

**Shaun Hall**

**Searching for the Source  
of Caml Type-Errors**

Computer Science Tripos

Churchill College

May 7, 2011



# Proforma

Name: **Shaun Anthony Hall**  
College: **Churchill College**  
Project Title: **Searching for the Source of Caml Type-Errors**  
Examination: **Part II Computer Science Tripos**  
Year **2011**  
Word Count: **11,983<sup>1</sup>**  
Project Originator: Shaun Hall  
Supervisor: John Wickerson

## Original Aims of the Project

To modify the OCaml compiler to produce type-error messages specifying locations that are closer, on average, to the location of the programmer type-error than those produced by the original compiler (when the program does not contain multiple, independent type-errors). The modified compiler should take no more than a few seconds on a modern machine to execute.

## Work Completed

The OCaml compiler was modified to include a module, `TELE`, which is executed when a type-error is discovered by the compiler, initiating a search-based procedure similar to `SEMINAL` [6] that attempts to find modifications to the program that cause it to become well-typed. `TELE` achieves a closer distance to the programmer type-error than `SEMINAL` and the OCaml compiler, makes fewer calls to the type-checker and executes quicker than `SEMINAL`.

## Special Difficulties

None.

---

<sup>1</sup>Calculated using TeXcount from <http://app.uio.no/ifi/texcount/index.html>

## **Declaration**

I, Shaun Anthony Hall of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Problem and its Solution . . . . .	1
1.2	Type-Checking and Type-Error Reporting . . . . .	1
<b>2</b>	<b>Preparation</b>	<b>5</b>
2.1	Technical Background . . . . .	5
2.1.1	An Example of a Typeability Implementation . . . . .	5
2.1.2	Type Checking Failure and Error Reporting . . . . .	6
2.1.3	Type Checking and Error Reporting in OCaml . . . . .	8
2.2	Precise Objectives . . . . .	9
2.2.1	Evaluation Strategy . . . . .	9
2.2.2	Requirements Analysis . . . . .	9
2.3	Practical Preparation . . . . .	10
2.4	Existing Work . . . . .	10
2.5	System Design . . . . .	11
2.5.1	Preliminary Design . . . . .	11
2.5.2	Final Design . . . . .	13
2.5.3	Other Benefits . . . . .	13
2.6	Design of the Searcher . . . . .	15
2.6.1	Precision and Recall of Interesting Nodes . . . . .	15
2.7	Design of the Enumerator . . . . .	17
2.8	Performance Analysis . . . . .	18
<b>3</b>	<b>Implementation</b>	<b>19</b>
3.1	Modification to the OCaml Compiler . . . . .	19
3.1.1	Compiler Pipeline . . . . .	19
3.1.2	OCaml Parse Tree Structure . . . . .	20
3.2	Searcher Implementation . . . . .	21
3.2.1	Maintaining Context and Recurring on AST Nodes . . . . .	23
3.2.2	Determining Interestingness . . . . .	24

3.2.3	Finding the type of Generic Replacements . . . . .	27
3.3	Enumerator Implementation . . . . .	28
3.3.1	Rearranging Function Arguments . . . . .	28
3.3.2	Tree Transformations . . . . .	28
3.3.3	Match Reassociation . . . . .	30
3.3.4	Collecting Identifiers for Identifier Enumeration . . . . .	31
3.4	Ranker Implementation . . . . .	32
3.5	Producing the Type-Error Message . . . . .	33
3.6	Evaluation Harness . . . . .	33
3.6.1	Calculating Distance Statistics . . . . .	33
3.7	Unit Tests . . . . .	35
<b>4</b>	<b>Evaluation</b>	<b>37</b>
4.1	Preprocessing the Evaluation Corpus . . . . .	37
4.2	Quantitative Evaluation . . . . .	39
4.2.1	Type-Error Location Comparison with OCaml . . . . .	39
4.2.2	Type-Error Location Comparison with SEMINAL . . . . .	42
4.2.3	Quality of Alternative Suggestions . . . . .	43
4.2.4	Timing Analysis . . . . .	45
4.3	Qualitative Evaluation . . . . .	47
4.3.1	Usefulness of TELE's Output . . . . .	47
4.3.2	Unquantified Benefits . . . . .	48
4.3.3	Generalising to Programs Outside the Corpus . . . . .	48
<b>5</b>	<b>Conclusion</b>	<b>51</b>
5.1	Was the Project a Success? . . . . .	51
5.2	Further Work . . . . .	52
5.3	Summary . . . . .	52
	<b>Bibliography</b>	<b>52</b>
	<b>A Project Proposal</b>	<b>55</b>

# List of Figures

1.1	An ill-typed program . . . . .	2
2.1	An OCaml code sample with a type-error message requiring the programmer to work out why a certain type was expected . . . . .	8
2.2	An OCaml code sample with a value definition that is inconsistent with its usages . . . . .	8
2.3	The OCaml compiler pipeline . . . . .	12
2.4	An initial architecture design . . . . .	12
2.5	A refined architecture diagram . . . . .	13
3.1	The modification to the OCaml compiler pipeline required for TELE . . . . .	20
3.2	The structure of the searcher function . . . . .	23
3.3	The structure of the function to determine interestingness . . . . .	25
3.4	The first syntax tree transformation enumeration . . . . .	29
3.5	The second syntax tree transformation enumeration . . . . .	30
3.6	The structure of the shell script to produce quantitative evaluation data. . . . .	34
4.1	An illustration of single and multiple type-errors . . . . .	38
4.2	Graphs comparing the distance of TELE, OCaml and SEMINAL to programmer type-errors in the unseen part of the corpus . . . . .	40
4.3	A comparison of the distance to the programmer error from TELE's top two suggestions . . . . .	44
4.4	Graph comparing the execution time per file with the amount of time spent type-checking . . . . .	45
4.5	Graphs comparing the time for each call to the type-checker and the distribution of the number of type-checker calls . . . . .	50

# List of Tables

2.1	Approaches to determine interestingness of different kinds of OCaml AST nodes . . . . .	15
4.1	Table showing the average number of type-checker calls made by TELE and SEMINAL when the programmer error was inside different kinds of node . . . . .	46

# Acknowledgements

I would like to sincerely thank Benjamin Lerner from the University of Washington for providing a corpus of ill-typed Caml programs collected from students participating in a programming course. Without this, the quantitative evaluation of the project would not have been possible, so the project would not have been implemented. Also, I would like to thank my supervisor, John Wickerson, for providing excellent advice and support throughout.



# Chapter 1

## Introduction

### 1.1 The Problem and its Solution

Whilst type inference reduces the need for the programmer to annotate programs with types, it often leads to inscrutable error messages for ill-typed programs. Typically, the reported location of the error is far from the programmer error that caused type inference to fail. The error message is based on how the type checker works and constructive suggestions can rarely be provided. This increases the time taken to correct type-errors, makes learning the language more difficult and reduces its popularity.

This project combats these issues for the Caml language by using a search based procedure that aims to report the location of the cause of the type-error, as well as one or more suggestions at this location that would cause the program to become well-typed. I have successfully implemented TELE (Type-Error Location Extension); an extension and modification to the OCaml compiler that produces type-error messages that are closer to the programmer error, on average, than those produced by the compiler. Furthermore, TELE produces type-error messages that are closer to the programmer type-error for an unseen corpus of ill-typed programs, makes fewer calls to the type-checker and executes quicker than the system on which this project is based, SEMINAL [6].

### 1.2 Type-Checking and Type-Error Reporting

Formal type systems can be used to detect errors in programs, either statically (at compile-time) or dynamically (at run-time), and prove properties of well-typed programs, such as being free from certain run-time errors. Other uses of type systems are to improve efficiency of executing well-typed programs by removing the need to check compatibility of operands at run-time and generalisation

```
1 let a = "hello,"
2 in let b = 3
3     in b ^ " world"
```

**Figure 1.1:** An ill-typed program

of program constructs through polymorphism. A desirable feature of an implementation is to provide type-error messages that describe why type inference fails when it does, and possibly provide suggestions to the user to correct the type-error.

The OCaml program in Figure 1.1 is ill-typed because the variable `b` is defined as an integer but used as a string, which would cause a run-time error. The OCaml compiler reports a type-error on line 3, because `b` is not a string, so cannot be concatenated with another string. However, the problem could also be reported on line 2, where `b` is defined as an integer, instead of a string. A more accurate description of the problem is that the definition of the variable `b` is inconsistent with its usage. The problem with the OCaml approach is that it always reports the location of the type-error at the point where type inference fails, however, well-typed bindings are often the cause of the type-error, and as discussed in Section 2.1.2, the location where type inference fails can vary depending on the implementation of the formal type system, even if there is only a single type-error in the program.

TELE combats this problem by searching for small syntactic changes to make ill-typed programs become well-typed, and producing type-error messages in the form of suggested modifications to the program. This helps the user understand why their program is ill-typed in a way that is independent of the implementation of the formal type system. The suggestions that TELE reports for the program in Figure 1.1 include replacing the variable `b` on line 3 with `a` and the integer `3` on line 2 with an expression of type `string`.

There are several existing approaches to produce more helpful type-error messages in functional languages [2, 5, 1, 9, 8] which are described in Section 2.4. One problem common to them all is that they have modified the type inference algorithm by adding error generation code, making it harder to maintain, more difficult to ensure correct (because it has less resemblance to the formal specification) and slower to execute for both ill-typed and well-typed programs.

The basis for this work is the SEMINAL tool developed by Lerner et al. [6] where all of the problems above are combated by intercepting the type-error message produced by the OCaml compiler when a program is ill-typed and initi-

ating a search-based procedure that uses the type-checker on candidate modified programs to see if they are well-typed. The project is an extension to the OCaml compiler and is only initiated when type inference fails, so compilation performance of well-typed programs remains the same – this is not necessarily the case for all of the existing approaches. Several different design decisions compared with SEMINAL were made to produce higher quality error messages and reduced speed of operation.



# Chapter 2

## Preparation

In this chapter I describe the unsatisfactory type-error messages produced by OCaml, and functional languages in general, by examining the original typeability implementation for ML [3], as well as describing the relevant theory and design decisions that needed to be understood before development of TELE could take place.

### 2.1 Technical Background

In this section, I describe the technical reasons that cause unsatisfactory type-error messages to be produced by examining Algorithm *W* (the original implementation of the ML type system), then extend the analysis to OCaml's type-checking and error reporting system.

#### 2.1.1 An Example of a Typeability Implementation

A typical type system judgement is a relation between types,  $\tau$ , typing environments,  $\Gamma$  and program phrases,  $e$ , written

$$\Gamma \vdash e : \tau.$$

A program  $M$  is said to be *well-typed*, if for some type,  $\tau$ ,

$$\emptyset \vdash M : \tau$$

and *ill-typed* otherwise. Making this judgement is an instance of the *typeability* problem that asks, given  $\Gamma$  and  $M$ : is there a  $\tau$  such that  $\Gamma \vdash M : \tau$  is derivable using the formal type system?

The implementation of a type system is decoupled from its formal specification, allowing the programmer to compile programs with compilers using different

type system implementations, with the same assurance that the formal type system provides. However, as described below, the production of type-error messages differs widely between compilers, thus requiring the programmer to be familiar with the compiler in order to resolve type-errors.

Consider a subset of the ML typing rules:

$$\text{(TAUT)} \frac{}{\Gamma \vdash x : \sigma} (x : \sigma \text{ IN } \Gamma) \qquad \text{(COMB)} \frac{\Gamma \vdash e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e e' : \tau}$$

Where  $\Gamma$  maps variables to type schemes,  $\sigma = \forall A(\tau)$  where  $A$  ranges over type variables.

Algorithm  $W$  implements the typeability problem by, given  $\Gamma$  and  $e$ , returning  $(S, \tau)$ , where  $\sigma = \forall A(\tau)$  and  $A = ftv(S\tau) - ftv(S\Gamma)$  are the most general type scheme and substitution, such that

$$S\Gamma \vdash e : \sigma$$

---

**Algorithm W.** Adapted and abridged from Damas et al. [3]

---

```

Input:  $(\Gamma, e)$ 
Output:  $(S, \tau)$ 
1: if  $e$  matches  $e_1 e_2$  then                                     /* application */
2:    $(S_1, \tau_1) \leftarrow W(\Gamma, e_1)$ 
3:    $(S_2, \tau_2) \leftarrow W(S_1\Gamma, e_2)$ 
4:    $V \leftarrow mgu(\tau_2 \rightarrow \beta, S_2\tau_1)$                  /* where  $\beta$  is fresh */
5:   return  $(VS_2S_1, V\beta)$ 
6: else if  $e$  matches  $x$  then                                       /* variable */
7:   if  $x \in dom(\Gamma)$  then
8:      $\forall A(\tau) \leftarrow \Gamma(x)$ 
9:     return  $(Id, \tau)$ 
10:  else
11:    FAIL
12:  end
13: else if  $e$  matches ... then
14:   ...
15: end

```

---

### 2.1.2 Type Checking Failure and Error Reporting

In this section, I show that Algorithm  $W$  contains various design decisions that result in type-checking failures at different places in the same ill-typed program

from other implementations of Algorithm  $W$ , even if the program contains a single programmer type-error. This means that the programmer may need to be aware of the compiler implementation details as well as the typing rules of the language in order to debug type-errors.

The applicative case could be modified to type-check the expression on the right before the expression on the left, so this algorithm would fail at a different point in the program than in Algorithm  $W$ , if both expressions contain a type-error:

```
fun x => ((x + (x ^ "hello")) (if x then x else 1))
```

It may also be the case that each expression is typeable on its own, but the other is not after applying the substitution  $S_1$  to its type environment (they are inconsistent expressions):

```
fun x => ((fn y => x+1) (if x then 0 else 1))
```

Again, this causes different implementations of the formal type system to fail at different points in the program. In addition, there is flexibility in the implementation of *mgu* (most general unifier), leading to failures at different places in the type structure for the same ill-typed program. It is clear that different implementations of the same type-system can fail when analysing different parts of the same program, even if they contain a single type-error (e.g. when two expressions are inconsistent).

Type-error reporters based on this algorithm keep track of the location in the source code that corresponds to each call to  $W$ , and report the location of the error as the location in the program corresponding to the most recently called  $W$  that failed [5]. Thus the programmer may be required to know the implementation details in order to fix the type-error quickly. In particular, this type-error reporter always assumes that *usages* of a well-typed binding are the problem when used in a context that is ill-typed.

If type unification fails, the error message will typically give the inferred type of the expression along with the expected type, which cannot be unified together. In Figure 2.1, taken from the corpus of ill-typed programs used to evaluate TELE, the programmer must find out why an argument had expected type `unit` by finding out why the function `loop` has an inferred type of `'a list -> 'b -> 'c -> 'd -> unit -> unit`. A much more satisfactory type-error message would not require the programmer to work out why a certain type was expected, and would instead explain why a certain type was expected, or suggest a modification to either location to make the types unifiable.

```

let rec loop movelist x y dir acc =
  if (movelist=[]) then
    acc
  else
    print_string "foo"
in
List.rev (loop movelist 0.0 0.0 0.0 [(0.0,0.0)])
      ^^^^^^^^^^^^^
(* Error: This expression has type 'a list
   but an expression was expected of type unit *)

```

**Figure 2.1:** An OCaml code sample with a type-error message requiring the programmer to work out why a certain type was expected

### 2.1.3 Type Checking and Error Reporting in OCaml

The OCaml type-error reporter makes broadly the same assumptions as the simple reporter in Section 2.1.2 [4]<sup>1</sup>. The OCaml code in Figure 2.2 shows that the usage of a well-typed value that causes a part of a program to be ill-typed is not always the location of the programmer error. The error shown is the error reported by OCaml and the actual error may be that the value `a` should have been declared as `3` instead of `"3"` on line 1.

```

1 let a = "3"
2   in if(a > 5) then (32 + 400 / (80 * a))
3       else if(a < 5) then -a
4       else 0
5
6 (* Error: The expression on line 2 (5) has type int but
7     an expression was expected of type string *)

```

**Figure 2.2:** An OCaml code sample with a value definition that is inconsistent with its usages

<sup>1</sup>typing/typecore.ml, lines 1025-1642, are an implementation of Algorithm *W*, with unification exceptions thrown on line 1021

## 2.2 Precise Objectives

In this section, I refine the success criteria given in the project proposal by describing the evaluation methods and both quantitative and qualitative requirements of the project.

### 2.2.1 Evaluation Strategy

The University of Washington has provided a corpus of approximately 2000 ill-typed but syntactically valid Caml programs that were collected from eleven students participating in a graduate programming course. The files are time-stamped, which allows programs to be analysed by looking at which part of the program the programmer modified to correct the type-error. Quantitative data will be collected by comparing the AST (abstract syntax tree) node distance (length of the shortest path) between the node that the programmer modified to correct the type-error and the nodes reported by TELE, SEMINAL and the OCaml compiler.

The programmers were using an unmodified version of the OCaml compiler, so if a modification did not correct the type-error (because of a poor OCaml type-error message), the same type-error should not be counted more than once. Otherwise, several positive results for TELE could be counted (if TELE produces a good type-error message) but if OCaml produces a good type-error message and TELE produces a poor type-error message then only a single negative result for TELE would be recorded.

### 2.2.2 Requirements Analysis

From the project proposal, the compiler produced must firstly, on average, produce type-errors with a location closer (with respect to distance between AST nodes) to the programmer error than the existing compiler, for programs that do not contain multiple, independent type-errors. Secondly, the compiler produced must take no more than a few seconds longer on a modern machine than the existing compiler for the majority of ill-typed programs in the corpus. Both criteria must be evaluated on a sample of the corpus that have not been tested with TELE to enable performance conclusions to be extended to ill-typed programs outside the corpus (assuming that the corpus is representative of ill-typed programs).

For good software engineering practice, the extension should be designed with testability in mind to allow monitoring of the behaviour of components, such as regression testing to ensure that previously fixed bugs are not re-introduced.

## 2.3 Practical Preparation

The OCaml compiler pipeline is written in OCaml (and compiled using bootstrapping). As well as providing easy integration with the OCaml compiler, the OCaml language is an ideal high level language for concisely implementing syntax tree manipulation and type-checking algorithms (many of which are easily defined recursively), making the choice of programming language for TELE obvious. I will learn the OCaml language and become familiar with OCaml’s formal type-system to develop the theory in Sections 2.6.1 and 3.2.2.

Unit testing will be implemented using OUnit, to allow monitoring of bugs and behaviour. Since the OCaml compiler source code is designed to be compiled on a UNIX system [4], a Linux machine will be used to implement the extension. I will write a shell script to implement the adding of ill-typed programs in the corpus to an analysis database to gather quantitative data.

I will use the spiral development methodology with iterations every fortnight because this naturally allows new features to be added to working prototypes as required by the timetable. The evaluation and testing phase of every fortnight will ensure that TELE satisfies its requirements.

Daily backups will be made to a subversion server hosted by the PWF, and weekly backups will be made to an external storage device, stored in a different site to the main development machine.

## 2.4 Existing Work

Existing work focusses on improving type-error message quality by modifying the type system. Program slice computation [5] works by separating the generation of type constraints (e.g. requiring an expression on the left of an application to have a function type) from solving them, and tracking the location in the source code that gives rise to each constraint. The constraints have a solution (a type substitution) if and only if the expression is typeable, and if the constraints are inconsistent, the minimal inconsistent subset is used to report precisely the parts of the program that give rise to the inconsistency. This exactly characterises the reason that typeability fails and every implementation of the modified type system will produce the same type-error messages.

Another approach attempts to see if type-checking can succeed modulo linear type isomorphisms [8] (once type inference fails) by modifying Algorithm *W*, replacing type unification with unification modulo isomorphisms (for example, allowing  $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$  to unify with  $((\alpha \times \beta) \rightarrow \gamma)$  by applying the morphism  $\lambda f.\lambda x.\lambda y.f(x, y)$ ). Morphisms that result in a well-typed program can then be

used to transform the syntax of the program to provide constructive suggestions that will help the programmer correct the type-error. However, this method is not sufficiently general to make helpful suggestions for all kinds of type-errors (for example, using a variable in different contexts where the expected type and inferred type are not unifiable modulo linear isomorphism, such as `int` and `bool`).

In both of these implementations, the time to type-check well-typed programs has increased and the correspondence between the modified type system and the formal type system has decreased, so it is harder to maintain and ensure correct.

The approach taken by Lerner et al. [6] combines the positive features of both of these implementations by providing constructive changes and producing error messages that characterise the reason that type-inference fails but does not require any modification to the underlying type-system. The type checking implementation is not required to produce any error message generation code, therefore the type system implementation has a close correspondence to the formal definition and well-typed programs type-check quickly. Instead, the type checker is used as an oracle for a search based procedure (initiated when type-inference fails) that looks for small changes to the program to correct the type-error. Although this seems similar to McAdam [8], the modifications to the program correspond directly to changes that the programmer could make (by changing the syntax), instead of modifying the types which are unable to capture all programmer errors and requires partial evaluation semantics to map the changes back to syntax level modifications.

Although all of these approaches produce higher quality type-error messages than OCaml (which reports failure in the program corresponding to the first point of failure of Algorithm *W*), the last approach is most desirable because it does not require any modification to the underlying type-system. This technique will be analysed to ensure that both success criteria can be fulfilled.

## 2.5 System Design

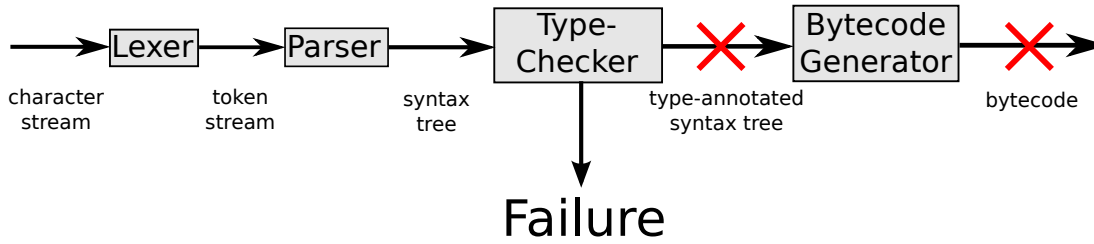
In this section, I present a preliminary design for TELE and then refine it further to ensure that both success criteria can be satisfied. Finally, I compare this design to SEMINAL and describe additional benefits of the design.

### 2.5.1 Preliminary Design

The OCaml bytecode compiler pipeline is given in Figure 2.3 [4]<sup>2</sup>. Since the extension will only search for similar programs when the program is ill-typed, the

---

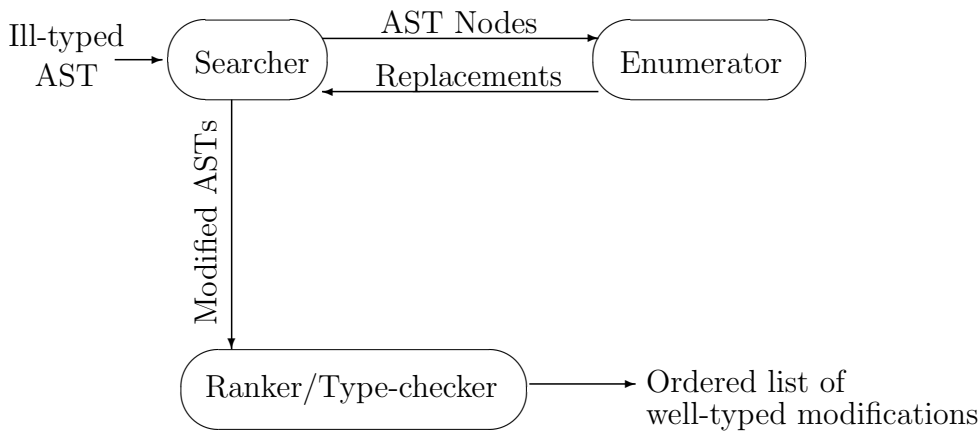
<sup>2</sup>`driver/compile.ml`, lines 127-137



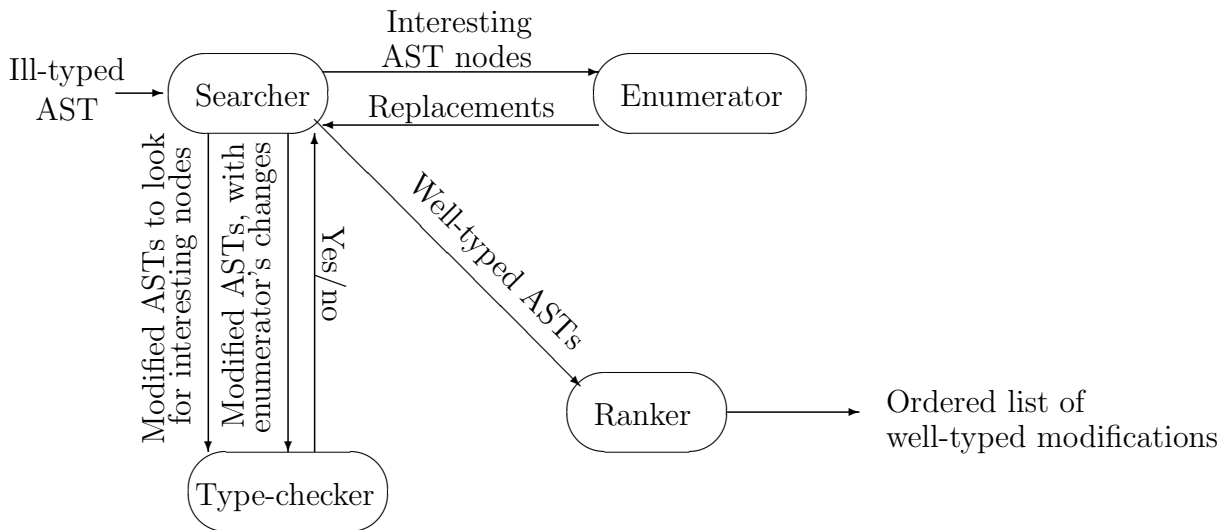
**Figure 2.3:** The OCaml compiler pipeline and the passage of an ill-typed but syntactically valid program through the pipeline

modifications to the program could be made at any stage in the pipeline before type inference, but will be made after the last successful change, parsing, so there is no possibility of causing an earlier stage to fail (provided the transformations preserve syntactic correctness).

In order to enable thorough test coverage, it is desirable to separate parts of TELE that make changes to specific parts of the abstract syntax tree (the enumerator), the part that maintains context and submits the program to the type-checker (the searcher) and the part that ranks successful modifications. This is shown in Figure 2.4. In this design, the searcher examines and modifies every node to attempt to correct the type-error.



**Figure 2.4:** An initial architecture design



**Figure 2.5:** A refined architecture diagram

## 2.5.2 Final Design

Figure 2.4 does not satisfy the requirements from Section 2.2.2 because the searcher is calling the type checker at least once for every node in the AST, hence is slow and may take more than a few seconds to execute (so does not satisfy the success criteria). Also, the enumerator may be carrying out redundant computation by making changes to a node that is well-typed.

The design in Figure 2.5 addresses these concerns by having the searcher use feedback from the type-checker to guide further search. Ideally, the searcher would be able to determine whether or not there exists a modification to a given node or at least one of its descendants to cause the program to become well-typed, so if no modifications exist, no descendants of the node are examined.

**Definition.** A node is *interesting* if modifications exist in the tree rooted at it to produce a well-typed program.

Note from Algorithm 1 that the searcher not only identifies interesting nodes, but also produces suggestions from them, as discussed in Section 3.2.3.

## 2.5.3 Other Benefits

When there are multiple locations in the program that could be modified to correct the error, TELE will suggest multiple modifications to the program, unlike the approach based on Algorithm *W*, which reports the error as location that

---

**Algorithm 1:** Algorithm for Figure 2.5

---

```

1: CandidateSols  $\leftarrow \emptyset$ 
2:  $Q \leftarrow \text{newQueue}()$ 
3:  $Q.\text{insert}(\text{AST-root-node})$ 
4: while  $Q$  is not empty do
5:    $N \leftarrow Q.\text{dequeue}()$ 
6:   if  $N$  is interesting then /* Searcher */
7:     CandidateSols  $\leftarrow$  CandidateSols  $\cup$  Tree[Gen-Repl $N$ / $N$ ]
8:     for all  $M \in \text{Modifications}(N)$  do /* Enumerator */
9:       if Tree[ $M/N$ ] is well-typed then
10:        CandidateSols  $\leftarrow$  CandidateSols  $\cup$  Tree[ $M/N$ ]
11:       end if
12:     end for
13:     for all  $C \in \text{Children}(N)$  do
14:        $Q.\text{insert}(C)$ 
15:     end for
16:   end if
17: end while
18: Solutions  $\leftarrow \text{Rank}(\text{CandidateSolutions})$  /* Ranker */
19: return Top $N$ (Solutions)

```

---

first caused inference to fail. This helps the programmer understand why type-inference failed, by reporting suggestions to each program segment in a set of inconsistent program segments.

As well as locating the programmer error accurately, the quality of error message can also be high, by making concrete syntactic suggestions that will correct the type-error. For example, general programming errors such as association (bracketing) errors can be reported by implementing a tree rotation modification in the enumerator, and errors that are common in the OCaml language can be detected, for example by enumerating syntactically similar constructs (such as  $[1, 2, 3]$  instead of  $[1; 2; 3]$ , or correcting pattern match association errors).

The highly modular structure will allow unit tests to be written for each component in isolation, allowing regression testing to reveal bugs and changes in behaviour in localised parts of the program.

Program Segment	Replacement Strategy
expression	raise E
$\tau$ (type constraint)	$\alpha$ (fresh type variable)
pattern $\rightarrow$ e	$\_ \rightarrow$ raise E
pattern	If the pattern is an identifier, attempt replacing by $\_$ , and every other identifier that is unbound in the neighbouring expression. Otherwise, recur on the pattern.
Type declaration	Recur until type identifiers are reached. Attempt replacing by all primitive types plus all previously defined user types until modification is successful
let id1 = ... and id2 = ... ... (top level bindings)	let id1 = raise E and id2 = raise E ...

**Table 2.1:** Approaches to determine interestingness of different kinds of OCaml AST nodes

## 2.6 Design of the Searcher

Recall that the goal of the searcher is to identify interesting nodes, which are nodes where there exists a modification in the subtree rooted at the node causing the whole program to become well-typed. The OCaml parse tree (excluding object-oriented features, signatures and modules) consists of a list of top-level statements, for example, value definitions, type definitions, exception declarations and expressions to be evaluated. Table 2.1 describes the replacement strategy to determine interestingness of each kind of AST node and the precision and recall of interesting nodes using these strategies is discussed in Section 2.6.1.

### 2.6.1 Precision and Recall of Interesting Nodes

In order to fully implement Algorithm 1 and achieve a search time that is proportional to the height of the AST, the searcher must be precise, i.e. every non-interesting node must be determined non-interesting (and for recall, every interesting node must be determined interesting) and, it must do so using a constant number of calls to the type-checker. The approaches given in Table 2.1 will

be analysed to see if these requirements are met.

The strategies for expressions, pattern-expressions and top level bindings are precise because the node is only deemed interesting if the modification causes the program to become well-typed (so by definition, the node is interesting). The strategies for patterns and type declarations are imprecise because they recur on nodes when it is not known whether or not they are interesting. The recall of interesting nodes by these strategies are analysed below.

### Expressions

If the rule below is valid, then every interesting node is identified (with one exception, given in Section 3.2.2).

$$\text{(EXPR)} \frac{\Gamma \vdash C[e] : \sigma}{\Gamma \vdash C[\mathbf{raise} \mathbf{E}] : \sigma'} \quad (\sigma' \text{ AT LEAST AS GENERAL AS } \sigma)$$

$$C ::= \_ \mid \text{Constructor}(e, \dots, e, C, e, \dots, e) \mid e_1 C \mid C e_2 \mid \dots$$

The context is defined to replace any expression unless it is a tuple given as an argument to a constructor because there is a syntactic restriction on the structure of arguments to constructors that requires the number of arguments to equal the arity of the constructor [10]. Although this rule is sound (follows trivially from induction over Owen's typing rules [10] provided the exception  $\mathbf{E}$  is fresh), I show in Section 3.2.2 that generalising the type of an expression can sometimes introduce a new type-error, thus this replacement does not identify all interesting nodes.

### Pattern-Expressions

Any interesting pattern-expression replaced by  $\_ \rightarrow \mathbf{raise} \mathbf{E}$  will result in a well-typed program (by induction over the OCaml typing rules [10] with one exception – see Section 3.2.2), so the interesting judgement recalls every interesting pattern-expression (with one exception).

### Patterns

Structural changes to patterns are not considered, so the strategy does not recall every interesting node.

### Type Declarations

The approach is similar to patterns – structural changes are not considered. This poor recall can be justified by considering the number of calls to the type checker required to enumerate all possible structural changes to a type structure, for every type declared in a program.

### Top Level Bindings

This strategy will underestimate interestingness for a value with the wrong name, so does not recall all interesting value definitions.

## 2.7 Design of the Enumerator

In this section, I present a table to describe some of the syntactic modifications attempted to each interesting node to produce type-error messages that suggest syntactic modifications. Some modifications were taken from Lerner et. al [6] and others as a result of the common errors I have encountered when writing test programs. In Section 3.3, I introduce additional enumerations after analysing a part of the corpus to look for common type-errors.

Expression	Replacement	Notes
<code>f a1 ... ak ak+1 ... an</code> for some $0 \leq k < n$	<code>f a1 ... ak ak+2 ... an</code> <code>f a1 ... ar ak+1 ar+1 ... an</code> for some $0 \leq r \leq n$ <code>f (a1, ..., ak, ak+1, ..., an)</code> <code>f a1 ... (ak ak+1) ... an</code> <code>f a1 ... ak (raise E) ak+1 ... an</code>	Haven't enumerated all possible changes in the interests of efficiency (e.g. permutations)
<code>identifier1</code>	<code>identifier2</code>	Try another identifier declared in the program
<code>record.fld := e</code>	<code>record.fld &lt;- e</code>	Mutable assignment instead of reference update.
<code>let x = e1 in e2</code>	<code>let rec x = e1 in e2</code>	Recursive flag in let bindings
<code>let x = e1</code> <code>  in let y = e2</code> <code>    in e3</code>	<code>let rec x = e1</code> <code>and y = e2</code> <code>  in e3</code>	Move nested lets into a simultaneous definition.

<code>[x, y, ...]</code>	<code>[x; y; ...]</code>	Change a list containing a single tuple to a list with each element as a separate node and vice versa
<code>[x1; ...; xk; xk+1; ...; xn]</code> for some $0 \leq k < n$	<code>[x1; ...; xk; ...; xn]</code>	List element deletion. Also swap, move elements and insert an element.
<code>let p:t = ...</code>	<code>let p = ...</code>	Remove type constraints

## 2.8 Performance Analysis

Section 2.6.1 shows that almost all non-interesting nodes can be determined non-interesting in a constant number of calls to the type-checker, so the number of calls to the type-checker increases linearly as a function of depth of the AST and the amount of time spent calculating what to type-check also increases at this rate, since the enumerations are made locally (a constant number for each interesting node). Ranking the suggestions should also increase slowly as a function of program size (for example by sorting the suggestions using a numerical desirability value for each suggestion). However, the time taken for each modified program to type-check scales approximately linearly with the textual size of the program [4]<sup>3</sup>. Therefore, for a given program, the overall cost of the search process is

$$O(n \log(n)) \quad (O(\log(n)) \text{ calls, each of which takes } O(n) \text{ time}) \\ + O((\log(n))(\log(\log(n)))) \quad (\text{for ranking - sort } O(\log(n)) \text{ suggestions})$$

where  $n$  is the number of nodes in the AST, so the cost of the search process is  $O(n \log(n))$ . The results from SEMINAL [6] indicate that the constant factor is low enough to be practical, with most 200 line programs completing in under one second, suggesting that the second success criterion can be satisfied, so the design presented in Section 2.5.2 will be implemented. The reasons given in Section 2.5.3 justify the choice of a search based procedure over alternative methods that modify the type system to produce higher quality type-error messages.

<sup>3</sup>typing/typemod.ml lines 720-890 – type\_struct is called once for every top level statement

# Chapter 3

## Implementation

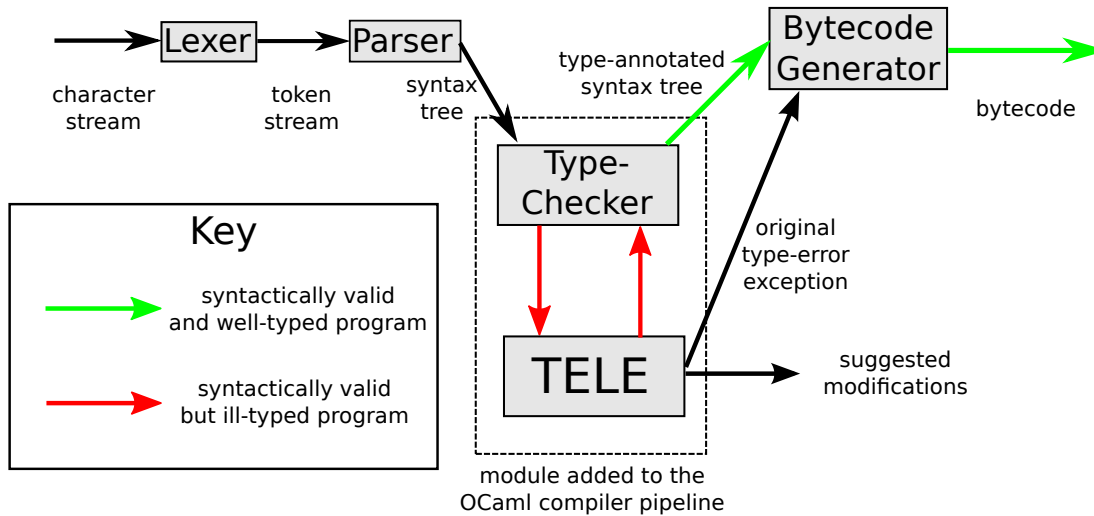
This chapter describes the implementation of TELE. Section 3.1 describes the modifications made to the OCaml compiler to add the TELE module and an evaluation database to collect quantitative data for the evaluation. Section 3.1.2 describes the OCaml parse tree implementation, which informs the implementation of the searcher and enumerator, described in Sections 3.2 and 3.3 respectively. The ranker implementation is described in Section 3.4. The implementation of the evaluation harness for collecting quantitative evaluation data is described in Section 3.6 and the details of unit testing are given in Section 3.7.

### 3.1 Modification to the OCaml Compiler

The TELE module was implemented as a single file to comply with the high degree of modularity in the OCaml compiler source code and to simplify integration with the compiler. To make the TELE module build automatically when the compiler is compiled, I used the `ocamldep` tool to modify the dependency database to produce a list of dependencies for each new and modified file in the compiler, which is read by the `Makefile`. New command line options, `-use-tele` and `-collect-ast` were added to allow TELE to be easily enabled or disabled and to collect the AST for quantitative evaluation, respectively.

#### 3.1.1 Compiler Pipeline

Since TELE will be invoked only if the program is syntactically valid and ill-typed, the OCaml compiler pipeline must be modified to initiate TELE when the program being compiled fails the type-checking stage. TELE will require the type-checking module to determine whether or not modified programs are well-typed, so to increase the simplicity of TELE's integration with the compiler,



**Figure 3.1:** The modification to the OCaml compiler pipeline required for TELE

the TELE module will be invoked for all syntactically valid programs, and if the program is well-typed, TELE passes the result from the type-checker to the next pipeline stage and initiates the search procedure otherwise. If the program is ill-typed, after producing suggested modifications, TELE will throw the original exception produced by the type-checker so the OCaml type-error message will also be produced and the required post-processing operations can take place.

The diagram in Figure 3.1 summarises the architectural modification to the compiler pipeline.

### 3.1.2 OCaml Parse Tree Structure

An OCaml AST consists of a list of `structure_items` [4]<sup>1</sup>. Each `structure_item` represents a toplevel declaration (e.g. a value/type definition, open/include statement, expression to be evaluated). Each different `structure_item` is a record containing the location in the source file represented and a `structure_item_desc` which is an AST for the program fragment that it represents.

Each of the types representing AST nodes of different forms have a similar structure, with each node containing the location in the source file and a description node containing the subtree rooted at this node. This structure affects the implementation of TELE in the following ways:

<sup>1</sup> `./parsing/parsetree.mli`, line 254

- AST nodes are of different types. This means that it is not possible to write a single function to operate on all kinds of nodes. Since such functions are required (for the searcher and the enumerator), the type `astItem` is implemented to provide a universal AST node type, discussed in Section 3.2.
- There may be more than one AST node that represents the same area of source code. For example, a function application argument without a label is contained within an expression node inside a label-expression node (with a blank label). This means that it is not sufficient to identify an AST node using only the location in the file. Instead, the type `nodeKind` is used in the evaluation to help manually tag files with the node containing the programmer error and identify the correct node reported by TELE. This is used to calculate the AST node distance between nodes to evaluate the effectiveness of type-error messages produced by TELE against OCaml and SEMINAL, as discussed in Section 3.6.

## 3.2 Searcher Implementation

From Figure 2.5, the searcher must implement:

1. the identification of interesting nodes;
2. maintaining context of ASTs past to the enumerator to determine which modifications made by the enumerator result in well-typed programs; and
3. submitting successful changes to the ranker.

To implement 1 and 2 with a single function, the OCaml AST nodes must belong to the same type. The AST could be converted into an AST where every node is of the same type, however, this deep conversion would have to be reversed to see if the modifications result in a well-typed program because the type-checker operates on OCaml ASTs of type `structure`. To avoid this overhead for every modified AST, I implemented a universal type, `astItem`, with a different constructor for each type of node in the OCaml AST. This means that there is no computational time wasted converting parts of the AST that are not interesting, and the original OCaml AST node can be recreated in one step by removing the constructor. In addition, `astItem` distinguishes between the same types of node that appear in different contexts to improve the efficiency of the searcher and the enumerator and provide more helpful type-error messages, for example between

nodes inside a `match` block that map patterns to expressions and definitions of values that map patterns to expressions.

Another type is required for the enumerator and the searcher to communicate the change made to the original node to aid ranking and the production of the type-error message. For example, the searcher may make a generic replacement (such as `raise E`), or an identifier replacement in a pattern and the enumerator may attempt a tree rotation or a reassociation of function arguments. This is implemented by the type `astModified` below. The fields `typedTree` and `astNodeType` are populated once the modification has been determined successful by the searcher and are used to produce helpful type-error messages from generic replacements, discussed in Section 3.2.3. After these fields have been populated, it represents the smallest possible package of information that could be useful to the ranker and to produce type-error messages. The context that represents the unchanged part of the AST is omitted from this type and maintained separately by the searcher because this is not required outside the searcher function.

```

1 type astModified =
2   {
3     ast: astItem; (* Modified node *)
4     change: changeType option; (* Change made from original *)
5     aloc: Location.t; (* Textual location of original node *)
6     typedTree: typedTreeOption;
7     descr: string;
8     adepth: int;
9     astNodeType: (astItemType option) list
10  }

```

The `findSolutions` function implements the searcher by taking an `astItem` representing the whole AST and a type-checker function from the OCaml pipeline and returning a list of unordered modifications that cause the program to become well-typed, which is passed to the ranker. The structure of this function is given in Figure 3.2. It modifies the program to include a new exception declaration at the top of the file (which should not already be declared in the program so that new type-errors are not introduced because of an incorrect number of arguments to the exception being raised) to allow the `raise E` replacement for expressions to compile and calls the `solFinder` function which is the main recursive searcher function to traverse the AST and collect modifications to the program that result in well-typed programs.

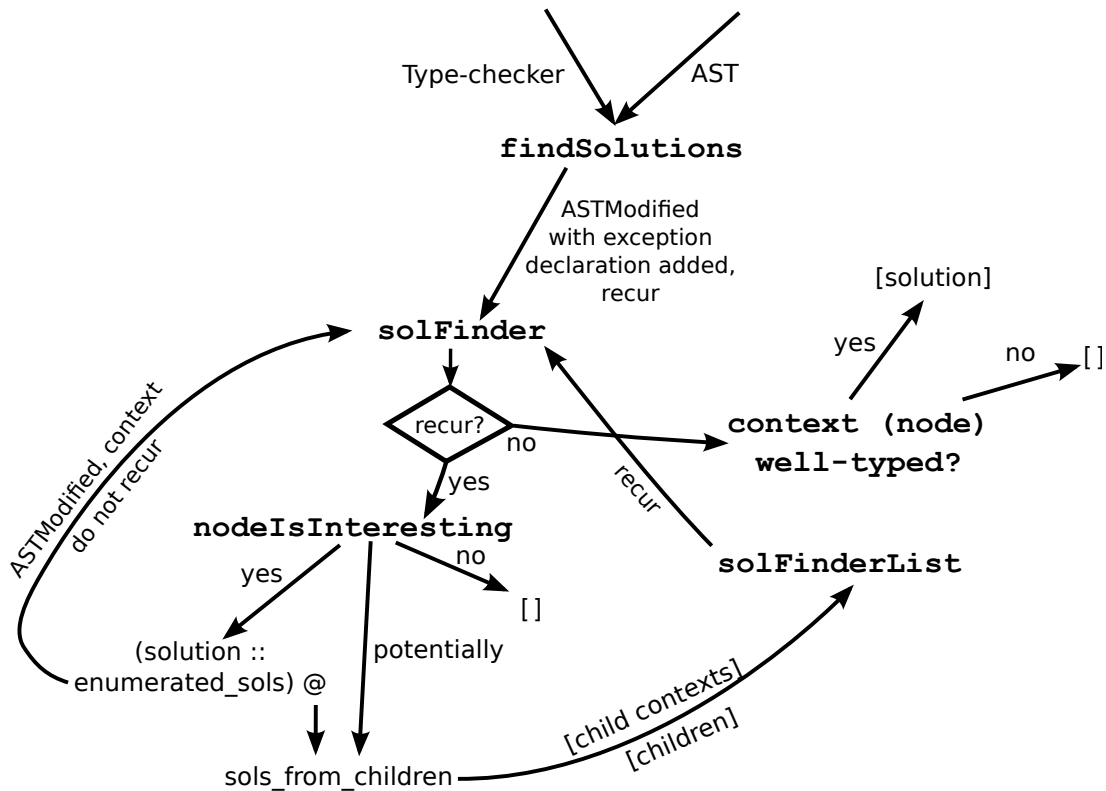


Figure 3.2: The structure of the searcher function

### 3.2.1 Maintaining Context and Recurring on AST Nodes

Context is required for the sole purpose of checking whether a modification has resulted in a well-typed program. It represents the unchanged part of the AST for each node to be modified. Algorithm 1 shows that if a node is interesting, then its children are explored, so the context must be produced for the children nodes given the context of the parent and then the search procedure is called recursively on the children. Context is implemented by a function of type `astItem -> structure` because this makes the production of context for a child node given the context for a parent node simple by wrapping the parent context in a function to add the information contained in the parent and sibling nodes. This means that for any `node` of type `astItem`, the type-checker can be called with `(context node)` to see if the program is well-typed. If the number of child nodes are not fixed, the list of contexts (one for each child) is produced using a context construction function which takes a list of the child nodes and a context function that takes a list of child nodes and returns the OCaml AST. This function is implemented by iterating along the children, producing a context function

for the  $i^{\text{th}}$  child that applies the parent context to the modified child appended between the  $(i - 1)^{\text{th}}$  and  $(i + 1)^{\text{th}}$  unmodified children.

Recurring on child nodes is implemented by producing a list of children nodes alongside a list of context functions as explained above and recursively calling the `solFinder` function for each child with its corresponding context, illustrated by Figure 3.2. This function takes an additional boolean argument, `recur` which determines whether or not the children of the node should be explored if it is deemed interesting. This allows enumerated modifications to be type-checked and added to the list of solutions alongside those produced by the searcher.

### 3.2.2 Determining Interestingness

Recall from Section 2.5.2 the definition of the interesting property – a node is interesting if there exists a modification in the subtree rooted at that node to produce a well-typed program.

As explained in Section 2.6, some nodes can be precisely determined interesting by making a generic replacement (e.g. `raise E` for expressions), some can be determined interesting in special cases (e.g. patterns where a variable can be renamed to a free variable inside its neighbouring expression to correct the type-error) and some cannot be determined interesting so the searcher recurs on all the children of that node.

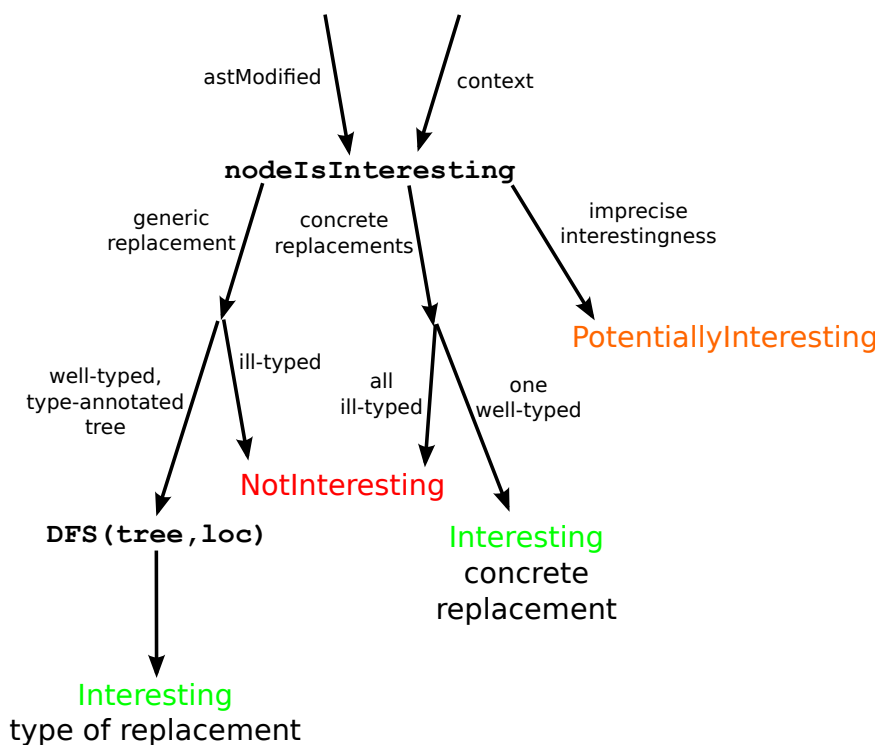
The function `nodeIsInteresting` implements the interestingness judgement of nodes by returning an instance of the type `interestingInfo`. If the node is interesting, the record returned is used to fill in the remaining fields of the `astModified` record so the modification forms a complete suggestion and a type-error message can be produced from the modification (see Section 3.5).

```

1 type interestingInfo =
2     NotInteresting
3     | Interesting of {tree: Typedtree.structure;
4                     nodeType: (astItemType option) list}
5     | PotentiallyInteresting

```

The structure of `nodeIsInteresting` is given in Figure 3.3. It implements the algorithm given in Table 2.1 by making generic replacements, trying to replace identifiers until one results in a well-typed program (in type definitions and patterns) and implementing recurring on children nodes when it is impossible to determine interestingness using the `PotentiallyInteresting` constructor. When a node is determined `PotentiallyInteresting`, the child nodes are explored but no suggestion is added and no enumerated suggestions are attempted



**Figure 3.3:** The structure of the function to determine interestingness

(since it is not known whether or not there exists a modification to that node to correct the type-error), as shown by Figure 3.2. For clarity of implementation, the identification of interesting nodes should be solely performed by the function `nodeIsInteresting`, and if a concrete modification is found as a side-effect of determining interestingness, this should be a suggested modification and modifications of the same nature should not be attempted by the enumerator.

The collection of free variables in expressions (for pattern enumeration) is implemented when required by the searcher because each pattern is examined by the searcher at most once. This is implemented by a function that traverses the tree to identify the expression that is associated with the pattern and then removes the duplicates from the list of free variables in the expression.

### Monomorphic Definition Typing and the Recall of Interesting Nodes

In this section, I explain why the `raise E` replacement of an expression does not recall all interesting nodes by first examining the OCaml typing rules [10] and then present an example to show that generalising the type of an expression can sometimes introduce a new type-error.

The environment is invalid if a value has a type containing free type variables because the conclusion of the rule

$$\frac{E \vdash \forall t : \mathbf{Type}}{E, (value\_name : \forall t) \vdash \mathbf{ok}} \text{ (JT\_EOK\_VALUE\_NAME)}$$

cannot be proved because the value type contains free type variables so the conclusion in the rule below cannot be proved:

$$\frac{E \vdash \mathbf{ok} \quad E \vdash idx \text{ bound}}{E \vdash \langle idx, num \rangle : \mathbf{Type}} \text{ (JT\_T\_VAR)}$$

This reduces the recall of interesting nodes because the replacement `raise E` may cause a type-error from a value having a type that contains free type variables but a replacement that instantiates the type-variables in the type of the value may resolve the type-error. For example, consider the ill-typed program:

```
let y = let x = ref []
        in (); x

(* Error: The type of this expression, '_a list ref,
contains type variables that cannot be generalized *)
```

A generic replacement does not resolve the type-error:

```
exception E
let y = let x = ref []
        in raise E; x

(* Error: The type of this expression, '_a list ref,
contains type variables that cannot be generalised *)
```

However, this node is interesting, and the replacement below yields a well-typed program:

```
let y = let x = ref []
        in x := [3]; x
```

This example illustrates the surprising result that generalising the type of an expression may cause the program to become ill-typed.

### Correction to Generic Replacement Strategy

To help produce suggestions for expressions that cannot be determined interesting using the generic `raise E` replacement, the exception thrown by the

type-checker is examined after every generic replacement, and if the error is caused by a value type containing type-variables that cannot be generalised, the node is marked `PotentiallyInteresting`, causing the children to be explored and in addition, enumerations are attempted for that node (unlike other `PotentiallyInteresting` nodes). This allows type-errors caused by type variables that cannot be generalised to be corrected provided an enumeration exists to correct the type-error. Note that this can cause nodes to be marked `PotentiallyInteresting` that are not interesting, for example, nodes inside a different definition from the one containing the type generalisation type-error, yielding a less efficient execution.

### 3.2.3 Finding the type of Generic Replacements

When a modification, represented by an `astModified` record causes the program to become well-typed, the function `nodeIsInteresting` returns a record as an argument to the `Interesting` constructor which contains the type of the node that has been modified as well as the type-annotated syntax tree. The type of the modified node is found by a depth first search of the type-annotated syntax tree to find a type-annotated node with the same location range in the source file as the original node. This finds the type of the correct node because when the generic replacement is made, it is assigned the same location data as the original node, which is stored in the `aloc` field of the `astModified` record.

The types of pattern-expression nodes are found by performing a depth first search for a pair of sibling nodes in the type-annotated tree to store the type of the pattern and the expression (since the pattern-expression replacement is `_ -> raise E`) inside the `astModified` record, and the types of value definitions are found and recorded similarly. For example, for the ill-typed program:

```

1 type foo = Foo1 of int | Foo2
2 type bar = Cons
3
4 let f x = match x with Foo1(y) -> y
5                | Cons -> 1

```

the searcher produces the suggestions:

```

Try replacing the match pattern-expression on line 4, chars 23 to 34 with
  Pattern type bar and body type int

```

```

Try replacing the match pattern-expression on line 5, chars 10 to 19 with
  Pattern type foo and body type int

```

### 3.3 Enumerator Implementation

In this section, I describe the implementation of the enumerator which looks for specific syntactic modifications to nodes that have been determined interesting by the searcher to produce more informative type-error messages than those produced from the searcher. The purpose of the enumerator is to return a list of enumerated nodes given an interesting node. It should not enumerate nodes that have been determined interesting by the searcher by making concrete replacements (type definitions and patterns) because the searcher has effectively already implemented the enumeration. I introduce and justify the modifications involving function argument reassociation and rotation, reassociation of pattern match clauses and finally, the collection of identifier replacements inside expressions. The enumerations in Section 2.7 have been implemented and some are discussed below.

#### 3.3.1 Rearranging Function Arguments

As described in Section 2.7, some permutations of function arguments will not be attempted to avoid calling the type-checker an exponential (in the number of arguments) number of times. Instead, the enumerations include removing each argument, inserting a generic argument before and after each argument, moving each argument to a new position, un-currying all the arguments and associating each pair of arguments in a new function application. This results in  $O(n^2)$  enumerations where  $n$  is the number of function arguments.

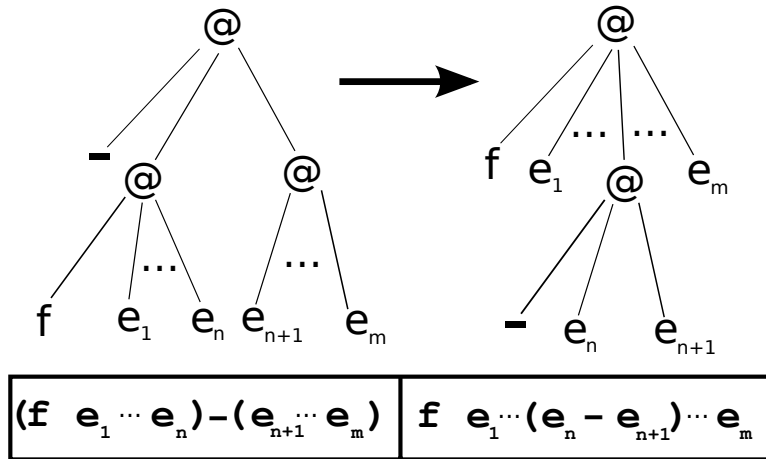
#### 3.3.2 Tree Transformations

The goal of the tree transformation enumerations is to correct common association errors in function applications excluding those that change the order of the arguments. After examining 120 ill-typed programs from four students in the seen part of the corpus to look for different kinds of association errors, there were only two different kinds of application association errors and both involved infix functions.

The first association error involves a programmer incorrectly assuming that an infix operator binds more tightly than function application, causing an expression to be incorrectly identified as a function. For example, the expression

f a b c-1 d e
---------------

is parsed as



**Figure 3.4:** The first syntax tree transformation enumeration

```
(f a b c) - (1 d e)
```

Figure 3.4 describes the syntax tree manipulation required to correct this type-error.

The second association error also involves a programmer assuming that an infix operator binds more tightly than function application but does not cause an expression to be labelled incorrectly as a function. For example,

```
float_to_int 1.0 /. 2.0 /. 4.0
```

is parsed as

```
((float_to_int 1.0) /. 2.0) /. 4.0
```

Figure 3.5 describes the syntax tree manipulation required to correct this type-error. Note that these enumerations can also correct type-errors with prefix functions, but the syntax manipulation corresponds to changing the order of the functions instead of adding parentheses.

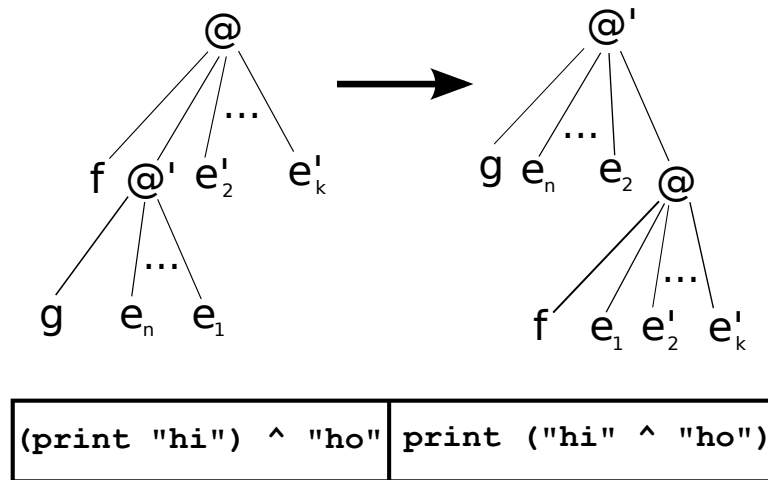


Figure 3.5: The second syntax tree transformation enumeration

### 3.3.3 Match Reassociation

Another common type-error encountered in the corpus is caused by associating a pattern-match clause with the wrong match expression. For example,

```
match x with pa1 ->
  match y with pb1 -> eb1
             |pb2 -> eb2
             |pa2 -> ea2
```

is parsed as

```
match x with pa1 ->
  (match y with pb1 -> eb1
   |pb2 -> eb2
   |pa2 -> ea2)
```

The match enumeration should correct any single logical match association error (because multiple type-error correction is not a feature of TELE). If there is a single logical association error, each inner match expression could have any number of pattern-expression clauses adjacent to the last clause that should be associated with the outer match expression. Algorithm 2 describes the implementation of the match reassociation algorithm.

---

**Algorithm 2:** Algorithm for enumerating match expressions corresponding to a single association error

---

```

1: EnumeratedSet  $\leftarrow \emptyset$ 
2: for all  $(p_i \rightarrow e_i)$  clauses in outer match expression do
3:   match  $e_i$  with “ match _ with  $(p'_0 \rightarrow e'_0) \dots (p'_m \rightarrow e'_m)$  ”
4:     for all  $(p'_j \rightarrow e'_j)$  clauses in inner match expression do
5:       EnumeratedSet  $\leftarrow$  EnumeratedSet  $\cup$ 
6:         { “ match _ with
7:            $(p_0 \rightarrow e_0) \dots (p_{i-1} \rightarrow e_{i-1})$ 
8:            $(p_i \rightarrow \text{match\_with } (p'_0 \rightarrow e'_0) \dots (p'_{j-1} \rightarrow e'_{j-1}))$ 
9:            $(p'_j \rightarrow e'_j) \dots (p'_m \rightarrow e'_m)$ 
10:           $(p_{i+1} \rightarrow e_{i+1}) \dots (p_n \rightarrow e_n)$  ” }
11:     end for
12:   end match
13: end for
14: return EnumeratedSet

```

---

### 3.3.4 Collecting Identifiers for Identifier Enumeration

The enumerator should enumerate identifiers that are inside expressions. Identifiers that can appear inside an expression include those that appear in patterns (binding identifiers in expressions), the names of values and their formal arguments.

The identifiers used to determine the interestingness of patterns were collected when the pattern was required because the set of identifiers for each pattern is independent of the other patterns and each pattern is examined at most once by the searcher. However, the set of candidate identifier replacements for identifiers inside expressions can be similar to the candidate set for other identifiers (e.g. for identifiers inside the same function), so the set of candidate replacements was collected once for all identifiers. Identifiers are collected by a function which traverses the AST, adding the identifiers of declared exception identifiers, formal function arguments, function/value names and identifiers within patterns. Identifiers within other expressions are also collected because these may refer to values inside libraries that are not declared in the source file, allowing typographical mistakes in library calls to be corrected if a correct reference has been made elsewhere.

### 3.4 Ranker Implementation

Smaller changes should be preferred over larger changes and the size of a modification should take into account the distance to a leaf node as well as the distance from the root. Enumerated changes should be preferred over generic suggestions because they describe the syntactic modification required to the program, rather than the more general suggestion of replacing a program fragment with another fragment of a certain type. The exact relative weights of different changes was initially estimated and then manually fitted to reduce the average distance to the programmer type-error in the seen part of the corpus to produce evaluation data using the unseen part of the corpus.

I divided the corpus of annotated ill-typed programs in the ratio 2:3, the former part to set ranking heuristics (weights of the enumerations and percentage depth of the change) and the latter part to evaluate TELE. I chose to give a smaller part of the corpus to fit the heuristics than is commonly given for supervised learning algorithms (Setchi et al. suggest that 65% of the corpus is used for training [11]) because the ranking heuristics should not need large adjustments since the SEMINAL heuristics were set without fitting them to any corpus and still produced type-error messages closer to the programmer error than OCaml [6]. Also, I wanted to have a sufficiently large number of ill-typed programs to produce statistically significant conclusions about TELE.

To improve TELE's performance further, when the only solution that TELE could find was to replace the whole program by `raise E`, I modified TELE to revert to OCaml's type-error message. This helps to reduce the distance to the type-error when it is within a node that cannot be determined interesting with a generic replacement and the appropriate enumeration has not been implemented (e.g. certain problems with type declarations).

The ranker takes an unsorted list of `astModifieds` which represent modifications to the program that cause the program to become well-typed and returns a list with the same elements, sorted according to desirability. The maximum distance to a leaf is calculated by a function by converting the subtree that represents the part of the AST that has been modified into a `genericAst`.

I also added an option to bypass the ranking system altogether for changes that are so desirable that they should always be the top ranked suggestion, such as adding the `rec` flag to allow a function to call itself which is a change with shallow depth but is a very easily made mistake in OCaml. The initial weights were set to prefer changes that add to the program (e.g. inserting an argument) to those that remove parts of the program and to prefer all enumerated changes over those produced from generic replacements by the searcher. Ambitious enumerations

such as tree rotations are ranked higher than other enumerations because they are very specific and correct errors that are easy to make.

### 3.5 Producing the Type-Error Message

The function `prettyPrintSolution` produces a string describing the modification represented by an `astModified`. The type-error message contains the kind of node, the relevant surrounding context (populated by the searcher during the recursion stage and stored in the `descr` field), the location of the modification in the source code and the nature of the modification including the type of the generic replacement or the identifier used to replace another identifier (stored in the `astNodeType` field). If the modification was produced from the searcher (has a `change` field of `None`), the function `prettyPrintTypes` is called with the value of the `astNodeType` field to produce the message describing the type of the replacement(s). This function uses the OCaml compiler library function `Printtyp.type_expr` to produce a string describing the type of the replacement.

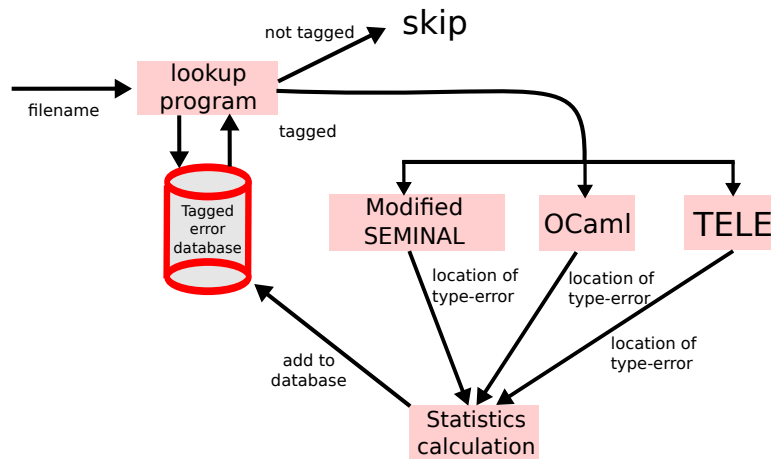
### 3.6 Evaluation Harness

Each manually tagged error location is of the form (`cnum_s`, `cnum_e`, `nodeKind`) which marks the character range in the source file and the kind of parsetree node that contains the error. This allows different nodes in the parsetree that have the same location in the source code (such as expressions containing identifiers and the identifiers themselves) to be distinguished, so can be used to accurately identify parsetree nodes.

I chose to write a shell script to call TELE with the `-collect-ast` argument for each file to be analysed because it is not possible to execute SEMINAL and TELE from an OCaml program because they are written for different versions of the compiler. The shell script only calls SEMINAL and TELE if the file has been manually annotated to reduce the time taken to collect the data. The SEMINAL source code was modified to only output the location range of the highest ranked suggestion. The structure of the shell script is given in Figure 3.6.

#### 3.6.1 Calculating Distance Statistics

In this section, I explain how the locations reported by TELE, SEMINAL and OCaml were converted into AST nodes and the method used to calculate distances between them to produce quantitative evaluation data.



**Figure 3.6:** The structure of the shell script to produce quantitative evaluation data.

The OCaml AST nodes do not all belong to the same type (see Section 3.1.2), the AST should be converted to a tree where every node belongs to the same type to allow a single function to calculate distance statistics. This is implemented by the function `convertToGenericAst` which converts ASTs of type `astItem` to `genericAst`, whilst retaining the location in the source file that each node maps to and the type of the original node (to enable disambiguation between different nodes with the same location in the source file).

Each `genericAst` node stores their parent and a lazy list of their children, so it is possible to calculate the distance (length of the shortest path) between any two nodes given only the two nodes.

```

1 type genericAst = { parent: genericAst option;
2                   children: unit -> (genericAst list);
3                   depth: int;
4                   nodeKind: nodeKind;
5                   loc: Location.t
6                   }

```

The location range in the source file corresponding to the locations reported by TELE, SEMINAL and OCaml are converted into `genericAst` nodes by the function `findDeepestNode` which uses `tag` to enable disambiguation between different nodes with the same location range in the source file.

### 3.7 Unit Tests

The OCaml unit testing framework, OUnit was used to implement unit tests for TELE. Due to the highly modular structure of TELE, I was able to implement separate tests for the searcher, the enumerator and the ranker. To test the searcher, a collection of fifteen ill-typed programs containing type-errors in every kind of node with a variety of surrounding contexts were created and the number of nodes that should have been identified as interesting were counted in each file. The enumerator tests were constructed similarly by creating a files that can have their type-errors resolved by each enumeration and testing for the presence of the appropriate enumeration in the list of solutions. The ranker tests were written by comparing the top suggestion with the expected suggestion (the programmer error that I tried to emulate in the source file). Although a ranker test failure is not as severe as a searcher or enumerator failure because it is not possible to accurately determine the programmer error by looking only at the source file, it does allow the implications of a particular ranking modification to be observed by examining which modifications have no longer been ranked most highly.

A total of 40 tests were written and were re-run after each change to TELE to monitor changes in each component, particularly the ranker because small changes in the weights may cause different suggestions to be the ranked highest. New tests were written when new enumerations were introduced.



# Chapter 4

## Evaluation

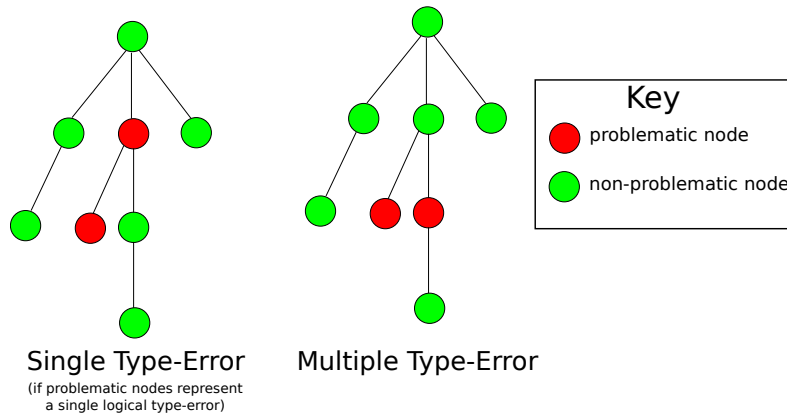
The quantitative evaluation consists of a comparison of the distances to the programmer’s type-errors from the locations reported by TELE, OCaml and SEMINAL, a comparison of execution times of TELE and SEMINAL and an analysis of performance as a function of the kind of node that contains the type-error. The ill-typed programs come from the unseen part of the corpus (the other part was used to set ranking heuristics) provided by the University of Washington and I explain how the location of the programmer’s type-error is identified in Section 4.1. The qualitative evaluation analyses the textual output of TELE for a range of ill-typed programs and evaluates the usefulness of TELE’s type-error messages for all ill-typed programs.

### 4.1 Preprocessing the Evaluation Corpus

TELE is designed to be able to correct type-errors that require modifications to one or more nodes that lie on the same path in the AST from the root node (because the algorithm only makes one modification at once). Therefore to evaluate TELE, only ill-typed programs that contain a single type-error were annotated. The usefulness of TELE for all ill-typed programs is discussed in Section 4.3.3.

**Definition.** A *single type-error* in an ill-typed program is defined as an error that is caused by a single logical error by the programmer and the nodes that require modification to correct this type-error have a closest common ancestor that also requires modification.

This amounts to a node in the AST containing an incorrect value or having incorrect form (e.g. wrong number of arguments) as well as errors with multiple nodes such as match re-association and application rotation (which modify the parent node and a child node), but not errors with multiple nodes that represent



**Figure 4.1:** An illustration of single and multiple type-errors

independent logical mistakes or where the deepest common ancestor of these nodes does not contain a programmer error.

The location of the type-error was determined primarily by examining later compilation attempts and identifying the part of the program that the programmer changed to resolve the type-error. Note that this was also used to determine whether or not the error was a single type-error or not, for example:

```
type foo = C of int

let f x y = (match x with C(s1) -> s1) ^
            (match y with C(s2) -> s2)
```

would be deemed a single type-error if a later compilation attempt contained:

```
type foo = C of string

let f x y = (match x with C(s1) -> s1) ^
            (match y with C(s2) -> s2)
```

but would be deemed a multiple type-error if a later compilation attempt corrected the error by correcting multiple usages of the type `foo`:

```
type foo = C of int

let f x y = (match x with C(s1) -> string_of_int(s1)) ^
            (match y with C(s2) -> string_of_int(s2))
```

If this information was unavailable (e.g. the compilation attempt was the last in the corpus for that student and assignment), knowledge of the programming task

was used (e.g. comparing the type signatures of the functions defined with those expected) to manually determine the location of the error. In a small number of cases (around 1%), it was not possible to accurately determine whether or not the type-error was a single type-error or a multiple type-error or the location of the programmer type-error, and in these cases, the files were not annotated.

## 4.2 Quantitative Evaluation

### 4.2.1 Type-Error Location Comparison with OCaml

On average, TELE is significantly closer to the programmer type-error than OCaml, with TELE being closer in 48 files, further away in 21 files and equal on the remaining 51 files. The average distance for TELE is 1.7 nodes and the average distance for OCaml is 2.8. From Figure 4.2, for expressions, OCaml frequently produces a type-error message that is very far from the programmer error while TELE’s type-error message remains close to the programmer type-error. This is mostly because as explained in Section 2.1.3, OCaml always assumes that the usage of a well-typed value that causes a type-error is the problem and never the definition of the value. For example, consider the ill-typed program from the corpus below:

```

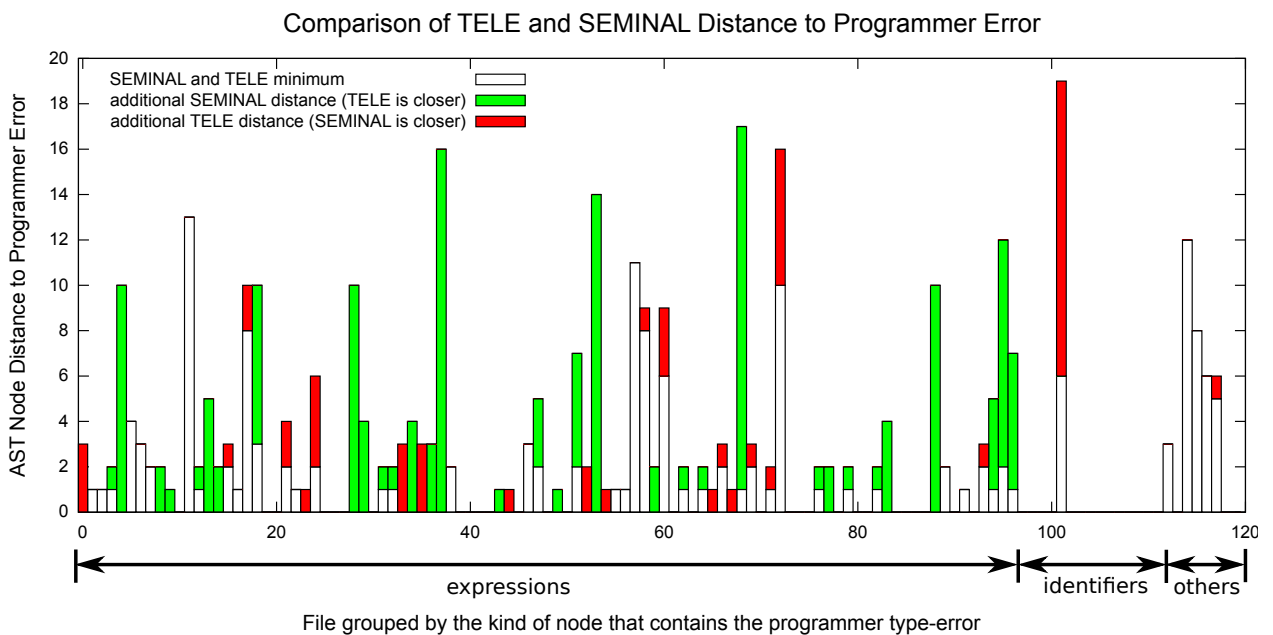
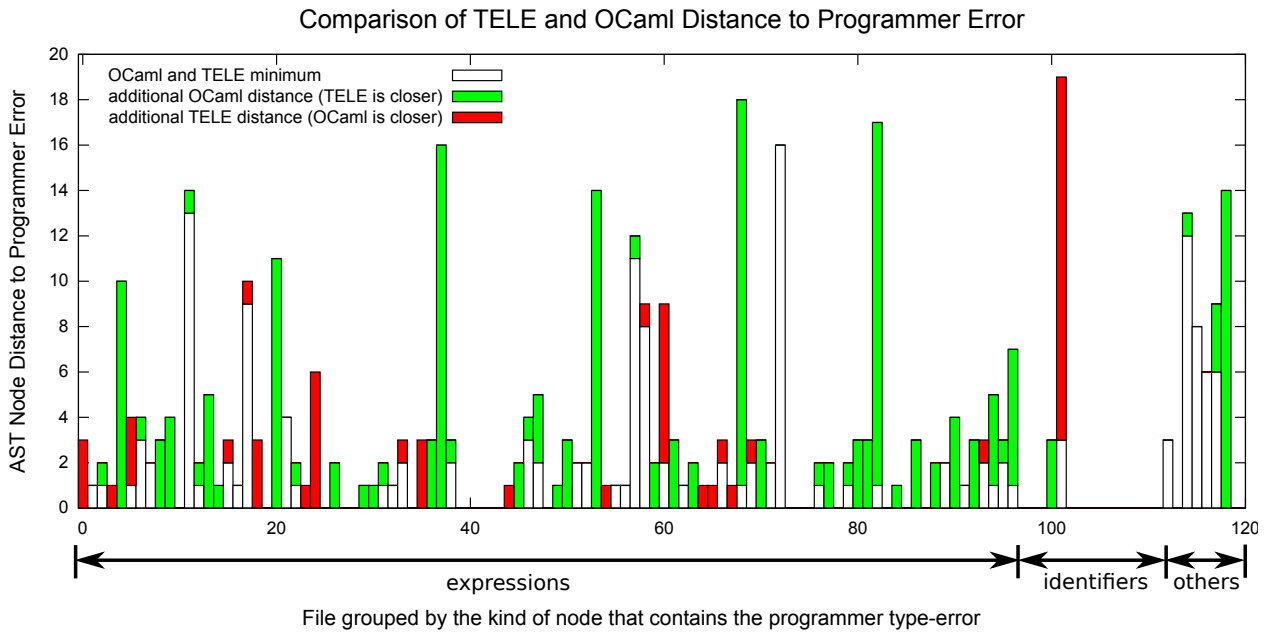
1 let rec computeFreeVars e =
2   match e with
3   | Var s -> raise Unimplemented
4   | Lam(s,e,lst) -> Some lst
5
6 ...
7 let ans3 = interp2 [] (fst (computeFreeVars ex1))
8 ...

```

OCaml: The expression on line 7 has type `string list option` but an expression was expected of type `'a * 'b`

TELE: Try replacing the body of the pattern match on line 4 with an expression of type: `'a`

In this case, TELE prefers changing `Some lst` to `computeFreeVars ex1` because the former node lies deeper in the AST.



**Figure 4.2:** Graphs comparing the distance of TELE, OCaml and SEMINAL to programmer type-errors in the unseen part of the corpus

In general, TELE will only make a suggestion if it produces a well-typed program, so if there are multiple uses conflicting with a single definition, the definition replacement is ranked more highly because replacing the definition is usually

a smaller change than replacing the common ancestor of multiple usages. This means that TELE decides to report the definition or usage of a binding that causes a type-error based on the structure of the file, unlike OCaml which has the static reporting policy of reporting the usage of a (well-typed) definition that causes the program to be ill-typed. The programmer error can either be inside the well-typed binding or the ill-typed usage and TELE's heuristics favour smaller changes. This dynamic policy produces type-error locations that are closer to the programmer error because the heuristics were set to prefer smaller changes and are clearly a good predictor at selecting the node in a conflicting set that the programmer will correct.

Figure 4.2 shows that when the programmer type-error is caused by an identifier node, TELE identifies the correct node 14/15 times and OCaml identifies the correct node 13/15 times. OCaml is accurate in this situation because if the identifier is undefined, type inference fails as soon as it encounters it, and thus the identifier is reported (assuming a single type-error). If the identifier is not undefined but incorrect, it may cause type-inference to fail in a different area, hence does not always locate the programmer error. TELE has a similarly high success rate because when a concrete identifier replacement is found (by the enumerator), a large desirability is assigned because the node is a leaf (so the change is small) and an additional score is added because enumerated replacements are preferred to generic replacements. However, a concrete replacement is not always found because the identifier replacement may not be present in the file but in an external library or built in to the language. In this case, the identifier replacement may not be ranked as highly because it is a generic replacement produced from the searcher, so other modifications (such as deleting the identifier) may be preferred.

Other kinds of node containing the programmer's type-error, including patterns and type definitions are far away from the locations reported by OCaml and TELE, with TELE and OCaml having an average distance of 4.4 and 6.6 nodes respectively. OCaml locates errors in patterns poorly for a similar reason as expressions – the patterns can be thought of as binders of components of the expression to be matched and the neighbouring expression uses these bindings. It never reports a well-typed pattern as the cause of a type-error and always reports the usage of the pattern-bound variable that causes an expression to be ill-typed, for example

```

type foo = C of int * float | D of string
let f x = match x with
           C(a,b) -> if(b = 2) then string_of_float(a)
                       else "not 2"
           |D(s) -> s

```

OCaml assumes that the binding is correct and reports a type-error caused by the usage of the variable:

```

OCaml: The expression '2' has type int but an expression
       was expected of type float

```

TELE does not assume that the binding or usage are correct and will suggest the highest ranked modification that causes the program to become well-typed. However, since patterns cannot be determined interesting accurately (see Section 2.6.1), enumerations are not attempted unless the pattern is an identifier (to reduce the number of calls to the type-checker), so the modification of swapping the arguments *a* and *b* in the pattern above is not attempted. This explains TELE's poor location of errors inside patterns. Similarly, problems with type declarations are located poorly by TELE (only those where the solution is to replace a type-identifier by a primitive or previously defined type are reported), however, if a type declaration contains an error that TELE can't correct, the only suggestion generated is to replace the whole program which is an unsatisfactory type-error message, so TELE reports OCaml's type-error message to produce a more helpful message.

## 4.2.2 Type-Error Location Comparison with SEMINAL

Figure 4.2 shows that TELE produces a type-error message that is closer to the programmer type-error than SEMINAL 33 times, further away 22 times and equal 65 times. The average distance to the type-error is 1.72 nodes for TELE compared with 2.53 for SEMINAL. The first reason for this is the difference in ranking strategy. SEMINAL measures the size of the change only by the textual range of the modification in the source code [7]<sup>1</sup> unlike TELE which measures size by the depth of the change in the AST as well as the distance to a leaf node. Therefore, TELE is able to select a modification to one of a pair of inconsistent statements in a way that more accurately measures the size of the modification as perceived by a programmer. Secondly, SEMINAL creates a signature for top level bindings

<sup>1</sup>Line 5675

that are well-typed [7]<sup>2</sup> (so they are not type-checked again). This causes successful modifications to be ignored since top-level modifications can contain the programmer type-error even if they are well-typed, for example in the ill-typed program below,

```

1 type foo = Cons of int
2
3 let f x = match x with Cons(s) ->
4           if String.length s > 3 then
5             print_string(s) else
6             print_string(s ^ ", hello")

```

TELE will suggest changing the type identifier `int` to `string` but SEMINAL will only suggest replacing the `Cons(s) -> if ...` pattern-expression, which is a much larger change.

Thirdly, unlike TELE, SEMINAL truncates the file so that top-level bindings after the binding containing the type-error reported by OCaml are removed [7]<sup>3</sup>. This means that suggestions produced by SEMINAL do not necessarily cause the program to become well-typed, so if the program contains a single type-error, the actual programmer type-error (which corrects the whole program) is competing against solutions that may not correct the type-error. For example, consider the ill-typed program:

```

let f x y = x + int_of_float(x);;
let _ = if (f 1 7.1 > 8) then print_string "rounded up"

```

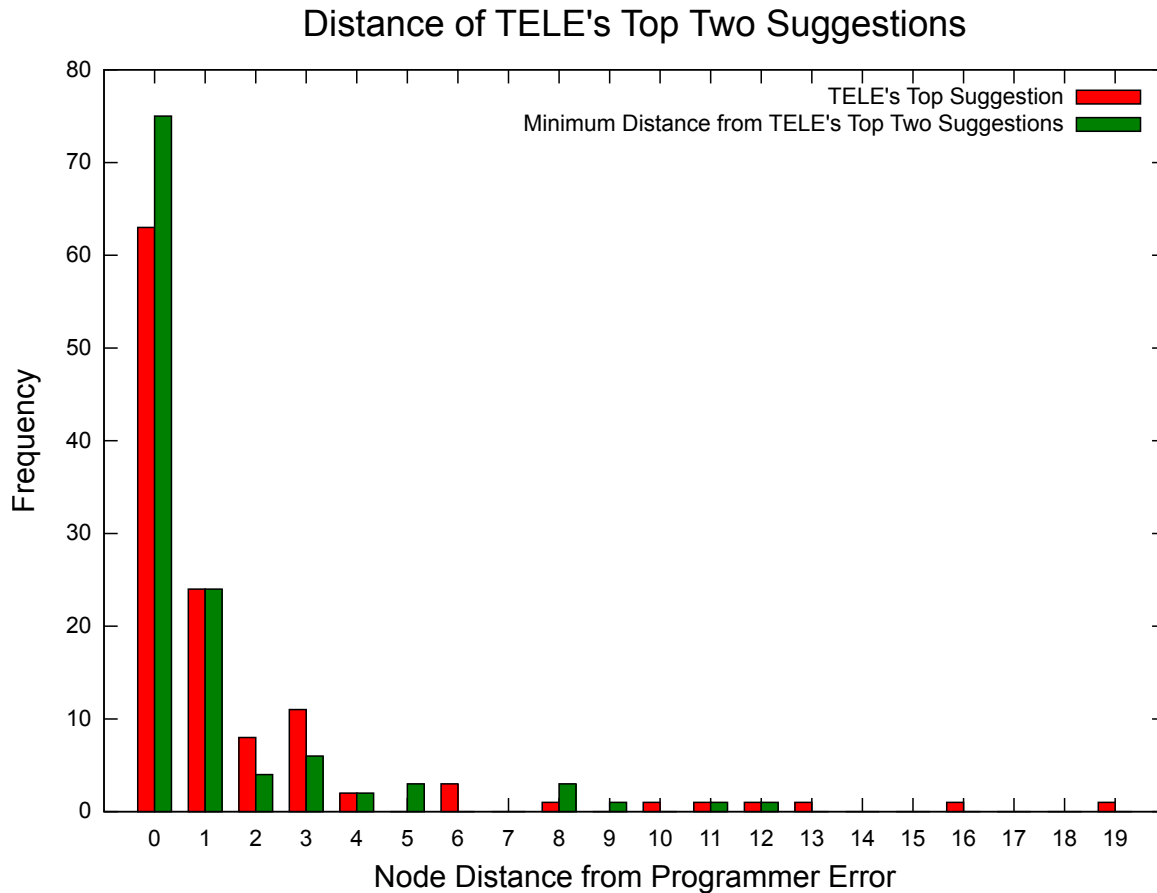
SEMINAL's modifications to the value `f` are produced regardless of usages of `f` in later bindings, so a modification that corrects the whole program (e.g. replacing the identifier inside `int_of_float` by `y`) may be ranked lower than a modification that does not correct the type-error (e.g. replacing the first `x` by `y`). Note that when later bindings contain independent type-errors, this approach produces higher quality type-errors, as discussed in Section 4.3.3.

### 4.2.3 Quality of Alternative Suggestions

TELE produces multiple, alternative modifications to correct the type-error (accessible with the `-verbose` flag) and these are presented in order of desirability. Figure 4.3 compares the minimum distance from the programmer type-error of the top two suggestions, compared to the top suggestion by TELE. It shows that

<sup>2</sup>Line 5872

<sup>3</sup>Line 2920

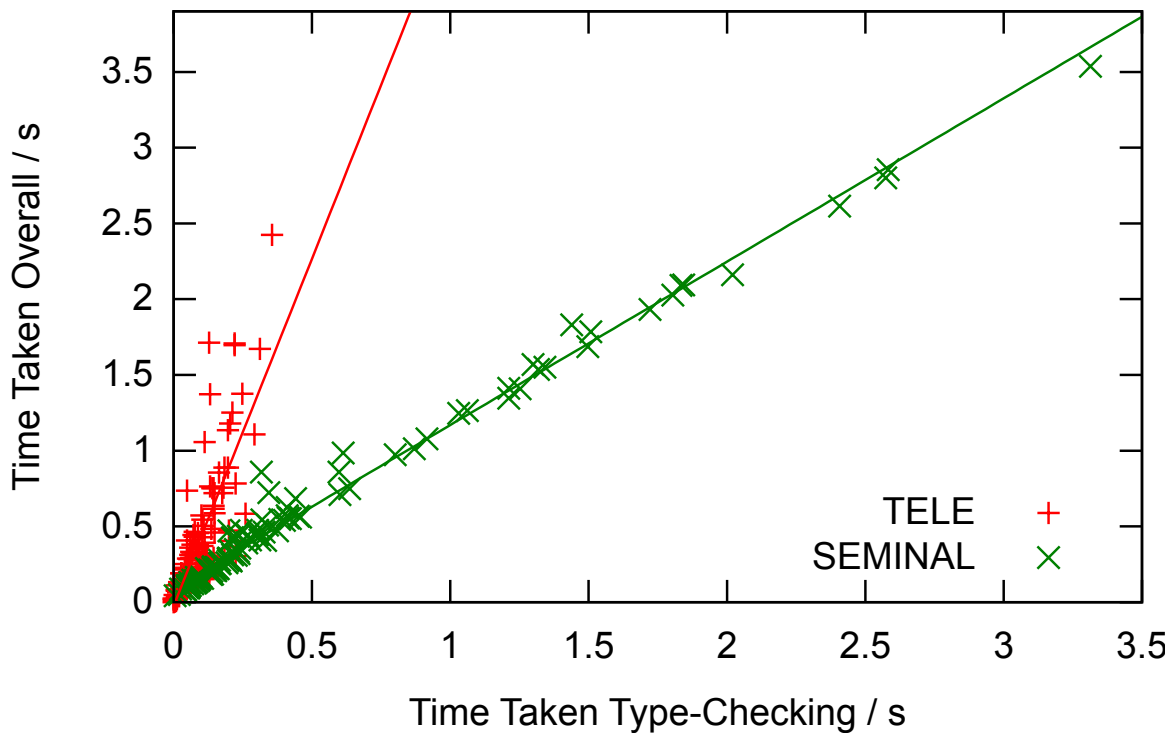


**Figure 4.3:** A comparison of the distance to the programmer error from TELE's top two suggestions

the second ranked type-error message is closer to the programmer type-error than the first a significant number of times. This suggests that the ranking heuristics could be refined to produce a top ranked type-error message that is even closer to the programmer type-error, on average. However, it is not possible to always rank the suggestion that equals the programmer type-error because one programmer may choose to make a small change to correct the type-error whilst another chooses to completely refactor the code.

Another advantage of viewing multiple suggestions is that it helps the programmer to understand why their program is ill-typed, for example, if there are a pair of inconsistent statements and the top two suggestions are to make modifications to each location in the inconsistent pair.

### Comparison of TELE and SEMINAL Execution Time



**Figure 4.4:** Graph comparing the execution time per file with the amount of time spent type-checking

#### 4.2.4 Timing Analysis

From Figure 4.4, 90% of TELE executions take less than one second, with the longest time being 2.4 seconds on my 2.4GHz machine. This is the additional time required over OCaml. This time may be noticeable by a human but as discussed in Section 4.3, the type-error messages are arguably more useful than those produced by OCaml (they also include the OCaml type-error message), so I would expect the programmer to save time debugging type-errors when using TELE instead of OCaml alone (although a user study has not been carried out).

Figure 4.4 shows that SEMINAL (with triage disabled) has a similar distribution of total execution time to TELE with a slightly higher mean. This can be explained by Figure 4.5 because SEMINAL makes a much larger number of calls to the type-checker than TELE. Although SEMINAL takes longer on average than TELE, the relative difference is small compared to the increase in time from using OCaml to TELE or SEMINAL, so is not a major disadvantage of SEMINAL. In

Kind of Node	Average Number of Type-Checker Calls	
	TELE	SEMINAL
Expression	16 ( $\sigma = 7$ )	209 ( $\sigma = 258$ )
Identifier	17 ( $\sigma = 8$ )	224 ( $\sigma = 320$ )
Pattern	10 ( $\sigma = 3$ )	52 ( $\sigma = 36$ )

**Table 4.1:** Table showing the average number of type-checker calls made by TELE and SEMINAL when the programmer error was inside different kinds of node

addition, Figure 4.4 shows that while SEMINAL spends almost all of its execution time type-checking candidate replacements, TELE spends only 25% of the time type-checking candidate replacements. This is because I decided to spend more time calculating which modifications should be attempted than SEMINAL (results in [6] show that SEMINAL’s slow execution time is caused by a large number of type-checker calls), which from Figure 4.5 vastly reduces the number of calls to the type-checker, reducing the overall execution time despite each call to the type-checker taking over two times longer on average than SEMINAL.

Figure 4.5 shows that type-checker call times have a very low variance in SEMINAL with a mean of 0.0024s compared to TELE’s mean of 0.006s. This is because SEMINAL creates a signature for all well-typed bindings so they are not repeatedly type-checked and it truncates the file after the first ill-typed binding [6]. Both of these factors decrease the execution time of the type-checker but produce type-error messages that are on average, further from the programmer type-error, as explained in Section 4.2.2. The variance of SEMINAL’s type-checking time is lower than TELE’s because SEMINAL submits at most one top level binding to the type-checker whereas TELE submits the whole file to the type-checker. Larger files take longer to type-check than smaller files, and Figure 4.5 shows that for a fixed file size, TELE has a low variance in the time taken per type-check, thus the reason for large variance in average type-checking time is the large variance in file sizes.

From Table 4.1, SEMINAL makes many more calls to the type-checker than TELE when the problematic node is an identifier or an expression. This is because SEMINAL:

- attempts enumerations for some nodes before they have been determined interesting [7]<sup>4</sup>, unlike TELE;

---

<sup>4</sup>Line 3528

- does not remove duplicate identifiers from the candidate identifier replacement list [7]<sup>5</sup>, unlike TELE; and
- attempts a larger number of enumerations for each node than TELE [7]<sup>6</sup>.

Both SEMINAL and TELE make significantly fewer calls to the type-checker when the type-error is within a pattern because subtrees rooted at patterns are typically more shallow than those rooted at expressions and few enumerations are attempted (TELE does not make structural pattern replacements and SEMINAL only attempts removal of sub-patterns and insertion of new patterns). This is because a very large number of replacements would be required to attempt all possible structural changes to patterns (e.g. replacing/introducing every constructor) and unlike expressions, there is no accurate way to determine interestingness of patterns (see Section 2.6.1) so these modifications would be attempted for many patterns. The key reason that TELE makes few calls to the type-checker is that the maximum number of type-checker calls is proportional to the depth of the AST because a node is only examined if its parent is interesting.

### 4.3 Qualitative Evaluation

#### 4.3.1 Usefulness of TELE's Output

The example below is taken from the corpus where the student takes six minutes to correct the type-error resulting from a match association error:

```

1 let rec format w k imdList =
2   match imdList with
3     hd::tl      ->
4       match hd with
5         (_,_,DocNil)  -> ...
6         | (i,_,DocGroup d) -> ...
7   | [] -> SNil

```

```

OCaml: This pattern ([]) matches values of type 'a list
       but a pattern was expected which matches values of type
       'b * Pprintdata.mode * Pprintdata.doc

```

<sup>5</sup>Line 2669

<sup>6</sup>Line 4142, SEMINAL attempts  $O(n^2)$  enumerations for each function with  $n$  arguments and recurs separately on the function with one fewer argument, resulting in  $O(n^3)$  enumerations of interesting functions.

TELE: 1. Try modifying the match expression on line 2 to line 6 by adding some clauses to the outer match

2. Try replacing the pattern-expression in match on line 7, with:

```
Pattern type int * Pprintdata.mode * Pprintdata.doc and
body type Pprintdata.sdoc
```

TELE's highest ranked type-error message is more helpful than OCaml in this case because it has identified the correct location of the programmer's type-error and a modification that the programmer implemented to correct the type-error. Sometimes the modification suggested by TELE is in a different location from the modification that the programmer will make to correct the type-error. However, such a type-error message still provides the programmer with information that the code in this location alone is preventing the rest of the program from being well-typed.

### 4.3.2 Unquantified Benefits

As well as producing type-error messages that are closer, on average, to the programmer type-error than OCaml, TELE has the additional benefits over OCaml.

- The type-error generation code from the OCaml type-checker can be removed, decreasing the time required to type-check well-typed programs and increasing the resemblance of the type-checking code to the formal definition of the type system, making it easier to verify correct and maintain.
- The implementation of the type-checker can be modified without producing different type-error messages.
- TELE helps the programmer to understand why the program is ill-typed, rather than why the implementation of type-checking failed. The former is more useful because it relates to the definition of the language rather than its implementation.

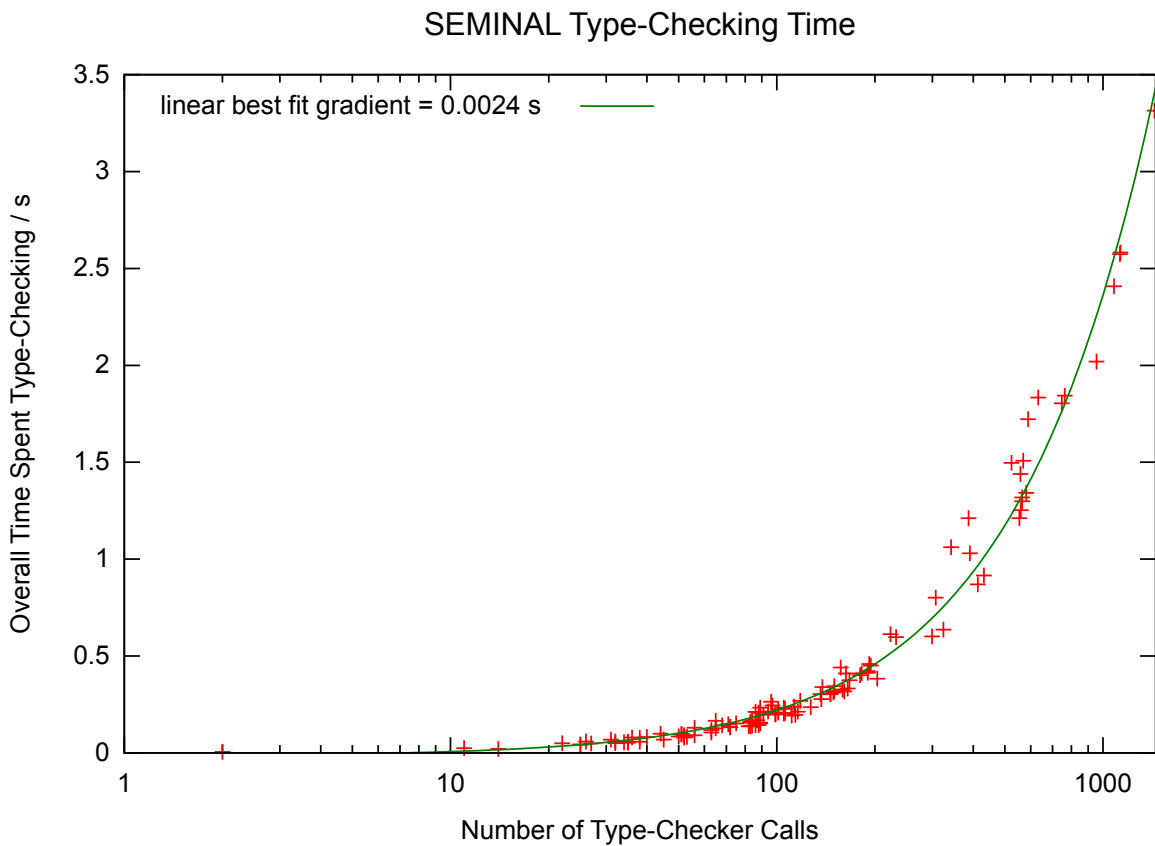
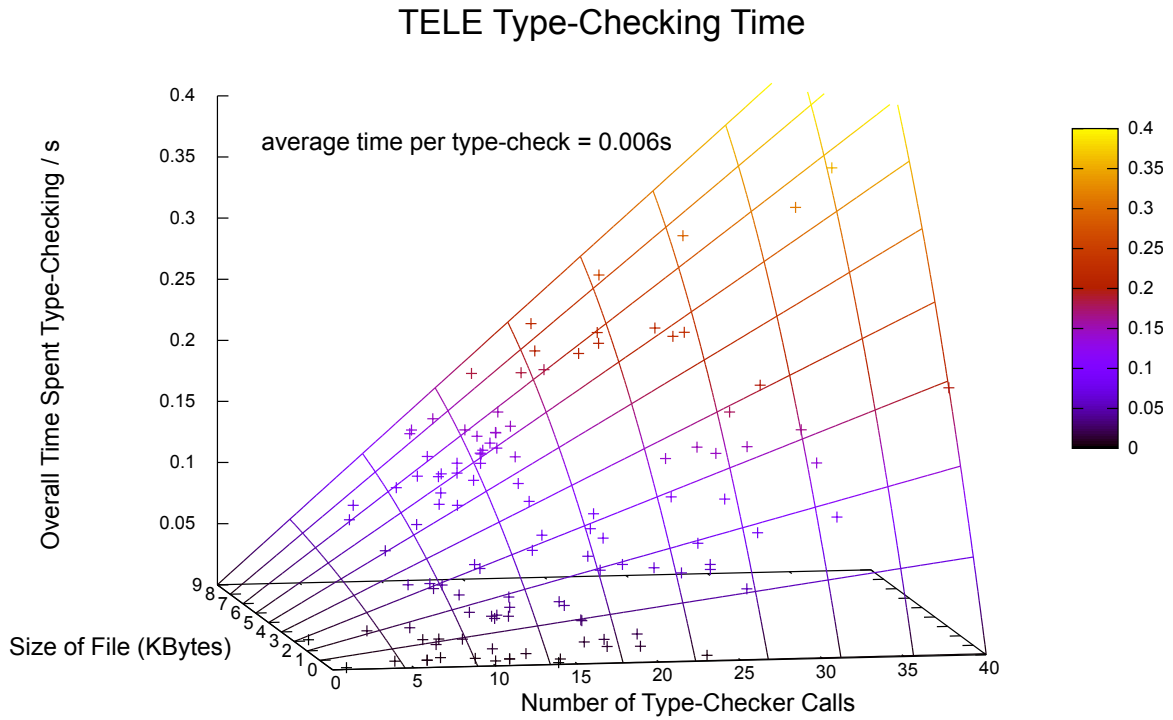
### 4.3.3 Generalising to Programs Outside the Corpus

The evaluation so far has focused on programs that contain a single programmer type-error (defined in Section 4.1). If a program contains multiple, independent programmer type-errors, the deepest node that TELE can report as problematic is the deepest common ancestor of these nodes because TELE does not make simultaneous generic replacements (and enumerator replacements correct errors with multiple nodes that are related e.g. tree rotations). The type-error produced

by TELE is therefore not very useful in this case. To improve performance, TELE could truncate the file after the first ill-typed binding, although when there is only a single type-error, this will decrease accuracy of type-error identification (see Section 4.2.2). This is the approach taken by SEMINAL and OCaml. Of the 964 ill-typed files examined in the corpus, 203 were deemed to contain a single type-error and the rest contained multiple type-errors. This seems to suggest that TELE is only useful approximately 20% of the time, however, the programmers were using the OCaml compiler which stops type-checking after reporting the first type-error in the file, which I believe encourages programmers to program in a way that introduces multiple type-errors more frequently than if they were using TELE.

The programs in the corpus were collected from novice OCaml programmers, so the results are encouraging, but do not definitively show that TELE reports type-errors that are closer to the programmer type-error than SEMINAL or OCaml for all OCaml programs. TELE may have also only been faster and more accurate than SEMINAL because it does not correct as many complicated type-errors (i.e. TELE does not attempt as many enumerations as SEMINAL). However, a large contributor to TELE's increased speed is having a smaller search space than SEMINAL (by only enumerating interesting nodes). In addition, whilst attempting more enumerations does make selecting the most desirable modification more difficult, TELE clearly improves on SEMINAL's ranking heuristic by evaluating the size of the modification by analysing the AST rather than the range in source code.

On balance, I would suggest using TELE alongside OCaml because TELE alone produces poor type-error messages when multiple type-errors are present but OCaml produces poor quality type-error messages compared to TELE when there is a single type-error or the programmer has made a common error. In addition, I believe that although the execution time of TELE may be noticeable, the overall time for the programmer to correct type-errors would be smaller when using TELE instead of OCaml, although a user study has not been carried out.



**Figure 4.5:** Graphs comparing the time for each call to the type-checker and the distribution of the number of type-checker calls

# Chapter 5

## Conclusion

### 5.1 Was the Project a Success?

From the project proposal, the project would be deemed a success if the modified compiler:

1. reports type-errors with a location closer (with respect to distance between AST nodes) to the programmer error than the existing compiler for programs that do not contain multiple, independent type-errors.

Status: **Achieved**. From Figures 4.2 and 4.3, the type-errors reported by TELE are, on average, 1.7 nodes away from the programmer type-error compared with OCaml's average of 2.8 nodes. The ill-typed programs used to produce this data were collected from 7 assignments and 11 students who were new to OCaml but not to programming. The results suggest, but do not definitively prove that TELE achieves a closer average distance to the programmer type-error for programs with a single type-error written by programmers with any level of expertise.

2. must take no more than a few seconds longer on a modern machine than the existing compiler for the majority of ill-typed programs in the set collected.

Status: **Achieved**. From Figure 4.4, 90% of all execution times are less than a second with the largest execution time in 120 files being 2.4 seconds on my 2.4GHz machine.

Furthermore, and perhaps surprisingly, TELE achieved a closer than average distance to the programmer type-error, quicker execution time and significantly

fewer calls to the type-checker than SEMINAL by making several different design decisions, as explained in Section 4.2.2 and 4.4.

As discussed in Section 4.3.3, I would suggest using TELE alongside OCaml instead of replacing it because of OCaml's high quality of error reporting when there are multiple, independent type-errors and TELE's high quality type-error messages when there is a single type-error. Furthermore, TELE is particularly useful as a tool for learners of the OCaml language because it specifically targets type-errors that are easily made because of the design of the language (such as using `()` instead of `[]`) and provides constructive error messages that describe why the program is ill-typed rather than why the type-checking implementation failed. SEMINAL may be more useful than TELE for experienced OCaml programmers because it attempts a wider range of modifications than those which correct mistakes that are easily made by novices.

## 5.2 Further Work

In order to produce type-error messages that are useful even when there are multiple, independent type-errors present, I would have liked to have implemented a triage feature (present in SEMINAL) to be able to modify multiple nodes at once by considering interestingness as a property of sets of nodes. Furthermore, I would have liked to have investigated the reduction in time for type-checking well-typed programs when the type-error generation code is removed from the OCaml type-checker and carried out a user study to see how the overall time taken for a programmer to correct a type-error is affected by using TELE instead of OCaml. To improve the ranking heuristics further, machine learning techniques could be used to fit the heuristics to the corpus (by minimising the average distance) or an interactive interface could be built for the programmer to identify the most helpful suggestion, allowing the heuristics to adapt in real time to the programmer.

## 5.3 Summary

In this project, I have become familiar with the OCaml language and compiler implementation, shell programming and Unix development. I have discovered several interesting properties of the OCaml type system (Sections 2.6.1 and 3.2.2) and successfully addressed the problem of poor quality type-error messages in OCaml by producing type-error messages that explain why type-inference fails instead of why the implementation of the formal type-system fails.

# Bibliography

- [1] Karen L. Bernstein and Eugene W. Stark. Debugging type errors. Technical report, State University of New York, Stony Brook, 1995.
- [2] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 193–204, New York, NY, USA, 2001. ACM.
- [3] Luis 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.
- [4] INRIA SICS Diffusion. OCaml compiler source code, version 3.12.0. <http://caml.inria.fr/download.en.html>, 08-02-2010.
- [5] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.
- [6] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. *SIGPLAN Not.*, 42(6):425–434, 2007.
- [7] Benjamin S. Lerner, Dan Grossman, and Craig Chambers. SEMINAL source code, version 0.0.3. [http://www.cs.washington.edu/homes/blerner/papers/seminal\\_prototype.html](http://www.cs.washington.edu/homes/blerner/papers/seminal_prototype.html), 19-01-2009.
- [8] Bruce McAdam. *Trends in functional programming*, pages 87–98. Intellect Books, Exeter, UK, 2002.
- [9] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 15–26, New York, NY, USA, 2003. ACM.

- [10] Scott Owens. A sound semantics for OCaml light. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems, ESOP'08/ETAPS'08*, pages 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11] Rossitza Setchi, Ivan Jordanov, Robert J. Howlett, and Lakhmi C. Jain, editors. *Knowledge-Based and Intelligent Information and Engineering Systems - 14th International Conference, KES 2010, Cardiff, UK, September 8-10, 2010, Proceedings, Part I*, volume 6276 of *Lecture Notes in Computer Science*. Springer, 2010.

**Appendix A**

**Project Proposal**

Shaun Hall  
Churchill College  
sh580

Computer Science Tripos Part II Project Proposal

## Searching for the Source of Caml Type-Errors

*12/10/10*

**Project Originator:** *Shaun Hall*

**Project Supervisor:** *John Wickerson*

**Signature:**

**Director of Studies:** *John Fawcett*

**Signature:**

**Overseers:** *Ann Copestake and Robert Harle*

**Signatures:**

# 1 Introduction and Description of the Work

Whilst type inference reduces the need for the programmer to annotate programs with types, it often leads to inscrutable error messages for ill-typed programs. Typically, the reported location of the error is far from the programmer error that caused type inference to fail. The error message is based on how the type checker works and constructive suggestions can rarely be provided.

The project will combat these issues for the Caml language by using a search based procedure that aims to report the location of the cause of the type-error, as well as one or more suggestions at this location that would cause the program to pass the type-check. A key extension is to be able to report multiple, independent type-errors.

Existing work in this area focuses on making changes to the Hindley-Milner type inference algorithm by constructing explanation graphs [2], implementing program slicing to find sets of inconsistent statements [3], extending type inference rules to operate on assumption environments that assign types to variables [1], using discriminative sum types [6] or checking if inference can succeed modulo linear type isomorphisms [5].

The problem with these approaches are that they have made the type inference algorithm:

1. harder to maintain, because it has been modified for error-message generation;
2. more difficult to ensure correct since the increased complexity needed for message generation has reduced the correspondence between the type checking code and the mathematical definition of the type system; and
3. slower to execute for well-typed and ill-typed programs since the extra error checking code is executed in both cases.

One possible option to solve the last two problems is to maintain two type-checking programs, one that has a close correspondence with the underlying mathematics and one that contains code to generate more desirable type-error messages that is called only when type inference using the former fails. However, this adds an extra degree of difficulty in maintaining the type-checkers since there are now two implementations and in addition, it becomes difficult to ensure that both type checkers are correct (that the error message generation checker will never deduce a type for a program that the other type checker deems ill-typed).

The project will follow the approach described in [4], where all three of the problems above are combated by intercepting the type-error message produced by the OCaml compiler when a program is ill-typed and initiating a search procedure that uses the type-checker on candidate changes to see if they are well-typed.

This compiler architecture consists of a *changer*, which takes an ill-typed AST (so is located after parsing and before type-annotation of the AST) and makes small changes that may

make the program well-typed, the *type-checker*, which is (mainly) being used by the changer to see if the modified programs are well-typed, and the *ranker* which prioritizes well-typed modifications and presents them appropriately to the user. In addition, the project hopes to have better performance than the compiler in [4], where performance was limited by the number of calls made to the type-checker, by spending more time analysing results from previously attempted changes.

## 2 Resources Required

The University of Washington has kindly provided 17MB of data collected from students who participated in a Caml course. The data consists of programs that were ill-typed (but parsing succeeded) from ten students' work on five assignments, who were using the standard Caml compiler. Each successive compilation attempt is timestamped, allowing the cause of each type-error to be determined manually by examining later compilation attempts. The assignments are also the same for all students, which can be used as an additional tool to determine if the project succeeds in locating the source of the type-error.

## 3 Starting Point

The existing OCaml compiler will be used as the starting point of the project, and an additional module will be written in the OCaml language that intercepts type-error messages and initiates the search procedure. Additional modifications may be required to allow presentation of the type-error messages generated. The project draws on material from the courses *Foundations of Computer Science*, *Compilers* and *Types*.

## 4 Substance and Structure of the Project

The aim of the project is to produce a modified Caml compiler that uses a search based OCaml compiler extension written in the OCaml language (as a module) that accepts Caml programs as input and produces type-error messages with more accurate locations and helpful suggestions than the existing compiler, with unchanged behaviour for well-typed programs.

The project has the following components:

1. the *changer*: takes an untyped AST that fails to pass the type-check and produces candidate ASTs that may cause type inference to succeed. This consists of a *searcher* and an *enumerator*.

2. the *searcher*: given an untyped AST that fails to pass the type-check, identifies nodes that cause the type-check to fail. Works by replacing nodes with a `raise Exception` node (which is always typeable) and seeing if the result becomes well-typed. To efficiently find the most localised node, the search will start from the root and work towards the leaves.
3. the *enumerator*: given an untyped AST node from the searcher, produces a list of modifications to that node which may cause type inference to succeed. For example, given the AST node for `f e1 e2 e3`, the enumerated changes will include `f e1 e3`, `f (e1, e2, e3)` and `f (e1 e2) e3`.
4. the *type-checker*: an unmodified type-checker from the OCaml compiler that is used by the searcher to see if the modifications by the enumerator were successful or not. The result is used to guide further search.
5. the *ranker*: this receives well-typed ASTs from the searcher and presents them to the user in the form of suggestions, in order of desirability according to some heuristics.

This modular structure yields a highly testable and maintainable compiler extension.

## 5 Success Criteria

The compiler produced must firstly, on average, produce type-errors with a location closer (with respect to distance between AST nodes) to the programmer error than the existing compiler for programs that do not contain multiple, independent type-errors. Multiple compilation attempts with the same type-error are counted only once and the first occurrence of the type-error is used as the representative. Secondly, the compiler produced must take no more than a few seconds longer on a modern machine than the existing compiler for the majority of ill-typed programs in the set collected.

## 6 Evaluation Strategy

The first part of the quantitative evaluation will comprise a comparison of the quality of error messages produced by the standard Caml compiler against those produced by the extended compiler implemented, per assignment, per student. The distance between the reported node and the node that is actually causing the error will be measured to implement this comparison. As mentioned in Section 2, the node actually causing the error will be found by manual inspection of the source code.

The second part comprises an observation of how error message location accuracy / time to generate the error message is affected by changing ranking heuristics, enumeration or

searching strategies. Also, compilation time with and without additional features (e.g. triage) will be compared.

Time permitting, the third part comprises a comparison of the kind of error in the AST node against the time taken to discover the error and number of calls made to the type-checker in the compiler implemented and the compiler implemented in [4] (will need to obtain benchmark programs from the authors to achieve this).

## 7 Extensions

1. Implement *trriage* to detect multiple, independent type-errors.
2. Reduce calls to the type-checker by implementing lazy evaluation of candidate ASTs and ordering candidate changes to reduce redundant calls (ASTs that are covered by more general ASTs or finding candidates that will be ranked lower than already successful changes).
3. Modify the Caml type-checker so that it does not contain any error generation code. Compare the speed of the extension implemented above this type checker against the original compiler for a range of well-typed programs.

## 8 Backup Strategy

All source files used in the project will be backed up weekly onto a local SD card, changes will be committed every day a change is made onto a PWF subversion server.

## 9 Timetable and Milestones

### Fortnight beginning 07/10/10

Research OCaml language and write some test modules. Write project proposal.

**Milestone:** Have written at least one OCaml module and submitted the project proposal.

### Fortnight beginning 21/10/10

Research and enumerate a suggested list of modifications for each kind of AST node in the Caml language. Research possible heuristics to determine desirability of modifications for each kind of node. Research the OCaml compiler architecture.

**Milestone:** Have a list of suggested modifications for every kind of Caml AST node along

with different ways these could be ranked. Understand the composition of the OCaml compiler and interactions between different components.

### **Fortnight beginning 04/11/10**

Design detailed architecture of compiler extension and integration with the OCaml compiler and research more complex aspects of the OCaml language.

**Milestone:** Have an architecture diagram and detailed explanation of interaction of the OCaml compiler with the search extension. Interfaces between internal components are defined.

### **Fortnight beginning 18/11/10**

Modify the OCaml compiler and implement a basic OCaml module to intercept type-error messages for ill-typed programs and have no effect on the compilation of well-typed programs.

**Milestone:** Have a modified OCaml compiler that makes a simple modification to the type-error generated and has unchanged behaviour for well-typed programs.

### **Fortnight beginning 02/12/10**

Implement the searcher, enumerator (based on work in the second fortnight) and a primitive ranker with easily modifiable heuristics for ranking and enumeration order. Write some simple unit tests to verify the implementations and enable regression testing later.

**Milestone:** Have a working searcher, enumerator and ranker that implement the interfaces defined in the fortnight beginning 04/11. Successful unit tests have been written for the searcher, enumerator and ranker.

### **Fortnight beginning 16/12/10**

Integrate the module with the OCaml compiler to produce new type-error messages for ill-typed programs. Prepare demonstration for progress meeting in January. Write progress report.

**Milestone:** Have a working extension to the OCaml compiler that produces a range of type-error messages with replacement suggestions. Have produced the progress report and working demonstration.

## Fortnight beginning 30/12/10

Slack time in case the progress report or demo are unfinished from the fortnight beginning 16/12. Research appropriate heuristics for the enumerator, ranker and searcher.

**Milestone:** Have a good awareness of the various heuristics that could be implemented for the enumerator, ranker and searcher and their relative strengths/weaknesses.

## Fortnight beginning 13/01/11

Devise ranking, enumeration and searching heuristics to improve location of the type-errors reported, using examples from the corpus of ill-typed programs. Time permitting, implement lazy evaluation of changes to reduce the number of calls to the type-checker.

**Milestone:** Have fulfilled the first success criterion (shorter average AST distance from reported error to actual error) for programs in the corpus that do not contain multiple, independent type-errors.

## Fortnight beginning 27/01/11

Slack time in case the enumerator, searcher or ranker are unfinished (or new bugs have been introduced that are revealed from running the unit tests from fortnight 02/12). Time permitting, implement triage to be able to detect and report multiple, independent type-errors. Compile the corpus of Caml programs with this compiler and fix noticeable bugs encountered in this process.

**Milestone:** All Caml programs in the data set run without noticeable bugs in the compiler and the triage extension has been implemented if there was enough time. Second success criterion satisfied.

## Fortnight beginning 10/02/11

Begin writing dissertation. Draw up structure of the dissertation with sections, sub-headings and short bullet points to cover in each section. This will help make the overall scope of the project more obvious when fully implementing each section.

**Milestone:** Structure of the dissertation is finished, containing the required sections and sub-headings and short bullet point summaries for each section.

## Fortnight beginning 24/02/11

Write Introduction and Preparation sections. Send to supervisor / DoS.

**Milestone:** Introduction and Preparation sections are complete.

## Fortnight beginning 10/03/11

Write the Implementation section. Implement suggestions from supervisor / DoS, as appropriate. Send latest edition to supervisor / DoS.

**Milestone:** The Implementation section is complete.

## Fortnight beginning 24/03/11

Slack time for finishing Introduction, Preparation and Implementation sections. Collect numerical data required for the evaluation.

**Milestone:** Introduction, Preparation and Implementation sections are complete and all numerical data has been collected for the evaluation.

## Fortnight beginning 07/04/11

Write up Evaluation. Send latest edition to supervisor / DoS. Implement the suggestions given from supervisor / DoS, as appropriate.

**Milestone:** Evaluation section and draft of dissertation are complete.

## Fortnight beginning 21/04/11

Focus on revision for examinations. Make final adjustments after feedback from the project supervisor. Send to supervisor / DoS and get the dissertation bound and printed after receiving approval from supervisor / DoS.

**Milestone:** Dissertation is finished, printed and bound, provided this has been agreed with supervisor / DoS.

## Fortnight beginning 05/05/11

Slack time for implementing any major suggestions from supervisor / DoS e.g. collecting more data and evaluating it. Get final version approved from supervisor / DoS.

**Milestone:** Dissertation is approved, finished, printed, bound and handed in before 20/05/11.

## References

- [1] Karen L. Bernstein and Eugene W. Stark. Debugging type errors. Technical report, 1995.

- [2] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 193–204, New York, NY, USA, 2001. ACM.
- [3] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.
- [4] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. *SIGPLAN Not.*, 42(6):425–434, 2007.
- [5] Bruce McAdam. How to repair type errors automatically. pages 87–98, 2002.
- [6] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 15–26, New York, NY, USA, 2003. ACM.