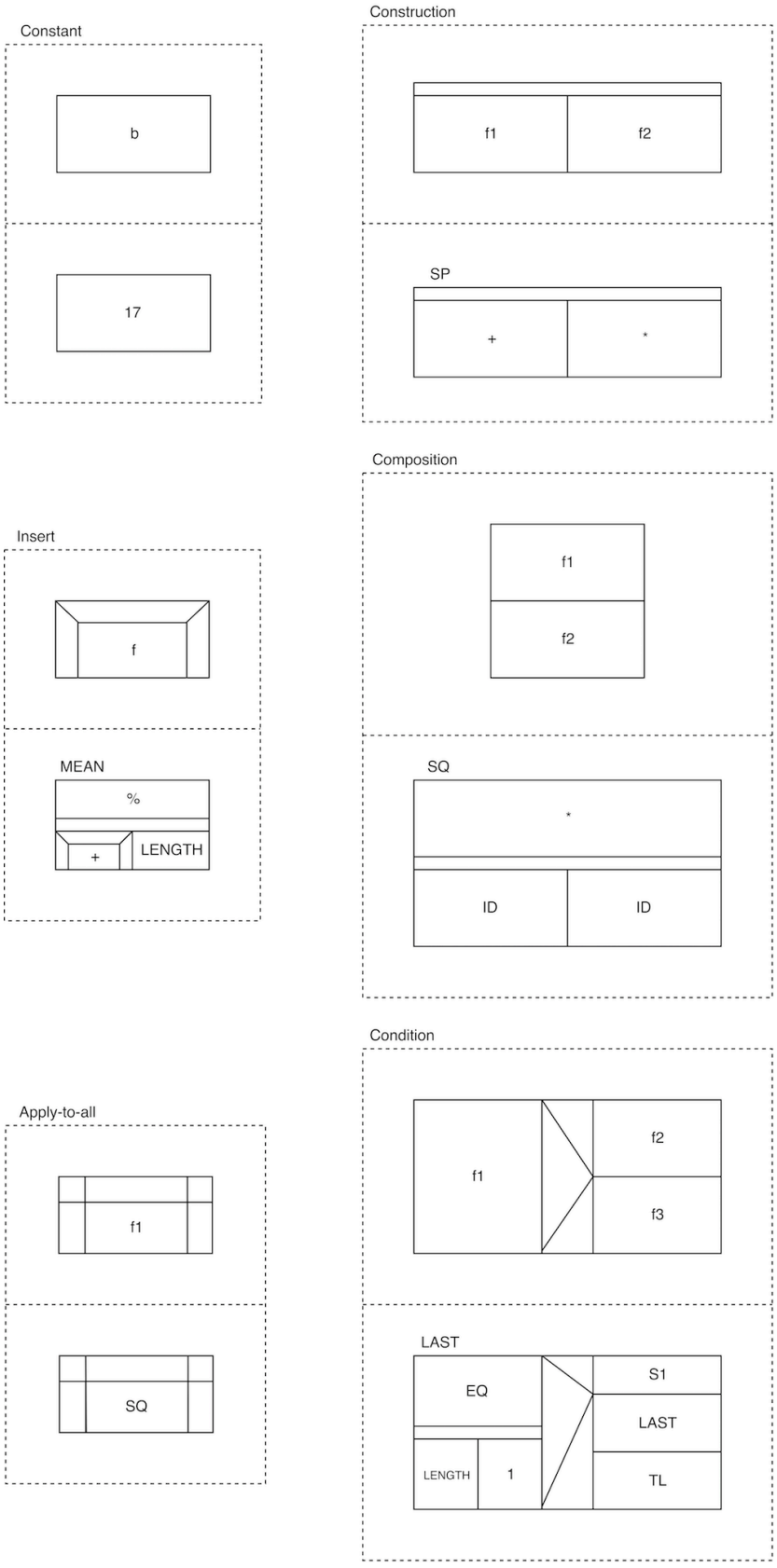

Figure 12: The 6 functional combining forms in GFP (Adapted from [30])

of Figure 12, starting from the top-left): the constant functional, the construction functional, the composition functional, the condition functional, the apply-to-all functional, and the insert functional.

Each representation is shown in an enclosing box with dotted lines, this box is split in two. The top half of the box shows the general form of the functional, the bottom shows an instance of it being used. In some of the uses a label is given in the top left-hand side, this denotes a named function.

The constant functional is the simplest functional and is represented by a rectangle. It's graphical form represents a function that maps all inputs (bar ??) to the symbol that can be seen in the rectangle. The use shown in Figure 12 is equivalent to `fn constant => 17` in SML.

The construction functional, represented by two rectangles of equal size side-by-side with a small bar on top of them, denotes the creation of a new function which when applied to an argument will return the tuple containing the result of applying the functions underneath the bar to that same argument. The given use defines a function SP that would be equivalent to the following SML function `fun SP (a, b) = (a+b, a*b)`. The general form of this functional could be described by the function `fun construction f1 f2 a = (f1 a, f2 a)`.

The composition functional, represented by two rectangles of equal size on top of each other, corresponds to function composition as commonly used in mathematics. The given example describes the function *SQ* (which returns the square of its argument) is equivalent to the following SML `fun SQ n = n * n`. The general form of this functional could be describe by the function `fun composition f1 f2 a = (f1 o f2) a` or `fun composition f1 f2 a = f1 (f2 a)`. (As such the function *SQ* could be built using functionals as `fun SQ n = composition (op *) (construction id id) n`, where `id` is the identity function `fun id x = x`).

The insert functional, represented by what looks like a box within a box, is equivalent to what many modern language implement as the *reduce* function. Given a list of arguments of the form $b1, b2, ..., bn$ the insert functional is equivalent to the expression $b1 f b2 f ... f bn$ (given that f is an infix operator). The equivalent in SML would be `fun insert f [h] = h | insert f (h::t) = f (h, insert f t)`. The given example defines a function $MEAN$ that calculates the mean of a sequence of numbers. We can represent this

in SML as `fun MEAN seq = composition (op div) (construction (insert (op +)) (List.length)) seq`.

The apply-to-all functional is synonymous with the commonly known `map` function, which given a function and a list will return the list containing the result of applying the given function to every element in the list. The equivalent SML would be `fun applyToAll f [] = [] | applyToAll f (h::t) = (f h)::(applyToAll f t)`, or simply `fun applyToAll f l = List.map f l`.

The condition functional is an if-then-else statement of the form `if f1 then f2 else f3`.

GFP represents one of the very first attempts at creating a visual functional programming language, having been implemented over three decades ago. Even with the limited graphic capabilities that were available at the time (1987), GFP allows for creation of arbitrary programs in a non-textual context. It is an impressive feat. The restriction on computing resources required simple, no frills representations; this should be seen as a feature, not a drawback. The representations create a treemap-like equivalent of a textual functional language.

There are limitations in the expressiveness of the program that can be built in GFP — due to it being variable free, as well as its simplistic type-system. Also, GFP does not support some of the core features that modern functional languages implement.

2.8.2 Najork & Golin [28]

Najork & Golin [28] extend the visual programming language Show-and-Tell (STL) [21] by implementing a polymorphic type system based on that described by Milner [27], as well as by allowing user-defined higher-order functions. They call this new language Enhanced-Show-and-Tell (ESTL).

STL — developed by Kimura, Choi, & Mack [21] — is a visual programming language. It represents constants, variables, and operations (functions), as boxes. Data flows between boxes through pipes (represented as arrows). ESTL introduces types through type icons (See Figure 13). ESTL defines four primitive types: Integer, Real, Character, and Boolean.

ESTL also defines constructors, or *typemakers*, for structure (tuple) and union types. The way in which these are represented are shown in Figure 14. Tuple types are represented by a box which encloses multiple other type icons or diagrams. Union types are represented by an enclosing box within which are multiple other type icons or diagrams which are split by

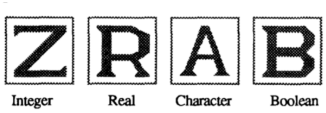

Figure 13: ESTL Primitive Type Icons (from Najork & Golin [28])

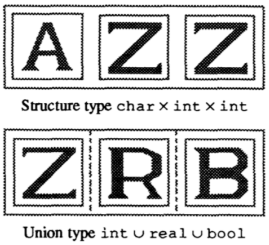

Figure 14: ESTL Union and Structure Typemakers (from Najork & Golin [28])

dotted/broken lines. A *type diagram* is a set of types that have been grouped/composed together using one of these typemakers. Type diagrams can be given a new icon, thus creating a new user-defined named type. A type icon along with its type diagram constitutes the declaration of a new type (See Figure 15 for an example of a definition of an enumeration type).

ESTL also provides the *void* type, equivalent in meaning to SML's `unit` type. It is defined as an all grey/black box, which names the empty type diagram (i.e. the empty tuple `()`), Figure 16 depicts this definition. Finally, ESTL provides the ability to use *type variables*, these are placeholders for types that may be used in definitions for polymorphic type diagrams (e.g. lists, trees). ESTL provides a predefined icon for a type variable, shown in Figure 17. Should multiple type variables be required in a definition, the T is sub-scripted appropriately. Figures 18 and 19 depict definitions for trees and lists. The definition for trees (Figure 19) also shows how the polymorphic type variable is instantiated.

This collection of constructs (a collection of primitive types, structure and union typemakers, a type naming convention, and type variables), gives ESTL a type system equivalent to that described by Milner [27] and that available in the family of ML languages. Najork & Golin go as far to state that these representation can "be viewed as a visual syntax for Milner's".

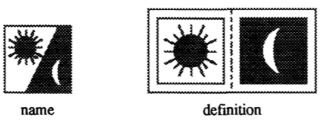

Figure 15: ESTL Definition of a Day/Night Enumeration Type (from Najork & Golin [28])

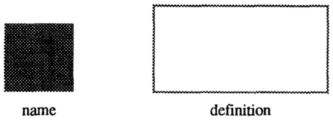

Figure 16: ESTL Definition of the Void Type (from Najork & Golin [28])

Figure 17: ESTL Polymorphic Type Variable Icon (from Najork & Golin [28])

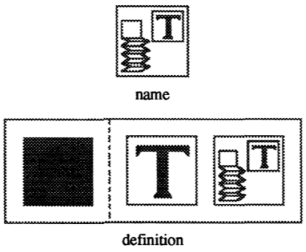

Figure 18: ESTL Definition of the Polymorphic List Type (from Najork & Golin [28])

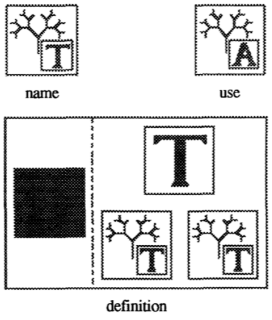

Figure 19: ESTL Definition of the Polymorphic Tree Type (from Najork & Golin [28])

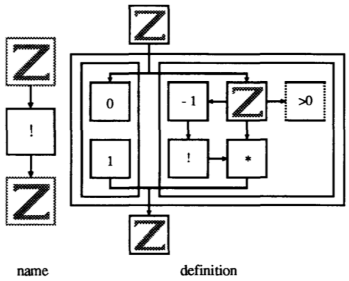

Figure 20: ESTL Definition of a Function Computing the Factorial of Its Input (from Najork & Golin [28])

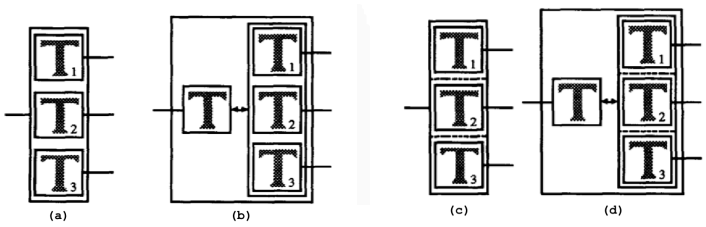

Figure 21: ESTL Structure & Union type constructors/selectors (from Najork & Golin [28])

ESTL implements a type inference system, and therefore does not require explicit type annotations or declarations. These can optionally be included. Figure 20 shows a definition of the factorial function, with type annotations included, explicitly defining its input and output types to be of Integer type.

ESTL also defines constructors and selectors for structure types, and type-filters for union types. These constructors allow for functionality synonymous to pattern matching in SML, as well as for dynamic construction of more complex types. Figure 21 depicts these ESTL constructs. Two constructors/selectors exist for both union and structure types, one anonymous (Figure 21a,c), the other named (Figure 21b,d). Data will flow from one side to the other, constituting either a selection, or a construction. In Figure 21 selection occurs when data is flowing left-to-right, and construction right-to-left. These can then be used to create very powerful functions; Figure 22 shows the definition of a preorder tree traversal function. In this definition we can see how both the structure and union type selector are used for pattern matching either on leaf/empty nodes or internal nodes.

The preorder binary-tree traversal function shown in Figure 22 also shows how ESTL allows user-defined higher-order polymorphic functions. ESTL introduces the notion of a *function slot* that is analogous in use to how a type variable is used in ESTL. A *function slot* is a placeholder parameter box within the function name icon. This *function slot* takes as input a function

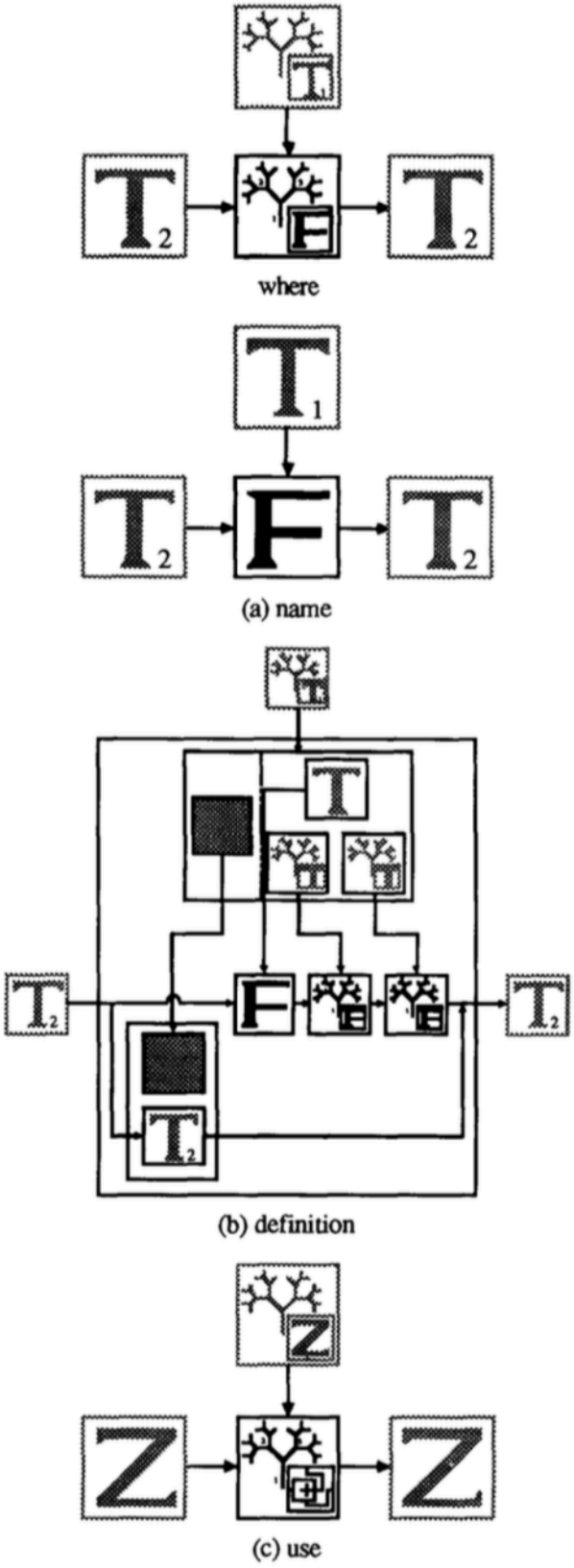

Figure 22: ESTL Preorder Tree Traversal Function (from Najork & Golin [28])

icon or function boxgraph. This parameter is then 'slotted in' wherever the function slot is referenced within the original function definition.

Najork & Golin [28] implement a powerful and expressive type system with functionality that is equivalent to that available in the core of modern textual-based functional programming languages such as Haskell and SML. Their work does not describe how or if they implement any special reporting system if their type inference algorithm fails. Given the visual nature of ESTL it may be unnecessary to do so as error may already be clearly identifiable through icon clashes. It could also be possible that the editing interface simply does not allow the joining of expressions if their types are not compatible.

A common theme in visual programming languages that implement type inference algorithm is that they evaluate the program for type correctness whenever a change is made to the program. The visual editing interface may then either warn the user or disallow the operation [7, 20, 34]. This is a feature that may only be feasibly implemented in visual programming language, as the language designers have finer control over how language constructs are 'written' and can therefore make choices of when it makes sense to type check the program. Making these choices would subvert some of the problems that exist in the type error report systems described in previous sections (§2.5, §2.6). By reporting type errors as soon as they are introduced the type-error location becomes much less ambiguous.

2.8.3 Yang & Michaelson [19]

Yang & Michaelson [19] create a visualisation for the type inference process that takes place during compilation in SML. The main purpose of this work was to improve the *understanding* of type checking and in turn type-errors for a subset of SML (As mentioned in §2.5 understanding type errors often requires a significant understanding of the underlying type-checking system that a language implements).

Instead of using icons to represented types, a method that many other visualisation efforts have taken, Yang & Michaelson use colours. Each base type is represented as a rectangle of solid colour. The colours that have been selected for the types are as followed: `int` in red, `real` in purple, `bool` in pink, and `string` in orange. They hope that while this approach may not necessarily allow the programmer to quickly identify the actual type of an expression it will help in identifying inconsistent type combinations. Figure 23 depicts the representations of the base types (black and white printing of these may not be allow for easy discrimination between type,

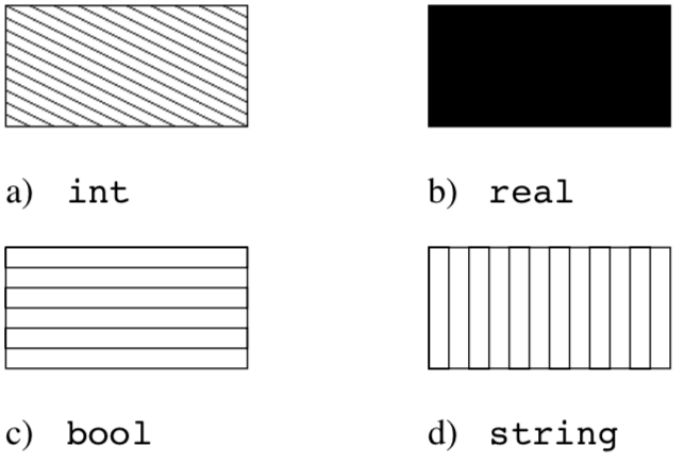

Figure 23: Yang & Michaelson's Representation of Base Types (from Yang & Michaelson [19])

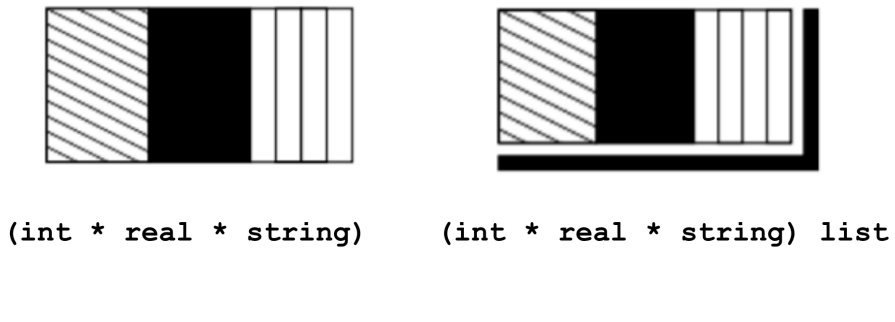

Figure 24: Yang & Michaelson's Representation of Tuple and List Types (from Yang & Michaelson [19])

as such Yang & Michaelson provided "schematic renderings" of the icon visualisations instead).

Yang & Michaelson [19] also provide visualisations for tuple types, and lists. They selected subset of SML does not include construction of datatypes, and therefore they do not provide constructs to support this (they do mention that extending support of their visualisation to a wider range of language constructs is 'under consideration'. However, no further document describing such work has been found by this author). Figure 24 shows the representation of tuples and lists that has been selected. Tuples are shown as a rectangle with equal 'vertical stripes' for the type of each element within the tuple. Lists are depicted a 'shadowed' rectangle, with the rectangle in the foreground depicting the type of the elements within the list.

Finally, they introduce a visualisation for function/arrow types. While they acknowledge that formally SML functions are mappings from single range to a single domain, they also

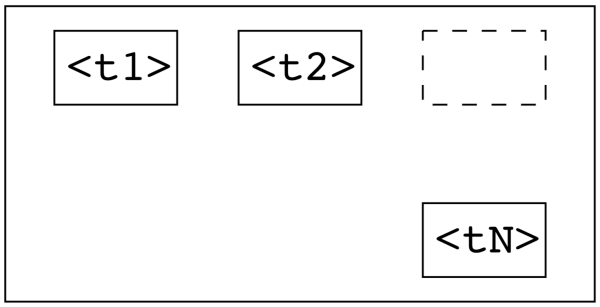

Figure 25: Yang & Michaelson's Representation of General Function Types
`t1 -> t2 -> ... -> tN`
(from Yang & Michaelson [19])

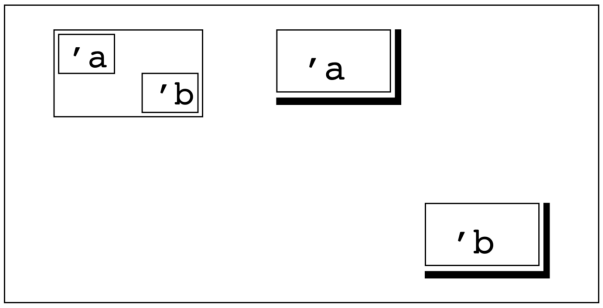

Figure 26: Yang & Michaelson's Representation of the type of the Map Function
`(`a -> `b) -> `a list -> `b list`
(from Yang & Michaelson [19])

acknowledge that it is common practice to treat curried functions as function that may be applied to several arguments (As discussed in Footnote 7). In accordance with this, their visualisations allow for this interpretation of functions with several arguments. Function types are represented as a rectangle containing the types of its arguments listed from top-left to top-right, with the return type in the bottom-right. Figure 25 depicts a generalised visualisation of a function type. Figure 26 shows the visualisation of the standard `map` function.

Figure 26 also shows how polymorphic type variables are represented, namely as rectangles containing the normal SML representation of a type variable (primed lowercase letters `` `a, `b, `c, ... ``).

Yang & Michaelson carried out an evaluation of their new visualisation. They asked 16 participants (split into two groups in order to account for order-effects), all of which were first year student at Heriot-Watt University that had been enrolled on a course on Functional Programming. Two sets of questions were devised (each with 20 questions) where each question presented a function and its type, and an argument, and four options for what the type of

applying the given function to the given argument could be (the result type). Participants were asked to select which result type they thought was correct. Each set of question had a textual version (representing type in the conventional manner) and a visual version (using the conventions introduced by Yang & Michaelson [19]). One group answered the visualised version of question set 1 followed by answering the textual version of set 2. The other group did the opposite, answering the textual version of question set 1, followed by a visualised version of question set 2. Participants were given 10 minutes to answer each set of questions

Yang & Michaelson [19] found that there were no significant advantages or disadvantages to using the visualised representation of types over the textual representation. In general they observed that participants made fewer mistakes when presented with the visualised representation, but failed to answer as many questions. These observations were however statistically insignificant. Yang & Michaelson suggest that an approach consisting of both visual and textual representation of type signature may be the most effective, appeasing both those that find visualisation helpful, and those that prefer textual representation.

2.8.4 Graph/Edge Layout

Many visualisations summarised in the previous sections, and the visualisation created as part of this project involve joining/linking together different objects with arrows or lines. The objects in these visualisations can be seen to be nodes, and the arrows/lines can be seen as edges. Graph/Edge layout algorithms are algorithms that can be used to calculate the optimum placement of nodes and edges, to avoid (if possible) undesirable properties appearing in the rendering of the graph (such as overlapping/crossing of edges). A well rendered graph is more easily read and therefore understood.

For the purposes of this project the graph layout algorithms made available through the D3 JavaScript library will be used. D3 was chosen after consultation with the project supervisor and on the advice of other members of teaching staff in the School of Mathematical and Computer Sciences at Heriot-Watt University.

3 Design & Implementation

This section of the document covers the work that has been achieved over the course of this project. It is be split into four subsections covering aspects of the project relating to front-end, back-end, the interface between the front and the back ends, as well as miscellaneous work carried out. Each subsection contains the requirements that were elicited at the beginning of the project, followed by a discussion of how these requirements were achievement, or why they were not. Any new requirements that became apparent over the course of the project are introduced (new requirements are shown in their respective tables with an asterisk (*) next to their requirement identifier/ID).

3.1 Back-End

The majority of this project was spent working on the *back-end*. The back-end refers to the Skalpel analysis engine that was discussed in §2.7. The Skalpel codebase[13] is fairly mature, consisting of over 55,000[14] lines of SML code and the current version control system (`git`) traces back the original 'commit' to October of 2011 (work was previously held under `svn`). In light of this, all additions/modification to the back-end followed previous design decisions made during development. This was done so that the project retains some consistency in its implementation and is not confusing for any future contributor. The core contributions made to the Skalpel back-end are summarised in the following list:

- Improved printing of constraint environments
- New AST traversal function which finds binder/accessor pairs
- New JSON output format
- Numerous small refactorings
- Improved documentation of how the back-end works through source-code comments

The overarching goal of the changes made to the back-end was to extend Skalpel to produce a new output format that contained the information required for the front-end to generate a visualisation.

[13] Available at `https://github.com/ultra-group/skalpel`.

[14] Found by running the following command in the base directory for the analysis engine source code (`/analysis-engines/standard-ml`): `find . -name "*.sml" -o -name "*.sig" | xargs wc -l`.

```
fun average weight list = let
    fun iterator (x, (sum,length)) = (sum + weight x, length +1)
    val (sum, length) = foldl iterator (0,0) list
in
    sum div length
end

fun find_best weight lists = let
    val average = average weight
    fun iterator (list, (best, max)) = let
      val avg_list = average list
    in
        if avg_list > max then
          (list, avg_list)
        else
       (best, max)
    end
    val (best, _) = foldl iterator (nil,0) lists
in
  best
end

val find_best_simple = find_best 1
```

Figure 27: SML code containing a type error (Adapted from Haack & Wells [14])

3.1.1 Requirements

A number of requirements were found during the initial requirement analysis conducted at the start of the project. These initial requirements are presented in Table 1.

Requirements BE-1, BE-2, and BE-3 relate to the most basic functionality required of the changes in the back-end. The example in Figure 3 was chosen as it uses a relatively powerful number of language constructs consisting of `val` bindings, anonymous functions, function application, and the `let` construct. Supporting these few features of SML are enough to describe any computable function[15]. However, this example is still very simplistic and does not offer use of many of the features that make SML such a powerful and expressive language. Because of this requirements BE-2 and BE-3 were updated (See BE-12 & BE-13) to target the code presented in Figure 27 (which is an adapted version of SML code taken from Haack & Wells [14] where `fun` bindings have been used to replace `val` bindings of anonymous functions). The example in Figure 27 was chosen as it is a more complex program that uses more features of SML. As well as this, it contains a more interesting type error that is not very easy to debug[16]. Updating

[15]By supporting variables, abstraction, and application. This subset of SML is as powerful as the λ-calculus (i.e. Turing complete). By supporting `val` and `let` it allows for some convenient reuse of expressions.

[16]There is a clash between a function type and the `int` type that occurs due to the manner in which the `weight` parameter is used in the functions `find_best` and `average`.

Id	Priority	Description	Status
BE-1	Must	Compilation must succeed	Achieved
BE-2	Must	Run on at least the example presented in Figure 3 without crashing	Achieved
BE-3	Must	Run on at least the example presented in Figure 3 and produce usable output	Achieved
BE-4	Must	Not introduce any regressions to original functionality	Achieved
BE-5	Must	Output must be correct	Partially Achieved
BE-6	Must	Not increase space/time complexity significantly	Achieved
BE-7	Must	Support let-polymorphism	Achieved
BE-8	Should	Deal with all errors gracefully (i.e. Not crash)	Achieved
BE-9	Should	Run on a non-trivial subset of SML	Achieved
BE-10	Should	Be extensible to more language features with minimal effort	Partially Achieved
BE-11	Won't	Implement the full SML language specification	Not Achieved
BE-12 (*)	Could	Run on at least the example presented in Figure 27 without crashing	Achieved
BE-13 (*)	Could	Run on at least the example presented in Figure 27 and produce usable output	Achieved
BE-14 (*)	Could	Support type and datatype declarations	Not Achieved

Table 1: Back-End Requirements

this target also leads to the satisfaction of requirement BE-9. Requirement BE-5 left *correct* as ambiguous during initial requirement analysis as it was unclear exactly what approach would be taken, and therefore what data would be outputted. However, now that the new output has been implemented (As discussed in §3.1.2), the notion of *correct* is that the new data outputted identifies only those identifiers that: (1) are defined and used within a users program, (2) appear within a relevant type error slice (as outputted by Skalpel), (3) that the identifiers are indeed related (i.e. the accessing identifier is correctly associated with its binding instance), (4) that if at least one relevant identifier pair is present, then *all* related and relevant identifiers are present. Requirement BE-14 relates to supporting `type` and `datatype` declarations. This was not achieved due to time constraints and is left for future work.

3.1.2 Discussion

Initially, this project aimed to pass more information from Skalpel's constraint generation and solving processes through to the final slicing operation. The idea was that this extra information

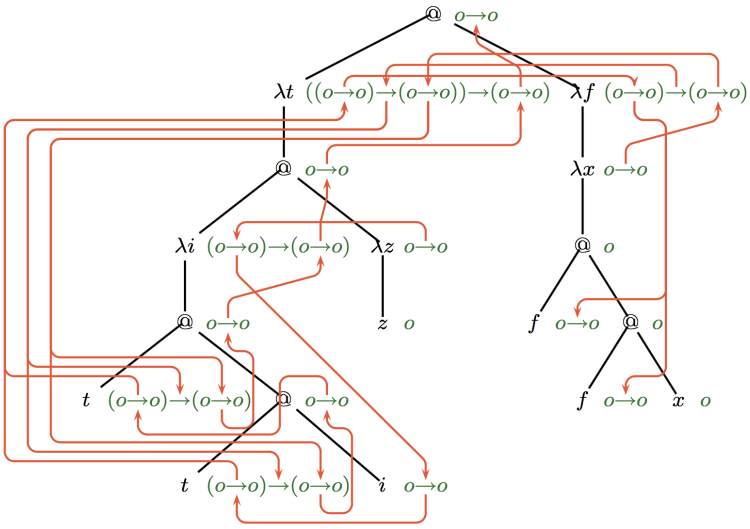

Figure 28: Flow Analysis Visualisation of $(\lambda t.(\lambda i.t(ti))(\lambda y.y))(\lambda f.\lambda x.f(fx))$[17] (from Wells [44])

could be used to describe the paths that unsolvable constraints have taken through a program and that these paths could then be visualised. These paths would then, in theory, help the user to understand in depth how different parts of their program are causing conflicting constraints from being generated. This would then lead to a better understanding of the type error and to fixing it. These paths were inspired by the type-flow analysis visualisation presented by Wells [44] shown in Figure 28.

This approach involved understanding and analysing Skalpel's constraint generation process manually. This proved to be problematic because the format which the debugging output used was not optimised for readability (by humans). This resulted in the need to implement a new pretty-printing function for constraint environments. A debugging option (`CONSTRAINT_SOLVING`) was updated to enable this new output. Figures 29 and 30 show the print-out of the initial constraint set generated for the SML program `fn y => y::y;`. There is a stark change in readability. The original output contained no line breaks at all[18], making it very difficult to extract any useful information. The new output prints out each level of the constraint tree on its respective level, making it more obvious, and less tedious, for the reader to see which constraints are related to each other and how. There is still room for improvement; an item left for future work (due to time/scope constraints) would be to explicitly draw lines between related contraints. This would make it even easier to see which constraints are related as it would remove the need for the reader to remember indentation levels. Imple-

[18]Line-breaks have been included here, so that the excerpt would fit in this document.

```
(CONSTRAINT_SOLVING) Unification.sml: Solving constraint:   TYPE_CONSTRAINT(((TYPE_VAR(t16,-,
    ↪ POLY,EQUALITY_TYPE_STATUS(UNKNOWN)),TYPE_CONSTRUCTOR (TYPENAME_CONSTRUCTION(arrow,
    ↪ BUILTIN_BASIS_CONS,l11),ROW_CONSTRUCTION([FIELD_CONSTRUCTION((2,l11):TYPE_CONSTRUCTOR (
    ↪ TYPENAME_CONSTRUCTION(list,BUILTIN_BASIS_CONS,l11),ROW_CONSTRUCTION([FIELD_CONSTRUCTION
    ↪ ((1,l11):TYPE_VAR(t15,-,POLY,EQUALITY_TYPE_STATUS(UNKNOWN)),l11)],-,l11),l11,
    ↪ EQUALITY_TYPE_STATUS(UNKNOWN)),l11),FIELD_CONSTRUCTION((1,l11):TYPE_CONSTRUCTOR (
    ↪ TYPENAME_CONSTRUCTION(record,BUILTIN_BASIS_CONS,l11),ROW_CONSTRUCTION([
    ↪ FIELD_CONSTRUCTION((2,l11):TYPE_CONSTRUCTOR (TYPENAME_CONSTRUCTION(list,
    ↪ BUILTIN_BASIS_CONS,l11),ROW_CONSTRUCTION([FIELD_CONSTRUCTION((1,l11):TYPE_VAR(t15,-,
    ↪ POLY,EQUALITY_TYPE_STATUS(UNKNOWN)),l11)],-,l11),l11,EQUALITY_TYPE_STATUS(UNKNOWN)),l11
    ↪ ),FIELD_CONSTRUCTION((1,l11):TYPE_VAR(t15,-,POLY,EQUALITY_TYPE_STATUS(UNKNOWN)),l11)
    ↪ ],-,l11),l11,EQUALITY_TYPE_STATUS(UNKNOWN)),l11)],-,l11),l11,EQUALITY_TYPE_STATUS(
    ↪ UNKNOWN))),[l1],[], contextDependancies=))
```

Figure 29: Original Constraint Environment Printout

menting something similar to the format used in the `CONSTRAINT_PATH` debug option seems like a good starting point.

After discussing this constraint-based approach with my project supervisor we felt that, given this projects time-scale, it would not be feasible to achieve a significant presentable amount of work by pursuing this approach. At the beginning of this project it was unknown exactly which approach would be possible or feasible within the scope of an honours project. Part of the work that needed to be carried out was to find out what kind of work would be required to extract extra information from some of Skalpel's analysis steps. It is unfortunate that this approach could not be taken further. However, the work achieved by improving the way in which the constraint environments are presented should be of great aid to any further exploration of this approach.

The implemented approach involves an analysis/traversal of the input programs AST combined with the knowledge of which program labels are relevant to a type error (this is determined by whether a program label is contained within a type error slice returned by Skalpel).

The idea is that bindings/declarations of identifiers are collected and are passed down the AST to the expressions for which these bindings would be 'in scope'. When an accessing occurrence of an identifier is traversed a pair containing the binding occurrence of the identifier and the accessing occurrence is recorded. The traversal algorithm accumulates a list of these identifier pairs and then filters this list to only contain bindings/accessing that are contained within the relevant type error slice. This approach does not support the full specification of SML, instead a reasonable powerful sub-set of SML (including anonymous functions, pattern matching (on primitive types), `fun` declarations, `let` expressions, `case` statements, tuples,

[18]The λ-expression presented is equivalent to the following SML code: `fun twice f x = f (f x); fun id z = z; twice (twice id)`. See footnote 8 for explanation of the `@` notation.

```
(CONSTRAINT_SOLVING) Unification.sml: Solving constraint:
TYPE_CONSTRAINT([l1],[],)
    TYPE_VAR(t16, -, POLY)
         EQUALITY_TYPE_STATUS(UNKNOWN)
    TYPE_CONSTRUCTOR(l1)
         TYPENAME_CONSTRUCTION(arrow, BUILTIN_BASIS_CONS, l1)
         ROW_CONSTRUCTION(-, l1)
              FIELD_CONSTRUCTION(l1)
                   LABEL_CONSTRUCTION(2, l1)
                   TYPE_CONSTRUCTOR(l1)
                        TYPENAME_CONSTRUCTION(list, BUILTIN_BASIS_CONS, l1)
                        ROW_CONSTRUCTION(-, l1)
                             FIELD_CONSTRUCTION(l1)
                                  LABEL_CONSTRUCTION(1, l1)
                                  TYPE_VAR(t15, -, POLY)
                                       EQUALITY_TYPE_STATUS(UNKNOWN)
                        EQUALITY_TYPE_STATUS(UNKNOWN)
              FIELD_CONSTRUCTION(l1)
                   LABEL_CONSTRUCTION(1, l1)
                   TYPE_CONSTRUCTOR(l1)
                        TYPENAME_CONSTRUCTION(record, BUILTIN_BASIS_CONS, l1)
                        ROW_CONSTRUCTION(-, l1)
                             FIELD_CONSTRUCTION(l1)
                                  LABEL_CONSTRUCTION(2, l1)
                                  TYPE_CONSTRUCTOR(l1)
                                       TYPENAME_CONSTRUCTION(list, BUILTIN_BASIS_CONS, l1)
                                       ROW_CONSTRUCTION(-, l1)
                                            FIELD_CONSTRUCTION(l1)
                                                 LABEL_CONSTRUCTION(1, l1)
                                                 TYPE_VAR(t15, -, POLY)
                                                      EQUALITY_TYPE_STATUS(UNKNOWN)
                                       EQUALITY_TYPE_STATUS(UNKNOWN)
                             FIELD_CONSTRUCTION(l1)
                                  LABEL_CONSTRUCTION(1, l1)
                                  TYPE_VAR(t15, -, POLY)
                                       EQUALITY_TYPE_STATUS(UNKNOWN)
                        EQUALITY_TYPE_STATUS(UNKNOWN)
         EQUALITY_TYPE_STATUS(UNKNOWN)
```

Figure 30: New Constraint Environment Printout

lists, and `val` declarations) is supported. Notable omissions include `type`, `datatype` declarations, as well as any of SML's module system constructs (e.g. `structure`, `signature`, and `functor` declarations). These were omitted due to time constraints. It was foreseen that it would be infeasible to support the full specification of SML and these constructs were deemed to be of lower priority (either because other language constructs are more widely used, or provide greater functionality). If an input program contains any unsupported parts of SML the traversal algorithm will raise an exception. This exception is handled by Skalpel which results in the binding/accessor list being replaced with the empty list. Only the newly created extra traversal of the AST is affected by this. The standard analysis Skalpel performs continues unhindered. The decision to return an empty list of binding/accessor pair was made so as to avoid any confusion that could be caused by outputting *some* of the offending accessing and not all of them. It was deemed that not drawing any would be less detrimental to a user's understanding than outputting partial information (outputting only some of the links may lead to a user disregarding part of the program because no relevant information was returned). Placeholder functions within the implementation exist that should make it relatively easy for support to be added for further language constructs in the future. This list of identifier pairs forms part of the new JSON output format, along with highlighting information that Skalpel originally included (functions were implemented to encode this information in JSON), the original source code, the error message, and Skalpel version information. Figure 31 shows this new output format for the SML program `fn y => y::y;`. The top-level JSON object contains five keys: `regions` contains the highlighting information, `links` containing the accessor and binding pairs, `msg` containing the error message, `source` containing the 'exploded'[19] source code, and `version` containing the information of the Skalpel version that generated the output. This new output format is enabled by passing the `-z <output_file>` flag to the Skalpel binary (i.e. `skalpel -z <output_file> <source_file>`).

The rationale behind collecting these accessor and binding pairs is that a significant number of type errors occurs due to multiple conflicting uses of variables. Chitil, Huch, & Simon [8] state that the natural way that users debug errors is by examining surrounding expressions for every occurrence of the identifier, in order find which use results in a conflicting type being inferred (interestingly, such debugging was observed in a majority of the participants that took

[19]The source code string is exploded into a two dimensional array of characters, allowing for easier access and reference of specific parts of the source through line and character numbers.

part in the evaluation described in §4). Collecting where these identifiers are used and defined should be enough information for a visualisation framework to give the user visual cues of where relevant identifiers are used (See §3.2.2 for discussion of how the visualisation created for this project does this).

3.1.3 Conclusion

The contributions to the back-end of Skalpel resulted in the creation of a new output format that may be able to become the only format that Skalpel needs to use. This would mean that only a single output format would be required for the multiple interfaces Skalpel makes available. Further to this, the output now provides richer information in the form of outputting a list of binder/accessor pair for identifiers. This new information provides the basis for a new attempt at visualising the type error slices generated by Skalpel.

Exploratory work was carried out relating to extracting more information from of Skalpel's constraint generation and solving process. This resulted in improved outputting of constraint environments. While the decision was taken not to pursue this approach, it remains a viable option for future projects to be based upon.

Additionally, where required and helpful, improvements to pre-existing code were made — through refactoring to improve readability, maintainability, or ease of extension. Skalpel's code base is large and has been written by a number of different authors over more than a decade. There is room for improvements in code style and design throughout the project (as is the case in all large projects) and the technical-debt that has accumulated should always be kept in check by making small 'repayments' where possible. One instance of this is the refactoring work done to the parameter parsing function implemented for Skalpel. This function was somewhat inconsistent in its style and contained significantly over-engineered logic that complicated the process of adding new command line parameters (which was necessary for this project). While it would have been possible to add the required functionality without changing much of the function, this seemed a less-than optimal approach to take for the health of the Skalpel codebase in the long term (many of the improvements describe in §3.3.2 were undertaken with this same thought in mind). The function was refactored to be simpler and more consistent and it should be fairly obvious how to implement any new command line parameters for future authors.

Finally, in order to help any future potential contributors to Skalpel understand some of the code of the Skalpel back-end, source-code comments were added explaining: the flow of

```
{ "regions": [
    { "type": "leaf",
      "color": "#FF0000",
      "weight": 1,
      "region": {
        "fromLine": 0,
        "fromColumn": 0,
        "toLine": 0,
        "toColumn": 1 } },
    { "type": "leaf",
      "color": "#FF0000",
      "weight": 1,
      "region": {
        "fromLine": 0,
        "fromColumn": 3,
        "toLine": 0,
        "toColumn": 3 } },
    { "type": "leaf",
      "color": "#FF0000",
      "weight": 1,
      "region": {
        "fromLine": 0,
        "fromColumn": 5,
        "toLine": 0,
        "toColumn": 6 } },
    { "type": "leaf",
      "color": "#FF0000",
      "weight": 1,
      "region": {
        "fromLine": 0,
        "fromColumn": 8,
        "toLine": 0,
        "toColumn": 11 } }
  ],
  "links": [
    { "bind": {   "identifier-string": "y",
                  "identifier-id": 2,
                  "region": { "fromLine": 0,
                              "fromColumn": 3,
                              "toLine": 0,
                              "toColumn": 3 } },
      "access": { "identifier-string": "y",
                  "identifier-id": 2,
                  "region": { "fromLine": 0,
                              "fromColumn": 8,
                              "toLine": 0,
                              "toColumn": 8 } } },
    { "bind": {   "identifier-string": "y",
                  "identifier-id": 2,
                  "region": { "fromLine": 0,
                              "fromColumn": 3,
                              "toLine": 0,
                              "toColumn": 3 } },
      "access": { "identifier-string": "y",
                  "identifier-id": 2,
                  "region": { "fromLine": 0,
                              "fromColumn": 11,
                              "toLine": 0,
                              "toColumn": 11 } } }
  ],
  "msg": "Circularity",
  "source": [["f", "n", " ", "y", " ", "=", ">", " ", "y", ":", ":", "y", ";"]],
  "version": { "commit": "4256a8630607d92e4ea6d5f219d5229f2e8c90e7",
               "compiler": "Poly\/ML"} }
```


Figure 31: New JSON Output Format

the code from entry-point through to slicing, the *interesting* manner in which Skalpel manages outputting files, as well as a general overview of how the slicing loop has been implemented. This is done in the hopes that future contributors can get 'up-to-speed' with the codebase faster and in turn begin contributing earlier in their project.

3.2 Front-End

The front-end portion of this project involved created a visualisation based on the new output that was created as part of the work carried out on the Skalpel back-end (discussed in §3.1). The front-end was designed as a completely separate project with its own codebase[20]. This was done to ensure that there was a separation of concerns between the analysis and the visualisation, reducing coupling. This new codebase/project is called *dissect*.

The new interface is browser-based. The decision to go browser-based (as opposed to using a native desktop application solutions such as JavaFX, Qt, or GTK) was made for the following reasons:

- No need for extra installation/updating
- Easier portability
- More consistent behaviour across platforms
- Tooling available (JS, CSS, and associated libraries)
- Ease of development

The main advantage of creating native desktop cations is in the performance boost. Because all of the intensive computation will be performed by the back-end this is not required by the front-end. Native applications have substantial development overhead. This is not the case in browser based-applications. Browser-based applications (when hosted) do not require the user to keep the application up-to-date (users are notorious for not updating software). This means that features can be added and made available instantly and bugs can be fixed without requiring any extra action from the user. Furthermore, the browser standard browser toolkit of HTML5, CSS3, and JavaScript is extremely powerful and flexible.

[20]Codebase hosted at `https://github.com/CGA1123/dissect`.

Id	Priority	Description	Status
FE-1	Must	Produce correct representations of type errors	Achieved
FE-2	Must	Not require additional installation by users	Partially Achieved
FE-3	Must	Run on Mozilla Firefox v57.0 or greater	Achieved
FE-4	Must	Run on at least the example presented in Figure 3	Achieved
FE-5	Should	Support all features supported by the new back-end output	Achieved
FE-6	Could	Run on all modern browser	Achieved
FE-7 (*)	Could	Support multiple slices	Achieved
FE-8 (*)	Could	Run on the example presented in Figure 27	Achieved

Table 2: Front End Requirements

3.2.1 Requirements

Table 2 contains the requirements that were elicited as part of the initial requirements analysis conducted at the beginning of this project and requirements that became apparent over the course of the project. Requirement FE-1 leaves the definition of 'correct' undefined. The definition of correct in this instance is similar to that described in §3.1.1. Namely, (1) the front-end should only highlight those parts of the program that are present within the error slice; and (2) the front-end should only draw links between program points that have been identified by the back-end as being relevant. FE-2 is marked as being partially achieved because it using dissect still requires the user to have Skalpel installed on their local machine. The user also needs to have access to the interface/wrapper script that has been created to generate the data from Skalpel's analysis into a format accepted by dissect (This script is discussed in §3.2.2). Requirements FE-4, FE-5, FE-7, and FE-8 are related to supporting features that are provided by the back-end and were derived from the back-end requirements. Requirements FE-3 and FE-6 are related to compatibility across browsers — the visualisation has been tested and developed using Firefox (`v59`) and Google Chrome (`v66`) and appear to work fully, the visualisation was briefly tested with Safari (`v11`) no noticeable incompatibilities were noticed.

3.2.2 Discussion

The interface was created using the D3 JavaScript library. Figures 32 & 34 show examples of what the front-end visualisations look like. The visualisation works by taking the binding/accessor pair information generated by the back-end and drawing links between these. The motivation behind this is to make the declarations and uses of identifiers more quickly identifi-

Figure 32: Example of Visualisation Generated by dissect

able within a program. By doing this it should become more quickly obvious where identifiers are being used in conflicting ways. Drawing links should guide the programmers attention to relevant parts of the program more easily as they do not have to search the program to find binding and accessing instances of an identifier themselves.

The browser interface requires the user to manually load a visualisation file as created by the interface script. It is important to note that the final file that the interface script generates is *not* equivalent to that created solely by running a source file through Skalpel with the appropriate command line options. The script provided wraps the file(s) outputted by Skalpel analysis into a JSON array which is assigned to the 'data' key in a new top-level JSON object. This is done to increase the interactivity of the front-end by enabling easily switching between many reported slices. This switching is done by clicking the Previous/Next buttons that can be seen in Figures 32 & 34. The front-end also re-implements the highlighting currently available in the command-line output.

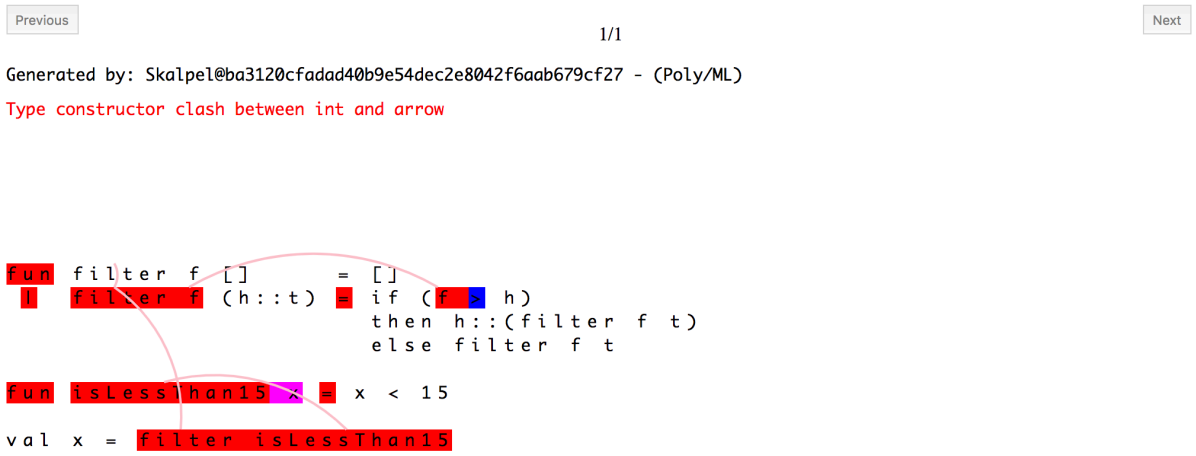

Figure 33: Example of Visualisation Generated by dissect

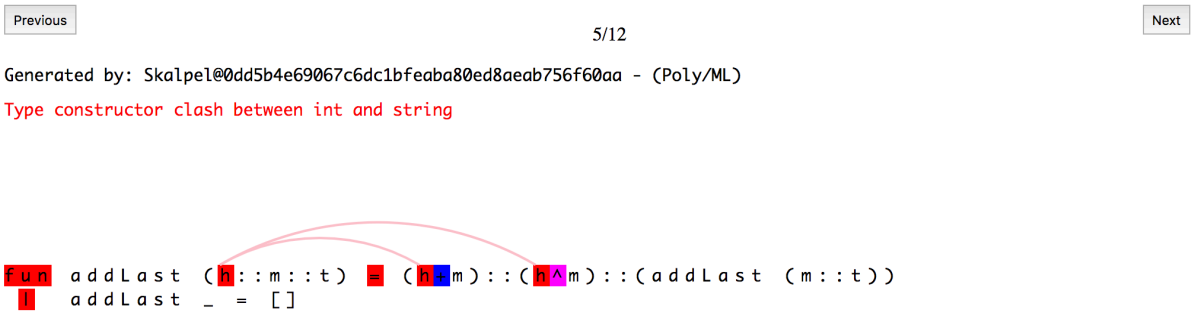

Figure 34: Example of Visualisation Generated by dissect

3.2.3 Conclusion

Work has been carried out to create a brand new interface for the Skalpel type-error slicer. The new JSON output format generated by the back-end is processed by a interface/wrapper script which creates another file that can then be inputted to the visualisation. The visualisation uses the information from Skalpel to display the source program with slice highlighting while also drawing links between identifier bindings and accessing. These links are rendered in order to help the user to *see* where identifiers are being used in potentially conflicting ways. In Figure 32 following the drawn links can help the user to see how the first argument given to the `find_best` function (the integer `1`) ends up being passed through as the first argument in the `average` function which applied to some argument `x` — this is how the conflict between a function and integer type is generated. Without these links the user would have to manually find how these different parts of the reported slices are inter-connected.

While this front-end is a good starting point for an approach consisting of visually linking different program points together, there are a number of challenges that still need to be overcome. Most significantly the current method of drawing links between two points is very primitive and does not employ any layout algorithm. This can lead to confusing visualisations where many of the links overlap or cross. This can severely impact the usefulness of these visualisations. This is discussed is further in §5.1.

3.3 Miscellaneous

As well as implementing the core functionality required for this project, working on top of the Skalpel codebase offered the opportunity to improve the project in general. This section describes some of the secondary objectives that have been achieved. While these requirements do not directly influence the success of the project, they eased the software engineering process. It is also a nice thing to try and leave any codebase in a slightly better state than when you started work on it.

These requirements are meant to improve the state of the codebase and to improve the 'start-up' time required for future users or developers to get Skalpel up and running on their local machines.

They focus mostly on updating the way in which Skalpel is built. The following lists highlights the core contributions:

Id	Priority	Description	Status
MS-1	Should	Integrate Skalpel's build process into a continuous integration (CI) service for at least one target SML compiler	Achieved
MS-2	Should	Integrate Skalpel's test framework into a CI service after building has succeeded	Achieved
MS-3	Should	Update the SML Doxygen patch to latest version	-
MS-4	Could	Integrate Skalpel's build process into a CI service for all target compilers	Partially Achieved
MS-5	Could	Integrate Skalpel's deployment process into a CI service for target platforms	Not Achieved
MS-6	Could	Integrate functionality with current web demo	Not Achieved / N/A

Table 3: Miscellaneous Requirements

- Improved/updated building process
- Automatic building of project upon changes being pushed upstream
- Automatic testing of project upon changes being pushed upstream
- Updated Doxygen patch

3.3.1 Requirements

Table 3 shows the miscellaneous requirements that were found during the requirement analysis performed at this beginning of the project. MS-1, MS-2, MS-4, and MS-5 are all related to automating the building, testing, and deployment processes of Skalpel as a project. All of these requirements were met to some degree, apart from MS-5 which was not attempted due to it not being crucial to development and being of lower priority. MS-6 was also not achieved, instead (as described in §3.2) a fully new front-end was developed. MS-3 was achieved fully.

3.3.2 Discussion

Prior to beginning this project the Skalpel codebase had been dormant for over three years. During this time many of Skalpel's dependencies had continued to be actively developed, sometimes introducing breaking changes in newer versions. Furthermore, over the last few years large improvements have been made to the tooling available (mostly free of charge for open source project) for automatically running custom scripts when changes are committed to a code-base.

Such services are commonly referred to as continuous integration (CI) services. CI services allow projects to be monitored on consistent infrastructure for breakages in the build processes as a result of recent changes made. In order to ensure that the changes made throughout this project did not result in an unusable program; it was seen as beneficial to take advantage of CI services to automate the building and testing of Skalpel. Because the codebase is hosted on GitHub, TravisCI [5] was used to provide these services due to the integrations that it provides with GitHub and the prior experience that this author has with the service. Automating the build process involved creating custom scripts to install the required dependencies on the CI infrastructure. As of writing, Skalpel is built against MLton version `20130715` and Poly/ML version `5.7.1`. Automatic building of SML/NJ was not supported due to encountering a number of problems installing it on the CI infrastructure. After installation Skalpel is built using both MLton and Poly/ML, if an error occurs the CI service will notify the person that committed the breaking change through email.

Skalpel's internal testing suite is also run on the MLton version of Skalpel upon every change. The testing suite is incredibly long (taking around 20 minutes on average to complete). Because of this it was deemed that only running the testing suite using the MLton compiled version of Skalpel (which is known to be faster that Poly/ML) would be satisfactory.

Skalpel uses (to some degree) Doxygen [1] to convert source code comments into documentation in various formats. Doxygen does not support SML by default, a patch was created by John Pirie to add this functionality to Doxygen. This patch was created in 2013 and Doxygen has since made significant (breaking) design changes. The patch was updated to work with newer version of Doxygen[21].

The build process of Skalpel had to be updated in order to keep up with the changes that were made to the MLton compiler. Skalpel requires MLton to be installed to compile with MLton and with Poly/ML. This is because some of the standard libraries that are packaged with MLton are not available through the installation of Poly/ML. In one of the latest updates MLton changed the file structures for many of the libraries which Skalpel required. This resulted in Skalpel not being able to be built for MLton or Poly/ML. The build files for Skalpel were updated to reflect these changes. Changes were made to Skalpel's Makefile to streamline the build process (removing duplicate lines, ensuring temporary files are removed, detecting the

[21] The updated patch has been verified to work against doxygen@`929ea15c46c55562862181f59ae2c6b00c046dc0` or version `1.8.14`.

building OS and making any required changes to parameters). These changes should ensure that Skalpel can be easily and cleanly installed/built from source on Darwin or Linux based machines for the foreseeable future.

Unfortunately no work was carried out to investigate the state of 'packaging' Skalpel. A script does exist that *should* allow for automatic packaging for major Linux distributions as well as macOS. However, it is unknown whether it is still functional for newer versions. Investigating this would be of great benefit to Skalpel, as it would greatly reduce the 'entry barrier' of installing Skalpel. Many people are not confident in building projects from source or simply prefer to use a package manage to do so for simplicity. This is however out of the scope for this project and is left for future work.

3.3.3 Conclusion

The miscellaneous contributions made to Skalpel centred around updating and streamlining the build process. This was done in order to make development of the new features described in §3.1 much easier, as well as to make the installation of Skalpel easier and more up to date.

The Skalpel GitHub repository was updated to integrate with TravisCI, which will trigger a build every time a new commit is pushed to the repository or any pull request is updated. This allows developers to quickly see whether their changes have affected the build status of the project as a whole on a consistent infrastructure (getting rid of any infrastructure specific changes that might erroneously cause the project to build successfully or unsuccessfully). Doing this also sets in some foundations for any extra processes that may want to be automated in the future, such as code-linting, packaging, or unit testing.

The previously outdated Doxygen patch that added support for SML was updated to work with the latest version of Doxygen, ensuring any new developers can easily get access to the documentation (and maybe even add to it or update it).

A new documentation file was added to the source code root (`BUILD.md`) giving instructions on how to build Skalpel, as well as links to installation instructions for required dependencies (such as MLton, Poly/ML, autoconf).

Id	Priority	Description	Status
IF-1	Must	Return data from the back-end	Partially Achieved
IF-2	Must	Input data to the back-end as specified by the user	Partially Achieved
IF-3	Must	Response is the result of executing the back-end on the user-provided code	Partially Achieved
IF-4	Should	Expose an HTTP POST endpoint	Not Achieved / N/A
IF-5	Should	Be able to deal with multiple simultaneous requests	Not Achieved / N/A

Table 4: Interface Requirements

3.4 Interface

This project was split into two distinct parts in terms of development: back-end (§3.1) and front-end (§3.2). These two parts must somehow be allowed to interact in some manner so that information generated by the back-end can be used by the front-end — this is called the 'interface' between the front and back ends.

3.4.1 Requirements

Initial requirement analysis was conducted under the assumption that the newly developed front-end would enable users to input source code directly and receive a response from this interface. The result of this analysis are presented in Table 4. Over the course of the project this initial design assumptions was discarded for a number of reason discussed in §3.4.2. As a result, requirements IF-3, IF-4, and IF-5 become redundant. Requirements IF-1, IF-2, and IF-3 remain relevant although the maturity of the features associated with there requirements are prototypes and have the potential to be 'fleshed-out', again the details are discussed in §3.4.2.

3.4.2 Discussion

The initial idea for the interface was to create a small web API, which would receive AJAX requests from the front-end containing a piece of SML code, run this code through the Skalpel back end and return the generated output. This idea was somewhat ambitious and also posed a lot of technical problems. The main motivator for this approach was that it would eliminate the requirement for a user to install Skalpel on their machine. Due to the changes to the build

process described in §3.3, the installation process should be less complicated for the end-user. This removed some of the motivation for this objective. Creating such an interface also poses many security concerns. The analysis performed by Skalpel is computationally expensive and it is not difficult to craft pieces of SML code that maximise the the computational cost of analysis. This opens up an attack vector for a denial of service through over consumption of resources available to the infrastructure hosting the interface. It would require careful design and monitoring to ensure that this is mitigated. Further to this, the original idea also leads to the need to maintain another codebase, keeping it up to date with potential future changes made to the back-end and the front-end. Taking into consideration all of these factors, and on the advice of the project supervisor, it was decided to not move forward with this original idea. The benefit that such an interface would provide is still significant. Providing simpler access to a very powerful learning and debugging tool merits further investigation and work. There are a number of courses at Heriot-Watt University — F28PL, F20FA, F20FB — that either *require* the use of SML or provide assignments suitable to implementation in SML. Minimising the effort required to access the analysis provided by Skalpel would likely prove beneficial to students, lecturers, and to Skalpel as an educational tool.

Instead of this, a small bash script has been created (as part of the front-end project) to automate the process of running Skalpel and then open the user's browser so that they may access the visualisation. This eases the process somewhat, but is not a comprehensive 'linking' between the front and back ends and could be improved upon. The details of this script are discussed in §3.2.

3.4.3 Conclusion

The initial idea for creating a communication channel between the front and back ends was very ambitious and and required a fully-fledged web API being created. While tools exist that can speed up this process considerably and the actual amount of development required to get a somewhat working version would likely not be very high, the resources required to create a longer-term solution that satisfies performance, usability, and security requirements were too great. A lightweight solution that provides a convenient manner of running Skalpel on a source file and opening the visualisation interface in a web browser was created. This interface constitutes minimal functionality and improvements could be made to further streamline the interaction between front and back end.The potential for creating a more comprehensive

interface is discussed further in §5.1.

4 Evaluation

In order to evaluate the effectiveness of the newly created representation of Skalpel type-error slices a user evaluation study was conducted. The purpose of this study was to investigate whether the newly created visualisation was more effective in conveying the type error present within a given program in comparison to the currently available representation Skalpel provides. The main difference between these representations (besides one being command-line-based and the other browser-based) are the new links that are rendered in the visualisation developed as part of this project.

4.1 Participants

Participants were recruited from within the Department of Computer Science at Heriot-Watt University. This included students, researchers, and lecturers. Participants had a variety of experience with statically type-inferred functional programming languages and a varying degree of knowledge of type inference systems. Recruitment was done via e-mail with the help of the Joe Wells, the project supervisor.

In total 11 participants took part in the evaluation. They were split into two groups: A and B.

4.2 Setup & Procedure

Before taking part in the evaluation participants were asked to give their informed consent by acknowledging the information set out in a copy of the document shown in Appendix A. Participants were also asked to complete a pre-evaluation questionnaire (Shown in Appendix B) to gauge their level of experience with SML-like languages, type systems, and visualisations.

Participants in Group A were shown the original error report as generated by Skalpel for the piece of code shown in Figure 27 (Shown in Figure 35) first. They were then shown the visualisation generated by dissect for the piece of code shown in Figure 37 (Shown in Figure 33). Participants in Group B were first shown the visualisation generated by dissect for the example presented in Figure 27 (Shown in Figure 32). Thereafter, they were shown the error report generated by Skalpel for the code presented in Figure 33 (Shown in Figure 36). This was done in order to try and minimise the order effects that may be present. This was also the reason why two different pieces of code were used during this evaluation. The data collected from the

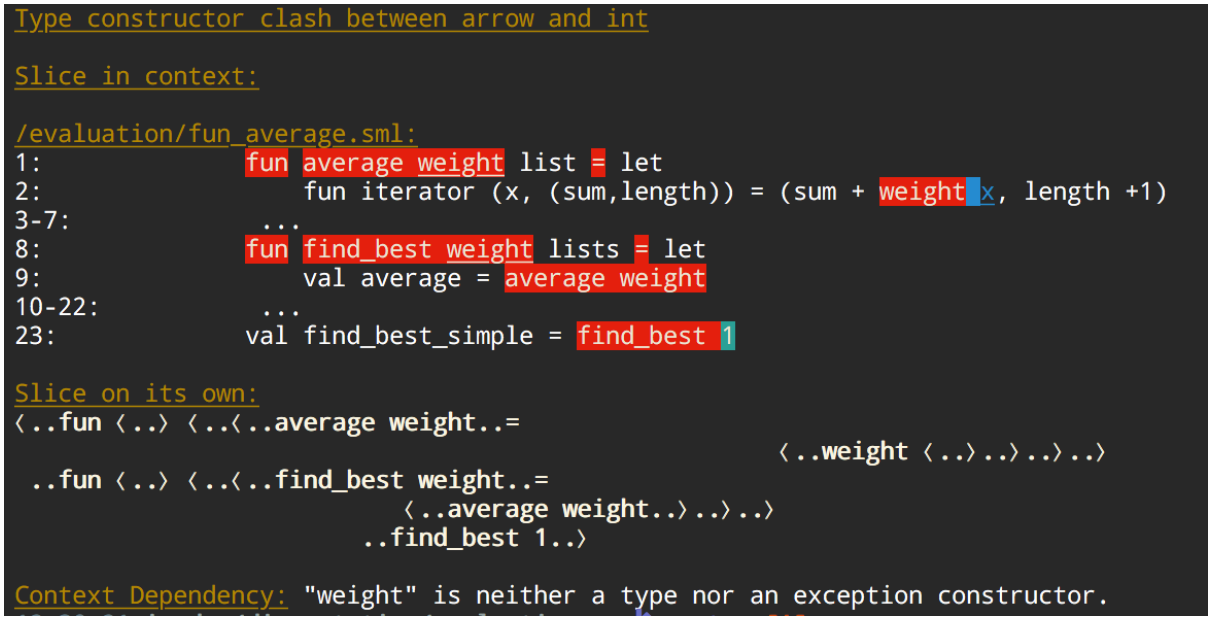

Figure 35: Skalpel Error Report

the second representation evaluated by a participant would not be usable if the same piece of code was used. This is because the participant would have already have worked out the type error which the code contained as part of the evaluation of the first representation they were shown — this would invalidate the data collected.

Participants were asked to complete the questionnaire shown in Appendix C after they felt that they understood the type error present in the present program. This questionnaire was based on the Usefulness, Satisfaction, Ease of Use (USE) questionnaire developed by Lund [25].

As well as completing the questionnaire participants were invited to add any comments, observation, or suggestions that they may have regarding both representations.

4.3 Results

Table 5 presents the results of the evaluation. Each possible answer to a question asked in the evaluation questionnaire was encoded as a number from 1 to 7 (1 being strongly disagree, 7 strongly agree). This table shows the average answer for each question for each representation which participants were shown (the dissect representation and the Skalpel representation). The fully encoded set of data can be found in Appendix D. The rightmost column shows the difference in the average answer between the dissect and Skalpel representation. A positive difference indicates that dissect scored better, while a negative indicated that the Skalpel representation was preferred. From this data we can see that in total the dissect representation was more

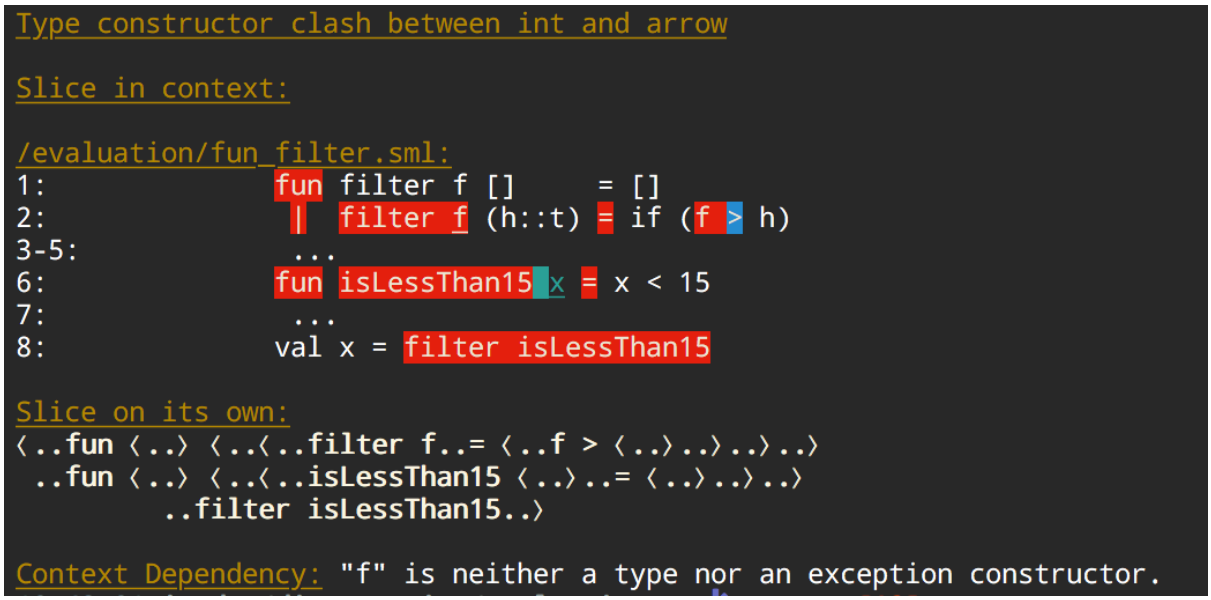

Figure 36: Skalpel Error Report

```
fun filter f []     = []
 |  filter f (h::t) = if (f > h)
                      then h::(filter f t)
                      else filter f t

fun isLessThan15 x = x < 15

val x = filter isLessThan15
```

Figure 37: SML code containing a type error

Question	Averages		
	Dissect	Skalpel	+/-
1	5.727	5.6364	0.0909
2	5.900	5.4545	0.4455
3	6.091	5.9091	0.1818
4	5.636	6.0000	-0.3636
5	5.900	5.7273	0.1727
6	5.818	5.0000	0.8182
7	5.000	3.8182	1.1818
8	5.727	5.0909	0.6364
9	5.636	5.2727	0.3636
10	5.727	4.8182	0.9091
11	5.200	4.4000	0.8000
12	4.909	4.0909	0.8182
13	6.100	5.7273	0.3727
14	5.545	5.4545	0.0909
15	5.182	5.1818	0.0000
16	5.636	5.0909	0.5455
17	5.091	4.7000	0.3909
18	4.455	4.1818	0.2727
19	5.818	5.5455	0.2727
20	5.727	5.5455	0.1818
21	5.455	4.8182	0.6364
AVG	5.537	5.1173	0.4199

Table 5: Average of Results of Evaluation Questions

successful overall by 0.492 points. Dissect was seen to perform better on all of the evaluation questions apart from question 4, which asked "It makes the things I want to accomplish easier to get done". Other questions related to ease of use and usefulness (such as questions 1, 2, 3, 5, 11, and 12) all show the dissect representation performing better. However, due to the small number of participants in the study it is difficult to reach any concrete conclusions from this data. This limitation and others, as well as other comments and observations made during the evaluation are discussed in §4.4.

4.4 Discussion

The results found by this study are subject to many limitations. The small number of participants means that the overall results can be easily swayed by one or two participants having

strong views on either representation being shown to them, making it difficult to generalise the findings. Participants also had a wide variety of experience level with SML (or similar languages) and type theory. From researchers with upward of 10 years experience that have written numerous papers on the subject of types to undergraduate students that have had limited exposure to functional programming usually only through university course. This variety of experiences brings with it a variety of expectations and biases. In general, observations made throughout the evaluation sessions found that participants with more experience programming with functional languages found the presented error reports to be too descriptive. These participants were used to being given succinct error messages pointing to a single point in a given program. They had become used to the state of error messages and had adapted their development workflows and debugging methods around this — they acknowledged that error reports can sometimes be less than optimal, but accept this as the way things are. Participants with less experience seemed to be more receptive to error messages being presented differently. This is most likely because they are not yet accustomed to any concrete representation. This means that they do not necessarily have any strong expectations as to what information should be present within an error report.

For the majority of the participants this evaluation was their first exposure to type errors being presented in slices as opposed to being presented with a single error location. This also caused some confusion. Participants were not able to utilise the information which they were provided to the fullest extent as they were not fully aware of what this information meant. Participants in general found it difficult to ignore those parts of the program that were not highlighted (and therefore were not in the slice). This may be explained by the fact that the evaluation was their first look at a new piece of source code. It seems natural to want to have some sort of context as to what the program is trying to achieve. It may be the case that if participants had a stronger understanding of the source code (i.e. if they had written it) then they would be more likely or willing to debug type errors focusing solely on the part of the program that are part of the error slice. This however would require a larger study where participants are asked to use Skalpel and dissect over a longer period of time.

Some participants commented that the error messages themselves were not detailed enough. Multiple participants mentioned that it would be helpful to output more detailed type information (e.g. instead of report a clash between the types ‘arrow’ and ‘int’, the ‘arrow’ type should be more specific). It was also mentioned that it may be beneficial to attempt to create

'human-like' error messages that would give more descriptive error messages. This comment was also made of the links drawn by dissect, it was suggested that an improvement would be to annotate these links with type information.

Another common criticism related to the lack of explanation given within the error reports with regards to what the different highlighting colours actually meant. Furthermore, it was mentioned that the majority of the sections were coloured in red which draws undue attention to parts of the slice that are auxiliary — impacting the efficiency of the debugging process.

Overall, participants seemed to react positively to the new representation. There were many valid criticism of both Skalpel and dissect which should be addressed. Dissect remains a very early prototype and participants mentioned several missing features. Many of the more experienced functional programmers were somewhat untrusting of being told that they only need to look at highlighted parts of the code to understand the type error — this may just be a reflex they have developed over time due to the poor error messages that they are used to. In general, although seemingly a strange idea in relation to functional programming[22] participants appeared to support a move towards creating more user-friendly error reports — whether this presents itself in terms of a browser-based visualisation, as improved command line tooling, or as a mix of both in the form of a integrated development environment remains to be determined. What is certain is that there remains a lot of potential for improvement.

[22]Perhaps they have given up hope on any chance of improvement, a sort of type-error reporting Stockholm Syndrome.

5 Conclusion

The aim of this project was to investigate how more information could be extracted from the analysis that Skalpel performs with the goal of then using this newly available information to create a new visual representation of type-error slices. This new representation would then ideally help programmers in their development and debugging process. Exploratory work was carried out to investigate potential ways in which information could be passed on from the analysis step to the reporting step. As as result of this a new pretty-printing function was implemented that should enable for easier analysis of constraint environments. A new AST traversal function was implemented that finds bindings and accessor pairs of identifiers that are relevant to a type-error slice. A new output format was designed and implemented that enables outputting highlighting information as well as the new identifier pairs. A new visualisation framework that is capable of visualising the information which Skalpel provides. A number of miscellaneous improvements were made to Skalpel in order to improve the overall 'health' of the project. This involved updating Skalpel's build process, integrating a CI server to provide automatic building and testing of the project, improvements to documentation, updating dependencies, and numerous improvements to source code.

5.1 Future Work

The work carried out as part of this project is by no means final. There is *a lot* of room for improvement to the features which this project has worked towards implementing. Throughout the course of this project — as a result of the research undertaken and numerous discussions with Joe Wells — a multitude of ideas and potential improvements to Skalpel and the overall type error reporting process have been thought of. What follows is a list of some of these ideas which may be able to form the basis for future projects or research.

A New "Manifesto for Good Type Error Reporting". Yang et al. [46] developed a manifesto almost two decades ago. However, it has never really been analysed or critiqued in depth. It would be interesting to analyse the work on type error reporting that has been carried out since the original manifesto was created and updating the manifesto to reflect what the type-error reporting community has found to be the most important factors in creating good reports (if any have changed).

Drawing Paths of Constraints Flowing Through Program Points. Skalpel uses a

constraint based approach for type checking. During Skalpel's analysis it will find many paths connecting the types associated to program point (Similarly to what is shown in Figure 28). Skalpel does not make this — potentially very valuable — type information it gathers during this analysis step available to the the slicer/reporter. It may be possible to use this extra information to visualise the visual path of unsolvable constraints flowing thorough program points.

Generating Human-Like Error Explanations. While Skalpel improves on the reported location of a type error, it does not make any significant changes to the generated error-message. In fact, some error messages (specifically related to function types) are less informative than those returned by conventional type-error reports. Work has already been carried out to create better textual explanations of type errors. It may be beneficial to combine these improvements into Skalpel. Work carried out by Yang, Michaelson, & Trinder [18] may prove to be a good basis for this.

Suggesting Potential Changes That Lead to Type-Correctness. Systems such as SEMINAL developed by Lerner et al. [24] automatically suggest changes to a source program that render it type correct. Leveraging the slices returned by Skalpel within such an approach could greatly increase efficiency of such an analysis. Using slice information would allow for the analysis of a wider section of a program (Skalpel type error slices contain *all* parts of the program contributing to a particular error) and therefore suggesting more potential fixes.

Developing a Edge-Layout Algorithm for dissect. The current edge layout process implemented is simplistic at best. For large slices containing many bindings and accessings of identifiers the edges can quickly become messy and confusing, contrary to the goal of drawing them in the first place. Developing an algorithm that tries to reduce the overlapping/crossing of edges over other edges or parts of the source code would be beneficial.

More Advanced Analysis of AST with Slice Information. The current AST traversal searched somewhat blindly for all occurrences of identifiers that are used and defined within a slice. It may be possible to do some more advanced traversal and information gathering under certain circumstances. One potential area for more analysis could be in taking more care during function applications. A more intelligent traversal algorithm may be able to collect and link together where *parameters* are used and defined. Type errors often occur due to improper application of functions to parameters of incompatible types.

Supporting More of SML. This project implements only a subset of SML. There are many

part of the language which still need to be implemented. Implementing support for `type` and `datatype` declaration is probably of highest priority as these are often used by programmers. Adding support for SML's module system would involve a more significant amount of work but would allow for larger products to make use of the newly created visualisation.

Ranking Type Error Slices. Skalpel may return more than one error slice and will present all of them in its error report, this may be overwhelming for the user. It may be interesting to develop some metrics that could be used to try and find the error slice that is most useful in explaining an error. These metrics could then be used to rank each error slice. Decisions could then be made regarding the order in which these are presented to the user or how many are presented to the user. Pavlinovic et al. [32] (Discussed in §2.6) developed a method for assigning weights to constraints to evaluate the 'usefulness' of a slice — a similar approach could be taken for Skalpel.

Investigating Highlighting. One of the ways in which Skalpel represents type error slices is by highlighting those parts of the program that are present within a given slice. Skalpel implements a colour scheme and a system of highlighting or underlining. This system is however somewhat poorly documented[23] and the error reports do not contain any sort of legend to inform the user of what the different colours or underlinings are trying to depict. Improving this situation would allow for much clearer communication of what Skalpel has found to the user.

Promoting Skalpel As An Education Tool. Skalpel has been developed at Heriot-Watt University for many years and is built as a tool to help ease understanding of type-errors. It is a missed opportunity for Skalpel not to be made available — or even perhaps formally introduced — to students studying Standard ML in courses such as F20PL, F20FA, and F20FB. Doing so would likely prove beneficial to students, lecturers, and Skalpel itself. Skalpel could be used by lecturers to give students more insight into how SML's underlying type system works. Students would be able to receive type errors that are more informative than what is currently available by default — potentially reducing frustration and in turn increasing levels of satisfaction and perhaps even level of achievement. Skalpel as a project would benefit by having a recurring group of entry-level users that could be used to conduct evaluations to continually improve the accessibility and level of understanding which Skalpel brings.

Improving The Interface Between Back & Front End. As described in §3.2.2 and

[23] Some fairly cryptic source-code level comments exist in `analysis-engines/standard-ml/error/ExtReg.sml`.

§3.4.2 the interface between the Skalpel back-end and dissect front-end is rather rudimentary at the moment. The initial vision for the interface was that of a web-service which would serve the dissect front-end. The dissect front-end would then implement functionality allowing a user to paste their SML code and send it to be analysed (via the interface). As such, the interface would need to make available an endpoint where SML code could be sent to for analysis. This endpoint would then respond with the result Skalpel analysing the given piece of code. Such a system presents a number of issues with regards to hosting, security, and usability. One worry is also whether this is worth it or if it is better to implement similar functionality though the development of an integrated development environment that offers a visualisation similar to that provided by dissect as part of it.

Improving Skalpel's Packaging And Deployment Process. Skalpel implements some methods for automatically creating packages for some popular GNU/Linux distributions as well as for macOS. These tools may be out of date. It would be beneficial to investigate the state of this tooling and if required update it. Adding an automatic packaging and deployment step to the continuous integration builds would be of great benefit to the project. Making up-to-date pre-built development versions of Skalpel available to users may encourage them to use Skalpel more often and to become involved in the development process. Making stable versions of Skalpel available for the most popular operating systems in an easily installable format would perhaps reduce barrier of entry to using Skalpel.

Analysing The Needs Of Different Skalpel User Groups. Skalpel has the potential to be used by a variety of different user groups. Students, teachers, researchers, and programmers may all potentially benefit from using Skalpel. Each of these groups however have different needs and expectation. It may be interesting to investigate what these needs are and how feasible they are to implement Skalpel.

5.2 Reflection

I think that the project as a whole was successful. Many contributions have been made to Skalpel as a project that should make the life of future contributors a bit easier. It is somewhat disappointing that the timescale of this project did not allow for more work to be completed — it would have been nice to continue to pursue the 'constraint-based', to implement some of the changes suggested during the evaluation, and to run a longer evaluation study. This project has allowed me to delve into an area of computer science that I feel a lot of undergraduate student

don't get the opportunity to investigate. Working with a large codebase and trying to extend it proved to be a challenge, but it is a skill that is important to learn as a software engineer — not everything is sunshine and rainbows, and not everything can be fixed.

There remains a lot more work to be done in relation to Skalpel, dissect, and type error reporting in general. I hope that the work described in this document — even if only the literature review — might be of help to any student or researcher interested in this subject.

6 References

[1] Doxygen. `http://www.stack.nl/~dimitri/doxygen/`. Accessed: 2018-04-10.

[2] Microsoft MakeCode. `https://makecode.com/`. Accessed: 2018-03-30.

[3] MIT App Inventor. `http://appinventor.mit.edu`. Accessed: 2018-03-30.

[4] Scratch. `http://scratch.mit.edu`. Accessed: 2018-03-30.

[5] TravisCI. `https://travis-ci.org`. Accessed: 2018-04-10.

[6] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[7] Margaret M. Burnett. Types and type inference in a visual programming language. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 238–243. IEEE Computer Society Press, 1993.

[8] Olaf Chitil, Frank Huch, and Axel Simon. Typeview: A toll for understanding type errors. In M. Mohnen and P. Koopman, editors, *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 63–69, 2008.

[9] Luís Damas. *Type assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[10] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[11] G. Frege. *Begriffsschrift, eine der Arithmetischen Nachgebildete Formelssprache des Reinen Denkens*. Halle, Germany, 1879.

[12] Michael J. C. Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1979.

[13] A. J. G. Gray and Peter King. *4th Year Dissertation 2017/2018: Calendar, guidelines, and other snippets*. Department of Computer Science, School of Mathematical and Computer Sciences, Heriot-Watt University, September 2017.

[14] Christian Haack and J.B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[15] Bastiaan Heeren. *Top quality type error Messages*. PhD thesis, Universiteit Utrecht, 2005.

[16] Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *POPL*, 1986.

[17] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, 1993.

[18] Yang Jun, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[19] Yang Jung and Greg Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, 2000.

[20] Joel Kelso. A visual representation for functional programs. Research report, Murdoch University, Perth, Western Australia, 1994.

[21] Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack. A visual language for keyboardless programming. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, Missouri, March 1986.

[22] Michael D. Lee and Douglas Vickers. Psychological approaches to data visualisation. Research Report AR-010-587, DSTO Electronics and Surveillance Research Laboratory, Salisbury, South Australia, July 1998.

[23] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, July 1998.

[24] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *PLDI*, 2007.

[25] Arnold M Lund. Measuring usability with the use questionnaire. *Usability Interface*, 8(2):3–6, 2001.

[26] Bruce J. McAdam. *On the Unification of Substitutions in Type Inference*, pages 137–152. Springer, Berlin, Heidelberg, 1999.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:248–375, 1978.

[28] M. A. Najork and E. Golin. Enhancing show-and-tell with a polymorphic type system and higher-order functions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 215–220, Oct 1990.

[29] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. *ACM SIGPLAN Notices*, 38(9):15–26, 2003.

[30] F G Pagan. A graphical fp language. *SIGPLAN Notices*, 22(3):21–39, March 1987.

[31] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 2nd edition, 1996.

[32] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Software Engineering & Management*, 2014.

[33] John Pirie. *New developments to Skalpel: A type error slicing method for explaining errors in type and effect systems*. PhD thesis, Heriot-Watt University, 2014.

[34] Jörg Poswig, Guido Vrankar, and Claudio Morara. VisaVis: a higher-order functional visual programming language. *Journal of Visual Languanges and Computing*, 5(1):83–111, March 1994.

[35] François Pottier and Didier Rémy. The essence of ML type inference. In Jan Fagerberg, David C. Mowery, and Richard R. Nelson, editors, *Adavanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[36] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. Skalpel: A constraint-based type error slicer for Standard ML. *Journal of Symbolic Computation*, 80:164–208, 2017.

[37] Thomas Schilling. Constraint-free type error slicing. In *International Symposium on Trends in Functional Programming*, pages 1–16. Springer, 2011.

[38] Eric L. Seidel and Ranjit Jhala. A collection of novice interactions with the OCaml Top-Level system, June 2017.

[39] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: Localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, October 2017.

[40] Ryan Stansifer. *ML Primer*. Prentice-Hall, Inc., 1992.

[41] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10:5–55, 1997.

[42] Mitchell Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 38–43, New York, NY, USA, 1986. ACM.

[43] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10:352–357, 1981.

[44] J. B. Wells. Explaining concepts in compositional type-based program analysis: Principality, intersection types, expansion, etc. `http://www.macs.hw.ac.uk/~jbw/slides/compositional-type-based-analysis.pdf.gz`.

[45] Jun Yang. Explaining type errors by finding the source of a type conflict. In *Scottish Functional Programming Workshop*, 1999.

[46] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting, 2000.

[47] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *POPL*, 2014.

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

Appendices

Appendix A Informed Consent Form

Human Subject Consent Form: "Towards Graphically Representing Skalpel Type-Error Slices"

Participation No:

The objective of the experiment is to evaluate the effectiveness of a new visual representation of Skalpel Type-Error slices.

The experiment will involve identifying a type error in two pieces of Standard ML code.

You will be required to complete a questionnaire related to the way in which the Standard ML code and the type error which it contains was represented.

All results will be held in strict confidence, ensuring the privacy of all participants. There will be no record kept that would allow your results to be tracked back to you. All data will also be held securely in a password protected computer system (for electronic data) or a locked office (for paper based data).

A feedback sheet (containing summaries of the data mentioned above) will be sent to all participants who request it, after the data has been analysed. Your participation in this experiment will have no effect on your marks for any subject at this, or any other university.

You may withdraw from the experiment at any time without prejudice, and any data already recorded will be deleted.

Project Student:	**Project Supervisor:**
Christian Gregg	**Joe Wells**
E-mail: cg23@hw.ac.uk	**Email: jbw@hw.ac.uk**

Name:
Signed:
Date:
Email Address:

I would like to receive information on the results of this study: Yes ☐ **No** ☐

Appendix B Pre-Evaluation Questionnaire

Pre-Evaluation Questionnaire

Participant #:___

1. How much experience have you had with Standard ML (or similar) languages?

   < 1 year 1 year 1+ years 5+ years 10+ years

2. How would you rate your level of expertise with regards to type systems?

   Basic Novice Intermediate Advanced Expert

3. How would you rate your level of expertise with regards to type errors in particular?

   Basic Novice Intermediate Advanced Expert

4. How would you rate your level of expertise with regards to visualisations of data?

   Basic Novice Intermediate Advanced Expert

5. How would you rate your level of expertise with regards to visualisations of programming languages?

Basic Novice Intermediate Advanced Expert

6. How would you rate your level of expertise with regards to visualisations of type systems?

Basic Novice Intermediate Advanced Expert

Appendix C Evaluation Questionnaire

Evaluation Questionnaire

Participant #:____

1. It helps me be more effective.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

2. It helps me be more productive.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

3. It is useful.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

4. It makes the things I want to accomplish easier to get done.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

5. It saves me time when I use it.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

6. It meets my needs.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

7. It does everything I would expect it to do.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

8. It is easy to use.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

9. It is simple to use.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

10. It is user friendly.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

11. It requires the fewest steps possible to accomplish what I want to do with it.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

12. Using it is effortless.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

13. I don’t notice any inconsistencies when I use it.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

14. Both occasional and regular users would like it.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

15. I am satisfied with it.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

16. I would recommend it to a friend.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

17. It works the way I want it to work.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

18. It is wonderful.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

19. It helps my understanding.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

20. It is easy to learn to use.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

21. It is clear and understandable.							
Strongly Disagree	Disagree	Slightly Disagree	Neutral	Slightly Agree	Agree	Strongly Agree	N/A

Any other comments, observations, suggestions:

Appendix D Evaluation Results

Participant (Group)	Pre Evaluation Questions					
	Q1	Q2	Q3	Q4	Q5	Q6
1 (A)	5	4	1	3	1	1
2 (B)	2	2	2	3	2	2
3 (A)	4	5	3	3	3	4
4 (B)	4	3	3	2	3	1
5 (A)	5	5	5	4	5	5
6 (B)	2	3	2	3	2	2
7 (A)	5	3	3	5	4	2
8 (B)	3	3	3	2	2	2
9 (A)	4	4	3	2	2	2
10 (B)	2	3	4	4	1	1
11 (A)	3	2	3	3	4	3

Table D1: Results of Pre-Evaluation Questions

Question	Dissect / Participant										
	1	2	3	4	5	6	7	8	9	10	11
1	7	7	6	2	6	6	6	6	6	5	6
2	7	7	6	3	6	7	6	6	4		7
3	7	7	6	3	6	7	6	6	6	6	7
4	7	7	5	3	6	6	6	5	6	5	6
5	7	7	5	5	6	7	6	6	4		6
6	7	7	6	5	6	7	5	6	4	5	6
7	5	7	6	3	3	7	5	6	3	3	7
8	7	7	6	5	6	6	6	5	6	4	5
9	7	6	6	6	5	6	6	6	6	3	5
10	7	7	6	6	5	5	6	6	6	3	6
11	7	7	5	3	5	6	2	6	6		5
12	7	7	4	6	3	5	3	6	4	3	6
13	7	7	6	5	6	7	4	6	6		7
14	7	7	6	3	5	7	5	6	4	4	7
15	6	7	6	5	4	6	5	5	4	3	6
16	7	7	6	4	5	7	5	6	4	5	6
17	6	7	6	3	3	7	4	6	3	4	7
18	5	7	4	2	3	6	5	6	4	1	6
19	6	7	5	5	5	7	6	4	6	6	7
20	7	7	5	5	6	6	6	4	6	5	6
21	7	7	6	4	6	6	5	6	5	1	7
AVG	6.7	7.0	5.6	4.1	5.0	6.4	5.1	5.7	4.9	3.9	6.2

Table D2: Results of Evaluation Questions for dissect

Question	Skalpel / Participant										
	1	2	3	4	5	6	7	8	9	10	11
1	7	7	4	5	6	6	5	5	6	5	6
2	7	7	4	5	6	6	5	5	4	5	6
3	7	7	5	5	6	6	5	6	6	5	7
4	7	7	5	5	6	7	5	6	6		6
5	7	7	5	4	6	6	5	6	5	5	7
6	7	7	4	4	6	6	4	4	4	4	5
7	4	7	4	3	3	5	2	2	4	3	5
8	7	7	3	5	4	6	3	3	6	5	7
9	7	7	3	5	4	7	4	3	6	5	7
10	7	7	6	3	3	7	3	2	6	3	6
11	6	7	4	3	3	6	2	4	3		6
12	5	7	3	3	3	6	2	2	5	3	6
13	7	7	7	4	6	7	3	6	4	5	7
14	7	7	7	3	3	7	4	4	6	5	7
15	6	7	6	4	5	6	2	6	5	4	6
16	6	7	6	3	5	7	2	5	4	5	6
17	6	7	6	3	3	6	3	4	3		6
18	5	7	4	2	3	6	1	6	4	1	7
19	7	7	4	4	6	6	3	7	5	5	7
20	7	7	4	3	6	7	5	6	4	5	7
21	7	7	5	3	3	6	3	5	6	2	6
AVG	6.5	7.0	4.7	3.8	4.6	6.3	3.4	4.6	4.9	4.2	6.3

Table D3: Results of Evaluation Questions for Skalpel