UNIVERSITY OF CALIFORNIA, SAN DIEGO

Data-Driven Techniques for Type Error Diagnosis

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Eric Lee Seidel

Committee in charge:

Professor Ranjit Jhala, Chair
Professor William Griswold
Professor Philip Guo
Professor James Hollan
Professor Sorin Lerner

2017

The Dissertation of Eric Lee Seidel is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

EPIGRAPH

Types are the leaven of computer programming;
They make it digestible.

Robin Milner

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am indebted to a great many people for helping me through these five years.

First, I want to thank Ranjit Jhala for being such a great advisor. He has always been supremely supportive and is brimming with ideas for interesting projects, even when you feel like you have none of your own. He also has a brilliant ability to force you to keep trying (usually failing) to explain your ideas, until finally he decides to step in and summarize what you've been struggling to say with the utmost clarity and precision. Finally, I want to thank Ranjit for pushing me to investigate using machine learning to predict the source of type errors; I was quite reluctant at first, but it turned out to be a very enlightening and productive direction!

I also thank Bill Griswold, Philip Guo, Jim Hollan, and Sorin Lerner for serving on my committee. In particular, I want to thank Jim for inviting me to his research group meetings and Bill for reading so many papers with me in my early years here. Thanks also to Ingolf Krueger, who enabled this great experience by recruiting me to UC San Diego in the first place.

Next, I want to thank all of my collaborators. Wes Weimer seems to have an endless supply of insightful questions that helped guide my research, and he also provided invaluable assistance in designing and running our user studies. Kamalika Chaudhuri and Huma Sibghat helped me quickly get up to speed in the foreign field of machine learning, and ensured that our models and experiments were on sound footing.

I want to thank my many labmates in the PL group at UCSD for making my time here so enjoyable. In particular, I thank Alexander Bakst, Valentin Robert, Zach Tatlock, and Niki Vazou for welcoming me into the group and providing a supportive environment.

I also want to thank Gabrielle Allen and Douglas Troeger for being early mentors during my undergraduate career. Gab invited me to work with her research group at LSU, which made me realize that I actually do enjoy programming when I have a purpose, and led to three publications before I even started grad school. Dr. Troeger inspired and nurtured my interest in functional programming with his Programming Paradigms course at CCNY. I surely wouldn't have even applied to grad school without them.

Last but not least, I want to thank my family for being supportive throughout this whole

experience. Grad school can be very difficult at times, to put it lightly, but my parents, my sister, and my wife Megan never wavered in their support and confidence that I could persevere.

Published Materials Adapted for this Dissertation

Chapter 1 contains material adapted from the following publications: E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In ICFP '16, 2016; E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *In submission to J. Funct. Programming*, 2017; and E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *In submission to OOPSLA '17*, 2017. The dissertation author was the primary investigator and author of these papers.

Chapter 3, in part, is a reprint of the material as it appears, or may appear, in: E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In ICFP '16, 2016; and E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *In submission to J. Funct. Programming*, 2017. The dissertation author was the primary investigator and author of these papers.

Chapter 4, in part, has been submitted for publication of the material as it may appear in: E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *In submission to OOPSLA '17*, 2017. The dissertation author was the primary investigator and author of this paper.

VITA

2012	Bachelor of Science, City College of New York
2016	Master of Science, University of California, San Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In ICFP '16, 2016

T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt free ivory. In Haskell '15, 2015

E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In ESOP '15, 2015

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In ICFP '14, 2014

N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In Haskell '14, 2014

E. L. Seidel. Metadata management in scientific computing. *JOCSE*, 3(2), 2012

W. L. Khoo, E. L. Seidel, and Z. Zhu. Designing a virtual environment to evaluate multimodal sensors for assisting the visually impaired. In ICCHP '12, 2012

G. Allen, F. Löffler, E. Schnetter, and E. L. Seidel. Component specification in the cactus framework: the cactus configuration language. In GRID '10, 2010

E. L. Seidel, G. Allen, S. Brandt, F. Löffler, and E. Schnetter. Simplifying complex software assembly: the component retrieval language and implementation. In TG '10, 2010

ABSTRACT OF THE DISSERTATION

Data-Driven Techniques for Type Error Diagnosis

by

Eric Lee Seidel

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Ranjit Jhala, Chair

Static type systems are a powerful tool for reasoning about the safety of programs. Global type inference eliminates one of the prime complaints against static types, that the annotation burden is too high. However, this introduces its own problems as the type checker must now make assumptions about what the programmer intended to do. A single incorrect assumption can lead the type checker to erroneously blame an expression far from the actual error the programmer made, which can be particularly confusing for newcomers who have not yet constructed a mental model for how the type checker works.

In this dissertation we present a pair of complementary techniques to *localize* and *explain* type errors, with an emphasis on the errors encountered by novice users.

We tackle the localization problem by using machine learning to learn a model of the errors made by students in an introductory course. Then, we use the model to produce a ranked list of likely error locations in new programs. Our models can be trained on a modest amount of data, *e.g.* a single instance of a course, and we envision a future where each introductory course is accompanied by a model of its students' errors.

To better explain the error to novice users, we present a runtime error that the type system would have prevented. We interleave type-checking and execution to search for a set of program inputs that would lead execution to a bad state, and present the execution trace to the user in an interactive debugger. This allows the user to explore why their program was rejected, and connects the dynamic (runtime) semantics to the static (typing) semantics.

We have evaluated our techniques empirically using a new dataset of ill-typed student programs collected from two instances of an undergraduate programming languages course at UC San Diego. We have also performed user studies with novice users, comparing the output of our techniques with the state of the art in type error diagnosis. Our results show that these are practical, lightweight techniques for improving the error messages produced by type checkers.

Chapter 1

Introduction

Static type systems are a marvelous invention. They allow programmers to rule out, at compile-time, entire classes of run-time failures, ranging from the mundane but ubiquitous null-pointer dereference to the construction of invalid SQL queries [22, 58] and beyond. Languages like OCaml and Haskell make the value-proposition for types even more appealing by automatically synthesizing the types for all program terms, without troubling the programmer for any annotations. Unfortunately, this automation comes at a price. Type annotations signify the programmer's intent, and help to correctly blame the erroneous sub-term when the code is ill-typed. In the absence of such signifiers, automatic type inference algorithms are prone to report type errors far from their source [111]. While this can seem like a minor annoyance to veteran programmers, Joosten et al. [46] have found that novices often focus their attention on the *location* reported and disregard the *message*.

In this dissertation we present two new, complementary techniques designed to help *localize* and *explain* type errors, drawing inspiration from the fields of automated program testing and machine learning. In the rest of this chapter we will set the stage for our contributions. First, we will motivate the program of type error diagnosis with a simple example, and will review the state of the art in type error diagnosis. Then, we will outline our novel contributions to the field.

```
1 let rec sumList xs =
2   match xs with
3   | []   -> []
4   | h::t -> h + sumList t
```

```
This expression has type 'a list
but an expression was expected of type int
```

Figure 1.1. (left) An ill-typed OCaml program that should sum the elements of a list, highlighting the location blamed by the OCaml compiler. (right) The error reported by OCaml.

1.1 A Running Example

Consider the OCaml program in Figure 1.1, which is supposed to sum the integers in a list. This program was written by an undergraduate student at UC San Diego, and works as follows. In functional languages like OCaml, lists are recursively defined as either the empty list (written `[]` and pronounced "nil"), or a single element `h` followed by the rest of the list `t` (written `h::t` and pronounced "h cons t")[1]. Given an input `xs`, the student `match`es it against the two forms a list can take. In the `[]` case she returns another empty list `[]`, and in the `h::t` case she adds `h` to the recursive sum of `t`.

The observant reader will notice that this program is incorrect, given *any* non-empty list of integers, the addition on line 4 will attempt to add an integer to `[]`, which is an invalid operation. In fact, the program is *ill-typed* and the OCaml compiler rejects it with the error message in Figure 1.1. Unfortunately, OCaml's error *blames* the recursive call to `sumList`, explaining that `sumList` returns a `list`, while the `+` operator requires an `int`. The real error is on line 3, where the student returns `[]` rather than `0` as the sum of an empty list.

As we will see throughout the rest of this chapter, this rather simple program is sufficient to illustrate many of the difficulties of automatically locating the source of a type error, and explaining it to the programmer.

1.2 The Hindley-Milner Type System

To illustrate the difficulty of pinpointing the source of type errors, let us consider the λ-calculus in Figure 1.2. In addition to the usual variables, λ-abstractions, and applications, we

[1] These variable names are conventional, `h` stands for "head" and `t` for "tail".

$$
\begin{array}{rrcl}
\textbf{Expressions} & e & ::= & x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid & n \mid e + e \\
& & \mid & [] \mid e :: e \mid \texttt{match}\ e \begin{cases} [] \to e \\ x :: x \to e \end{cases} \\
& n & ::= & 0, 1, -1, \ldots \\
\textbf{Types} & t & ::= & \alpha \mid \texttt{int} \mid t \to t \mid [t] \\
& s & ::= & \forall \overline{\alpha}.t \\
\textbf{Environments} & \Gamma & ::= & \{\overline{x : s}\} \\
\textbf{Substitutions} & \theta & ::= & \{\overline{\alpha \mapsto t}\}
\end{array}
$$

Figure 1.2. A simple λ-calculus with integers and lists.

have equipped the language with integers and lists, so that it can easily represent our `sumList` program. The types of our language are split into *monomorphic* types t and *polymorphic* types s.

To describe type inference, we will also need type environments and substitutions. A type environment Γ is a mapping from program variables x to (polymorphic) types s, written $\{\overline{x : s}\}$, and denotes that each variable x_i has a quantified type s_i. Type environments are used to propagate typing information from outer expressions to inner expressions. Types and type environments may have *free* type variables. The free variables of a type, written $FV(s)$ and defined in Figure 1.3, are all type variables that are not quantified over. The free variables of a type environment is the union of the free variables of the types it contains. A substitution is a mapping from *type* variables α to types t, written $\{\overline{\alpha \mapsto t}\}$ and defined in Figure 1.3, and denotes that each type variable α_i has been *refined* to a type t_i. A substitution may be applied to a type or type environment, written $\theta(s)$, and replaces free variables by the corresponding type. The application of a substitution may similarly be lifted to operate over type environments. The composition of two substitutions, written $\theta_2\theta_1$, denotes the substitution formed by first applying θ_1 and then θ_2. Substitutions are used to propagate typing information from inner expressions back to the outer expressions.

OCaml's type system, like many other typed functional languages, is based on the Hindley-Milner type system [44, 70], which we have extended to our language in Figure 1.4.

$$
\begin{array}{rcl}
FV(\alpha) & = & \{\alpha\} \\
FV(\texttt{int}) & = & \varnothing \\
FV(t_1 \to t_2) & = & FV(t_1) \cup FV(t_2) \\
FV(\forall \overline{\alpha}.t) & = & FV(t) - \overline{\alpha}
\end{array}
$$

$$
\begin{array}{rcl}
\theta(\alpha) & = & \begin{cases} t & \text{if } \alpha \mapsto t \in \theta \\ \alpha & \text{otherwise} \end{cases} \\
\theta(\texttt{int}) & = & \texttt{int} \\
\theta(t_1 \to t_2) & = & \theta(t_1) \to \theta(t_2) \\
\theta(\forall \overline{\alpha}.t) & = & \{\alpha' \mapsto t' \mid \alpha' \notin \overline{\alpha}\}(t)
\end{array}
$$

Figure 1.3. Free variable computation and application of substitutions.

The type system is written as a set of inference rules for typing judgments of the form $\Gamma \vdash e : t$, which can be read as "in the environment Γ, the expression e has type t."

As written, this system cannot be used as an algorithm for computing the type of an expression. For instance, consider the Lam rule. The premise says that e should have type t_2 in an environment where x has type t_1, but where did t_1 come from? We have no way of knowing at this point what choice of t_1 will lead to a successful type inference, and trying all types is not an option as there are an infinite number. Thus, a type inference algorithm must defer the choice of t_1 until it has examined the body e to determine how x is used.

The traditional Damas-Milner Algorithm $\mathcal{W}$ [25], extended to our language in Figure 1.5, solves this issue by binding x to a fresh type variable α (*i.e.* one that does not occur in the environment). It takes as input a typing environment Γ and an expression e, and returns a substitution θ and an inferred type t. In the λ case, it then applies the substitution to the function type $\alpha \to t$ before returning it, crucially refining α based on the body of the lambda.

A key component of Algorithm $\mathcal{W}$ is Robinson's unification algorithm, $\mathcal{U}$ [90], which takes two types t_1 and t_2 and returns a substitution θ such that $\theta(t_1) = \theta(t_2)$. For example, $\mathcal{U}(\alpha_1 \to \texttt{int},\ [\texttt{int}] \to \alpha_2) = \{\alpha_1 \mapsto [\texttt{int}],\ \alpha_2 \mapsto \texttt{int}\}$. It is the combination of fresh type variables to defer the choice of concrete types and Robinson's $\mathcal{U}$ to instantiate them that allows $\mathcal{W}$ to efficiently compute the type of any expression without any user annotations.

Unfortunately, that is also precisely the source of the poor error messages associated with type inference. Unification is not guaranteed to succeed, *e.g.* there is no substitution of type variables that would unify `int` and `[int]`. $\mathcal{W}$ traverses the program from the bottom up, collecting typing constraints at each expression, and halts with an error when it detects an

Typing $\boxed{\Gamma \vdash e : t}$

$$\textsc{Var}\frac{\Gamma(x) = \forall \overline{\alpha}.t \quad \overline{\alpha'} \text{ are fresh}}{\Gamma \vdash x : \{\overline{\alpha \mapsto \alpha'}\}(t)} \qquad \textsc{Lam}\frac{\Gamma \cup \{x : t_1\} \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2}$$

$$\textsc{App}\frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2} \qquad \textsc{Let}\frac{\Gamma \vdash e_1 : t_1 \quad \overline{\alpha} = FV(t_1) - FV(\Gamma) \quad \Gamma \cup \{x : \forall \overline{\alpha}.t_1\} \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

$$\textsc{Lit}\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \textsc{Plus}\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\textsc{Nil}\frac{\alpha \text{ is fresh}}{\Gamma \vdash [] : [\alpha]} \qquad \textsc{Cons}\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : [t]}{\Gamma \vdash e_1 :: e_2 : [t]}$$

$$\textsc{Match}\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \cup \{x_1 : t_1,\ x_2 : [t_1]\} \vdash e_3 : t_2}{\Gamma \vdash \texttt{match}\ e_1 \begin{cases} [] \to e_2 \\ x_1 :: x_2 \to e_3 \end{cases} : t_2}$$

Figure 1.4. A Hindley-Milner-style type system for the language in Figure 1.2.

inconsistent constraint during a call to $\mathcal{U}$. Thus, the placement of calls to $\mathcal{U}$ determines which expression will be blamed when a program is ill-typed.

Given our `sumList` program $\mathcal{W}$ will infer that the `[]` case returns a list while the `h::t` case returns an integer, which violates the Match rule's constraint that both branches must have the same type t_2. This violation will manifest as an error in the second $\mathcal{U}$ call in $\mathcal{W}$'s `match` case, and $\mathcal{W}$ will thus blame the entire `match` expression even though the error was in the base case.

In this case $\mathcal{W}$'s error report is actually not so bad, a well-written error message could convey that the error is due to the two branches having different types, which may be sufficient to isolate the source in the base case. But in general, this behavior of bubbling up of typing constraints from the leaves of the program can produce very poor errors [57, see Fig. 1 for a particularly pathological example].

$$
\begin{array}{lcl}
\mathcal{W} & : & \Gamma \times e \to \theta \times t \\
\mathcal{W}(\Gamma,\ x) & = & \text{let } \Gamma(x) = \forall\overline{\alpha}.t \\
 & & \text{in } (\varnothing,\ \{\overline{\alpha \mapsto \alpha'}\}(t)), \quad \overline{\alpha'} \text{ are fresh} \\
\mathcal{W}(\Gamma,\ \lambda x.e) & = & \text{let } (\theta,\ t) = \mathcal{W}(\Gamma \cup \{x : \alpha\}, e), \quad \alpha \text{ is fresh} \\
 & & \text{in } (\theta,\ \theta(\alpha \to t)) \\
\mathcal{W}(\Gamma,\ e_1\ e_2) & = & \text{let } (\theta_1,\ t_1) = \mathcal{W}(\Gamma,\ e_1) \\
 & & \quad (\theta_2,\ t_2) = \mathcal{W}(\theta_1(\Gamma),\ e_2) \\
 & & \quad \theta_3 = \mathcal{U}(\theta_2(t_1),\ t_2 \to \alpha), \quad \alpha \text{ is fresh} \\
 & & \text{in } (\theta_3\theta_2\theta_1,\ \theta_3(\alpha)) \\
\mathcal{W}(\Gamma,\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \text{let } (\theta_1,\ t_1) = \mathcal{W}(\Gamma,\ e_1) \\
 & & \quad \overline{\alpha} = FV(t_1) - FV(\Gamma) \\
 & & \quad \Gamma' = \Gamma \cup \{x_1 : \forall\overline{\alpha}.t_1\} \\
 & & \quad (\theta_2,\ t_2) = \mathcal{W}(\theta_1(\Gamma'),\ e_2) \\
 & & \text{in } (\theta_2\theta_1,\ t_2) \\
\mathcal{W}(\Gamma,\ n) & = & (\varnothing,\ \texttt{int}) \\
\mathcal{W}(\Gamma,\ e_1 + e_2) & = & \text{let } (\theta_1,\ t_1) = \mathcal{W}(\Gamma,\ e_1) \\
 & & \quad (\theta_2,\ t_2) = \mathcal{W}(\theta_1(\Gamma),\ e_2) \\
 & & \quad \theta_3 = \mathcal{U}(\theta_2(t_1),\ \texttt{int}) \\
 & & \quad \theta_4 = \mathcal{U}(t_2,\ \texttt{int}) \\
 & & \text{in } (\theta_4\theta_3\theta_2\theta_1,\ \texttt{int}) \\
\mathcal{W}(\Gamma,\ [\,]) & = & (\varnothing,\ [\alpha]), \quad \alpha \text{ is fresh} \\
\mathcal{W}(\Gamma,\ e_1 :: e_2) & = & \text{let } (\theta_1,\ t_1) = \mathcal{W}(\Gamma,\ e_1) \\
 & & \quad (\theta_2,\ t_2) = \mathcal{W}(\theta_1(\Gamma),\ e_2) \\
 & & \quad \theta_3 = \mathcal{U}(t_2,\ \theta_2([t_1])) \\
 & & \text{in } (\theta_3\theta_2\theta_1,\ \theta_3(t_2)) \\
\mathcal{W}(\Gamma,\ \texttt{match}\ e_1 \begin{cases} [\,] \to e_2 \\ x_1 :: x_2 \to e_3 \end{cases}) & = & \text{let } (\theta_1,\ t_1) = \mathcal{W}(\Gamma,\ e_1) \\
 & & \quad \theta_2 = \mathcal{U}(t_1,\ [\alpha]),\ \alpha \text{ is fresh} \\
 & & \quad (\theta_3,\ t_2) = \mathcal{W}(\theta_2(\Gamma),\ e_2) \\
 & & \quad \Gamma' = \Gamma \cup \{x_1 : \alpha,\ x_2 : [\alpha]\} \\
 & & \quad (\theta_4,\ t_3) = \mathcal{W}(\theta_3\theta_2(\Gamma'),\ e_3) \\
 & & \quad \theta_5 = \mathcal{U}(t_3,\ \theta_4(t_2)) \\
 & & \text{in } (\theta_5\theta_4\theta_3\theta_2\theta_1,\ \theta_5(t_3))
\end{array}
$$

Figure 1.5. Algorithm $\mathcal{W}$, adapted to our language.

1.3 Prior Work on Diagnosing Type Errors

Algorithm $\mathcal{W}$'s poor error reports were noticed soon after its introduction [111], and improving them has been a popular area of research ever since. In this section we will review the state of the art in type error diagnosis according to the following three high-level approaches:

1. *localizing* errors to a specific (set of) term(s);
2. *explaining* the error to the programmer; and
3. automatically *fixing* the error for the programmer.

1.3.1 Localizing Type Errors

The location reported by a type error is likely to be the first place the programmer looks for issues, so providing an accurate location could greatly reduce the time spent debugging.

Alternate Traversal Strategies

Noting that the placement of unification calls determines where errors are reported, several authors have proposed alternate traversal strategies. Lee et al. [57] describe a "folklore" algorithm $\mathcal{M}$ that traverses the program top down, rather than from the bottom up, pushing constraints inward from outer expressions. Thus, while $\mathcal{W}$ returned both a substitution and a type, $\mathcal{M}$ takes an *expected* type as input, and returns only a substitution. Lee et al. also prove that $\mathcal{M}$ always terminates sooner that $\mathcal{W}$ would, *i.e.* at an expression deeper in the tree.

Figure 1.6 contains a selection of $\mathcal{M}$'s rules that are relevant to the error in our `sumList` program. Note how the `match` rule no longer does any unification, it just makes a series of recursive calls. Rather, the + rule is now responsible for checking that its surrounding context expects it to return an `int`. This is a subtle change from $\mathcal{W}$'s behavior, but as a result $\mathcal{M}$ will blame the + expression for producing an `int` rather than the `match` expression. This is better, though still not ideal as the actual error is in the `[]` case.

Another issue, present in both algorithm $\mathcal{M}$ and $\mathcal{W}$, is that constraints are propagated from one branch to others, known as the "left-to-right" bias [69]. This bias is the reason that $\mathcal{M}$

$$
\begin{array}{lcl}
\mathcal{M} & : & \Gamma \times e \times t \to \theta \\
\mathcal{M}(\Gamma,\ e_1 + e_2,\ t) & = & \text{let } \theta_1 = \mathcal{U}(t,\ \texttt{int}) \\
 & & \quad \theta_2 = \mathcal{M}(\theta_1(\Gamma),\ e_1,\ \texttt{int}) \\
 & & \quad \theta_3 = \mathcal{M}(\theta_2\theta_1(\Gamma)),\ e_2,\ \texttt{int}) \\
 & & \text{in } \theta_3\theta_2\theta_1 \\
\mathcal{M}(\Gamma,\ \texttt{match}\ e_1 \begin{cases} [] \to e_2 \\ x_1 :: x_2 \to e_3 \end{cases},\ t) & = & \text{let } \theta_1 = \mathcal{M}(\Gamma,\ e_1,\ [\alpha_1]),\ \alpha_1 \text{ is fresh} \\
 & & \quad \theta_2 = \mathcal{M}(\theta_1(\Gamma),\ e_2,\ t) \\
 & & \quad \Gamma' = \Gamma \cup \{x_1 : \alpha_1,\ x_2 : [\alpha_1]\} \\
 & & \quad \theta_3 = \mathcal{M}(\theta_2\theta_1(\Gamma')),\ e_3,\ \theta_2(t)) \\
 & & \text{in } \theta_3\theta_2\theta_1
\end{array}
$$

Figure 1.6. A selection of rules from algorithm $\mathcal{M}$, extended to our language.

(and most type-checkers in practice) blames the `h::t` case rather than the `[]` case; however, it is not limited to expressions that create a branch in the program's control-flow. Rather, the constraint propagation happens between branches of the program's abstract syntax tree. See, *e.g.* the $e_1\ e_2$ case of $\mathcal{W}$ — we apply the substitution θ_1 from e_1 to the environment when checking e_2. Thus, any expression with multiple children will be subject to the left-to-right bias.

Instead of propagating the constraints collected in one branch to others, McAdam [69] and Yang [115] suggest a *symmetric* traversal that checks each branch independently of the others and then reports an error when merging two inconsistent sets of constraints from the branches. Mechanically, this means that we no longer apply the substitution from one branch to the other, but rather unify the substitutions from both branches after they have been individually checked. An important, and unfortunate, consequence of unifying substitutions between branches is that we can no longer implement type variables with mutable references to avoid the overhead of passing explicit substitutions around, as is commonly done in production-grade compilers. One might also think that we would be right back at the issue we observed with algorithm $\mathcal{W}$, with an error reported at the `match` expression, but Yang annotates each constraint with its source location, so that they can report the conflict between the `[]` on line 3 and the `+` on line 4.

Type Error Slicing

Tip et al. [107] and Haack et al. [37] extend the idea of McAdam and Yang, and compute a full type error *slice*, *i.e.* all of the sub-expressions that are required for the error to manifest and no more. For example, one type error slice for `sumList` would be the `match` expression and its two children, the `[]` and the `+` expression, which [37] would report as follows:

```
Type error: type constructor clash, endpoints list vs. int.
match .. with | [] -> [] | h::t -> .. + ..
```

There is at least one other error slice for `sumList`, which includes the `[]`, the `+`, the recursive `sumList` call, and the `let rec` binder, which captures the issue that a list can be passed back through the recursive call to the `+` operator. In general there may be many distinct error slices for the same error, and while computing one slice can be done efficiently, computing all slices is exponential.

Neubauer et al. [75] present a decidable type system based on discriminative sum types, in which all terms are typeable and type derivations contain all type errors in a program. They then use the typing derivation to slice out the parts of the expression related to each error. Rahli et al. [85, 86] investigate what is required to support slicing for a full programming language, and present a type error slicer for the entirety of SML. Sagonas et al. [93] use type error slices to explain errors in the optional Dialyzer [63] system for Erlang. Schilling [94] shows how to how to turn any type checker into a slicer by treating it as a black-box.

A drawback of type error slicers is that they typically involve rewriting the type checker to use a specialized constraint language and solver. Production compilers for languages like OCaml and Haskell generally feature more advanced type languages than Hindley-Milner with heavily optimized type checkers, so the prospect of a full rewrite to support slicing is probably quite daunting.

Further, while type error slicers can guarantee enough information to diagnose an error, they can fall into the opposite trap of providing *too much* information, producing a slice that is not much smaller than the original program. In other words, a type error slicer will produce

every possible expression that could be blamed for the error, but some expressions are more likely to be at fault than others.

Finding Likely Sources of Errors

Thus, recent work has focused on finding the *most likely* source of a type error. Zhang et al. [118, 119] use Bayesian reasoning to search the constraint graph for constraints that participate in many unsatisfiable paths and relatively few satisfiable paths, based on the intuition that the program should be mostly correct. Pavlinovic et al. [79, 80] translate the localization problem into a MaxSMT problem, asking an off-the-shelf solver to find the smallest set of constraints that can be removed such that the resulting system is satisfiable. Loncaric et al. [65] improve the scalability of Pavlinovic et al. by reusing the existing type checker as a theory solver in the Nelson-Oppen [74] style, and thus require only a MaxSAT solver. All three of these techniques support *weighted* constraints to incorporate knowledge about the frequency of different errors, but only Pavlinovic et al. use the weights, setting them to the size of the term that induced the constraint.

Of these three techniques, only Pavlinovic et al. can isolate the source of the type error in `sumList` to the `[]` case, as they weight constraints by expression size. Zhang et al. and Loncaric et al. cannot distinguish between the erroneous `[]` and the correct `+`, and present both as equally likely sources.

1.3.2 Explaining Type Errors

The techniques we have discussed so far have focused primarily on the task of *localizing* a type error, but a good error report should also *explain* the error. Wand [111], Beaven et al. [6], and Duggan et al. [27] attempt to explain type errors by collecting the chain of inferences made by the type checker — essentially the typing derivation — and presenting them to the user. For example, an explanation of the error in `sumList` in the style of Beaven et al. might look like the following:

```
A type error was detected in the case analysis of >> xs <<.
```

```
The types of the two branches,
>> 'a list << and >> int <<,
are not unifiable.
** Why does the >> [] << branch have type >> 'a list << ?
   The expression >> [] << has type >> 'a list << .
** Why does the >> h::t << branch have type >> int << ?
   The >> + << operator returns a value of type >> int << .
```

Such explanations, when presented in natural language, can become quite lengthy. As an attempt to compress the explanation Yang et al. [116] present a visualization of the inference process. Gast [34] produces a slice enhanced by arrows showing the dataflow from sources with different types to a shared sink, borrowing the insight of dataflows-as-explanations from MrSpidey [31].

Interactive Explanations

Static explanations of type errors, as seen above, run the risk of overwhelming the user with too much information, it may be preferable to treat type error diagnosis as an *interactive* debugging session. Bernstein et al. [8] extend the type inference procedure to handle *open* expressions (*i.e.* with unbound variables). This allows users to interactively query the type checker for the types of sub-expressions. Thus, a user may be able to quickly examine the expressions she believes to be relevant rather than having to sift through a static explanation.

Still, the user may have a fundamental misunderstanding of the type system, leading her to engage in a long series of queries that are not actually relevant to the error. As a remedy, Chitil [19] proposes *algorithmic debugging* of type errors, presenting the user with a sequence of yes-or-no questions about the inferred types of sub-expressions. Thus, she will be guided by the system to a specific explanation for the error in a finite amount of time.

Programmatic Explanations

The best explanation of a type error, however, might be given by an expert, *e.g.* the compiler or library author. Hage et al. [38] catalog a set of heuristics for improving the quality of error messages by examining errors made by novices. Marceau et al. [67, 68] study the effectiveness of error messages in novice environments and present suggestions for improving their quality and consistency. Heeren et al. [41], Christiansen [20], and Serrano et al. [102] extend the ability to customize error messages to library authors, enabling *domain-specific* errors. The 8.0 release of the Glasgow Haskell Compiler[2] incorporates these ideas, allowing library authors to supply custom errors when type-class resolution or type-family reduction fail, but not for ordinary unification failures.

1.3.3 Fixing Type Errors

Finally, some techniques go beyond explaining or locating a type error, and actually attempt to *fix* the error automatically. Lerner et al. [60] searches for fixes by enumerating a set of local mutations to the program and querying the type checker to see if the error remains. Chen et al. [17] use a notion of *variation-based* typing to track choices made by the type checker and enumerate potential changes that would fix the error. They also extend the algorithmic debugging technique of Chitil by allowing the user to enter the expected type of specific sub-expressions and suggesting fixes based on these desired types [18].

1.4 Our Contributions

The thesis of this dissertation is that we can adapt the wealth of work in automated program testing and machine learning to the task of providing better diagnostic information for type errors. To that end we present three concrete contributions:

1. In Chapter 2 we present a dataset of novice interactions with the OCaml type checker, which will form the backbone of our evaluation. This dataset contains thousands of

[2] https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors

ill-typed programs from over one hundred different students, the largest set of novice type errors we are aware of.

2. In Chapter 3 we use techniques from automated program testing to search for *witnesses* to type errors, *i.e.* input vectors that would cause the program to crash. Once we have found a witness, we compute an execution trace that *demonstrates* how the program goes wrong, and provide an interactive debugger with which students can explore the erroneous computation.

3. In Chapter 4 we use machine learning techniques to *learn* a model of where type errors are most likely to occur in our students' programs. This model allows us to make significantly more accurate predictions of where the error is most likely to be found.

Our contributions are presented and evaluated in the context of OCaml programs written by novice programmers, but they are not restricted to this domain. Rather, we chose OCaml as the target language as it has a powerful type system with global inference, showcasing both the advantages of static typing and the limitations of type inference. We use novice programs for our benchmarks as we feel novices are the most in need of assistance in diagnosing type errors. In our experience, working with the type system is quite pleasant once you have gotten used to it; however, getting to that point can be difficult.

Endnotes

Acknowledgments

This chapter contains material adapted from the following publications: E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In ICFP '16, 2016; E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *In submission to J. Funct. Programming*, 2017; and E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *In submission to OOPSLA '17*, 2017. The dissertation author was the primary investigator and author of these papers.

Chapter 2

A Dataset of Novice Type Errors

In this chapter we describe a collection of novice interactions with the OCaml compiler — including, importantly, type errors — that we gathered at UC San Diego over two quarters of the undergraduate CSE 130 course (IRB #140608). We have made the anonymized data publicly available [97], and hope that other researchers will find it as valuable as we have.

The CSE 130 course is an upper-level (*i.e.* generally consisting of third- and fourth-year students) course that introduces students to typed functional languages, specifically OCaml. For most students, this course will be their first exposure to both functional programming and type systems with global inference. We generally spend the first five weeks covering basic functional programming in OCaml, and then spend the last five weeks in Scala covering more advanced concepts like traits and monads (in the guise of `for-yield` comprehensions). In the OCaml portion of the course we cover standard functional idioms like (tail-) recursion, higher-order functions, and user-defined algebraic datatypes.

We recruited students from two instances of the course, Spring 2014 (SP14) and Fall 2015 (FA15), to use an instrumented version of the OCaml-Top[1] editor, which logged each of their interactions with the OCaml top-level system. 46 students from the SP14 quarter and 56 students from the FA15 quarter participated, for a total of 102 participants. The participants used our instrumented editor to complete the first three programming assignments, which involved writing 23 OCaml programs.

Figure 2.1 shows the main interface of OCaml-Top, which contains an editor pane on

[1] https://www.typerex.org/ocaml-top.html

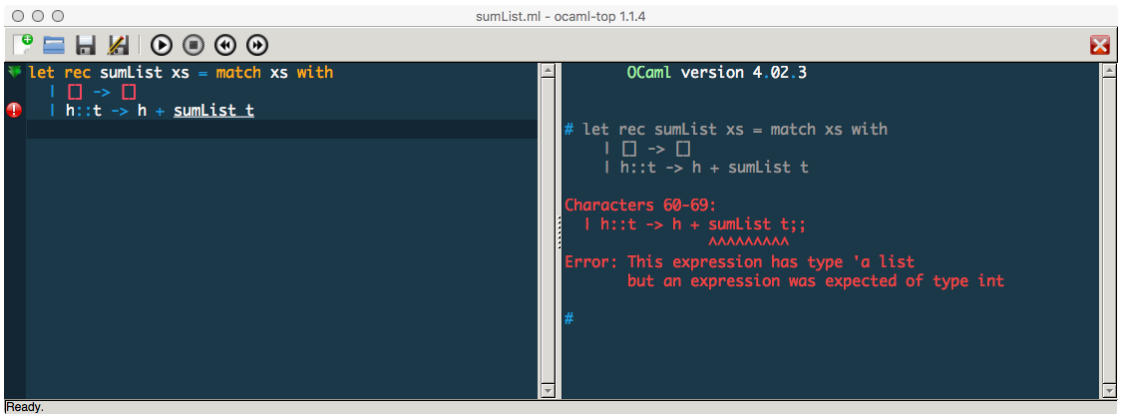


Figure 2.1. The OCaml-Top editor.

the left and an instance of the OCaml top-level interpreter on the right. The students interact with the top-level system by selecting text in the editor and pressing the "play" button in the toolbar, which sends the selected text to the interpreter for evaluation. The editor also maintains a offset into the open file to track how much of the file has been evaluated; this allows it to intelligently send the next definition to the interpreter if no text is selected. The toolbar also has a "stop" button to abort evaluation (*e.g.* to abort infinite loops), a "rewind" button to restart the interpreter, and a "fast-forward" button to load the entire file into the interpreter. OCaml-Top always sends each definition to the interpreter individually, even when using the "fast-forward" button, this will become important when we extract ill-typed programs from the interactions.

We instrumented OCaml-Top to record each of the student's interactions with the top-level interpreter. Specifically, we modified it to log an event each time a student pressed one of the four interaction buttons on the toolbar (or used the equivalent keyboard shortcuts). This gives rise to three kinds of interaction events:

Eval The student sent one or more definitions to be evaluated by the interpreter, by pressing either "play" or "fast-forward". In addition to logging the event, we logged both the offsets into the file that mark delimit the evaluated text, and the list of evaluated definitions.

Abort The student aborted evaluation by pressing "stop".

Stop The student restarted the interpreter by pressing "rewind".

```
{
    "file": "hw1.ml" | "hw2.ml" | "hw3.ml",
    "time": number,
    "body": string,
    "cursor": number,
    "event": {
        "type": "abort" | "eval" | "stop",
        "region": {
            "start": number,
            "stop": number
        }
    },
    "ocaml": [{
        "in": string,
        "out": string,
        "type": "scope" | "syntax" | "type" | "",
        "min": string
    }]
}
```

Figure 2.2. Format of the post-processed interaction events as JSON objects.

For each event we also recorded the filename to identify the homework the student was working on, the current UNIX timestamp, the entire body of the file, and the offset of the cursor into the file. Thus, for each participating student we can see and replay, with fine granularity, the steps they took to solve the programming assignments.

We then post-processed the interaction traces to add the OCaml interpreter's responses to the students' submissions. Recall that OCaml-Top always sends single top-level definitions to the interpreter, which maintains an environment of defined types and functions. This is inconvenient for extracting ill-typed programs, as the vast majority of submitted definitions will depend on other definitions that were submitted previously. Thus, we modified the OCaml interpreter to track dependencies between top-level function and type definitions, so that for each definition a student submitted, we could produce a self-contained, *minimal* program that would have the same behavior.

For each submitted definition, we collected the interpreter's response and the minimal self-contained program, and classified the response as either a syntax error, a scoping error (*e.g.* an unbound variable), a type error, or no error. As we are primarily concerned with type

errors in this work, we did not capture the actual result of evaluating the definition, *i.e.* the resulting value, but it would be easy to extend the replay procedure to do so. We then stored the post-processed interaction traces as sequences of JSON objects, in the format described by Figure 2.2. From this format it is quite convenient to run various analyses, *e.g.* what are the hardest assignments (measured by time spent or by errors encountered), what is the relative frequency of various errors, *etc.*, though for this work we will only use the dataset as a source of type errors and fixes.

Chapter 3

Dynamic Witnesses for Static Type Errors

We have noticed a common theme in the existing literature on type error diagnosis: errors are always presented in terms of the static type system, and yet (static) type systems are meant to rule out certain types of *dynamic* errors. We believe this may be particularly confusing for novice users, who must simultaneously develop a mental model of the dynamic (evaluation) semantics and the static (typing) semantics of the language they are learning. Furthermore, given the rise of dynamic languages like Python and Javascript as teaching languages, novices may be more familiar with reasoning about the dynamic semantics of a program than the static semantics. Thus, by connecting the static type error to the dynamic error it would prevent, we might help novices understand the type system better.

In this chapter we propose a new approach that explains static type errors by *dynamically* witnessing how an ill-typed program goes wrong. We have developed NanoMaLy, an interactive tool that uses the source of the ill-typed function to automatically synthesize the result on the bottom-left in Figure 3.1, which shows how the recursive calls reduce to a configuration where the program "goes wrong" — *i.e.* the `int` value `0` is to be added to the `list` value `[]`. We achieve this via three concrete contributions.

1. Finding Witnesses

Our first contribution is an algorithm for searching for *witnesses* to type errors, *i.e.* inputs that cause a program to go wrong (§ 3.2). This problem is tricky when we cannot rely on static type information, as we must avoid the trap of *spurious* inputs that cause irrelevant problems

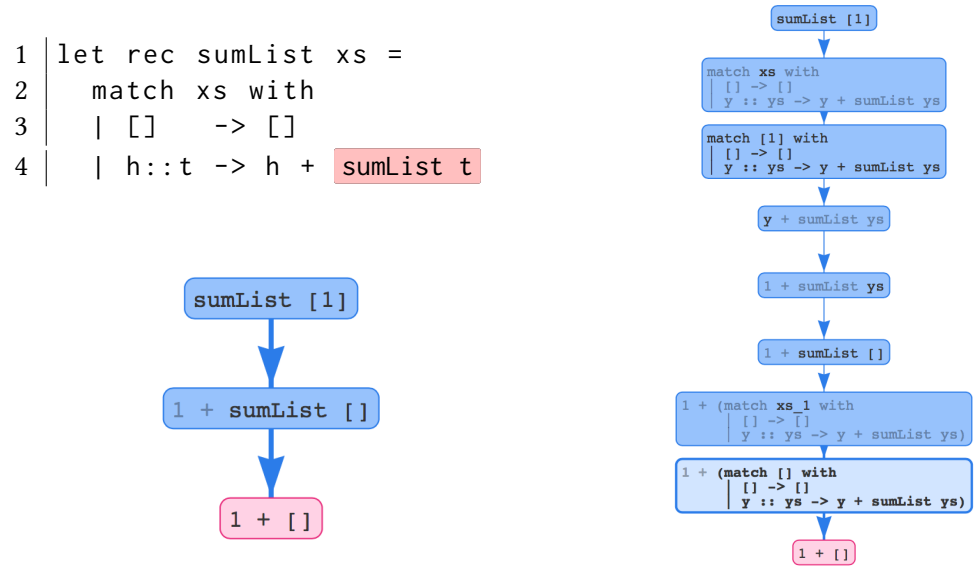


Figure 3.1. (top-left) The ill-typed `sumList` function highlighting the error location reported by OCaml. (bottom-left) Dynamically witnessing the type error in `sumList`, showing only function call-return pairs. (right) The same trace, fully expanded to show each small-step reduction in the computation.

that would be avoided by picking values of a different, relevant type. We solve this problem by developing a novel operational semantics that combines evaluation and type inference. We execute the program with *holes* — values whose type is unknown — as the inputs. A hole remains abstract until the evaluation context tells us what type it must have, for example the parameters to an addition operation must both be integers. Our semantics conservatively instantiates holes with concrete values, dynamically inferring the type of the input until the program goes wrong. We prove that our procedure synthesizes *general* witnesses, which means, intuitively, that if a witness is found for a given ill-typed function, then, *for all* (inhabited) input types, there exist values that can make the function go wrong.

Given a witness to a type error, the novice may still be at a loss. The standard OCaml interpreter and debugging infrastructure expect well-typed programs, so they cannot be used to investigate *how* the witness causes the program to crash. More importantly, the execution itself may be quite long and may contain details not relevant to the actual error.

2. Visualizing Witnesses

Our second contribution is an interactive visualization of the execution of purely functional OCaml programs, well-typed or not (§ 3.3). We extend the semantics to also build a *reduction graph* which records all of the small-step reductions and the context in which they occur. The graph lets us visualize the sequence of steps from the source witness to the stuck term. The user can interactively expand the computation to expose intermediate steps by selecting an expression and choosing a traversal strategy. The strategies include many of the standard debugging moves, *e.g.* stepping *forward* or *into* or *over* calls, as well stepping or jumping *backward* to understand how a particular value was created, while preserving a context of the intermediate steps that allow the user to keep track of a term's provenance.

We introduce a notion of *jump-compressed* traces to abstract away the irrelevant details of a computation. A jump-compressed trace includes only function calls and returns. For example, the trace in the bottom-left of Figure 3.1 is jump-compressed. Jump-compressed traces are similar to stack traces in that both show a sequence of function calls that lead to a crash. However, jump-compressed traces also show the return values of successful calls, which can be useful in understanding why a particular path was taken.

3. Evaluating Witnesses

Of course, the problem of finding witnesses is undecidable in general. In fact, due to the necessarily conservative nature of static typing, there may not even exist any witnesses for a given ill-typed program. Thus, our approach is a heuristic that is only useful if it can find *compact* witnesses for *real-world* programs. Our third contribution is an extensive evaluation of our approach on two different sets of ill-typed programs obtained by instrumenting compilers used in beginner's classes (§ 3.4). The first is the UW dataset [60], standard in the literature, comprising 284 ill-typed programs. The second comes from the new dataset described in Chapter 2, comprising 4,407 ill-typed programs. We show that for both datasets, our technique is able to generate witnesses for around 85% of the programs, in under a second in the vast majority of cases. Furthermore, we show that a simple interactive strategy yields compact

counterexample traces with at most 5 steps for 60% of the programs, and at most 10 steps for over 80% of the programs. We can even use witnesses to *localize* type errors with a simple heuristic that treats the values in a "stuck" term as *sources* of typing constraints and the term itself as a *sink*, achieving around 70% accuracy in locating the source of the error.

The ultimate purpose of an error report is to help the programmer *comprehend* and *fix* problematic code. Thus, our final contribution is a user study that compares NanoMaLy's dynamic witnesses against OCaml's type errors along the dimension of comprehensibility (§ 3.4.6). Our study finds that students given one of our witnesses are consistently more likely to correctly explain and fix a type error than those given the standard error message produced by the OCaml compiler.

All together, our results show that in the vast majority of cases, (novices') ill-typed programs *do* go wrong, and that the witnesses to these errors can be helpful in understanding the source of the error. This, in turn, opens the door to a novel dynamic way to explain, understand, and appreciate the benefits of static typing.

3.1 Overview

We start with an overview of our approach to explaining (static) type errors using *witnesses* that (dynamically) show how the program goes wrong. We illustrate why generating suitable inputs to functions is tricky in the absence of type information. Then we describe our solution to the problem and highlight the similarity to static type inference, Finally, we demonstrate our visualization of the synthesized witnesses.

3.1.1 Generating Witnesses

Our goal is to find concrete values that demonstrate how a program "goes wrong".

Problem: Which inputs are bad?

One approach is to randomly generate input values and use them to execute the program until we find one that causes the program to go wrong. Unfortunately, this approach quickly runs aground. Recall the erroneous `sumList` function from Figure 3.1. What *types* of inputs

should we test `sumList` with? Values of type `int list` are fair game, but values of type, say, `string` or `bool` will cause the program to go wrong in an *irrelevant* manner. Concretely, we want to avoid testing `sumList` with any type other than `int list` because any other type would cause it to get stuck immediately in the `match` expression.

Solution: Don't generate inputs until forced.

Our solution is to avoid generating a concrete value for the input at all, until we can be sure of its type. The intuition is that we want to be as lenient as possible in our tests, so we make no assumptions about types until it becomes clear from the context what type an input must have. This is actually quite similar in spirit to type inference.

To defer input generation, we borrow the notion of a "hole" from SmallCheck [92]. A hole — written v^{α} — is a *placeholder* for a value v of some unknown type α. We leave all inputs as uninstantiated holes until they are demanded by the program, *e.g.* due to a built-in operation like the `match` expression.

Narrowing Input Types

Primitive operations, data construction, and case-analysis *narrow* the types of values. For concrete values this amounts to a runtime type check, we ensure that the value has a type compatible with the expected type. For holes, this means we now know the type it should have (or in the case of compound data we know *more* about the type) so we can instantiate the hole with a value. The value may itself contain more holes, corresponding to components whose type we still do not know. Consider the `fst` function:

```
let fst p = match p with
  (a, b) -> a
```

The case analysis tells us that `p` must be a pair, but it says *nothing* about the contents of the pair. Thus, upon reaching the case-analysis we would generate a pair containing fresh holes for the `fst` and `snd` component. Notice the similarity between instantiation of type variables and instantiation of holes. We can compute an approximate type for `fst` by approximating the

types of the (instantiated) input and output, which would give us:

```
fst : (α1 * α2) -> α1
```

We call this type approximate because we only see a single path through the program, and thus will miss narrowing points that only occur in other paths.

Returning to `sumList`, given a hole as input we will narrow the hole to a `list`[1] upon reaching the `match` expression. At this point we construct a random `list`[2] with new holes as the values for the instantiation and concrete execution takes over. Assuming we have generated a non-empty list, we will move into the `h::t` branch and reach the `+` expression, which will narrow `h` to a concrete `int`. We will then recurse via `sumList t` until we reach the last element of the list, at which point `sumList t` will return the empty list `[]` and the program will crash as expected at the `+` expression.

Witness Generality

We show in § 3.2.3 that our lazy instantiation of holes produces *general witnesses*. That is, we show that if "executing" a function with a hole as input causes the function to "go wrong", then there is *no possible* type for the function. In other words, for *any* types you might assign to the function's inputs, there exist values that will cause the function to go wrong.

Problem: How many inputs does a function take?

There is another wrinkle, though; how did we know that `sumList` takes a single argument instead of two (or none)? It is clear, syntactically, that `sumList` takes *at least* one argument, but in a higher-order language with currying, syntax can be deceiving. Consider the following definition:

```
let incAllByOne = List.map (+ 1)
```

Is `incAllByOne` a function? If so, how many arguments does it take? The OCaml compiler deduces that `incAllByOne` takes a single argument because the *type* of `List.map` says it takes

[1] At this point we do not know it must be an `int list`.
[2] With standard heuristics [21] to favor small values.

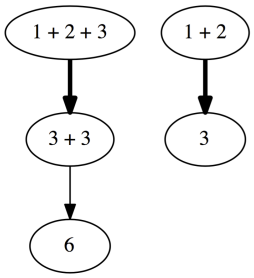


Figure 3.2. The reduction graph for `1+2+3`, highlighting the edges produced by reducing `1+2+3` to `3+3`.

two arguments, and it is partially applied to `(+ 1)`. As we are dealing with ill-typed programs we do not have the luxury of typing information.

Solution: Search for saturated application.

We solve this problem by deducing the number of arguments via an iterative process. We add arguments one-by-one until we reach a *saturated* application, *i.e.* until evaluating the application returns a value other than a lambda.

3.1.2 Visualizing Witnesses

We have described how to reliably find witnesses to type errors in OCaml, but this does not fully address our original goal — to *explain* the errors. Having identified an input vector that triggers a crash, a common next step is to step through the program with a *debugger* to observe how the program evolves. The existing debuggers and interpreters for OCaml assume a type-correct program, so unfortunately we cannot use them off-the-shelf. Instead we extend our search for witnesses to produce an execution trace.

Reduction Graph

Our trace takes the form of a reduction graph, which records small-step reductions in the context in which they occur. For example, evaluating the expression `1+2+3` would produce the graph in Figure 3.2. Notice that when we transition from `1+2+3` to `3+3` we collect both that

edge *and* an edge from the sub-term 1+2 to 3. These additional edges allow us to implement two common debugging operations *post-hoc*: "step into" to zoom in on a specific function call, and "step over" to skip over uninteresting computations.

Interacting with the graph

The reduction graph is useful for formulating and executing traversals, but displaying it all at once would quickly become overwhelming. Our interaction begins by displaying a big-step reduction, *i.e.* the witness followed by the stuck term. The user can then progressively fill in the hidden steps of the computation by selecting a visible term and choosing one of the applicable traversal strategies — described in § 3.3 — to insert another term into the visualization.

Jump-compressed Witnesses

It is rare for the initial state of the visualization to be informative enough to diagnose the error. Rather than abandon the user, we provide a short-cut to expand the witness to a *jump-compressed* trace, which contains every function call and return step. The jump-compressed trace abstracts the computation as a sequence of call-response pairs, providing a high-level overview of steps taken to reach the crash, and a high level of compression compared to the full trace. For example, the jump-compressed trace in Figure 3.1 contains 4 nodes compared to the 19 in the fully expanded trace. Our benchmark suite of student programs shows that jump-compression is practical, with an average jump-compressed trace size of 7 nodes and a median of 5.

3.2 Type-Error Witnesses

Next, we formalize the notion of type error witnesses as follows. First, we define a core calculus within which we will work (§ 3.2.1). Second, we develop a (non-deterministic) operational semantics for ill-typed programs that precisely defines the notion of a *witness* (§ 3.2.2). Third, we formalize and prove a notion of *generality* for witnesses, which states, intuitively, that if we find a single witness then for *every possible* type assignment there exist inputs that are guaranteed to make the program "go wrong" (§ 3.2.3). Finally, we refine the operational semantics

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & \mathbf{e} & ::= & e \mid \texttt{stuck} \\
& e & ::= & v \mid x \mid e\ e \mid e + e \\
& & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid & \langle e, e \rangle \mid \texttt{match}\ e \left\{ \langle x, x \rangle \rightarrow e \right. \\
& & \mid & e :: e \mid [] \\
& & \mid & \texttt{match}\ e \begin{cases} [] \rightarrow e \\ x :: x \rightarrow e \end{cases} \\
\textbf{\textit{Values}} & v & ::= & n \mid b \mid \lambda x.e \mid v^{\alpha} \\
& & \mid & \langle v, v \rangle \mid l \\
& l & ::= & v ::^{t} v \mid []^{t} \\
\textbf{\textit{Integers}} & n & ::= & 0, 1, -1, \ldots \\
\textbf{\textit{Booleans}} & b & ::= & \texttt{true} \mid \texttt{false} \\
\textbf{\textit{Types}} & t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{fun} \\
& & \mid & t \times t \mid [t] \mid \alpha \\
\textbf{\textit{Substitutions}} & \sigma & ::= & \{v_1^{\alpha_1} \mapsto v_n, \ldots, v_n^{\alpha_n} \mapsto v_n\} \\
& \theta & ::= & \{\alpha_1 \mapsto t_n, \ldots, \alpha_n \mapsto t_n\} \\
\textbf{\textit{Contexts}} & C & ::= & \bullet \mid C\ e \mid v\ C \\
& & \mid & C + e \mid v + C \\
& & \mid & \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid & \langle C, e \rangle \mid \langle v, C \rangle \\
& & \mid & \texttt{match}\ C \left\{ \langle x, x \rangle \rightarrow e \right. \\
& & \mid & C :: e \mid v :: C \\
& & \mid & \texttt{match}\ C \begin{cases} [] \rightarrow e \\ x :: x \rightarrow e \end{cases}
\end{array}
$$

Figure 3.3. Syntax of λ^H

into a *search procedure* that returns concrete (general) witnesses for ill-typed programs § (3.2.4). We have formalized and tested our semantics and generality theorem in PLT-Redex [30]. Detailed proofs for the theorems in this section can be found in Appendix A.

3.2.1 Syntax

Figure 3.3 describes the syntax of λ^H, a simple lambda calculus with integers, booleans, pairs, and lists. As we are specifically interested in programs that *do* go wrong, we include an explicit `stuck` term in our syntax. We write **e** to denote terms that may be `stuck`, and e to denote terms that may not be stuck.

Holes

Recall that a key challenge in our setting is to find witnesses that are meaningful and do not arise from choosing values from irrelevant types. We solve this problem by equipping our term language with a notion of a *hole*, written v^{α}, which represents an *unconstrained* value v that may be replaced with *any* value of an unknown type α. Intuitively, the type holes α can be viewed as type variables that we will *not* generalize over. A *normalized* value is one that is not a hole, but which may internally contain holes. For example $\langle v_1[\alpha_1], v_2[\alpha_2]\rangle$ is a normalized value.

Substitutions

Our semantics ensure the generality of witnesses by incrementally *refining* holes, filling in just as much information as is needed locally to make progress (inspired by the manner in which SmallCheck uses lazy evaluation [92]). We track how the holes are incrementally filled in, by using value (resp. type) *substitutions* σ (resp. θ) that map value (resp. type) holes to values (resp. types). The substitutions let us ensure that we consistently instantiate each hole with the same (partially defined) value or type, regardless of the multiple contexts in which the hole appears. This ensures we can report a concrete (and general) witness for any (dynamically) discovered type errors.

A *normalized* value substitution is one whose co-domain is comprised of normalized values. In the sequel, we will assume and ensure that all value substitutions are normalized. We ensure additionally that the co-domain of a substitution does not refer to any elements of its domain, *i.e.* when we extend a substitution with a new binding we apply the substitution to itself. We will use the notation $\theta + \{\alpha \mapsto t\}$ for the extension of a type (*resp.* value) substitution, to distinguish it from a simple set union.

3.2.2 Semantics

Recall that our goal is to synthesize a value that demonstrates why (and how) a function goes wrong. We accomplish this by combining evaluation with type inference, giving us a form

of dynamic type inference. Each primitive evaluation step tells us more about the types of the program values. For example, addition tells us that the addends must be integers, and an if-expression tells us the condition must be a boolean. When a hole appears in such a context, we know what type it must have in order to make progress and can fill it in with a concrete value.

The evaluation relation is parameterized by a pair of functions, *narrow* (narrow) and *generate* (gen), that "dynamically" perform type-checking and hole-filling respectively.

Narrowing Types

The procedure $\mathsf{narrow}(v, t, \sigma, \theta)$, defined in Figure 3.4, takes as input a value v, a type t, and the current value and type substitutions, and refines v to have type t by yielding a triple of either the same value and substitutions, or yields the stuck state if no such refinement is possible. In the case where v is a hole, it first checks in the given σ to see if the hole has already been instantiated and, if so, returns the existing instantiation. For convenience, narrow uses a variant of Robinson's $\mathcal{U}$ [90] that unifies a set of types, and that takes and updates an existing substitution. As the value substitution is normalized, in the first case of narrow we do not need to narrow the result of the substitution, the sub-hole will be narrowed when the context demands it.

Generating Values

The (non-deterministic) $\mathsf{gen}(t, \theta)$ in Figure 3.5 takes as input a type t and returns a value of that type. For base types the procedure returns an arbitrary value of that type. For functions it returns a lambda with a *new* hole denoting the return value. For unconstrained types (denoted by α) it yields a fresh hole constrained to have type α (denoted by ν^{α}). When generating a $[t]$ we must take care to ensure the resulting tree is well-typed. For a polymorphic type $[\alpha]$ or $\alpha_1 \times \alpha_2$ we will place holes in the generated value; they will be lazily filled in later, on demand.

$$
\begin{array}{lll}
\mathsf{narrow} & : v \times t \times \sigma \times \theta \to \langle v \cup \mathtt{stuck}, \sigma, \theta \rangle & \\
\mathsf{narrow}(v^\alpha, t, \sigma, \theta) & \doteq \begin{cases} \langle v, \sigma, \theta' \rangle & \textbf{if } v = \sigma(v^\alpha), \ \theta' = \mathcal{U}(\{\alpha, t, \mathsf{ty}(v)\}, \theta) \\ \langle \mathtt{stuck}, \sigma, \theta \rangle & \textbf{if } v = \sigma(v^\alpha) \\ \langle v, \sigma + \{v^\alpha \mapsto v\}, \theta' \rangle & \textbf{if } \theta' = \mathcal{U}(\{\alpha, t\}, \theta), \ v = \mathsf{gen}(t, \theta') \end{cases} \\
\mathsf{narrow}(n, \mathtt{int}, \sigma, \theta) & \doteq \langle n, \sigma, \theta \rangle & \\
\mathsf{narrow}(b, \mathtt{bool}, \sigma, \theta) & \doteq \langle b, \sigma, \theta \rangle & \\
\mathsf{narrow}(\lambda x.e, \mathtt{fun}, \sigma, \theta) & \doteq \langle \lambda x.e, \sigma, \theta \rangle & \\
\mathsf{narrow}(\langle v_1, v_2 \rangle, t_1 \times t_2, \sigma, \theta) & \doteq \langle \langle v_1, v_2 \rangle, \sigma, \theta'' \rangle & \textbf{if } \theta' = \mathcal{U}(\{\mathsf{ty}(v_1), t_1\}, \theta), \ \theta'' = \mathcal{U}(\{\mathsf{ty}(v_2), t_2\}, \theta') \\
\mathsf{narrow}([]^{t_1}, [t_2], \sigma, \theta) & \doteq \langle []^{t_1}, \sigma, \theta' \rangle & \textbf{if } \theta' = \mathcal{U}(\{t_1, t_2\}, \theta) \\
\mathsf{narrow}(v_1 ::^{t_1} v_2, [t_2], \sigma, \theta) & \doteq \langle v_1 ::^{t_1} v_2, \sigma, \theta' \rangle & \textbf{if } \theta' = \mathcal{U}(\{t_1, t_2\}, \theta) \\
\mathsf{narrow}(v, t, \sigma, \theta) & \doteq \langle \mathtt{stuck}, \sigma, \theta \rangle &
\end{array}
$$

Figure 3.4. Narrowing values

Steps and Traces

Figure 3.6 describes the small-step contextual reduction semantics for λ^H. A configuration is a triple $\langle \mathbf{e}, \sigma, \theta \rangle$ of an expression e or the stuck term `stuck`, a value substitution σ, and a type substitution θ. We write $\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow \langle \mathbf{e}', \sigma', \theta' \rangle$ if the state $\langle \mathbf{e}, \sigma, \theta \rangle$ transitions in a *single step* to $\langle \mathbf{e}', \sigma', \theta' \rangle$. A (finite) *trace* τ is a sequence of configurations $\langle \mathbf{e}_0, \sigma_0, \theta_0 \rangle, \ldots, \langle \mathbf{e}_n, \sigma_n, \theta_n \rangle$ such that $\forall 0 \leq i < n$, we have $\langle \mathbf{e}_i, \sigma_i, \theta_i \rangle \hookrightarrow \langle \mathbf{e}_{i+1}, \sigma_{i+1}, \theta_{i+1} \rangle$. We write $\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow^\tau \langle \mathbf{e}', \sigma', \theta' \rangle$ if τ is a trace of the form $\langle \mathbf{e}, \sigma, \theta \rangle, \ldots, \langle \mathbf{e}', \sigma', \theta' \rangle$. We write $\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow^* \langle \mathbf{e}', \sigma', \theta' \rangle$ if $\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow^\tau \langle \mathbf{e}', \sigma', \theta' \rangle$ for some trace τ.

Primitive Reductions

Primitive reduction steps — addition, if-elimination, function application, and data construction and case analysis — use narrow to ensure that values have the appropriate type (and that holes are instantiated) before continuing the computation. Importantly, beta-reduction *does not* type-check its argument, it only ensures that "the caller" v_1 is indeed a function.

gen	$:$	$t \times \theta \to v$	
$\mathsf{gen}(\alpha, \theta)$	$\doteq$	$\mathsf{gen}(\theta(\alpha), \theta)$	if $\alpha \in dom(\theta)$
$\mathsf{gen}(\texttt{int}, \theta)$	$\doteq$	n	non-det.
$\mathsf{gen}(\texttt{bool}, \theta)$	$\doteq$	b	non-det.
$\mathsf{gen}(t_1 \times t_2, \theta)$	$\doteq$	$\langle \mathsf{gen}(t_1, \theta), \mathsf{gen}(t_2, \theta) \rangle$	
$\mathsf{gen}([t], \theta)$	$\doteq$	l	non-det.
$\mathsf{gen}(\texttt{fun}, \theta)$	$\doteq$	$\lambda x.v^\alpha$	v, α are fresh
$\mathsf{gen}(\alpha, \theta)$	$\doteq$	v^α	v is fresh

Figure 3.5. Generating values

Recursion

Our semantics lacks a built-in `fix` construct for defining recursive functions, which may surprise the reader. Fixed-point operators often cannot be typed in static type systems, but our system would simply approximate its type as `fun`, apply it, and move along with evaluation. Thus we can use any of the standard fixed-point operators and do not need a built-in recursion construct.

3.2.3 Generality

A key technical challenge in generating witnesses is that we have no (static) type information to rely upon. Thus, we must avoid the trap of generating *spurious* witnesses that arise from picking irrelevant values, when instead there exist perfectly good values of a *different* type under which the program would not have gone wrong. We now show that our evaluation relation instantiates holes in a *general* manner. That is, given a lambda-term f, if we have $\langle f\ v^\alpha, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma, \theta \rangle$, then *for every* concrete type t, we can find a value v of type t such that $f\ v$ goes wrong.

Theorem 1 (Witness Generality)**.** *For any lambda f, if $\langle f\ v^\alpha, \varnothing, \varnothing \rangle \hookrightarrow^\tau \langle \texttt{stuck}, \sigma, \theta \rangle$, then for every (inhabited [3]) type t there exists a value v of type t such that $\langle f\ v, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma', \theta' \rangle$.*

We need to develop some machinery in order to prove this theorem. First, we show how our evaluation rules encode a dynamic form of type inference, and then we show that the

[3] All types in λ^H are inhabited, but in a larger language like OCaml this may not be true.

Evaluation $\boxed{\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow \langle \mathbf{e}, \sigma, \theta \rangle}$

$$\textsc{Plus-G}\ \frac{\begin{array}{c}\langle n_1, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta) \\ \langle n_2, \sigma'', \theta'' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma', \theta') \\ n = n_1 + n_2\end{array}}{\langle C\,[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle C\,[n], \sigma'', \theta'' \rangle}$$

$$\textsc{Plus-B1}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta)}{\langle C\,[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\textsc{Plus-B2}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma, \theta)}{\langle C\,[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\textsc{If-G1}\ \frac{\langle \mathtt{true}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C\,[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2], \sigma, \theta \rangle \hookrightarrow \langle C\,[e_1], \sigma', \theta' \rangle}$$

$$\textsc{If-G2}\ \frac{\langle \mathtt{false}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C\,[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2], \sigma, \theta \rangle \hookrightarrow \langle C\,[e_2], \sigma', \theta' \rangle}$$

$$\textsc{If-B}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C\,[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\textsc{App-G}\ \frac{\langle \lambda x.e, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{fun}, \sigma, \theta)}{\langle C\,[v_1\ v_2], \sigma, \theta \rangle \hookrightarrow \langle C\,[e\,[v_2/x]], \sigma', \theta' \rangle}$$

$$\textsc{App-B}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{fun}, \sigma, \theta)}{\langle C\,[v_1\ v_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\textsc{Match-Pair-G}\ \frac{\alpha_1, \alpha_2 \text{ are fresh} \qquad \langle \langle v_1, v_2 \rangle, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C\left[\mathtt{match}\ v\ \left\{\langle x_1, x_2 \rangle \rightarrow e\right\}\right], \sigma, \theta \rangle \hookrightarrow \langle C\,[e\,[v_1/x_1]\,[v_2/x_2]], \sigma_1, \theta_1 \rangle}$$

$$\textsc{Match-Pair-B}\ \frac{\alpha_1, \alpha_2 \text{ are fresh} \qquad \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C\left[\mathtt{match}\ v\ \left\{\langle x_1, x_2 \rangle \rightarrow e\right\}\right], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle}$$

Figure 3.6. Evaluation relation for λ^H

Evaluation (*ctd.*) $\boxed{\langle \mathbf{e}, \sigma, \theta \rangle \hookrightarrow \langle \mathbf{e}, \sigma, \theta \rangle}$

$$\textsc{Nil-G}\ \frac{\alpha \text{ is fresh}}{\langle C\,[[]]\,, \sigma, \theta\rangle \hookrightarrow \langle C\,[[]^{\alpha}]\,, \sigma, \theta\rangle}$$

$$\textsc{Cons-G}\ \frac{\begin{array}{c} t = \mathsf{ty}(v_1) \\ \langle v_2', \sigma_2, \theta_2\rangle = \mathsf{narrow}(v_2, [t], \sigma_1, \theta_1)\end{array}}{\langle C\,[v_1 :: v_2]\,, \sigma, \theta\rangle \hookrightarrow \langle C\left[v_1 ::^{t} v_2'\right], \sigma_2, \theta_2\rangle}$$

$$\textsc{Cons-B}\ \frac{\begin{array}{c} t = \mathsf{ty}(v_1) \\ \langle \mathtt{stuck}, \sigma_2, \theta_2\rangle = \mathsf{narrow}(v_2, [t], \sigma_1, \theta_1)\end{array}}{\langle C\,[v_1 :: v_2]\,, \sigma, \theta\rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_2, \theta_2\rangle}$$

$$\textsc{Match-List-G1}\ \frac{\alpha \text{ is fresh} \qquad \langle []^{t}, \sigma_1, \theta_1\rangle = \mathsf{narrow}(v, [\alpha], \sigma, \theta)}{\langle C\left[\mathtt{match}\ v \begin{cases} [] \to e_1 \\ x_1 :: x_2 \to e_2 \end{cases}\right], \sigma, \theta\rangle \hookrightarrow \langle C\,[e_1]\,, \sigma_1, \theta_1\rangle}$$

$$\textsc{Match-List-G2}\ \frac{\alpha \text{ is fresh} \qquad \langle v_1 ::^{t} v_2, \sigma_1, \theta_1\rangle = \mathsf{narrow}(v_1, [\alpha], \sigma, \theta)}{\langle C\left[\mathtt{match}\ v \begin{cases} [] \to e_1 \\ x_1 :: x_2 \to e_2 \end{cases}\right], \sigma, \theta\rangle \hookrightarrow \langle C\,[e_2\,[v_1/x_1]\,[v_2/x_2]]\,, \sigma_1, \theta_1\rangle}$$

$$\textsc{Match-List-B}\ \frac{\alpha \text{ is fresh} \qquad \langle \mathtt{stuck}, \sigma_1, \theta_1\rangle = \mathsf{narrow}(v, [\alpha], \sigma, \theta)}{\langle C\left[\mathtt{match}\ v \begin{cases} [] \to e_1 \\ x_1 :: x_2 \to e_2 \end{cases}\right], \sigma, \theta\rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_1, \theta_1\rangle}$$

Figure 3.6. Evaluation relation for λ^{H} (*ctd.*)

$$
\begin{array}{lcl}
\mathrm{ty}(n) & \doteq & \texttt{int} \\
\mathrm{ty}(b) & \doteq & \texttt{bool} \\
\mathrm{ty}(\lambda x.e) & \doteq & \texttt{fun} \\
\mathrm{ty}(\langle v_1, v_2 \rangle) & \doteq & \mathrm{ty}(v_1) \times \mathrm{ty}(v_2) \\
\mathrm{ty}([]^t) & \doteq & [t] \\
\mathrm{ty}(v_1 ::^t v_2) & \doteq & [t] \\
\mathrm{ty}(v^\alpha) & \doteq & \alpha
\end{array}
$$

Figure 3.7. The *dynamic type* of a value.

witnesses found by evaluation are indeed maximally general.

The Type of a Value

The *dynamic type* of a value v is defined as a function $\mathrm{ty}(v)$ shown in Figure 3.7. The types of primitive values are defined in the natural manner. The types of functions are *approximated*, which is all that is needed to ensure an application does not get stuck. For example,

$$\mathrm{ty}(\lambda x.x + 1) = \texttt{fun}$$

instead of $\texttt{int} \to \texttt{int}$. The types of (polymorphic) trees are obtained from the labels on their values, and the types of tuples directly from their values.

Dynamic Type Inference

We can think of the evaluation of $f\ v^\alpha$ as synthesizing a partial instantiation of α, and thus *dynamically inferring* a (partial) type for f's input. We can extract this type from an evaluation trace by applying the final type substitution to α. Formally, we say that if $\langle f\ v^\alpha, \varnothing, \varnothing \rangle \hookrightarrow^\tau \langle \mathbf{e}, \sigma, \theta \rangle$, then the *partial input type* of f up to τ, written $\rho_\tau(f)$, is $\theta(\alpha)$.

Compatibility

A type s is *compatible* with a type t, written $s \sim t$, if $\exists\theta.\ \theta(s) = \theta(t)$. That is, two types are compatible if there exists a type substitution that maps both types to the same type. A value v is *compatible* with a type t, written $v \sim t$, if $\mathrm{ty}(v) \sim t$, that is, if the dynamic type of v is

compatible with t.

Type Refinement

A type s is a *refinement* of a type t, written $s \preceq t$, if $\exists\theta.s = \theta(t)$. In other words, s is a refinement of t if there exists a type substitution that maps t directly to s. A type t is a *refinement* of a value v, written $t \preceq v$, if $t \preceq \mathrm{ty}(v)$, *i.e.* if t is a refinement of the dynamic type of v.

Preservation

We prove two preservation lemmas. First, we show that each evaluation step refines the partial input type of f, thus preserving type compatibility.

Lemma 2. *If* $\tau \doteq \langle f\ v^\alpha, \varnothing, \varnothing\rangle, \ldots, \langle e, \sigma, \theta\rangle$ *and* $\tau' \doteq \tau, \langle e, \sigma, \theta\rangle \hookrightarrow \langle e', \sigma', \theta'\rangle$ *(i.e.* τ' *is a single-step extension of* τ*) and* $\rho_\tau(f) \neq \rho_{\tau'}(f)$ *then* $\theta' = \theta + \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n\}$.

Proof. By case analysis on the evaluation rules. α does not change, so if the partial input types differ then $\theta \neq \theta'$. Only narrow can change θ, via $\mathcal{U}$, which can only extend θ. ■

Second, we show that at each step of evaluation, the partial input type of f is a refinement of the instantiation of v^α.

Lemma 3. *For all traces* $\tau \doteq \langle f\ v^\alpha, \varnothing, \varnothing\rangle, \ldots, \langle e, \sigma, \theta\rangle$, $\rho_\tau(f) \preceq \sigma(v^\alpha)$.

Proof. By induction on τ. In the base case $\tau = \langle f\ v^\alpha, \varnothing, \varnothing\rangle$ and α trivially refines v^α. In the inductive case, consider the single-step extension of τ, $\tau' = \tau, \langle e', \sigma', \theta'\rangle$. We show by case analysis on the evaluation rules that if $\rho_\tau(f) \preceq \sigma(v^\alpha)$, then $\rho_{\tau'}(f) \preceq \sigma'(v^\alpha)$. ■

Incompatible Types Are Wrong

For all types that are *incompatible* with the partial input type up to τ, there exists a value that will cause f to get stuck in *at most* k steps, where k is the length of τ.

Lemma 4. *For all types* t*, if* $\langle f\ v^\alpha, \varnothing, \varnothing\rangle \hookrightarrow^\tau \langle e, \sigma, \theta\rangle$ *and* $t \nsim \rho_\tau(f)$*, then there exists a* v *such that* $\mathrm{ty}(v) = t$ *and* $\langle f\ v, \varnothing, \varnothing\rangle \hookrightarrow^* \langle \mathtt{stuck}, \sigma', \theta'\rangle$ *in at most* k *steps, where* k *is the length of* τ.

Proof. We can construct v from τ as follows. Let

$$\tau_i = \langle f\ v^\alpha, \varnothing, \varnothing\rangle, \ldots, \langle e_{i-1}, \sigma_{i-1}, \theta_{i-1}\rangle, \langle e_i, \sigma_i, \theta_i\rangle$$

be the shortest prefix of τ such that $\rho_{\tau_i}(f) \nsim t$. We will show that $\rho_{\tau_{i-1}}(f)$ must contain some other hole α' that is instantiated at step i. Furthermore, α' is instantiated in such a way that $\rho_{\tau_i}(f) \nsim t$. Finally, we will show that if we had instantiated α' such that $\rho_{\tau_i}(f) \sim t$, the current step would have gotten `stuck`.

By Lemma 2 we know that $\theta_i = \theta_{i-1} + \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n\}$. We will assume, without loss of generality, that $\theta_i = \theta_{i-1} + \{\alpha' \mapsto t'\}$. Since θ_{i-1} and θ_i differ only in α' but the resolved types differ, we have $\alpha' \in \rho_{\tau_{i-1}}(f)$ and $\rho_{\tau_i}(f) = \rho_{\tau_{i-1}}(f)\,[t'/\alpha']$. Let s be a concrete type such that $\rho_{\tau_{i-1}}(f)\,[s/\alpha'] = t$. We show by case analysis on the evaluation rules that $\langle e_{i-1}, \sigma_{i-1}, \theta_{i-1} + \{\alpha' \mapsto s\}\rangle \hookrightarrow \langle \texttt{stuck}, \sigma, \theta\rangle$.

Finally, by Lemma 3 we know that $\rho_{\tau_{i-1}}(f) \preceq \sigma_{i-1}(v^\alpha)$ and thus $\alpha' \in \sigma_{i-1}(v^\alpha)$. Let $u = \mathrm{gen}(s, \theta)$ and $v = \sigma_{i-1}(v^\alpha)\left[u/v'^{\alpha'}\right][s/\alpha']$. $\langle f\ v, \varnothing, \varnothing\rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma, \theta\rangle$ in i steps. ■

Proof of Theorem 1. Suppose τ witnesses that f gets stuck, and let $s = \rho_\tau(f)$. We show that *all* types t have stuck-inducing values by splitting cases on whether t is compatible with s.

Case $s \sim t$**:** Let $\tau = \langle f\ v^\alpha, \varnothing, \varnothing\rangle, \ldots, \langle \texttt{stuck}, \sigma, \theta\rangle$. The value $v = \sigma(v^\alpha)$ demonstrates that $f\ v$ gets stuck.

Case $s \nsim t$**:** By Lemma 4, we can derive a v from τ such that $\mathrm{ty}(v) = t$ and $f\ v$ gets stuck.

■

3.2.4 Search Algorithm

So far, we have seen how a trace leading to a stuck configuration yields a general witness demonstrating that the program is ill-typed (*i.e.* goes wrong for at least one input of every type). In particular, we have shown how to non-deterministically find a witnesses for a function of a *single* argument.

We must address two challenges to convert the semantics into a *procedure* for finding witnesses. First, we must resolve the non-determinism introduced by gen. Second, in the presence of higher-order functions and currying, we must determine how many concrete values to generate to make execution go wrong (as we cannot rely upon static typing to provide this information.)

The witness generation procedure GenWitness is formalized in Figure 3.9. Next, we describe its input and output, and how it addresses the above challenges to search the space of possible executions for general type error witnesses.

Inputs and Outputs

The problem of generating inputs is undecidable in general. Our witness generation procedure takes two inputs: (1) a search bound k which is used to define the *number* of traces to explore[4] and (2) the target expression e that contains the type error (which may be a curried function of multiple arguments). The witness generation procedure returns a list of (general) witness expressions, each of which is of the form $e\ v_1 \ldots v_n$. The *empty* list is returned when no witness can be found after exploring k traces.

Modeling Semantics

We resolve the non-determinism in the operational semantics (§ 3.2.2) via the procedure

$$\mathsf{eval} : e \longrightarrow \langle v \cup \texttt{stuck}, \sigma, \theta \rangle^*$$

Due to the non-determinism introduced by gen, a call eval(e) returns a *list* of possible results of the form $\langle v \cup \texttt{stuck}, \sigma, \theta \rangle$ such that $\langle e, \varnothing, \varnothing \rangle \hookrightarrow^* \langle v \cup \texttt{stuck}, \sigma, \theta \rangle$.

Currying

We address the issue of currying by defining a procedure Saturate(e), defined in Figure 3.8, that takes as input an expression e and produces a *saturated* expression of the form

[4] We assume, without loss of generality, that all traces are finite.

Saturate	:	$e \to e$	
Saturate(e)	=	**case** eval(e) **of**	
$\langle \lambda x.e, \sigma, \theta \rangle, \ldots$	$\to$	Saturate($e\ v^{\alpha}$)	(v, α are fresh)
_	$\to$	e	

Figure 3.8. Generating a saturated application.

$e\ v_1^{\alpha_1} \ldots v_n^{\alpha_n}$ that *does not* evaluate to a lambda. This is achieved with a simple loop that keeps adding holes to the target application until evaluating the term yields a non-lambda value.

Generating Witnesses

Finally, Figure 3.9 summarizes the overall implementation of our search for witnesses with the procedure GenWitness(k, e), which takes as input a bound k and the target expression e, and returns a list of witness expressions $e\ v_1 \ldots v_n$ that demonstrate how the input program gets stuck. The search proceeds as follows.

1. We invoke Saturate(e) to produce a *saturated* application e_{sat}.
2. We take the first k traces returned by eval on the target e_{sat}, and
3. We extract the substitutions corresponding to the `stuck` traces, and use them to return the list of witnesses.

We obtain the following corollary of Theorem 1:

Corollary (Witness Generation)**.** *If*

$$\text{GenWitness}(k,\ e) = \langle e\ v_1 \ldots v_n, \sigma, \theta \rangle, \ldots$$

then for all types $t_1 \ldots t_n$ there exist values $w_1 \ldots w_n$ such that

$$\langle e\ w_1 \ldots w_n, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma', \theta' \rangle$$

Proof. For any function f of multiple arguments, we can define f' as the uncurried version of f that takes all of its arguments as a single nested pair, and then apply Theorem 1 to f'. ■

$$
\begin{array}{lcll}
\mathsf{GenWitness} & : & \mathsf{Nat} \times e \rightarrow e^* & \\
\mathsf{GenWitness}(n,\ e) & = & \{\sigma(e_{sat}) \mid \sigma \in \Sigma\} & \\
\quad \textbf{where} & & & \\
\quad\quad e_{sat} & = & \mathsf{Saturate}(e) & (1) \\
\quad\quad res & = & \mathsf{take}(n,\ \mathsf{eval}(e_{sat})) & (2) \\
\quad\quad \Sigma & = & \{\sigma \mid \langle \texttt{stuck}, \sigma, \theta \rangle \in res\} & (3)
\end{array}
$$

Figure 3.9. Generating witnesses.

3.3 Explaining Type Errors With Traces

A trace, on its own, is too detailed to be a good explanation of the type error. One approach is to use the witness input to step through the program with a *debugger* to observe how the program evolves. This route is problematic for two reasons. First, existing debuggers and interpreters for typed languages (*e.g.* OCaml) typically require a type-correct program as input. Second, we wish to have a quicker way to get to the essence of the error, *e.g.* by skipping over irrelevant sub-computations, and focusing on the important ones.

In this section we present a novel way to debug executions. First, we develop a notion of a *reduction graphs* and extend our semantics with a form of *tracing* so that they incrementally collect the edges in the graph (§ 3.3.1). Next, we express a set of common *interactive debugging* steps as graph traversals (§ 3.3.2), yielding an novel interactive debugger that allows the user to effectively visualize *how* the program goes (wrong).

3.3.1 Tracing Semantics

Reduction Graphs

A *steps-to* edge is a pair of expressions $e_1 \rightsquigarrow e_2$, which intuitively indicates that e_1 reduces, in a single step, to e_2. A *reduction graph* is a set of steps-to edges:

$$G ::= \{e \rightsquigarrow e, \ldots\}$$

Tracing Semantics

We extend the transition relation (§ 3.2.2) to collect the set of edges corresponding to the reduction graph. Concretely, we extend the operational semantics to a relation of the form $\langle e, \sigma, \theta, G\rangle \hookrightarrow \langle e', \sigma', \theta', G'\rangle$ where G' collects the edges of the transition.

Collecting Edges

Next, we describe the general recipe for extending a transition rule to collect edges. The general recipe for collecting steps-to edges is to record the consequent of each original rule in the trace. That is, each original judgment $\langle e, \sigma, \theta\rangle \hookrightarrow \langle e', \sigma', \theta'\rangle$ becomes $\langle e, \sigma, \theta, G\rangle \hookrightarrow \langle e', \sigma', \theta', G \cup \{e \rightsquigarrow e'\}\rangle$. As the translation is mechanical, we will not discuss it further.

3.3.2 Interactive Debugging

Next, we show how to build a visual interactive debugger from the traced semantics, by describing the visualization *state i.e.* what the user sees at any given moment, the set of *commands* available to user and what they do, and finally how we use a command to *update* the visualization state. In what follows, for clarity of exposition, we assume we have a (global) trace: $\langle e_0, \varnothing, \varnothing, \varnothing\rangle \hookrightarrow \langle e_n, \sigma, \theta, G\rangle$, where e_0 and e_n are the *initial* and *final* expressions respectively.

Visualization State

A *visualization state* V is a *directed graph* whose vertices are expressions and whose edges are such that each vertex has at most one predecessor and at most one successor. In other words, the visualization state looks like a set of linear lists of expressions as shown in Figure 3.10. The *initial state* is the graph containing a single edge linking the initial and final expressions.

Visualization Context

The *visualization context* of each expression e in the visualization state V is the (unique) linear chain in which the expression e belongs. We write $\mathrm{Root}(V, e)$ for the *first* (or root) expression appearing in the visualization context of e in V.

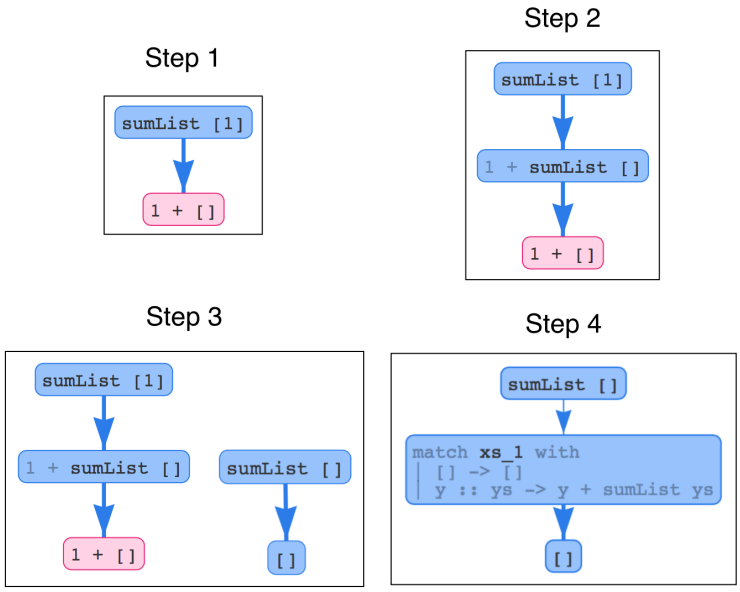


Figure 3.10. A sequence of interactions with the trace of `sumList [1]`. The stuck term is red, in each node the redex is highlighted. Thick arrows denote a multi-step transition, thin arrows denote a single-step transition. We start in step 1. In step 2 we jump forward from the witness to the next function call. In step 3 we step into the recursive `sumList []` call, which spawns a new "thread" of execution. In step 4 we take a single step forward from `sumList []`.

Commands

Our debugger supports the following *commands*, each of which is parameterized by a single expression (vertex) selected from the (current) visualization state:

- StepF, StepB: show the result of a single step forward or backward respectively,
- JumpF, JumpB: show the result of taking multiple steps (a *"big"* step) upto the first beta-reduction forward or backward respectively,
- StepInto: show the result of stepping into a function call in a sub-term, isolating it from the context,
- StepOver: show the result of skipping over a function call in a sub-term.

$$
\begin{array}{lcl}
\mathsf{StepF}(V, e) & \doteq & e' \quad \text{where } e \rightsquigarrow e' \in G \\
\mathsf{StepB}(V, e) & \doteq & e' \quad \text{where } e' \rightsquigarrow e \in G \text{ and } e' \in \mathsf{Path}(V, e) \\
\mathsf{JumpF}(V, e) & \doteq & \begin{cases} e' & \text{if } e' = v\ v' \\ \mathsf{JumpF}(V, e') & \text{otherwise} \end{cases} \quad \text{where } e' = \mathsf{StepF}(V, e) \\
\mathsf{JumpB}(V, e) & \doteq & \begin{cases} e' & \text{if } e' = v\ v' \\ \mathsf{JumpB}(V, e') & \text{otherwise} \end{cases} \quad \text{where } e' = \mathsf{StepB}(V, e) \\
\mathsf{StepInto}(V, e) & \doteq & e'\,[v'/x] \quad \text{if } e = C[v\ v'] \text{ and } v\ v' \rightsquigarrow e'\,[v'/x] \\
\mathsf{StepOver}(V, e) & \doteq & C[v''] \quad \text{if } e = C[v\ v'] \text{ and } v\ v' \rightsquigarrow^* v'' \in G \\
\mathsf{Path}(V, e) & \doteq & \{e' \mid \mathsf{Root}(V, e) \rightsquigarrow^* e' \in G \text{ and } e' \rightsquigarrow^* e \in G\}
\end{array}
$$

Figure 3.11. Rules for computing the *next* term given a visualization state V, selected term e and command.

Update

Figure 3.11 shows how we compute the *next* expression (to be added to the visualization state) given the current visualization state V, command *Cmd* and selected expression e. It is straightforward to then *update* the visualization graph by adding the new term before (resp. after) the selected expression e if the command was a step or jump forward (resp. backward), or to create a new visualization context if the command was StepInto.

3.4 Evaluation

We have implemented a prototype of our search procedure and trace visualization for a purely functional subset of OCaml — with polymorphic types and records, but no modules, objects, or polymorphic variants — in a tool called NanoMaLy. We treat explicit type signatures, *e.g.* `(x : int)`, as primitive operations that narrow the type of the wrapped value. In our implementation we instantiated gen with a simple random generation of values, which we will show suffices for the majority of type errors.

Evaluation Goals

There are four questions we seek to answer with our evaluation:

1. ***Witness Coverage*** (§ 3.4.2, 3.4.3) How many ill-typed programs *admit* witnesses?
2. ***Witness Complexity*** (§ 3.4.4) How *complex* are the traces produced by the witnesses?
3. ***Witness Utility*** (§ 3.4.5, 3.4.6) How *helpful* are the witnesses in debugging type errors?
4. ***Witness-based Blame*** (§ 3.4.7) Can witnesses be used to *locate* the source of an error?

In the sequel we present our experimental methodology (§ 3.4.1) and then answer the above questions. However, for the impatient reader, we first summarize our main results:

1. Most Type Errors Admit Witnesses

Our prime result is that the vast majority of static type errors, around 85%, do in fact admit a dynamic witness. Further, NanoMaLy efficiently synthesizes witnesses with its randomized search; it can synthesize a witness for over 75% of programs in under one second, *i.e.* fast enough for interactive use.

2. Jump-Compressed Traces Are Small

We find that our jump-compression heuristic effectively abstracts the pedestrian details of computation, compressing the median trace with 14–15 single-step reductions to only 4 jumps. Over 80% of programs have a jump-compressed trace with at most 10 jumps, providing a bird's-eye view from which we can launch a more in-depth investigation.

3. Witnesses Help Novices

A witness should also help programmers *understand* and *fix* type errors. We use a set of ill-typed student programs to show that NanoMaLy's witnesses effectively demonstrate the runtime error that the type system prevented. Furthermore, we find, in a study of undergraduate students, that NanoMaLy's witnesses lead to more accurate diagnoses and fixes of type errors than OCaml's type error messages.

4. Witnesses Assign Blame

Finally, we present a simple heuristic that allows us to use witnesses to *automatically* assign blame for type errors. We treat the values inside the stuck term as *sources* of typing constraints and the stuck term itself as a *sink*, producing a slice of the program that likely contains the error. Using this heuristic, NanoMaLy's witnesses are competitive with the state-of-the-art localization tools Mycroft and SHErrLoc.

3.4.1 Methodology

We answer the first two questions on two sets of ill-typed programs, *i.e.* programs that were rejected by the OCaml compiler because of a type error. The first dataset comes from the Spring 2014 dataset described in Chapter 2, which includes 4,407 distinct, ill-typed OCaml programs from a cohort of 46 students. The second dataset — widely used in the literature — comes from a graduate-level course at the University of Washington [61], from which we extracted 284 ill-typed programs. Both datasets contain relatively small programs, the largest being 348 SLoC; however, they demonstrate a variety of functional programming idioms including (tail) recursive functions, higher-order functions, and polymorphic and algebraic data types.

We answer the third question in two steps. First, we present a qualitative evaluation of NanoMaLy's traces on a selection of programs drawn from the UCSD dataset. Second, we present a quantitative user study of students in the University of Virginia's Spring 2016 undergraduate Programming Languages (CS 4501) course. As part of an exam, we presented the students with ill-typed OCaml programs and asked them to (1) *explain* the type error, and (2) *fix* the type error (IRB #2014009900). For each problem the students were given the ill-typed program and either OCaml's error message or NanoMaLy's jump-compressed trace.

We answer the last question on a subset of the UCSD dataset. For each ill-typed program compiled by a student, we identify the student's *fix* by searching for the first type-correct program that the student subsequently compiled. We then use an expression-level *diff* [59] to determine which sub-expressions changed between the ill-typed program and the student's fix,

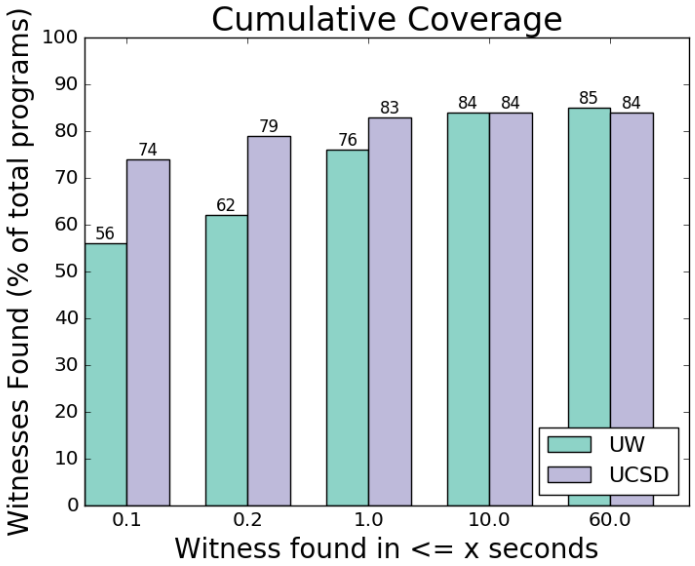


Figure 3.12. Results of our coverage testing. Our random search successfully finds witnesses for 76–83% of the programs in under one second, improving to 84–85% in under 10 seconds.

and treat those expressions as the source of the type error.

3.4.2 Witness Coverage

We ran our search algorithm on each program for 1,000 iterations, with the entry point set to the function that OCaml had identified as containing a type error. Due to the possibility of non-termination we set a timeout of one minute total per program. We also added a naïve check for infinite recursion; at each recursive function call we check whether the new arguments are identical to the current arguments. If so, the function cannot possibly terminate and we report an error. While not a *type error*, infinite recursion is still a clear bug in the program, and thus valuable feedback for the user.

Results

The results of our experiments are summarized in Figures 3.12 and 3.13. In both datasets our tool was able to find a witness for over 75% of the programs in under one second, *i.e.* fast enough to be integrated as a compile-time check. If we extend our tolerance to a 10 second timeout, we reach 84% coverage, and if we allow a 60 second search, we hit a maximum of

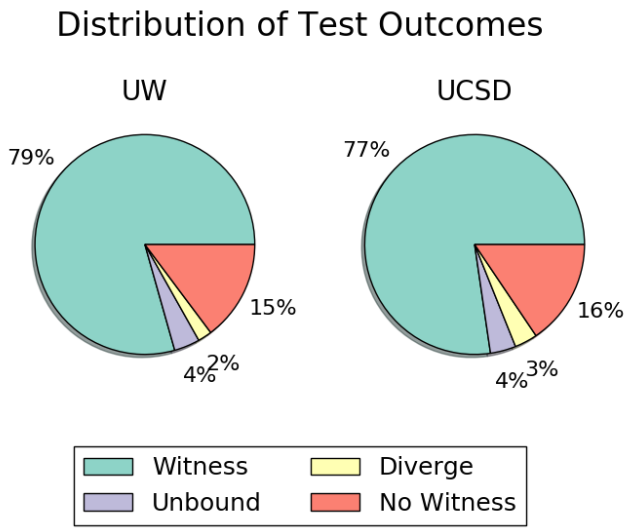


Figure 3.13. Distribution of test outcomes. In both datasets we detect actual type errors at least 77% of the time, unbound variables or constructors 4% of the time, and diverging loops 2–3% of the time. For the remaining 15–16% of the programs we are unable to provide any useful feedback.

84–85% coverage. Interestingly, while the vast majority of witnesses corresponded to a type-error, as expected, 4% triggered an unbound variable error (even though OCaml reported a type error) and 3% triggered an infinite recursion error. For the remaining 15–16% of programs we were unable to provide any useful feedback as they either completed 1,000 tests successfully, or timed out after one minute. While a more advanced search procedure, *e.g.* dynamic-symbolic execution, could likely uncover more errors, our experiments suggest that type errors are coarse enough (or that novice programs are *simple* enough) that these techniques are not necessary.

3.4.3 How safe are the "safe" programs?

An immediate question arises regarding the 15–16% of programs for which we could not synthesize a witness: are they *actually* safe (*i.e.* is the type system being too conservative), or did NanoMaLy simply fail to find a witness?

To answer this question, we investigated the 732 UCSD programs for which we failed to find a witness. We used a combination of automatic and manual coding to categorize these programs into four classes. The first class is easily detected by NanoMaLy itself, and thus

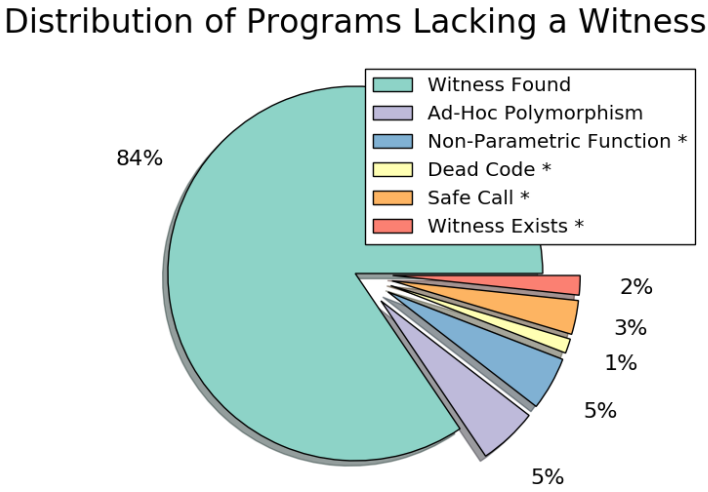


Figure 3.14. Results of our investigation into programs where NanoMaLy did not produce a witness. A "*" denotes that the percentage is an estimate based on a random sampling of 50 programs.

admits a precise count. This left us with 504 programs that required manual coding; we selected a random sample of 50 programs to investigate, and will report results based on that sample. Figure 3.14 summarizes the results of our investigation — we note the classes that were based on the random sample with a "*".

Ad-Hoc Polymorphism

We found that for 5% programs NanoMaLy got stuck when it tried to compare two holes. OCaml provides polymorphic equality and comparison operators, overloading them for each type. While convenient to use, they pose a challenge for NanoMaLy's combination of execution and inference. For example, consider the following ill-typed factorial function, parameterized by a lower bound.

```
let rec fac n m =
  if n <= m then
    true
  else
    n * fac (n - 1) m
```

When given `fac`, NanoMaLy will generate two fresh holes $\nu_1^{\alpha_1}$ and $\nu_2^{\alpha_2}$ and proceed directly into the `n <= m` comparison. We cannot (yet) instantiate either hole because we have no constraints on the αs (we know they must be equal, but nothing else), and furthermore we do not know what constraints we may encounter later on in the program. Thus, we cannot perform the comparison and proceed, and must give up our search for a witness, even though one obviously exists, any pair of `n` and `m` such that `n <= m` is false.

Extending NanoMaLy with support for symbolic execution would alleviate this issue, as we could then begin symbolically executing the program until we learn how to instantiate `n` and `m`. Alternatively, we could *speculatively* instantiate both `n` and `m` with some arbitrary type, and proceed with execution until we discover a type error. This speculative instantiation is, of course, unsound; we would have to take care to avoid reporting frivolous type errors that were caused by such instantiations. We would need to track which holes were instantiated speculatively to distinguish type errors that would have happened regardless, as in `fac`, from type errors that were caused by our instantiation.

Further, suppose that our speculative instantiation induces a frivolous type error. For example, suppose we are given

```
let bad x y =
  if x < y then
    x *. y
  else
    0.0
```

and choose to (speculatively) instantiate `x` and `y` as `int`s and proceed down the "true" branch. We will quickly discover this was the wrong choice, as they are immediately narrowed to `float`s. We must now backtrack and try a different instantiation, but we no longer need to choose one at random. Since our instantiation was speculative, and `x` and `y` were *originally* holes, we can treat the `*.` operator as a normal narrowing point with two holes. This tells us that the *correct* instantiation was in fact `float`, and we can then proceed as normal from the backtracking point with a concrete choice of `float`s. Thus, it appears that speculative instantiation of holes may

be a useful, lightweight alternative to symbolic execution for our purposes.

Non-Parametric Function Type *

5% of programs lack a witness in our semantics due to our non-parametric `fun` type for functions. Recall that our goal is to expose the runtime errors that would have been prevented by the type systems. At runtime, it is always safe to call a function, thus we give functions a simple type `fun` that says they may be applied, but says nothing about their inputs or outputs. But consider the following `clone` function, which is supposed to produce a list containing `n` copies of the input `x`.

```
let rec clone x n =
  if n > 0 then
    clone [x] (n - 1)
  else
    []
```

Unfortunately, the student instead constructs an `n`-level nested list containing a single `x`. The OCaml compiler rejects this program because the recursive call to `clone` induces a cyclic typing constraint `'a = 'a list`, capturing the fact that each call increases the nesting of the list. NanoMaLy fails to catch this because we do not track the types of the inputs to `clone`.

We note, however, that `clone` cannot go wrong; it is perfectly safe to repeatedly enclose a list inside another (disregarding the fact that the nested list is never returned). Still, such a function would be very difficult to *call* safely, as the programmer would have to reason about the dependency between the input `n` and the nesting of the output list, which cannot be expressed in OCaml's type system.

Thus, it is not particularly satisfying that NanoMaLy fails to produce a witness here; a possible solution could be to track the types of the inputs, and demonstrate to the user how they change between recursive calls. This would require maintaining a typing environment of variables in addition to the environments we maintain for holes. We would have to modify the rule App-G from Figure 3.6 to additionally narrow the function's type against the concrete

inputs. However, we would want to ensure that this narrow cannot fail — it is preferable to report a stuck term as that provides a fuller view of the error. Rather, we would note which evaluation steps induced incompatible type refinements, and if a traditional witness cannot be found, we could then report a trace expanded to show precisely these steps. This represents only a modest extension to our semantics, and would be interesting to explore further.

Dead Code and "Safe" Function Calls *

4% of programs contained type errors that were unreachable, either because they were dead code, or because the student called the function with inputs that could not trigger the error.

1% contained type errors that were unreachable by any inputs, often due to overlapping patterns in a `match` expression. While technically safe, dead code is generally considered a maintenance risk, as the programmer may not realize that it is dead [112] or may accidentally bring it back to life [103]. Thus, a warning like that provided by OCaml's pattern exhaustiveness checker would be helpful.

A further 3% included a function call where the student supplied ill-typed inputs, but the path induced by the call did not contain an error. Consider the following `assoc` function, which looks up a key in an *association list*, returning a default if it cannot be found.

```
let rec assoc (d, k, l) = match l with
  | (ki, vi)::tl ->
     if ki = k then
       vi
     else
       assoc (d, k, tl)
  | _ -> d

let _ = assoc ([], 123, [(123, "sad"); (321, "happy")])
```

The student's definition of `assoc` is correct, but OCaml rejects their subsequent call because the

default value `[]` is incompatible with the `string` values in the list. In this particular call the key `123` is in the list, so the default will not be used (even if it were, there would not be an error) and OCaml's complaint is moot. Of course, OCaml cannot be expected to know that this particular call is safe, its type system is not sophisticated enough to express the necessary conditions.

Witness Exists *

We found that only 2% of programs admit a witness that NanoMaLy was unable to discover. Slightly over half involved synthesizing a *pair* of specially-crafted inputs that would result in the function returning values of incompatible types. The rest required synthesizing an input that would trigger a particular path through the program, and would likely have been caught by symbolic execution.

Summary

Our investigation suggests that the vast majority of programs for which we fail to find a witness do not, in fact, admit a witness. These programs were generally cases where OCaml's type system was overly conservative. Of course, the conservatism is somewhat justified as each case pointed to code that would be difficult to use or maintain; it would be interesting to investigate how demonstrate these issues in an intuitive manner.

3.4.4 Witness Complexity

For each of the ill-typed programs for which we could find a witness, we measure the complexity of the generated trace using two metrics.

1. ***Single-step:*** The size of the trace after expanding all of the single-step edges from the witness to the stuck term, and
2. ***Jump-compressed:*** The size of the jump-compressed trace.

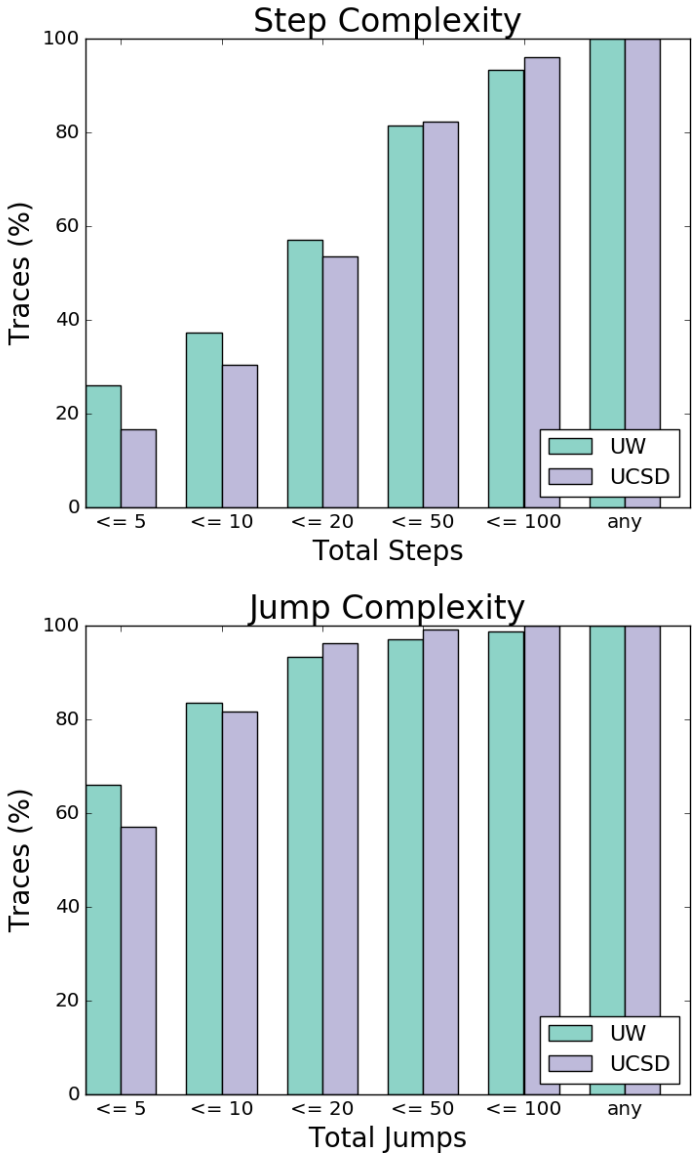


Figure 3.15. Complexity of the generated traces. Over 80% of the combined traces have a jump complexity of at most 10, with an average complexity of 7 and a median of 5.

Results

The results of the experiment are summarized in Figure 3.15. The average number of single-step reductions per trace is 17 for the UCSD dataset (42 for the UW dataset) with a maximum of 2,745 (*resp.* 982) and a median of 15 (*resp.* 15). The average number of jumps per trace is 7 (*resp.* 9) with a maximium of 353 (*resp.* 221) and a median of 4 (*resp.* 4). In both datasets about 60% of traces have at most 5 jumps, and 80% or more have at most 10 jumps.

3.4.5 Qualitative Evaluation of Witness Utility

Next, we present a *qualitative* evaluation that compares the explanations provided by NanoMaLy's dynamic witnesses with the static reports produced by the OCaml compiler and SHErrLoc, a state-of-the-art fault localization approach [118]. In particular, we illustrate, using a series of examples drawn from student programs in the UCSD dataset, how NanoMaLy's jump-compressed traces can get to the heart of the error. Our approach highlights the conflicting values that cause the program to get stuck, rather that blaming a single one, shows the steps necessary to reach the stuck state, and does not assume that a function is correct just because it type-checks. For each example we will present: (1) the code; (2) the error message returned OCaml; (3) the error locations returned by OCaml and SHErrLoc; and (4) NanoMaLy's jump-compressed trace.

Example: Recursion with Bad Operator

The recursive function `sqsum` should square each element of the input list and then compute the sum of the result.

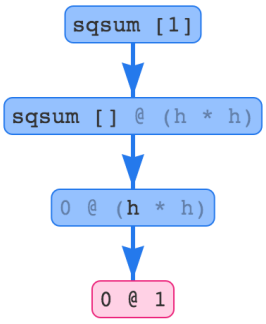
```
1  let rec sqsum xs = match xs with
2    | [] -> 0
3    | h::t -> sqsum t @ (h * h)
```

Unfortunately the student has used the list-append operator `@` instead of `+`. Both OCaml and SHErrLoc blame the *wrong location*, the recursive call `sqsum t`, with the message

```
This expression has type
```

```
    int
  but an expression was expected of type
    'a list
```

NanoMaLy produces a trace showing how the evaluation of `sqsum [1]` gets stuck.



The trace highlights the entire stuck term (not just the recursive call), emphasizing the *conflict* between `int` and `list` rather than assuming one or the other is correct.

Example: Recursion with Bad Base Case

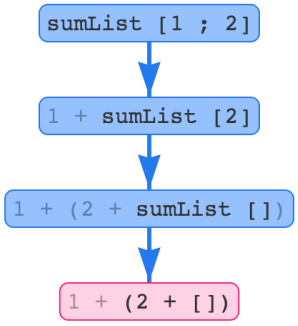
The function `sumList` should add up the elements of its input list.

```
1  let rec sumList xs = match xs with
2    | []    -> []
3    | y::ys -> y + sumList ys
```

Unfortunately, in the base case, it returns `[]` instead of `0`. SHErrLoc blames the base case, and OCaml assumes the base case is correct and blames the *recursive call* on line 3:

```
  This expression has type
    'a list
  but an expression was expected of type
    int
```

Both of the above are parts of the full story, which is summarized by NanoMaLy's trace showing how `sumList [1; 2]` gets stuck at `2 + []`.



The trace clarifies (via the third step) that the `[]` results from the recursive call `sumList []`, and shows how it is incompatible with the subsequent + operation.

Example: Bad Helper Function that Type-Checks

The function `digitsOfInt` should return a list of the digits of the input integer.

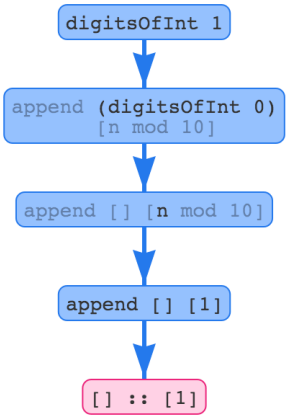
```
1   let append x xs =
2     match xs with
3     | [] -> [x]
4     | _  -> x :: xs
5
6   let rec digitsOfInt n =
7     if n <= 0 then
8       []
9     else
10      append (digitsOfInt (n / 10)) [n mod 10]
```

Unfortunately, the student's append function *conses* an element onto a list instead of appending two lists. Though incorrect, append still type-checks and thus OCaml and SHErrLoc blame the *use-site* on line 10.

```
This expression has type
  int
but an expression was expected of type
```

```
'a list
```

In contrast, NanoMaLy makes no assumptions about append, yielding a trace that illustrates the error on line 4, by highlighting the conflict in consing a list onto a list of integers.



Example: Higher-Order Functions

The higher-order function `wwhile` is supposed to emulate a traditional while-loop. It takes a function `f` and repeatedly calls `f` on the first element of its output pair, starting with the initial b, till the second element is `false`.

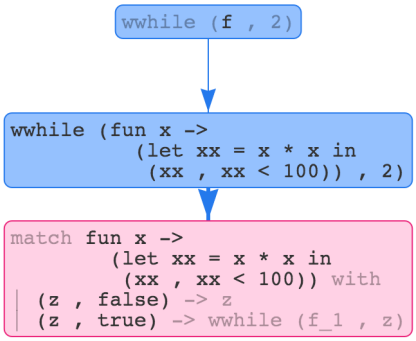
```
1   let rec wwhile (f,b) =
2     match f with
3     | (z, false) -> z
4     | (z, true)  -> wwhile (f, z)
5
6   let f x =
7     let xx = x * x in
8     (xx, (xx < 100))
9
10  let _ = wwhile (f, 2)
```

The student has forgotten to *apply* `f` at all on line 2, and just matches it directly against a pair. This faulty `wwhile` definition nevertheless typechecks, and is assumed to be correct by both

OCaml and SHErrLoc which blame the use-site on line 10.

```
This expression has type
  int -> int * bool
but an expression was expected of type
  'a * bool
```

NanoMaLy synthesizes a trace that draws the eye to the true error: the `match` expression on line 2, and highlights the conflict in matching a function against a pair pattern.



By highlighting conflicting values, *i.e.* the source and sink of the problem, and not making assumption about function correctness, NanoMaLy focusses the user's attention on the piece of code that is actually relevant to the error.

3.4.6 Quantitative Evaluation of Witness Utility

We assigned four problems to the 60 students in the course: the `sumList`, `digitsOfInt`, and `wwhile` programs from § 3.4.5, as well as the following `append` program

```
1  let append x l =
2    match x with
3    | []   -> l
4    | h::t -> h :: t :: l
```

which triggers an occurs-check error on line 4. For each problem the students were given the ill-typed program and either OCaml's error or NanoMaLy's jump-compressed trace; the full

user study is available in Appendix B. Due to the nature of an in-class exam, not every student answered every question; we received between 13 and 28 (out of a possible 30) responses for each problem-tool pair.

We then instructed four annotators (one of whom is an author, the other three are teaching assistants at UCSD) to classify the answers as correct or incorrect. We performed an inter-rater reliability (IRR) analysis to determine the degree to which the annotators consistently graded the exams. As we had more than two annotators assigning nominal ("correct" or "incorrect") ratings we used Fleiss' kappa [32] to measure IRR. Fleiss' kappa is measured on a scale from 1, indicating total agreement, to -1, indicating total disagreement, with 0 indicating random agreement.

Finally, we used a one-sided Mann-Whitney U test [66] to determine the significance of our results. The null hypothesis was that the responses from students given NanoMaLy's witnesses were drawn from the same distribution as those given OCaml's errors, *i.e.* NanoMaLy had no effect. Since we used a one-sided test, the alternative to the null hypothesis is that NanoMaLy had a *positive* effect on the responses. We reject the null hypothesis in favor of the alternative if the test produces a significance level $p < 0.05$, a standard threshold for determining statistical significance.

Threats to Validity

Measuring understanding is a difficult task; the following summarize the threats to the validity of our results.

Construct. We used the correctness of the student's explanation of, and fix for, the type error as a proxy for her understanding, but it is possible that other metrics would produce different results.

Internal. We assigned students randomly to two groups. The first was given OCaml's errors for `append` and `digitsOfInt`, and NanoMaLy's trace for `sumList` and `wwhile`; the second was given the opposite assignment of errors and traces. This assignment ensured that: (1) each

student was given OCaml and NanoMaLy problems; and (2) each student was given an "easy" and "hard" problem for both OCaml and NanoMaLy. Students without sufficient knowledge of OCaml could affect the results, as could the time-constrained nature of an exam. For these reasons we excluded any answers left blank from our analysis.

External. Our experiment used students in the process of learning OCaml, and thus may not generalize to all developers. The four programs were chosen manually, via a random selection and filtering of the programs in the UCSD dataset. In some cases we made minor simplifying edits (*e.g.* alpha-renaming, dead-code removal) to the programs to make them more understandable in the short timeframe of an exam; however, we never altered the resulting type-error. A different selection of programs may lead to different results.

Conclusion. We collected exams from 60 students, though due to the nature of the study not every student completed every problem. The number of complete submissions ranges from 13 (for the NanoMaLy version of `wwhile`) to 28 (for the OCaml version of `sumList`), out of a maximum of 30 per program-tool pair. Our results are statistically significant in only 2 out of 8 tests; however, collecting more responses per test pair was not possible as it would require having students answer the same problem twice (once with OCaml and once with NanoMaLy).

Results

The measured kappa values were $\kappa = 0.72$ for the explanations and $\kappa = 0.83$ for the fixes; while there is no formal notion for what consititutes strong agreement [54], kappa values above 0.60 are often called "substantial" agreement [55]. Figure 3.16 summarizes a single annotator's results, which show that students given NanoMaLy's jump-compressed trace were consistently more likely to correctly explain and fix the type error than those given OCaml's error message. Across each problem the NanoMaLy responses were marked correct $10 - 30\%$ more often than the OCaml responses, which suggests that the students who had access to NanoMaLy's traces had a better understanding of the type errors; however, only the append tests were statistically significant at $p < 0.05$.

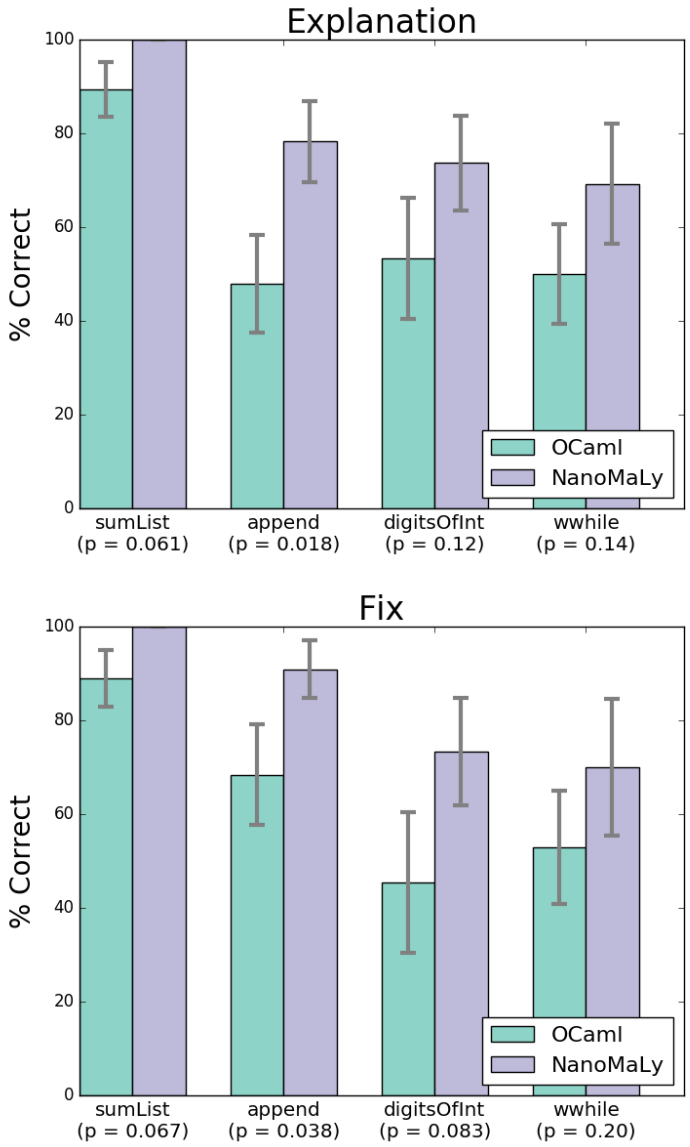


Figure 3.16. A classification of students' explanations and fixes for type errors, given either OCaml's error message or NanoMaLy's jump-compressed trace. The students given NanoMaLy's jump-compressed trace consistently scored better ($\geq$ 10%) than those given OCaml's type error. We report the result of a one-sided Mann-Whitney U test for statistical significance in parentheses.

3.4.7 Locating Errors with Witnesses

We have seen that NanoMaLy can effectively synthesize witnesses to explain the majority of (novice) type errors, but a good error report should also help *locate* the source of the error. Thus, our final experiment seeks to use NanoMaLy's witnesses as localizations.

As discussed in § 3.4.1, we recorded each interaction of our students with the OCaml top-level system. This means that, in addition to collecting ill-typed programs, we collected subsequent, fixed versions of the same programs. For each ill-typed program compiled by a student, we identify the student's *fix* by searching for the first type-correct program that the student subsequently compiled. We then use an expression-level *diff* [59] to determine which sub-expressions changed between the ill-typed program and the student's fix, and treat those expressions as the source of the type error.

Not all ill-typed programs will have an associated fix; furthermore, at some point a "fix" becomes a "rewrite". We do not wish to consider the "rewrites", so we discard outliers where the fraction of expressions that have changed is more than one standard deviation above the mean, establishing a diff threshold of 40%. This accounts for roughly 14% of programs pairs we discovered, leaving us with 2,710 program pairs.

For each pair of an ill-typed program and its fix, we run NanoMaLy and collect two sets of source locations: (1) the source location corresponding to the stuck term; and (2) the source locations that *produced* the values inside the stuck term. Intuitively, these two classes of locations correspond to *sinks* and *sources* for typing constraints. For example, in the `sqsum` program from § 3.4.5 the stuck term is `0 @ 1`. This corresponds to the call to `@` on line 3, and contains the literal `0` from line 2 and the value `1` produced by the `*` on line 3.

We compare NanoMaLy's witness-based predictions against a baseline of the OCaml compiler as well as the state-of-the-art localization tools SHErrLoc and Mycroft. SHErrLoc [118] attempts to predict the most likely source of a type error by searching the typing constraint graph for constraints that participate in many unsatisfiable paths and few satisfiable paths. Mycroft [65] reduces the localization problem to MaxSAT by searching for a minimal subset of constraints that can be removed, such that the resulting system is satisfiable. Both tools

produce a *set* of equally-likely expressions to blame for the error (in practice the set contains only a few expressions), similar to NanoMaLy's witness-based predictions.

We evaluate each tool based on whether *any* of its predictions identifies a changed expression. There were a number of programs where Mycroft or SHErrLoc encountered an unsupported language feature or timed out after one minute, or where NanoMaLy failed to produce a witness. We discard all such programs in our evaluation to level the playing field, around 15% for each tool, leaving us with a benchmark set of 1,759 programs.

Threats to Validity

Our benchmarks were drawn from students in an undergraduate course at UCSD and may not be representative of other student bodies. We mitigate this threat with a large empirical evaluation of 1,759 programs, drawn from a cohort of 46 students. A similar threat is that students are not industrial programmers, thus our results may not translate to large-scale software engineering. However, in our experience programmers are able to construct a mental model of type systems after sufficient exposure, at which point traditional error reports may suffice. We are thus particularly interested in aiding novice programmers as they learn to work with the type system.

Our definition of the next well-typed program as the intended ground truth answer is another threat to validity. Students might submit multiple well-typed "rewrites" between the initial ill-typed program and the final intended answer. Our approach to discarding outliers is intended to mitigate this threat. A similar threat is our removal of programs where any of the tools could not produce an answer. It may be, for example, that Mycroft and SHErrLoc are particularly effective on programs that do not admit dynamic witnesses. Finally, our use of student fixes as oracles for the source of type errors assumes that students are able to correctly identify the source. As the students are in the process of learning OCaml and the type system, this assumption may be faulty, *expert* users may disagree with the student fixes. We believe, however, that it is reasonable to use student fixes as oracles, as the student is the best judge of what she *intended* to do.

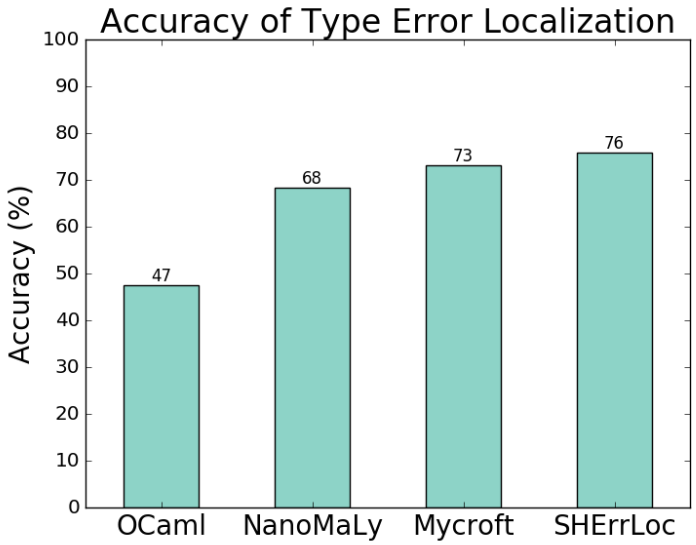


Figure 3.17. Accuracy of type error localization. NanoMaLy's witness-based predictions outperform OCaml by 21 points, and are competitive with the state-of-the-art tools Mycroft and SHErrLoc.

Results

Figure 3.17 summarizes our results, which show that NanoMaLy's witnesses are competitive with Mycroft and SHErrLoc in automatically locating the source of a type error. NanoMaLy, Mycroft, and SHErrLoc all outperform the OCaml compiler, which is not surprising given that they can produce multiple possible error locations, while the OCaml compiler is limited to one predicted error location. Interestingly, while all tools have a median of 2 predicted error locations per program, Mycroft and SHErrLoc have a long tail with a maximum of 22 (*resp.* 11) locations, while NanoMaLy's maximum is 5 locations. We also note that while Mycroft and SHErrLoc were designed specifically to *localize* type errors, NanoMaLy's foremost purpose is to *explain* them, we consider its ability to localize type errors an added benefit.

3.4.8 Discussion

To summarize, our experiments demonstrate that NanoMaLy finds witnesses to type errors: (1) with high coverage in a timespan amenable to compile-time analysis; (2) with traces that have a low median complexity of 5 jumps; (3) that are more helpful to novice programmers

than traditional type error messages; and (4) that can be used to automatically locate the source of a type error.

There are, of course, drawbacks to our approach. Four that stand out are: (1) coverage limits due to random generation; (2) dealing with explosions in the size of generated traces; (3) our use of a non-parametric function type; and (4) handling ad-hoc polymorphism.

Random Generation

Random test generation has difficulty generating highly constrained values, *e.g.* red-black trees or a pair of equal integers. If the type error is hidden behind a complex branch condition NanoMaLy may not be able to trigger it. Exhaustive testing and dynamic-symbolic execution can address this short-coming by performing an exhaustive search for inputs (*resp.* paths through the program). As our experiments show, however, novice programs do not appear to require more advanced search techniques, likely because they tend to be simple.

Trace Explosion

Though the average complexity of our generated traces is low in terms of jumps, there are some extreme outliers. We cannot reasonably expect a novice user to explore a trace containing 50+ terms and draw a conclusion about which pieces contributed to the bug in their program. Enhancing our visualization to slice out program paths relevant to specific values [81], would likely help alleviate this issue, allowing users to highlight a confusing value and ask: "Where did this come from?"

Non-Parametric Function Type

As we discussed in § 3.4.3 some ill-typed programs lack a witness in our semantics due to our use of a non-parametric type `fun` for functions. These programs cannot "go wrong", strictly speaking, but would be very difficult to *use* in practice. We also note that many of these programs induce cyclic typing constraints, causing infinite-type errors, which in our experience can be particularly difficult to debug (and to explain to novices). Better support for these programs would be welcome. For example, we might track how the types of inputs change

between recursive calls. If we cannot find a traditional witness, we could then produce a trace expanded to show these particular steps.

Ad-Hoc Polymorphism

Also discussed in § 3.4.3, our approach can only support ad-hoc polymorphism (*e.g.* type-classes in Haskell or polymorphic comparison functions in OCaml) in limited cases where we have enough typing information at the call-site to resolve the overloading. This issue is uncommon in OCaml (we detected it in around 5% of our benchmarks), but it would surely be exacerbated by a language like Haskell, which makes heavy use of overloading. We suspect that either dynamic-symbolic execution or speculative instantiation of holes would allow us to handle ad-hoc polymorphism, but defer a proper treatment to future work.

3.5 Related Work

In this section we connect our work to related efforts in testing and program exploration.

Running Ill-Typed Programs

Vytiniotis et al. [110] extend the Haskell compiler GHC to support compiling ill-typed programs, but their intent is rather different from ours. Their goal was to allow programmers to incrementally test refactorings, which often cause type errors in distant functions. They replace any expression that fails to type check with a *runtime* error, but do not check types at runtime. Bayne et al. [5] also provide a semantics for running ill-typed (Java) programs, but in constrast transform the program to perform nearly all type checking at run-time. The key difference between Bayne *et al.* and our work is that we use the dynamic semantics to automatically search for a witness to the type error, while their focus is on incremental, programmer-driven testing.

Testing

NanoMaLy is at its heart a test generator, and as such, builds on a rich line of work. Our use of holes to represent unknown values is inspired by the work of Runciman, Naylor, and Lindblad [64, 73, 92], who use lazy evaluation to drastically reduce the search space for

exhaustive test generation, by grouping together equivalent inputs by the set of values they force. An exhaustive search is complete (up to the depth bound), if a witness exists it will be found, but due to the exponential blowup in the search space the depth bound can be quite limited without advanced grouping and filtering techniques. Our search is not exhaustive; instead we use random generation to fill in holes on demand. Random test generation [21, 24, 78] is by its nature incomplete, but is able to check larger inputs than exhaustive testing as a result.

Instead of enumerating values, which may trigger the same path through the program, one might enumerate paths. Dynamic-symbolic execution [14, 35, 106] combines symbolic execution (to track which path a given input triggers) with concrete execution (to ensure failures are not spurious). The system collects a path condition during execution, which tracks symbolically what conditions must be met to trigger the current path. Upon successfully completing a test run, it negates the path condition and queries a solver for another set of inputs that satisfy the negated path condition, *i.e.* inputs that will not trigger the same path. Thus, it can prune the search space much faster than techniques based on enumerating values, but is limited by the expressiveness of the underlying solver.

Our operational semantics is amenable to dynamic-symbolic execution, one would just need to collect the path condition and replace our implementation of gen by a call to the solver. We chose to use lazy, random generation instead because it is efficient, does not incur the overhead of an external solver, and produces high coverage for our domain of novice programs.

A function's type is a theorem about the its behavior. Thus, NanoMaLy's witnesses can be viewed as *counter-examples*, thereby connecting it to work on using test cases to find counter-examples prior to starting a proof [15, 101].

Program Exploration

Flanagan et al. [31] describe a static debugger for Scheme, which helps the programmer interactively visualize problematic source-sink flows corresponding to soft-typing errors. The debugger allows the user to explore an abstract reduction graph computed from a static value

set analysis of the program. In contrast, NanoMaLy generates witnesses and allows the user to explore the resulting dynamic execution. Perera et al. [81] present a tracing semantics for functional programs that tags values with their provenance, enabling a form of backwards program slicing from a final value to the sequence of reductions that produced it. Notably, they allow the user to supply a *partial value* — containing holes — and present a partial slice, containing only those steps that affected the the partial value. This system is designed to answer questions of the form "Where did this value come from?" and thus is focused on backward exploration. In contrast, our visualization supports forward *and* backward exploration, though our backward steps are more limited. Specifically, we do not support selecting a value and inserting the intermediate terms that preceded it while ignoring unrelated computation steps.

Endnotes

Acknowledgments

This chapter, in part, is a reprint of the material as it appears, or may appear, in: E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In ICFP '16, 2016; and E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *In submission to J. Funct. Programming*, 2017. The dissertation author was the primary investigator and author of these papers.

Chapter 4

Learning To Blame

In the previous chapter we presented a technique designed to help explain type errors, by searching for a witness to the error. We found that, in addition to explaining the error, the witnesses can be used to localize the error, though they are not quite as effective as the state of the art in type error localization.

In this chapter we tackle the problem of error localization, building on top of recent work that *ranks* potentially erroneous terms by the likelihood that they are the source of the error. At a high-level, these techniques analyze the set of typing constraints to find the minimum (weighted) subset that, if removed, would make the constraints satisfiable and hence, assertion-safe [47] or well-typed [17, 65, 79, 118]. The finger of blame is then pointed at the sub-terms that yielded those constraints. This minimum-weight approach suffers from two drawbacks. First, they are not *extensible*: the constraint languages and algorithms for computing the minimum weighted subset must be designed afresh for different kinds of type systems and constraints [65]. Second, and perhaps most importantly, they are not *adaptable*: the weights are fixed in an ad-hoc fashion, based on the *analysis designer's* notion of what kinds of errors are more likely, rather than adapting to the kinds of mistakes programmers actually make in practice.

We introduce NATE[1], a *data-driven* approach to error localization based on supervised learning (see [52] for a survey). NATE analyzes a large corpus of training data — pairs of ill-typed programs and their subsequent fixes — to automatically *learn a model* of where errors are most likely to be found. Given a new ill-typed program, NATE simply executes the model to generate

[1] "Numeric Analysis of Type Errors"; any resemblance to persons living or dead is purely coincidental.

a list of potential blame assignments ranked by likelihood. We evaluate NATE by comparing its precision against the state-of-the-art on a set of over 5,000 ill-typed OCAML programs drawn from the dataset we collected in Chapter 2. We show that, when restricted to a *single* prediction, NATE's data-driven model is able to correctly predict the exact sub-expression that should be changed 72% of the time, 28 points higher than OCAML and 16 points higher than the state-of-the-art SHERRLOC tool. Furthermore, NATE's accuracy surpasses 85% when we consider the top *two* locations and reaches 91% if we consider the top *three*. We achieve these advances by identifying and then solving three key challenges.

Challenge 1: Acquiring Labeled Programs

The first challenge for supervised learning is to acquire a corpus of training data, in our setting a set of ill-typed programs *labeled* with the exact sub-terms that are the actual cause of the type error. Prior work has often enlisted expert users to manually judge ill-typed programs and determine the "correct" fix [*e.g.* 60, 65], but this approach does not scale well to a dataset large enough to support machine learning. Worse, while expert users have intimate knowledge of the type system, they may have a blind spot with regards to the kinds of mistakes novices make, and cannot know in general what novice users intended.

Our *first contribution* (§ 4.1) is a set of more than 5,000 labeled programs [97], giving us an accurate ground truth of the kinds of errors and the (locations of the) fixes that novices make in practice. We obtain this set by observing that software development is an iterative process; programmers eventually fix their own ill-typed programs, perhaps after multiple incorrect exploratory attempts. To exploit this observation we instrumented the OCAML compiler to collect fine-grained traces of student interactions over two instances of an undergraduate Programming Languages course at UC San Diego (IRB #140608), as described in Chapter 2. We then post-process the resulting time-series of programs submitted to the OCAML compiler into a set of pairs of ill-typed programs and their subsequent *fixes*, the first (type-) correct program in the trace suffix. Finally, we compute the blame labels using a *tree-diff* between the two terms to find the exact sub-terms that changed in the fix.

Challenge 2: Modeling Programs as Vectors

Modern supervised learning algorithms work on *feature vectors*: real-valued points in an n-dimensional space. While there are standard techniques for computing such vectors for documents, images, and sound (respectively word-counts, pixel-values, and frequencies), there are no similarly standard representations for programs.

Our *second contribution* (§ 4.2) solves this problem with a simple, yet expressive, representation called a *Bag-of-Abstracted-Terms* (BOAT) wherein each program is represented by the *bag* or multiset of (sub-) terms that appears inside it; and further, each (sub-) term is *abstracted* as a feature vector comprising the numeric values returned by *feature abstraction* functions applied to the term. We can even recover *contextual* information from the parent and child terms by *concatenating* the feature vectors of each term with those of its parent and children (within a fixed window). We have found this representation to be particularly convenient as it gives us flexibility in modeling the syntactic and semantic structure of programs while retaining compatibility with off-the-shelf classifiers, in contrast to, *e.g.*, Raychev et al. [88], who had to develop their own variants of classifiers to obtain their results.

Challenge 3: Training Precise Classifiers

Finally, the last and most important challenge is to use our BOAT representation to train classifiers that are capable of *precisely* pinpointing the errors in real programs. The key here is to find the right set of feature abstractions to model type errors, and classification algorithms that lead to precise blame assignments. Fortunately, our BOAT model allows us a great deal of latitude in our choice of features. We can use abstraction functions to capture different aspects of a term ranging from syntactic features (*e.g.* is-a-data-constructor, is-a-literal, is-an-arithmetic-operation, is-a-function-application, *etc.*), to semantic features captured by the type system (*e.g.* is-a-list, is-an-integer, is-a-function, *etc.*). We can similarly model the blame labels with a simple feature abstraction (*e.g.* is-changed-in-fix).

Our *third contribution* (§ 4.3) is a systematic evaluation of our data-driven approach using different classes of features like the above, and with four different classification algorithms:

logistic regression, decision trees, random forests, and neural networks. We find that NATE's models *generalize* well between instances of the same undergraduate course, outperforming the state of the art by at least 16 percentage points at predicting the source of a type error. We also investigate which features and classifiers are most effective at localizing type errors, and empirically characterize the importance of different feature sets. In particular, we find that while machine learning over syntactic features of each term in isolation performs worse than existing purely constraint-based approaches (*e.g.* OCAML, SHERRLOC), augmenting the data with a single feature corresponding to the *type error slice* [107] brings our classifiers up to par with the state-of-the-art, and further augmenting the data with *contextual* features allows our classifiers to outperform the state-of-the-art by 16 percentage points.

Thus, by combining modern statistical methods with domain-specific feature engineering, NATE opens the door to a new data-driven path to precise error localization. In the future, we could *extend* NATE to new languages or forms of correctness checks by swapping in a different set of feature abstraction functions. Furthermore, our data-driven approach allows NATE to *adapt* to the kinds of errors that programmers (in particular novices, who are in greatest need of precise feedback) actually make rather than hardwiring the biases of compiler authors who, by dint of their training and experience, may suffer from blind spots with regards to such problems. In contrast, our results show that NATE's data-driven diagnosis can be an effective technique for localizing errors by collectively learning from past mistakes.

4.1 Overview

Let us start with an overview of NATE's approach to localizing type errors by collectively learning from the mistakes programmers actually make.

The Problem

Consider our familiar `sumList` program, reproduced in Figure 4.1. The program is meant to add up the integers in a list, but the student has accidentally given the empty list as the base case, rather than `0`. The OCAML compiler collects typing constraints as it traverses the

```
1 | let rec sumList xs =
2 |   match xs with
3 |   | []    -> []
4 |   | h::t -> h + sumList t
```

```
This expression has type 'a list
but an expression was expected of type int
```

Figure 4.1. (left) An ill-typed OCaml program that should sum the elements of a list, with highlights indicating three possible blame assignments based on: (1) the OCaml compiler; (2) the fix made by the programmer; and (3) minimizing the number of edits required. (right) The error reported by OCaml.

program, and reports an error the moment it finds an inconsistent constraint. In this case it blames the recursive call to `sumList`, complaining that `sumList` returns a `list` while an `int` was expected by the + operator. This *blame* assignment is inconsistent with the programmer's intention and may not help the novice understand the error.

It may appear obvious to the reader that `[]` is the correct expression to blame, but how is a type checker to know that? Indeed, recent techniques like SHErrLoc and Mycroft [65, 79, 118] fail to distinguish between the `[]` and + expressions in Figure 4.1; it would be equally valid to blame *either* of them alone. The `[]` on line 3 could be changed to `0`, or the + on line 4 could be changed to either `@` (list append) or `::`, all of which would give type-correct programs. Thus, these state-of-the-art techniques are forced to either blame *both* locations, or choose one *arbitrarily*.

Solution: Localization via Supervised Classification

Our approach is to view error localization as a *supervised classification* problem [52]. A *classification* problem entails learning a function that maps *inputs* to a discrete set of output *labels* (in contrast to *regression*, where the output is typically a real number). A *supervised* learning problem is one where we are given a *training set* where the inputs and labels are known, and the task is to learn a function that accurately maps the inputs to output labels and *generalizes* to new, yet-unseen inputs. To realize the above approach for error localization as a practical tool, we have to solve four sub-problems.

1. How can we acquire a *training set* of blame-labeled ill-typed programs?

2. How can we *represent* blame-labeled programs in a format amenable to machine learning?

3. How can we find *features* that yield predictive models?

4. How can we use the models to give localized *feedback* to the programmer?

4.1.1 Step 1: Acquiring a Blame-Labeled Training Set

The first step is to gather a training set of ill-typed programs, where each erroneous subterm is explicitly labeled. Prior work has often enlisted expert users to curate a set of ill-typed programs and then *manually* determine the correct fix [*e.g.* 60, 65]. This method is suitable for *evaluating* the quality of a localization (or repair) algorithm on a small number (*e.g.* 10s–100s) of programs. However, in general it requires a great deal of effort for the expert to divine the original programmer's intentions. Consequently, is difficult to scale the expert-labeling to yield a dataset large enough (*e.g.* 1000s of programs) to facilitate machine learning. More importantly, this approach fails to capture the *frequency* with which errors occur in practice.

Solution: Interaction Traces

We solve both the scale and frequency problems by instead extracting blame-labeled data sets from *interaction traces*. Software development is an iterative process. Programmers, perhaps after a lengthy (and sometimes frustrating) back-and-forth with the type checker, eventually end up fixing their own programs. Thus, we can use the interaction traces described in Chapter 2 to extract a training set of ill-typed programs and fixes. For each ill-typed program in a particular programmer's trace, we find the *first subsequent* program in the trace that type checks and declare it to be the fixed version. From this pair of an ill-typed program and its fix, we can extract a *diff* of the abstract syntax trees, and then assign the blame labels to the *smallest* sub-tree in the diff.

Example

Suppose our student fixed the `sumList` program in Figure 4.1 by replacing `[]` with `0`, the diff would include only the `[]` expression. Thus we would determine that the `[]` expression

(and *not* the + or the recursive call `sumList t`) is to blame.

4.1.2 Step 2: Representing Programs as Vectors

Next, we must find a way to translate highly structured and variable sized *programs* into fixed size n-dimensional numeric *vectors* that are needed for supervised classification. While the Programming Languages literature is full of different program representations, from raw token streams to richly-structured abstract syntax trees (AST) or control-flow graphs, it is unclear how to embed the above into a vector space. Furthermore, it is unclear whether recent program representations that are amenable to one learning task, *e.g.* code completion [43, 89] or decompilation [10, 88] are suitable for our problem of assigning blame for type errors.

Solution: Bags-of-Abstracted-Terms

We present a new representation of programs that draws inspiration from the theory of abstract interpretation [23]. Our representation is parameterized by a set of *feature abstraction* functions, (abbreviated to feature abstractions) $f_1, \ldots, f_n$, that map terms to a numeric value (or just $\{0, 1\}$ to encode a boolean property). Given a set of feature abstractions, we can represent a single program's AST as a *bag-of-abstracted-terms* (BOAT) by: (1) decomposing the AST (term) t into a *bag* of its constituent sub-trees (terms) $\{t_1, \ldots, t_m\}$; and then (2) representing each sub-term t_i with the n-dimensional vector $[f_1(t_i), \ldots, f_n(t_i)]$. Working with ASTs is a natural choice as type-checkers operate on the same representation.

Modeling Contexts

Each expression occurs in some surrounding *context*, and we would like the classifier to be able make decisions based on the context as well. The context is particularly important for our task as each expression imposes typing constraints on its neighbors. For example, a + operator tells the type checker that both *children* must have type `int` and that the *parent* must accept an `int`. Similarly, if the student wrote `h sumList t` *i.e.* forgot the +, we might wish to blame the application rather than `h` because `h` *does not* have a function type. The BOAT representation makes it easy to incorporate contexts: we simply *concatenate* each term's feature

vector with the *contextual features* of its parent and children.

4.1.3 Step 3: Feature Discovery

Next, we must find a *good* set of features, that is, a set of features that yields predictive models. Our BOAT representation enables an iterative solution by starting with a simple set of features, and then repeatedly adding more and more to capture important aspects needed to improve precision. Our set of feature abstractions captures the *syntax*, *types*, and *context* of each expression.

Syntax and Type Features

We start by observing that at the very least, the classifier should be able to distinguish between the `[]` and `+` expressions in Figure 4.1 because they represent different *syntactic* expression forms. We model this by introducing feature abstractions of the form is-`[]`, is-`+`, *etc.*, for each of a fixed number of expression forms. Modeling the syntactic class of an expression gives the classifier a basic notion of the relative frequency of blame assignment for the various program elements, *i.e.* perhaps `[]` is *empirically* more likely to be blamed than `+`. Similarly, we can model the *type* of each sub-expression with features of the form is-`int`, is-`bool`, *etc.*. We will discuss handling arbitrary, user-defined types in § 4.4.

Contextual Features: Error Slices

Our contextual features include the syntactic class of the neighboring expressions and their inferred types (when available). However, we have found that the most important contextual signal is whether or not the expression occurs in a minimal type error slice [37, 107] which includes *a minimal subset* of all expressions that are necessary for the error to manifest. (That is, replacing any subterm with `undefined` or `assert false` would yield a well-typed program.) We propose to use type error slices to communicate to the classifier which expressions could *potentially* be blamed — a change to an expression outside of the minimal slice cannot possibly fix the type error. We empirically demonstrate that the type error slice is so important (§ 4.3.3) that it is actually beneficial to automatically discard expressions that are not part of

the slice, rather than letting the classifier learn to do so. Indeed, this domain-specific insight is crucial for learning classifiers that significantly outperform the state-of-the-art.

Example

When Nate is tasked with localizing the error in the example program of Figure 4.1, the `[]` and `+` sub-terms will each be given their own feature vector, and we will ask the classifier to predict for each *independently* whether it should be blamed. Table 4.1 lists some of the sub-expressions of the example from Figure 4.1, and their corresponding feature vectors.

Table 4.1. Example Feature Vectors

Expression	Is-[]	Match-[]-P	Size	Type-Int-C1	Type-[]	In-Slice
`[]`	1	1	1	0	1	1
`hd + sumList tl`	0	1	5	1	0	1
`sumList tl`	0	0	3	0	1	1
`tl`	0	0	1	0	1	0

A selection of the features we would extract from the `sumList` program in Figure 4.1. A feature is considered *enabled* if it has a non-zero value, and *disabled* otherwise. A "-P" suffix indicates that the feature describes the parent of the current expression, a "-Cn" suffix indicates that the feature describes the n-th (left-to-right) child of the current expression. Note that, since we rely on a partial typing derivation, we are subject to the well-known traversal bias and label the expression `sumList tl` as having type $[\cdot]$. The model will have to learn to correct for this bias.

4.1.4 Step 4: Generating Feedback

Finally, having trained a classifier using the labeled data set, we need to use it to help users localize type errors in their programs. The classifier tells us whether or not a sub-term *should be* blamed (*i.e.* has the blame label) but this is not yet particularly suitable as *user feedback*. A recent survey of developers by Kochhar et al. [51] found that developers are unlikely to examine more than around five potentially erroneous locations before falling back to manual debugging. Thus, we should limit our predictions to a select few to be presented to the user.

Solution: Rank Locations by Confidence

Fortunately, many machine learning classifiers produce not only a predicted label, but also a metric that can be interpreted as the classifier's *confidence* in its prediction. Thus, we *rank* each expression by the classifier's confidence that it should be blamed, and present only the top-k predictions to the user (in practice $k = 3$). The use of ranking to report the results of an analysis is popular in other problem domains [see, *e.g.* 53]; we focus explicitly on the use of data-driven machine learning confidence as a ranking source. In § 4.3 we show that NATE's ranking approach yields a high-precision localizer: when the top three locations are considered, at least one matches an actual student fix 91% of the time.

4.2 Learning to Blame

In this section, we describe our approach to localizing type errors, in the context of λ^{ML}(Figure 4.2), a simple lambda calculus with integers, booleans, pairs, and lists. Our goal is to instantiate the blame function of Figure 4.3, which takes as input a Model of type errors and an ill-typed program e, and returns an ordered list of subexpressions from e paired with the confidence score C that they should be blamed.

A Model is produced by train, which performs supervised learning on a training set of feature vectors $\mathcal{V}$ and (boolean) labels $\mathcal{B}$. Once trained, we can evaluate a Model on a new input, producing the confidence C that the blame label should be applied. We describe multiple Models and their instantiations of train and eval (§ 4.2.3).

Of course, the Model expects feature vectors $\mathcal{V}$ and blame labels $\mathcal{B}$, but we are given program pairs. So our first step must be to define a suitable translation from program pairs to feature vectors and labels, *i.e.* we must define the extract function in Figure 4.3. We model features as real-valued functions of terms, and extract a feature vector for each *subterm* of the ill-typed program (§ 4.2.1). Then we define the blame labels for the training set to be the subexpressions that changed between the ill-typed program and its subsequent fix, and model blame as a function from a program pair to the set of expressions that changed (§ 4.2.2). The extract function, then, extracts features from each subexpression and computes the blamed

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& \mid & n \mid e + e \\
& \mid & b \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \langle e, e \rangle \mid \texttt{match}\ e \left\{ \langle x, x \rangle \rightarrow e \right. \\
& \mid & [] \mid e :: e \mid \texttt{match}\ e \begin{cases} [] \rightarrow e \\ x :: x \rightarrow e \end{cases} \\
n & ::= & 0, 1, -1, \ldots \\
b & ::= & \texttt{true} \mid \texttt{false} \\
t & ::= & \alpha \mid \texttt{bool} \mid \texttt{int} \mid t \rightarrow t \mid t \times t \mid [t]
\end{array}
$$

Figure 4.2. Syntax of λ^{ML}

$$
\begin{array}{lcl}
\mathcal{V} & \doteq & [\mathcal{R}] \\
C & \doteq & \{r \in \mathcal{R} \mid 0 \leq r \leq 1\} \\
\text{features} & : & [e \rightarrow \mathcal{R}] \\
\text{label} & : & e \times e \rightarrow [e] \\
\text{extract} & : & [e \rightarrow \mathcal{R}] \rightarrow e \times e \rightarrow [\mathcal{V} \times \mathcal{B}] \\
\text{train} & : & [\mathcal{V} \times \mathcal{B}] \rightarrow \text{Model} \\
\text{eval} & : & \text{Model} \rightarrow \mathcal{V} \rightarrow C \\
\hline
\text{blame} & : & \text{Model} \rightarrow e \rightarrow [e \times C]
\end{array}
$$

Figure 4.3. A high-level API for converting program pairs to feature vectors and labels.

expressions according to label.

4.2.1 Features

The first issue we must tackle is formulating our learning task in machine learning terms. We are given programs over λ^{ML}, but learning algorithms expect to work with *feature vectors* $\mathcal{V}$ — vectors of real numbers, where each column describes a particular aspect of the input. Thus, our first task is to convert programs to feature vectors.

We choose to model a program as a *set* of feature vectors, where each element corresponds an expression in the program. Thus, given the `sumList` program in Figure 4.1 we would first split it into its constituent sub-expressions and then transform each sub-expression into a single feature vector. We group the features into five categories, using Table 4.1 as a running

example of the feature extraction process.

Local syntactic features

These features describe the syntactic category of each expression e. In other words, for each production of e in Figure 4.2 we introduce a feature that is enabled (set to 1) if the expression was built with that production, and disabled (set to 0) otherwise. For example, the Is-[] feature in Table 4.1 describes whether an expression is the empty list [].

We distinguish between matching on a list vs. on a pair, as this affects the typing derivation. We also assume that all pattern matches are well-formed — *i.e.* all patterns must match on the same type. Ill-formed match expressions would lead to a type error; however, they are already effectively localized to the match expression itself. We note that this is not a *fundamental* limitation, and one could easily add features that specify whether a match *contains* a particular pattern, and thus have a match expression that enables multiple features.

Contextual syntactic features

These are similar to local syntactic features, but lifted to describe the parent and children of an expression. For example, the Match-[]-P feature in Table 4.1 describes whether an expression's *parent* matches on a list. If a particular e does not have children (*e.g.* a variable x) or a parent (*i.e.* the root expression), we leave the corresponding features disabled. This gives us a notion of the *context* in which an expression occurs, similar to the *n-grams* used in linguistic models [33, 43].

Expression size

We also propose a feature representing the *size* of each expression, *i.e.* how many sub-expressions does it contain? For example, the Size feature in Table 4.1 is set to three for the expression `sumList tl` as it contains three expressions: the two variables and the application itself. This allows the model to learn that, *e.g.*, expressions closer to the leaves are more likely to be blamed than expressions closer to the root.

Typing features

A natural way of summarizing the context in which an expression occurs is with *types*. Of course, the programs we are given are *untypeable*, but we can still extract a *partial* typing derivation from the type checker and use it to provide more information to the model.

A difficulty that arises here is that, due to the parametric type constructors $\cdot \to \cdot$, $\cdot \times \cdot$, and $[\cdot]$, there is an *infinite* set of possible types — but we must have a *finite* set of features. Thus, we abstract the type of an expression to the set of type constructors it *mentions*, and add features for each type constructor that describe whether a given type mentions the type constructor. For example, the type `int` would only enable the `int` feature, while the type `int` $\to$ `bool` would enable the $\cdot \to \cdot$, `int`, and `bool` features.

We add these features for parent and child expressions to summarize the context, but also for the current expression, as the type of an expression is not always clear *syntactically*. For example, the expressions `tl` and `sumList tl` in Table 4.1 both enable Type-[], as they are both inferred to have a type that mentions $[\cdot]$.

Note that our use of typing features in an ill-typed program subjects us to *traversal bias* [69]. For example, the `sumList tl` expression might alternatively be assigned the type `int`. Our models will have to learn good localizations in spite of this bias (see § 4.3).

Type error slice

Finally, we wish to distinguish between changes that could fix the error, and changes that *cannot possibly* fix the error. Thus, we compute a minimal type error *slice* for the program (*i.e.* the set of expressions that contribute to the error), and add a feature that is enabled for expressions that are part of the slice. The In-Slice feature in Table 4.1 indicates whether an expression is part of such a minimal slice, and is enabled for all of the sampled expressions except for `tl`, which does not affect the type error. If the program contains multiple type errors, we compute a minimal slice for each error.

In practice, we have found that In-Slice is a particularly important feature, and thus include a post-processing step that discards all expressions where it is disabled. As a result,

the `tl` expression would never actually be shown to the classifier. We will demonstrate the importance of In-Slice empirically in § 4.3.3.

4.2.2 Labels

Recall that we make predictions in two stages. First, we use eval to predict for each subexpression whether it should be blamed, and extract a confidence score C from the Model. Thus, we define the output of the Model to be a boolean label, where "false" means the expression *should not* change and "true" means the expression *should* change. This allows us to predict whether any individual expression should change, but we would actually like to predict the *most likely* expressions to change. Second, we *rank* each subexpression by the confidence C that it should be blamed, and return to the user the top k most likely blame assignments (in practice $k = 3$).

We identify the fixes for each ill-typed program with an expression-level diff [59]. We consider two sources of changes. First, if an expression has been removed wholesale, *e.g.* if $f\ x$ is rewritten to $g\ x$, we will mark the expression f as changed, as it has been replaced by g. Second, if a new expression has been inserted *around* an existing expression, *e.g.* if $f\ x$ is rewritten to $f\ x + 1$, we will mark the application expression $f\ x$ (but not f or x) as changed, as the + operator now occupies the original location of the application.

4.2.3 Learning Algorithms

Recall that we formulate type error detection at a single expression as a supervised classification problem. This means that we are given a training data set $S : [\mathcal{V} \times \mathcal{B}]$ of labeled examples, and our goal is to use it to build a *classifier*, *i.e.* a rule that can predict a label b for an input v. Since we apply the classifier on each expression in the program to determine those that are the most likely to be type errors, we also require the classifier to output a *confidence score* that measures how sure the classifier is about its prediction.

There are many learning algorithms to choose from, existing on a spectrum that balances expressiveness with ease of training (and of interpreting the learned model). In this section we consider four standard learning algorithms: (1) logistic regression, (2) decision trees, (3) random

forests, and (4) neural networks. A thorough introduction to these techniques can be found in introductory machine learning textbooks [*e.g.* 40].

Below we briefly introduce each technique by describing the rules it learns, and summarize its advantages and disadvantages. For our application, we are particularly interested in three properties – expressiveness, interpretability and ease of generalization. Expressiveness measures how complex prediction rules are allowed to be, and interpretability measures how easy it is to explain the cause of prediction to a human. Finally ease of generalization measures how easily the rule generalizes to examples that are not in the training set; a rule that is not easily generalizable might perform poorly on an unseen test set even when its training performance is high.

Logistic Regression

The simplest classifier we investigate is logistic regression: a linear model where the goal is to learn a set of weights W that describe the following model for predicting a label b from a feature vector v:

$$\Pr(b = 1|v) = \frac{1}{1 + e^{-W^\top v}}$$

The weights W are learnt from training data, and the value of $\Pr(b|v)$ naturally leads to a confidence score C. Logistic regression is a widely used classification algorithm, preferred for its simplicity, ease of generalization, and interpretability. Its main limitation is that the prediction rule is constrained to be a linear combination of the features, and hence relatively simple. While this can be somewhat mitigated by adding higher order (quadratic or cubic) features, this often requires substantial domain knowledge.

Decision Trees

Decision tree algorithms learn a tree of binary predicates over the features, recursively partitioning the input space until a final classification can be made. Each node in the tree contains a single predicate of the form $v_j \leq t$ for some feature v_j and threshold t, which determines whether a given input should proceed down the left or right subtree. Each leaf is

labeled with a prediction and the fraction of correctly-labeled training samples that would reach it; the latter quantity can be interpreted as the decision tree's confidence in its prediction. This leads to a prediction rule that can be quite expressive depending on the data used to build it.

Training a decision tree entails finding both a set of good partitioning predicates and a good ordering of the predicates based on data. This is usually done in a top-down greedy manner, and there are several standard training algorithms such as C4.5 [84] and CART [13].

Another advantage of decision trees is their ease of interpretation — the decision rule is a white-box model that can be readily described to a human, especially when the tree is small. However, the main limitation is that these trees often do not generalize well, though this can be somewhat mitigated by *pruning* the tree.

Random Forests

Random forests improve generalization by training an *ensemble* of distinct decision trees and using a majority vote to make a prediction. The agreement among the trees forms a natural confidence score. Since each classifier in the ensemble is a decision tree, this still allows for complex and expressive classifiers.

The training process involves taking N random subsets of the training data and training a separate decision tree on each subset — the training process for the decision trees is often modified slightly to reduce correlation between trees, by forcing each tree to pick features from a random subset of all features at each node.

The diversity of the underlying models tends to make random forests less susceptible to the overfitting, but it also makes the learned model more difficult to interpret.

Neural Networks

The last (and most complex) model we use is a type of neural network called a *multi-layer perceptron* (see Nielsen [76] for an introduction to neural networks). A multi-layer perceptron can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to

the output. The output of an internal node v is

$$h_v = g(\sum_{j \in N(v)} W_{jv} h_j)$$

where $N(v)$ is the set of nodes in the previous layer that are adjacent to v, W_{jv} is the weight of the (j, v) edge and h_j is the output of node j in the previous layer. Finally g is a non-linear function, called the activation function, which in recent work is commonly chosen to be the *rectified linear unit* (ReLU), defined as $g(x) = \max(0, x)$ [72]. The number of layers, the number of neurons per layer, and the connections between layers constitute the *architecture* of a neural network. In this work, we use relatively simple neural networks which have an input layer, a single hidden layer and an output layer.

A major advantage of neural networks is their ability to discover interesting combinations of features through non-linearity, which significantly reduces the need for manual feature engineering, and allows high expressivity. On the other hand, this makes the networks particularly difficult to interpret and also difficult to generalize unless vast amounts of training data are available.

4.3 Evaluation

We have implemented our technique for localizing type errors for a purely functional subset of OCaml with polymorphic types and functions. We seek to answer four questions in our evaluation:

- **Blame Accuracy** How often does Nate blame a *correct* location for the error? (§ 4.3.2)
- **Feature Utility** Which program *features are required* to localize errors? (§ 4.3.3)
- **Interpretability** Do the models match our intuition about type errors? (§ 4.3.5)
- **Blame Utility** Do Nate's blame assignments help users diagnose type errors? (§ 4.3.6)

In the sequel we present our experimental methodology § 4.3.1 and then drill into how we evaluated each of the questions above. However, for the impatient reader, we begin with a quick

summary of our main results:

1. Data Beats Algorithms

Our main result is that for type error localization, data is indeed unreasonably effective [39]. When trained on student errors from one instance of an undergraduate course and tested on another instance, NATE's most sophisticated *neural network*-based classifier's top-ranked prediction blames the correct sub-term 72% of the time — a good 16 points higher than the state-of-the-art SHERRLOC's 56%. However, even NATE's simple *logistic regression*-based classifier is correct 61% of the time, *i.e.* 5 points better than SHERRLOC. When the top three predictions are considered, NATE is correct 91% of the time.

2. Slicing Is Critical

However, data is effective *only* when irrelevant sub-terms have been sliced out of consideration. In fact, perhaps our most surprising result is that type error slicing and local syntax alone yields a classifier that is 10 points better than OCAML and on par with SHERRLOC. That is, once we focus our classifiers on slices, purely local syntactic features perform as well as the state-of-the-art.

3. Size Doesn't Matter, Types Do

We find that (after slices) typing features provide the biggest improvement in accuracy. Furthermore, we find contextual syntactic features to be mostly (but not entirely) redundant with typing features, which supports the hypothesis that the context's *type* nicely summarizes the properties of the surrounding expressions. Finally, we found that the *size* of the sub-expression was not very useful. This was unexpected, as we thought smaller expressions would be simpler, and hence, more likely causes.

4. Models Learn Typing Rules

Finally, by investigating a few of the predictions made by the *decision tree*-based models, we found that the models appear to capture some simple and intuitive rules for predicting

well-typedness. For example, if the left child of an application is a function, then the application is likely correct.

4.3.1 Methodology

We answer our questions on the two datasets gathered in Chapter 2, which we will briefly describe again. We recorded each interaction with the OCaml top-level system while students in our undergraduate Programming Languages course worked on 23 programs from the first three homework assignments, capturing ill-typed programs and, crucially, their subsequent fixes. The first dataset comes from the Spring 2014 class (SP14), with a cohort of 46 students. The second comes from the Fall 2015 class (FA15), with a cohort of 56 students. The extracted programs are relatively small, but they demonstrate a range of functional programming idioms, *e.g.* higher-order functions and (polymorphic) algebraic data types.

Feature Selection

We extract 282 features from each sub-expression in a program, including:

1. 45 local syntactic features. In addition to the syntax of λ^{ML}, we support the full range of arithmetic operators (integer and floating point), equality and comparison operators, character and string literals, and a user-defined arithmetic expressions. We discuss the challenge of supporting other types in § 4.4.

2. 180 contextual syntactic features. For each sub-expression we additionally extract the local syntactic features of its parent and first, second, and third (left-to-right) children. If an expression does not have a parent or children, these features will simply be disabled. If an expression has more than three children, the classifiers will receive no information about the additional children.

3. 55 typing features. In addition to the types of λ^{ML}, we support `int`s, `float`s, `char`s, `string`s, and the user-defined `expr` mentioned above. These features are extracted for each sub-expression and its context.

4. One feature denoting the size of each sub-expression.

5. One feature denoting whether each sub-expression is part of the minimal type error slice. We use this feature as a "hard" constraint, sub-expressions that are not part of the minimal slice will be preemptively discarded. We justify this decision in § 4.3.3.

Blame Oracle

Recall from § 4.2.2 that we automatically extract a blame oracle for each ill-typed program from the (AST) diff between it and the student's eventual fix. A disadvantage of using diffs in this manner is that students may have made many, potentially unrelated, changes between compilations; at some point the "fix" becomes a "rewrite". We do not wish to consider the "rewrites" in our evaluation, so we discard outliers where the fraction of expressions that have changed is more than one standard deviation above the mean, establishing a diff threshold of 40%. This accounts for roughly 14% of each dataset, leaving us with 2,712 program pairs for SP14 and 2,365 pairs for FA15.

Accuracy Metric

All of the tools we compare (with the exception of the standard OCaml compiler) can produce a list of potential error locations. However, in a study of fault localization techniques, Kochhar et al. [51] show that most developers will not consider more than around five potential error locations before falling back to manual debugging. Type errors are relatively simple in comparison to general fault localization, thus we limit our evaluation to the top three predictions of each tool. We evaluate each tool on whether a changed expression occurred in its top one, top two, or top three predictions.

Blame Utility

Finally, to test the explanatory power of our blame assigments, we ran a user study at the University of Virginia (UVA IRB #2014009900). We included three problems in an exam in the Spring 2017 session of UVA's undergraduate Programming Languages course (CS 4501). We

presented the 31 students in the course with ill-typed OCaml programs and asked them to (1) *explain* the type error, and (2) *fix* the type error. For each problem the student was given the ill-typed program and either SHErrLoc or Nate's blame assignment, with no error message.

4.3.2 Blame Accuracy

First, we compare the accuracy of our predictions to the state of the art in type error localization.

Baseline

We provide two baselines for the comparison: a random choice of location from the minimized type error slice, and the standard OCaml compiler.

State of the Art

Mycroft [65] localizes type errors by searching for a minimal subset of typing constraints that can be removed, such that the resulting system is satisfiable. When multiple such subsets exist it can enumerate them, though it has no notion of which subsets are *more likely* to be correct, and thus the order is arbitrary. SHErrLoc [118] localizes errors by searching the typing constraint graph for constraints that participate in many unsatisfiable paths and comparatively few satisfiable paths. It can also enumerate multiple predictions, in descending order of likelihood.

Comparing source locations from multiple tools with their own parsers is not trivial. Our experimental design gives the state of the art tools the "benefit of the doubt" in two ways. First, when evaluating Mycroft and SHErrLoc, we did not consider programs where they predicted locations that our oracle could not match with a program expression: around 6% of programs for Mycroft and 4% for SHErrLoc. Second, we similarly ignored programs where Mycroft or SHErrLoc timed out (after one minute) or where they encountered an unsupported language feature: another 5% for Mycroft and 12% for SHErrLoc.

Our Classifiers

We evaluate five classifiers, each trained on the full feature set. These include:

Logistic A logistic regression trained with a learning rate $\eta = 0.001$, an L_2 regularization rate $\lambda = 0.001$, and a mini-batch size of 200.

Tree A decision tree trained with the CART algorithm [13] and an impurity threshold of 10^{-7} (used to avoid overfitting via early stopping).

Forest A random forest [12] of 30 estimators, with an impurity threshold of 10^{-7}.

MLP-10 and MLP-500 Two multi-layer perceptron neural networks, both trained with $\eta = 0.001$, $\lambda = 0.001$, and a mini-batch size of 200. The first MLP contains a single hidden layer of 10 neurons, and the second contains a hidden layer of 500 neurons. This gives us a measure of the complexity of the MLP's model, *i.e.* if the model requires many compound features, one would expect MLP-500 to outperform MLP-10. The neurons use rectified linear units (ReLU) as their activation function, a common practice in modern neural networks.

All classifiers were trained for 20 epochs on one dataset — *i.e.* they were shown each program 20 times — before being evaluated on the other. The logistic regression and MLPs were trained with the Adam optimizer [50], a variant of stochastic gradient descent that has been found to converge faster.

Results

Figure 4.4 shows the results of our experiment. Localizing the type errors in our benchmarks amounted, on average, to selecting one of 3 correct locations out of a slice of 10. Our classifiers consistently outperform the competition, ranging from 61% Top-1 accuracy (86% Top-3) for the Logistic classifier to 72% Top-1 accuracy (91% Top-3) for the MLP-500. Our baseline of selecting at random achieves 30% Top-1 accuracy (58% Top-3), while OCaml achieves a Top-1 accuracy of 44%. Interestingly, one only needs two *random* guesses to outperform

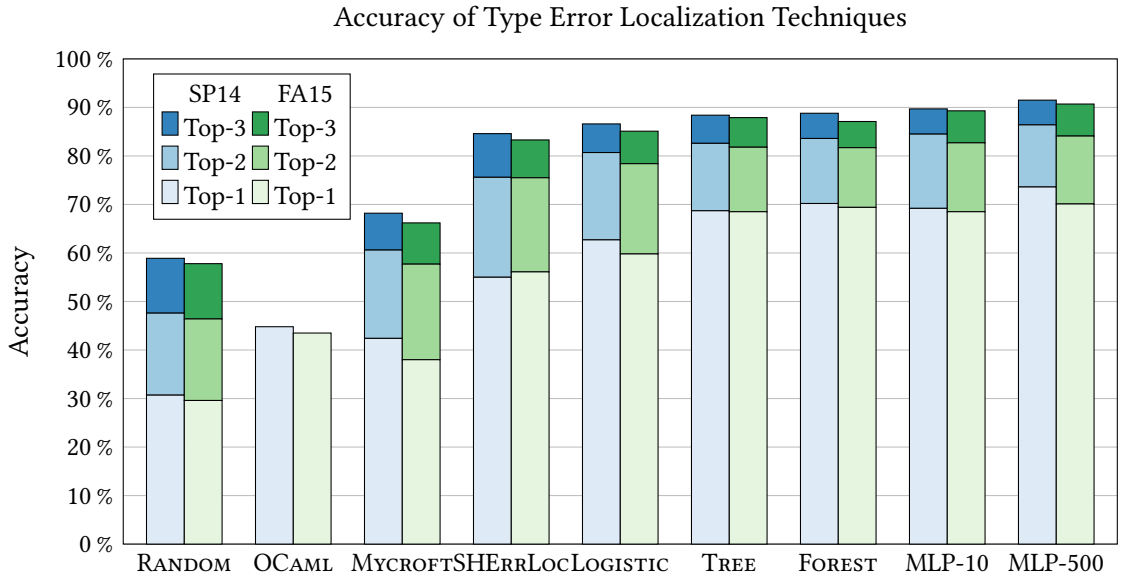


Figure 4.4. Results of our comparison of type error localization techniques. We evaluate all techniques separately on two cohorts of students from different instances of an undergraduate Programming Languages course. Our classifiers were trained on one cohort and evaluated on the other. All of our classifiers outperform the state-of-the-art techniques Mycroft and SHErrLoc.

OCaml, with 47% accuracy. SHErrLoc outperforms both baselines, and comes close to our Logistic classifier, with 56% Top-1 accuracy (84% Top 3), while Mycroft underperforms OCaml at 40% Top-1 accuracy.

Surprisingly, there is little variation in accuracy between our classifiers. With the exception of the Logistic model, they all achieve around 70% Top-1 accuracy and around 90% Top-3 accuracy. This suggests that the model they learn is relatively simple. In particular, notice that although the MLP-10 has 50× *fewer* hidden neurons than the MLP-500, it only loses around 4% accuracy. We also note that our classifiers consistently perform better when trained on the FA15 programs and tested on the SP14 programs than vice versa.

4.3.3 Feature Utility

We have shown that we can train a classifier to effectively localize type errors, but which of the feature classes from § 4.2.1 are contributing the most to our accuracy? We focus specifically on feature *classes* rather than individual features as our 282 features are conceptually

grouped into a much smaller number of *categorical* features. For example, the syntactic class of an expression is conceptually a feature but there are 45 possible values it could take; to encode this feature for learning we split it into 45 distinct binary features. Analyses that focus on individual features, *e.g.* ANOVA, are difficult to interpret in our setting, as they will tell us the importance of the binary features but not the higher-level categorical features. Thus, to answer our question we investigate the performance of classifiers trained on various subsets of the feature classes.

Type Error Slice

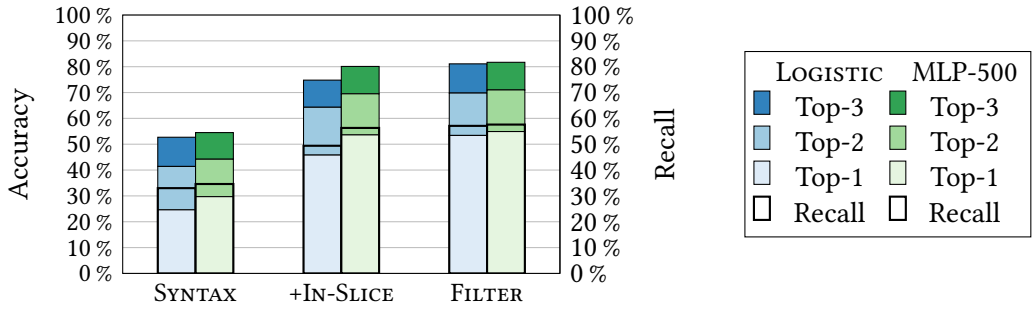
First we must justify our decision to automatically exclude expressions outside the minimal type error slice from consideration. Thus, we compare our classifiers on three sets of features:

1. A baseline with only local syntactic features and no preemptive filtering by In-Slice.
2. The features of (1) extended with In-Slice.
3. The same features as (1), but we preemptively discard samples where In-Slice is disabled.

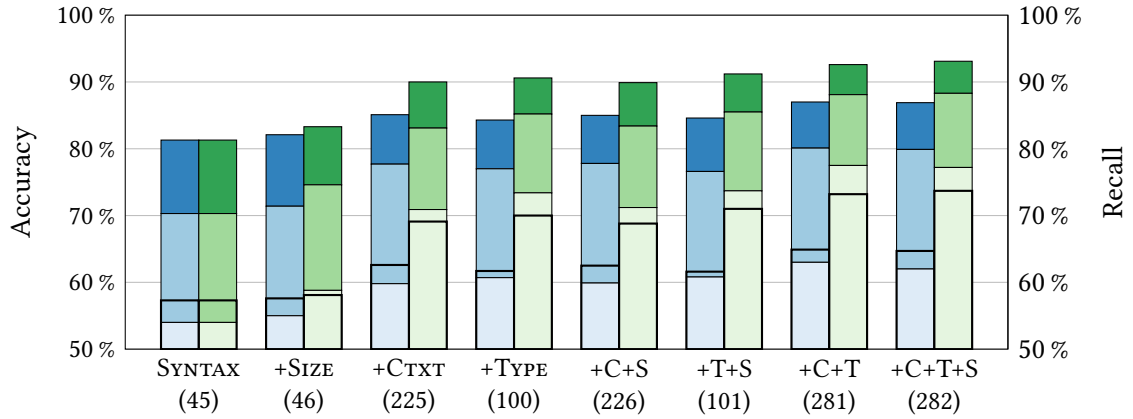
The key difference between (2) and (3) is that a classifier for (2) must *learn* that In-Slice is a strong predictor. In contrast, a classifier for (3) must only learn about the syntactic features, the decision to discard samples where In-Slice is disabled has already been made by a human. This has a few additional advantages: it reduces the set of candidate locations by a factor of 7 on average, and it guarantees that any prediction made by the classifier can fix the type error. We expect that (2) will perform better than (1) as it contains more information, and that (3) will perform better than (2) as the classifier does not have to learn the importance of In-Slice.

We tested our hypothesis with the Logistic and MLP-500[2] classifiers, cross-validated ($k = 10$) over the combined SP14/FA15 dataset. We trained for a single epoch on feature sets (1) and (2), and for 8 epochs on (3), so that the total number of training samples would be roughly

[2] A layer of 500 neurons is excessive when we have so few input features — we use MLP-500 for continuity with the surrounding sections.



(a) Impact of type error slice on blame accuracy.



(b) Impact of additional features on blame accuracy. Starting from a baseline of local syntactic features, we add each combination of expression size, contextual syntactic, and typing features. The total number of features is given in parentheses.

Figure 4.5. Results of our experiments on feature utility.

equal for each feature set. In addition to accuracy, we report each classifier's *recall* — *i.e.* "How many true changes can we remember?" — defined as

$$\frac{|\text{predicted} \cap \text{oracle}|}{|\text{oracle}|}$$

where predicted is limited to the top 3 predictions, and oracle is the student's fix, limited to changes that are in the type error slice. We make the latter distinction as: (1) changes that are not part of the type error slice are noise in the data set; and (2) it makes the comparison easier to interpret since oracle never changes.

Results

Figure 4.5a shows the results of our experiment. As expected, the baseline performs the worst, with a mere 25% Logistic Top-1 accuracy. Adding In-Slice improves the results substantially with a 45% Logistic Top-1 accuracy, demonstrating the importance of a minimal error slice. However, filtering out expressions that are not part of the slice *further* improves the results to 54% Logistic Top-1 accuracy. Interestingly, while the MLP-500 performs similarly poor with no error slice features, it recovers nearly all of its accuracy after being given the error slice features. Top-1 accuracy jumps from 29% to 53% when we add In-Slice, and only improves by 1% when we filter out expressions that are not part of the error slice. Still, the accuracy gain comes at zero cost, and given the other benefits of filtering by In-Slice— shrinking the search space and guaranteeing our predictions are actionable — we choose to filter all programs by In-Slice.

Contextual Features

We investigate the relative impact of the other three classes of features discussed in § 4.2.1, assuming we have discarded expressions not in the type error slice. For this experiment we consider again a baseline of only local syntactic features, extended by each combination of (1) expression size; (2) contextual syntactic features; and (3) typing features. As before, we perform a 10-fold cross-validation, but we train for a full 20 epochs to make the differences more apparent.

Results

Figure 4.5b summarizes the results of this experiment. The Logistic classifier and the MLP-500 start off competitive when given only local syntactic features, but the MLP-500 quickly outperforms as we add features.

Size is the weakest feature, improving Logistic Top-1 accuracy by less than 1% and MLP-500 by only 4%. In contrast, the contextual syntactic features improve Logistic Top-1 accuracy by 5% (*resp.* 16%), and the typing features improve Top-1 accuracy by 6% (*resp.* 18%).

Furthermore, while Size does provide some benefit when it is the only additional feature, it does not appear to provide any real increase in accuracy when added alongside the contextual or typing features. This is likely explained by *feature overlap*, *i.e.* the contextual features of "child" expressions additionally provide some information about the size of the subtree.

As one might expect, the typing features are more beneficial than the contextual syntactic features. They improve Top-1 accuracy by an additional 1% (*resp.* 3%), and are much more compact — requiring only 55 typing features compared to 180 contextual syntactic features. This aligns with our intuition that types should be a good summary of the context of an expression. However, typing features do not appear to *subsume* contextual syntactic features, the MLP-500 gains an additional 4% Top-1 accuracy when both are added.

4.3.4 Threats to Validity

Although our experiments demonstrate that our technique can pinpoint type errors more accurately than the state of the art and that our features are relevant to blame assignment, our results may not generalize to other problem domains or program sets.

One threat to validity associated with supervised machine learning is overfitting (*i.e.* learning a model that is too complex with respect to the data). A similar issue that arises in machine learning is model stability (*i.e.* can small changes to the training set produce large changes in the model?). We mitigate these threats by: (1) using separate training and testing datasets drawn from distinct student populations (§ 4.3.2), demonstrating the generality of our models; and (2) via cross-validation on the joint dataset (§ 4.3.3), which demonstrates the stability of our models by averaging the accuracy of 10 models trained on distinct subsets of the data.

Our benchmarks were drawn from students in an undergraduate course and may not be representative of other student populations. We mitigate this threat by including the largest empirical evaluation of type error localization that we are aware of: over 5,000 pairs of ill-typed programs and fixes from two instances of the course, with programs from 102 different students. We acknowledge, of course, that students are not industrial programmers and our results may

not translate to large-scale software development; however, we are particularly interested in aiding novice programmers as they learn to work inside the type system.

A related threat to construct validity is our definition of the immedate next well-typed program as the intended ground truth answer (see § 4.1, Challenge 2). Students may, in theory, submit intermediate well-typed program "rewrites" between the original ill-typed program and the final intended answer. Our approach to discarding outliers (see § 4.3) is designed to mitigate this threat.

Our removal of program pairs that changed too much, where our oracle could not identify the blame of the other tools, or where the other tools timed out or encountered unsupported language features is another threat to validity. It is possible that including the programs that changed excessively would hurt our models, or that the other tools would perform better on the programs with unsupported language features. We note however that (1) outlier removal is a standard technique in machine learning; and (2) our Top-1 accuracy margin is large enough that even if we assumed that SHErrLoc were perfect on all excluded programs, we would still lead by 9 points.

Examining programs written in OCaml as opposed to Haskell or any other typed functional language poses yet another threat, common type errors may differ in different languages. OCaml is, however, a standard target for research in type error localization and thus our choice admits a direct comparison with prior work. Furthermore, the functional core of OCaml that we support does not differ significantly from the functional core of Haskell or SML, all of which are effectively lambda calculi with a Hindley-Milner-style type system.

Finally, our use of student fixes as oracles assumes that students are able to correctly identify the source of an error. As the students are in the process of learning the language and type system, this assumption may be faulty. It may be that *expert* users would disagree with many of the student fixes, and that it is harder to learn a model of expert fixes, or that the state of the art would be better at predicting expert fixes. As we have noted before, we believe it is reasonable to use student fixes as oracles because the student is the best judge of what she *intended*.

4.3.5 Interpreting Specific Predictions

Next, we present a *qualitative* evaluation that compares the predictions made by our classifiers with those of SHErrLoc. In particular, we demonstrate, with a series of example programs from our student dataset, how our classifiers are able to use past student mistakes to make more accurate predictions of future fixes. We also take this opportunity to examine some of the specific features our classifiers use to assign blame. For each example, we provide (1) the code; (2) SHErrLoc's prediction; (3) our Tree's prediction; and (4) an *explanation* of why our classifier made its prediction, in terms of the features used and their values. We choose the Tree classifier for this section as its model is more easily interpreted than the MLP. We also exclude the Size feature from the model used in this section, as it makes the predictions harder to motivate, and as we saw in § 4.3.3 it does not appear to contribute significantly to the model's accuracy.

We explain the predictions by analyzing the paths induced in the decision tree by the features of the input expressions. Recall that each node in a decision tree contains a simple predicate of the features, *e.g.* "is feature v_j enabled?", which determines whether a sample will continue down the left or right subtree. Thus, we can examine the predicates used and the values of the corresponding features to explain *why* our Tree made its prediction. We will focus particularly on the enabled features, as they generally provide more information than the disabled features. Furthermore, each node is additionally labeled with the ratio of "blamed" vs "not-blamed" training expressions that passed through it. We can use this information to identify particularly important decisions, *i.e.* we consider a decision that changes the ratio to be more interesting than a decision that does not.

Failed Predictions

We begin with a few programs where our classifier fails to make the correct prediction. For these programs we will additionally highlight the correct blame location.

Constructing a List of Duplicates. Our first program is a simple recursive function `clone` that takes an item `x` and a count `n`, and produces a list containing `n` copies of `x`.

```
1  let rec clone x n =
2    let loop acc n =
3      if n <= 0 then
4        acc
5      else
6        clone ([x] @ acc) (n - 1) in
7    loop [] n
```

The student has defined a helper function `loop` with an accumulator `acc`, likely meant to call itself tail-recursively. Unfortunately, she has called `clone` rather than `loop` in the `else` branch, this induces a cyclic constraint `'a = 'a list` for the `x` argument to `clone`.

Our top prediction coincides with SHErrLoc (and OCaml), blaming the the first argument to `clone` rather than the occurrence of `clone` itself. We confess that this prediction is difficult to explain by examining the induced paths. In particular, it only references the expression's context, which is surprising. Much clearer is why we fail to blame the occurrence of `clone`, the two enabled features on the path are: (1) the parent is an application; and (2) `clone` has a function type. The model seems to have learned that programmers typically call the correct function.

Currying Considered Harmful? Next, another ill-fated attempt at `clone`.

```
1  let rec clone x n =
2    let rec loop x n acc =
3      if n < 0 then
4        acc
5      else
6        loop (x, (n - 1), (x :: acc)) in
7    loop (x, n, [])
```

The issue here is that OCaml functions are *curried* by default — *i.e.* they take their arguments one at a time — but our student has called the inner `loop` with all three arguments in a tuple. Many experienced functional programmers would choose to keep `loop` curried and rewrite the calls, however our student decides instead to *uncurry* `loop`, making it take a tuple of arguments. SHErrLoc blames the recursive call to `loop` while our classifier blames the tuple of arguments — a reasonable suggestion, but not the answer the student expected.

We fail to blame the definition of `loop` because it is defining a function. First, note that we represent `let f x y = e` as `let f = fun x -> fun y -> e`, thus a change to the pattern `x` would be treated as a change to the outer `fun` expression. With this in mind, we can explain our failure to blame the definition of `loop` (the outer `fun`) as follows: (1) it has a function type; (2) its child is a `fun`; and (3) its parent is a `let`. Thus it appears to the model that the outer `fun` is simply part of a function definition, a common and innocuous phenomenon.

Correct Predictions

Next, we present a few indicative programs where our first prediction is correct, and all of the other tools' top three predictions are incorrect.

Extracting the Digits of an Integer. Consider first a simple recursive function `digitsOfInt` that extracts the digits of an `int`.

```
1  let rec digitsOfInt n =
2    if n <= 0 then
3      []
4    else
5      [n mod 10] @ [ digitsOfInt (n / 10) ]
```

Unfortunately, the student has decided to wrap the recursive call to `digitsOfInt` with a list literal, even though `digitsOfInt` already returns an `int list`. Thus, the list literal is inferred to have type `int list list`, which is incompatible with the `int list` on the left of the `@` (list append) operator. Both SHErrLoc and the OCaml compiler blame the recursive call for

returning a `int list` rather than `int`, but the recursive call is correct!

As our Tree correctly points out (with high confidence), the fault lies with the list literal *surrounding* the recursive call, remove it and the type error disappears. An examination of the path induced by the list literal reveals that our Tree is basing its decision on the fact that (1) the expression is a list literal; (2) the child expression is an application, whose return type mentions `int`; and (3) the parent expression's type mentions `list`. Interestingly, Tree incorrectly predicts that the child application should change as well, but it is less confident of this prediction and ranks it below the correct blame assignment.

Padding a list. Our next program, `padZero`, is given two `int lists` as input, and must left-pad the shorter one with enough zeros that the two output lists have equal length. The student first defines a helper `clone`.

```
1  let rec clone x n =
2    if n <= 0 then
3      []
4    else
5      x :: clone x (n - 1)
```

Then she defines `padZero` with a branch to determine which list is shorter, followed by a `clone` to zero-pad it.

```
1  let padZero l1 l2 =
2    let n = List.length l1 - List.length l2 in
3    if n < 0 then
4      (clone 0 ((-1) * n) @ l2, l2)
5    else
6      (l1, clone 0 n :: l2)
```

Alas, our student has accidentally used the `::` operator rather than the `@` operator in the `else` branch. SHErrLoc and OCaml correctly determine that she cannot cons the `int list` returned

by `clone` onto `l2`, which is another `int list`, but they decide to blame the call to `clone`, while our Tree correctly blames the `::` constructor.

Examining the path induced by the `::`, we can see that our Tree is influenced by the fact that: (1) `::` is a constructor; (2) the parent is a tuple; and (3) the leftmost child is an application. We note that first fact appears to be particularly significant; an examination of the training samples that reach that decision reveals that, before observing the Is-Constructor feature the classifier is slightly in favor of predicting "blame", but afterwards it is heavily in favor of predicting "blame". Many of the following decisions change the balance back towards "no blame" if the "true" path is taken, but the `::` constructor always takes the "false" path. It would appear that our Tree has learned that constructors are particularly suspicious, and is looking for exceptions to this general rule.

Our Tree correctly predicts that the recursive call blamed by SHErrLoc should not be blamed; a similar examination suggests that the crucial observation is that the recursive call's parent is a data constructor application.

4.3.6 Blame Utility

We have demonstrated in the preceding sections that we can produce more *accurate* blame assignments by learning from the collective mistakes of prior students; however, users are the final judge of the *utility* of an error message. Thus, in this final experiment we ask whether Nate's correct blame assignments aid users in *understanding* type errors more than incorrect assignments.

We assigned three problems to the students in our user study: the `padZero` and `mulByDigit` programs from § 4.3.5, as well as the following `sepConcat` program

```
1  let rec sepConcat sep sl =
2    match sl with
3    | [] -> ""
4    | h::t ->
5        let f a x = a ^ (sep ^ x) in
```

```
6         let base = [] in
7         List.fold_left f base sl
```

where the student has erroneously returned the empty list, rather than the empty string, in the base case of the fold. For each problem the students were additionally given either Nate's correct blame assignment or SHErrLoc's incorrect blame assignment, with no error message. The full user study is available in Appendix C. Due to the nature of an in-class exam, not every student answered every question, but we always received at least 12 (out of a possible 15 or 16) responses for each problem-tool pair. This session of the course was taught in Reason,[3] a dialect of OCaml with a more C-like syntax, and thus for the study we transcribed the programs to Reason syntax.

We then instructed three annotators (one of whom is an author, the others are graduate students at UCSD) to classify the answers as correct or incorrect. We performed an inter-rater reliability (IRR) analysis to determine the degree to which the annotators consistently graded the exams. As we had more than two annotators assigning nominal ("correct" or "incorrect") ratings we used Fleiss' kappa [32] to measure IRR. Fleiss' kappa is measured on a scale from 1, indicating total agreement, to −1, indicating total disagreement, with 0 indicating random agreement.

Finally, we used a one-sided Mann-Whitney U test [66] to determine the significance of our results. The null hypothesis was that the responses from students given Nate's blame were drawn from the same distribution as those given SHErrLoc's, *i.e.* Nate had no effect. Since we used a one-sided test, the alternative to the null hypothesis is that Nate had a *positive* effect on the responses. We reject the null hypothesis in favor of the alternative if the test produces a significance level $p < 0.05$, a standard threshold for determining statistical significance.

Results

The measured kappa values were $\kappa = 0.68$ for the explanations and $\kappa = 0.77$ for the fixes; while there is no formal notion for what consititutes strong agreement [54], kappa values above

[3] https://reasonml.github.io

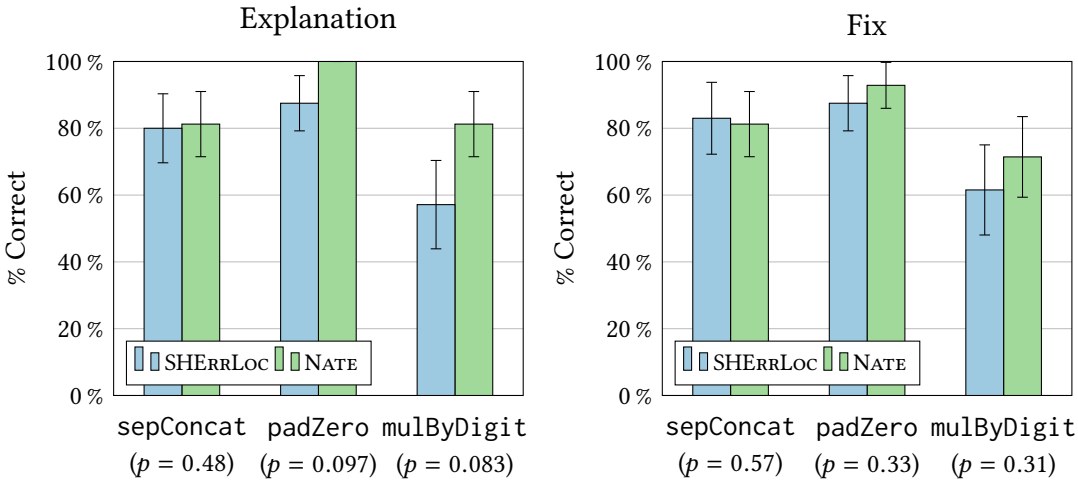


Figure 4.6. A classification of students' explanations and fixes for type errors, given either SHErrLoc or Nate's blame assignment. The students given Nate's location generally scored better than those given SHErrLoc's. We report the result of a one-sided Mann-Whitney U test for statistical significance in parentheses.

0.60 are often called "substantial" agreement [55]. Figure **??** summarizes a single annotator's results, which show that students given Nate's blame assignment were generally more likely to correctly explain and fix the type error than those given SHErrLoc's. There was no discernible difference between Nate and SHErrLoc for `sepConcat`; however, Nate responses for `padZero` and `mulByDigit` were marked correct 5–25% more often than the SHErrLoc responses. While the results appear to show a trend in favor of Nate, they do not rise to the level of statistical significance in this experiment; further investigation is merited.

Threats to Validity

Measuring understanding is difficult, and comes with its own set of threats.

Construct. We used the correctness of the student's explanation of, and fix for, the type error as a proxy for her understanding, but it is possible that other metrics would produce different results. A further threat arises from our decision to use Reason syntax rather than OCaml. Reason and OCaml differ only in syntax, the type system is the same; however, the difference in syntax may affect students' understanding of the programs. For example, Reason

uses the notation `[h, ...t]` for the list "cons" constructor, in contrast to OCaml's `h::t`. It is quite possible that Reason's syntax could help students remember that `h` is a single element while `t` is a list.

Internal. We assigned students randomly to two groups. The first was given SHErrLoc's blame assignment for `sepConcat` and `mulByDigit`, and Nate's blame for `padZero`; the second was given the opposite assignment. This ensured that each student was given SHErrLoc and Nate problems. Students without sufficient knowledge of Reason could affect the results, as could the time-constrained nature of an exam. Thus, we excluded any answers left blank from our analysis.

External. Our experiment used students in the process of learning Reason, and thus may not generalize to all developers. The three programs were chosen manually, via a random selection and filtering of the programs from the SP14 dataset, where Nate's top prediction was correct but SHErrLoc's was incorrect. A different selection of programs may lead to different results.

Subjects. We collected exams from 31 students, though due to the nature of the study not every student completed every problem. The number of complete submissions was always at least 12 out of a maximum of 15 or 16 per program-tool pair.

4.4 Limitations

We have shown that we can outperform the state of the art in type error localization by learning a model of the errors that programmers make, using a set of features that closely resemble the information the type checker sees. In this section we highlight some limitations of our approach and potential avenues for future work.

User-Defined Types

Probably the single biggest limitation of our technique is that we have (a finite set of) features for specific data and type constructors. Anything our models learn about errors made with the `::` constructor or the `list` type cannot easily be translated to new, user-defined datatypes the model has never encountered. We can mitigate this, to some extent, by adding generic syntactic features for data constructors and `match` expressions, but it remains to be seen how much these help. Furthermore, there is no obvious analog for transferring knowledge to new type constructors, which we have seen are both more compact and helpful.

As an alternative to encoding information about *specific* constructors, we might use a more abstract representation. For example, instead of modeling `x :: 2` as a `::` constructor with a right child of type `int`, we might model it as some (unknown) constructor whose right child has an incompatible type. We might symmetrically model the `2` as an integer literal whose type is incompatible with its parent. Anything we learn about `::` and `2` can now be transferred directly to yet unseen types, but we run the risk generalizing *too much* — *i.e.* perhaps programmers make different mistakes with `lists` than they do with other types, and are thus likely to choose different fixes. Balancing the trade-off between specificity and generalizability appears to be a challenging task.

Additional Features

There are a number of other features that could improve the model's ability to localize errors, that would be easier to add than user-defined types. For example, each occurrence of a variable knows only its type and its immediate neighbors, but it may be helpful to know about *other* occurrences of the same variable. If a variable is generally used as a `float` but has a single use as an `int`, it seems likely that the latter occurrence (or context) is to blame. Similarly, arguments to a function application are not aware of the constraints imposed on them by the function (and vice versa), they only know that they are occurring in the context of an application. Finally, *n-grams* on the token stream have proven effective for probabilistic modeling of programming languages [33, 43], we may find that they aid in our task as well. For

example, if the observed tokens in an expression diverge from the n-gram model's predictions, that indicates that there is something unusual about the program at that point, and it may signal an error.

Independent vs Joint Predictions

We treat each sub-expression as if it exists in a vacuum, but in reality the program has a rich *graphical* structure, particularly if one adds edges connecting different occurrences of the same variable. Raychev et al. [88] have used these richer models to great effect to make *interdependent* predictions about programs, *e.g.* de-obfuscating variable names or even inferring types. One could even view our task of locating the source of an error as simply another property to be predicted over a graphical model of the program. One of the key advantages of a graphical model is that the predictions made for one node can influence the predictions made for another node, this is known as *structured learning*. For example, if, given the expression `1 + true`, we predict `true` to be erroneous, we may be much less likely to predict `+` as erroneous. We compensate somewhat for our lack of structure by adding contextual features and by ranking our predictions by "confidence", but it would be interesting to see how structured learning over graphical models would perform.

4.5 Related Work

In this section we connect our work to related efforts in fault localization.

Fault Localization

Given a defect, *fault localization* is the task of identifying "suspicious" program elements (*e.g.* lines, statements) that are likely implicated in the defect (*i.e.* that should be changed to fix the defect) — thus, type error localization can be viewed as an instance of fault localization. The best-known fault localization technique is likely Tarantula, which uses a simple mathematical formula based on measured information from dynamic normal and buggy runs [45]. Other similar approaches, including those of Chen et al. [16] and Abreu et al. [1, 2] consider alternate features of information or refined formulae and generally obtain more precise results; see Wong

et al. [113] for a survey. While some researchers have approached such fault localization with an eye toward optimality (*e.g.* Yoo et al. [117] determine optimal coefficients), in general such fault localization approaches are limited by their reliance on either running tests or including relevant features. For example, Tarantula-based techniques require a normal and a buggy run of the program. By contrast, we consider incomplete programs with type errors that may not be executed in any standard sense. Similarly, the features available influence the classes of defects that can be localized. For example, a fault localization scheme based purely on control flow features will have difficulty with cross-site scripting or SQL code injection attacks, which follow the same control flow path on normal and buggy runs (differing only in the user-supplied data). Our feature set is comprised entirely of syntactic and typing features, a natural choice for type errors, but it would likely not generalize to other defects.

Endnotes

Acknowledgments

This chapter, in part, has been submitted for publication of the material as it may appear in: E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *In submission to OOPSLA '17*, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusion

The goal of this work has been to improve the diagnostic feedback that compilers provide when a program with no type annotations fails to type-check. To that end, we have made three key contributions that advance the state of the art in type error diagnosis.

Contribution 1: A Dataset of Novice Type Errors

Our first contribution was a new dataset of novice interactions with the OCaml top-level interpreter (in particular, type errors they encountered and their fixes). The dataset contains thousands of ill-typed programs written by over one hundred undergraduate students at UC San Diego, as well as the subsequent fixes. This is the largest set of novice type errors that we are aware of, has formed the backbone of our evaluation, and will hopefully be similarly useful to other researchers in the future.

Contribution 2: Dynamic Witnesses for Static Type Errors

Second, we presented a novel technique for explaining type errors in terms of the underlying runtime error the type system prevented. We interleave type inference and execution to search for a witness to the type error, a set of inputs that would cause the program to crash at runtime. We borrow the notion of "holes" from the automatic testing literature to avoid spurious witnesses by delaying the selection of a concrete input until execution has reached a point where we can be sure of its type. Once our search procedure finds a witness, we compute a full execution trace that demonstrates how the program would evolve and eventually crash.

We present this trace to the user in an interactive debugger that allows the user to explore the erroneous computation in a familiar setting.

We proved that our search procedure produces general witnesses, *i.e.* if we can find a witness the program must be untypeable. We showed empirically that most novice type errors, around 85%, admit witnesses, and that the vast majority can be found in under one second by our search procedure. We also found that students who were given our witnesses were more likely to correctly explain and fix a type error than students who were just given OCaml's error message. Finally, we found that our witnesses can also serve as a localization method for type errors by treating the stuck term as a sink for typing constraints and the values contained within it as sources. Our witness-based localizations are substantially more accurate than OCaml's errors and competitive with the state of the art.

Contribution 3: Data-Driven Diagnosis of Type Errors

Finally, we presented a novel technique for localizing type errors based on observations of past errors and their fixes. We use machine learning to train a classifier that predicts, given a term from an ill-typed program, whether the term is likely to be changed in the eventual fix (*i.e.* is that term to blame for the error). Given a new ill-typed program, we run the classifier for all program terms and use its confidence score to rank the terms by the likelihood that they should be blamed, selecting only the top three to present to the user. The classifier makes predictions based solely on the syntax and types of the term and its immediate parent and children, and, crucially, whether the term is part of a minimal type error slice.

Our classifier's top-ranked prediction is at least 16 percentage points more accurate than the state of the art in type error localization, and given three predictions it exceeds 90% accuracy. Furthermore, the classifier can be trained on a modest amount of data; we obtained our results by training on programs from a single instance of our undergraduate programming languages course at UC San Diego. This makes us confident that even if our model does not generalize to programs from other courses (or more broadly, to arbitrary OCaml programs), it is quite reasonable for instructors to train models of the specific errors made by students in

their courses.

5.1 Future Work

We will conclude this dissertation with a brief discussion of some exciting future directions for this line of work.

Other Type Systems and Analyses

This dissertation has focused on improving type errors for typed functional languages based on the Hindley-Milner type system, but there are a great many other type systems in use. Thus, one promising direction for future work would be adapting our techniques to other systems.

Both NanoMaLy and Nate have been designed to be parametric in the type system, so in principle it should be straightforward to adapt them to other languages and type systems. NanoMaLy's use of the type system is mostly confined to the narrow procedure that performs type-checking, thus one would need to adapt narrow to the target type system. If the dynamic semantics of the target language differ significantly from OCaml's, one would also need to replace the evaluation rules, but this is not a significant burden either. NanoMaLy's evaluation rules are just those of OCaml, with a call to narrow inserted before every primitive reduction. In contrast, Nate only uses the type system as a source of features, so one would only need to extract alternative features from the target type system.

We will next briefly outline a few classes of type systems and how supporting them might differ from OCaml.

Dependent Types Dependent types [9, 11, 77] and their close cousins, refinement types [28, 91, 105, 114], allow the programmer to specify complex invariants on their programs and data, and can statically prevent many runtime errors that OCaml cannot. These systems have been used to prove the absence out-of-bounds accesses [91, 108, 114], complex data-structure invariants (*e.g.* red-black tree balancing) [48, 108], security policies [7, 105], and even compiler correctness [62].

As one might expect, it is much harder to prove such properties about your programs than traditional type-safety, and thus programmers using such systems may spend much more time investigating type errors. Another consequence of the increased expressiveness is that these systems generally do not support global type inference as it becomes undecidable. Thus, the debugging type errors in these systems is more about *understanding* the error than *localizing* it. A crucial question the programmer must answer is whether the error is a legitimate bug in her program, or if the type checker simply lacks enough information to *prove* the program correct.

NanoMaLy's approach to searching for witnesses to type errors could thus be quite helpful in these systems. The presence of a witness proves that there is an actual bug, while the absence may suggest that the programmer needs to supply some additional lemmas to convince the type checker. We have done some preliminary work in this area, showing that dependent and refinement type signatures can be thought of as generators and oracles for comprehensive test suites [101]. However, that work only checked that a function satisfies its top-level signature, we did not search for witnesses to the misuse of other functions internally. Petiot et al. [82] present a technique for determining if a proof failure is due to a legitimate bug or a lack of available information, but they only evaluate it in the context of first-order, imperative C programs. It would be very interesting to apply these techniques to dependent and refinement type systems for functional languages.

Objects Objects are a common feature in popular languages like C++, Java, and C#, offering code reuse via inheritance and behavioral abstraction via interfaces. A core feature of type systems that support objects is *subtyping*, allowing values of the sub-type to be seamlessly used anywhere values of the super-type are expected. While these languages are gradually adopting type inference for local variables, they generally require type annotations for functions[1] as the addition of subtyping would force the compiler to guess the programmer's intent for the input types. For instance, did she want the function to accept objects of a specific type, or objects of any type that implement a particular interface?

[1] OCaml's own object system is a notable exception here.

Since the type checker is (generally) not responsible for guessing the programmer's intent in these systems, we suspect that type error localization is unlikely to be as serious problem as it is in OCaml; however, these languages may still benefit from NanoMaLy's approach to explaining type errors in terms of runtime errors. Novice users, in particular, may find it easier to understand the error when presented as a concrete runtime trace. Bayne et al. [5] demonstrate a tool similar to NanoMaLy that allows programmers to execute ill-typed Java programs, though their aim is user-driven testing of a program in spite of potentially irrelevant type errors, while ours is automated explanations of type errors. Furthermore, subtyping adds a similar question of whether the type error is legitimate or if a function was simply being too conservative in its input and output types, thus testing may help guide the programmer to a solution as suggested above.

Information Flow Control Several authors have proposed type systems for tracking who may access or modify certain pieces of data, *e.g.* medical records or paper reviews, in order to ensure confidentiality [26, 42, 71, 83, 104]. These systems typically associate a security label — ranging from simple "high" or "low" security label, to a set of privileged actors — in addition to a type with each object in the system, and ensure that only actors with sufficient privileges can access restricted objects. A major complication from traditional type-checking is that *implicit* information flows must be tracked in addition to explicit flows. While an explicit information flow could be returning a row from a database table, an implicit flow occurs when the program makes an observable decision based on a piece of information, *e.g.* by branching on the value of an object. Implicit flows must be restricted so that malicious users cannot infer privileged data by carefully constructed inputs to a system; a common tactic is to conservatively propagate security labels via implicit flows.

As with objects, these type systems do not perform global inference for the security labels, thus the type checker is not trying to infer the programmer's intent; however, the implicit flows can create a similar issue of errors being reported far from their source. Thus, we suspect that these systems could benefit from both approaches presented in this dissertation:

NATE's data-driven localizations could improve the accuracy of error reports, and NANOMALY's witnesses could help explain the errors in terms of the undesirable leaking of privileged data.

Fixing Type Errors

Throughout this entire dissertation we have focused on localizing and explaining type errors, but it would be nice to have a tool that could simply *fix* the error without any user intervention.

In addition to the techniques we mentioned in § 1.3.3 there is a wealth of existing work in the broader field of automatic program repair [see 56, § 4, for a survey] that we could draw from. A core challenge in program repair is fault localization, *i.e.* determining where the repair should take place. Thus, NATE's ability to accurately locate the source of type errors could provide a strong foundation on top of which to build repair systems.

Another exciting opportunity for future work would be using machine learning to predict fixes *in addition* to blame labels. In fact, this could be a very natural extension of our work on NATE, we can frame it as a classification problem as follows. Given a term from an ill-typed program, that we have identified as a candidate for blame, we want to predict which local syntactic feature should be enabled for the corresponding term in the program's fix. For example, in our `sumList` example, we would like the classifier to predict that the IS-INT feature should be enabled, instead of the IS-[] feature, in the fix. The problem then becomes a *multi-label* classification problem, as there are many possible syntactic features to choose from, but these problems are also well studied in the machine learning literature.

This approach could work nicely for relatively simple fixes like for `sumList`, or a use of the integer `+` rather than the floating-point `+.`, *etc.*, but many fixes will require multiple edits to the program. For example, even adding an extra argument to a function call would be a multi-edit fix in OCAML: first we would need to insert a new application node with the old node as its left child, then we would need to synthesize a new term for the right child. Clearly, such a fix cannot be generated by NATE as is, since there are an infinite number of possible terms but a finite number of labels that we can predict. However, there has been much work in the machine

learning community on models that can generate structured data, most notably images [36] and text [4], but also program terms [87, 89]. A sizable challenge to adopting these techniques is that they tend to require vast amounts of data to learn a precise model, and we have a comparatively small amount of type-error data. However, all is not lost. The type-error data allowed us to predict an accurate location for the fix, but the fix itself should be a type-*correct* program, and there are an abundance of type-correct programs publicly available in software repositories like GitHub. So it may be possible to train a precise model of type-correct programs using publicly available data, and then use that model to generate structured fixes to ill-typed programs.

Appendix A

Proofs for Section 3.2

Proof of Lemma 3. By induction on τ. In the base case $\tau = \langle f\ v^\alpha, \varnothing, \varnothing \rangle$ and α is trivially a refinement of v^α. In the inductive case, consider the single-step extension of τ, $\tau' = \tau, \langle e', \sigma', \theta' \rangle$. We show by case analysis on the evaluation rules that if $\theta(\alpha) \preceq \sigma(v)$, then $\theta'(\alpha) \preceq \sigma'(v)$.

We can immediately discharge all of the *-B rules as the calls to narrow return `stuck`. An examination of narrow shows that if narrow returns `stuck` then σ and θ are unchanged.

Case Plus-G: We narrow v_1 and v_2 to `int`, so we must consider the narrow$(v^\alpha, t, \sigma, \theta)$ and narrow$(n, \texttt{int}, \sigma, \theta)$ cases. The narrow$(n, \texttt{int}, \sigma, \theta)$ case is trivial as it does not change σ or θ. In the narrow$(v^\alpha, t, \sigma, \theta)$ case we will either find that $v \in \sigma$ or we will generate a fresh `int` and extend σ. Note that when we extend σ we also extend θ due to the call to $\mathcal{U}$, thus in the $v^\alpha \in \sigma$ sub-cases we cannot actually refine either v or α and thus the refinement is preserved. When we extend σ with a binding for v, the call to $\mathcal{U}$ ensures that we add a compatible binding for α if one was not already in θ, thus the refinement relation must continue to hold.

Case If-G{1,2}: Similar to Plus-G.

Case App-G: Similar to Plus-G.

Case Nil-G: This step cannot change σ or θ thus the refinement continues to hold trivially.

Case Cons-G: We narrow v_2 to $[t]$, so we must consider three cases of narrow.

narrow$(v^\alpha, t, \sigma, \theta)$**:** Similar to Plus-G.

narrow($[]^{t_1}, [t_2], \sigma, \theta$)**:** This case may extend θ but not σ, so the refinement continues to hold trivially.

narrow($v_1 ::^{t_1} v_2, [t_2], \sigma, \theta$)**:** Same as $[]^{t_1}$.

Case Match-List-G{1,2}: Similar to Plus-G.

Case Match-Pair-G: Similar to Plus-G.

■

Proof of Lemma 4. We can construct v from τ as follows. Let

$$\tau_i = \langle f\ \nu^\alpha, \varnothing, \varnothing\rangle, \ldots, \langle e_{i-1}, \sigma_{i-1}, \theta_{i-1}\rangle, \langle e_i, \sigma_i, \theta_i\rangle$$

be the shortest prefix of τ such that $\rho_{\tau_i}(f) \nsim t$. We will show that $\rho_{\tau_{i-1}}(f)$ must contain some other hole α' that is instantiated at step i. Furthermore, α' is instantiated in such a way that $\rho_{\tau_i}(f) \nsim t$. Finally, we will show that if we had instantiated α' such that $\rho_{\tau_i}(f) \sim t$, the current step would have gotten `stuck`.

Since θ_{i-1} and θ_i differ only in α' but the resolved types differ, we have $\alpha' \in \rho_{\tau_{i-1}}(f)$ and $\rho_{\tau_i}(f) = \rho_{\tau_{i-1}}(f)\,[t'/\alpha']$. Let s be a concrete type such that $\rho_{\tau_{i-1}}(f)\,[s/\alpha'] = t$. We show by case analysis on the evaluation rules that

$$\langle e_{i-1}, \sigma_{i-1}, \theta_{i-1} + \{\alpha' \mapsto s\}\rangle \hookrightarrow \langle \texttt{stuck}, \sigma, \theta\rangle$$

Case Plus-G: Here we narrow v_1 and v_2 to `int`, so the first case of narrow must apply (narrow(n, `int`, σ, θ) cannot apply as it does not change θ). In particular, since we extended θ_{i-1} with $\alpha' \mapsto t'$ we know that $\alpha' = \alpha$ and $t' =$ `int`. Let s be any concrete type that is incompatible with `int` and $\theta_s = \theta_{i-1} + \{\alpha \mapsto s\}$, narrow($\nu^\alpha$, `int`, σ_{i-1}, θ_s]) = $\langle$`stuck`$, \sigma_{i-1}, \theta_s\rangle$.

Case Plus-B{1,2}: These cases cannot apply as narrow does not update θ when it returns `stuck`.

Case IF-G{1,2}: Similar to PLUS-G.

Case IF-B: This case cannot apply as narrow does not update θ when it returns `stuck`.

Case APP-G: Similar to PLUS-G.

Case APP-B: This case cannot apply as narrow does not update θ when it returns `stuck`.

Case NIL-G: This case cannot apply as it does not update θ.

Case CONS-G: Here we narrow v_2 to $[t]$, so we must consider three cases of narrow.

narrow$(v^\alpha, t, \sigma, \theta)$**:** Similar to PLUS-G.

narrow$([]^{t_1}, [t_2], \sigma, \theta)$**:** For this case to extend θ with $\alpha' \mapsto t'$, either t_1 or t_2 must contain α'. Let s be any concrete type that is incompatible with t' and $\theta_s = \theta_{i-1} + \{\alpha \mapsto s\}$, narrow$(v^\alpha, \texttt{int}, \sigma_{i-1}, \theta_s]) = \langle \texttt{stuck}, \sigma_{i-1}, \theta_s \rangle$.

narrow$(v_1 ::^{t_1} v_2, [t_2], \sigma, \theta)$**:** Same as $[]^{t_1}$.

Case CONS-B: This case cannot apply as narrow does not update θ whe it returns `stuck`.

Case MATCH-LIST-G{1,2}: Here we narrow v to $[\alpha]$, so we must consider three cases of narrow.

narrow$(v^\alpha, t, \sigma, \theta)$**:** Similar to PLUS-G.

narrow$([]^{t_1}, [t_2], \sigma, \theta)$**:** This case cannot extend θ with $\alpha' \mapsto t'$ as we use a fresh α, which cannot be referenced by $\rho_{\tau_{i-1}}(f)$, in the call to narrow, and thus it cannot apply.

narrow$(v_1 ::^{t_1} v_2, [t_2], \sigma, \theta)$**:** Same as $[]^{t_1}$.

Case MATCH-LIST-B: This case cannot apply as narrow does not update θ whe it returns `stuck`.

Case MATCH-PAIR-G Here we narrow v to $\alpha_1 \times \alpha_2$, so we must consider two cases of narrow.

narrow$(v^\alpha, t, \sigma, \theta)$**:** Similar to PLUS-G.

narrow($\langle v_1, v_2\rangle, t_1 \times t_2, \sigma, \theta$)**:** This case cannot extend θ with $\alpha' \mapsto t'$ as we use a fresh α_1 and α_2, which cannot be referenced by $\rho_{\tau_{i-1}}(f)$, in the call to narrow, and thus it cannot apply.

Case Match-Pair-B: This case cannot apply as narrow does not update θ whe it returns `stuck`.

Finally, by Lemma 3 we know that $\rho_{\tau_{i-1}}(f) \preceq \sigma_{i-1}(v)$ and thus $\alpha' \in \sigma_{i-1}(v^{\alpha})$. Let $u = \mathsf{gen}(s, \theta)$ and $v = \sigma_{i-1}(v)\left[u/v'^{\alpha'}\right][s/\alpha']$, $\langle f\ v, \varnothing, \varnothing\rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma, \theta\rangle$ in i steps.

■

Appendix B

NanoMaLy User Study

B.1 Version A

9 Debugging and Functional Programming (16 points)

Consider these OCaml programs that *do not type-check* and their corresponding error messages (including the implicated code, shown underlined). Each has comments detailing what the program *should* do as well as sample invocations that *should* type-check.

```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h ::  t ::  ys

assert( append [1] [2] = [1;2] ) ;;

This expression has type
    'a list
but an expression was expected of type
    'a
The type variable 'a occurs inside 'a list
```

```
(* "digitsOfInt n" returns "[]" if "n" is
    not positive, and otherwise returns the
    list of digits of "n" in the order in
    which they appear in "n".  *)
let rec append x xs =
    match xs with
    | [] -> [x]
    | _  -> x ::  xs

let rec digitsOfInt n =
    if n <= 0 then
       []
    else
       append (digitsOfInt (n/10))
              [n mod 10]

assert( digitsOfInt 99 = [9;9] ) ;;

This expression has type
    int
but an expression was expected of type
    'a list
```

(a) *[2 pts]* Why is the `append` program not well-typed?

(b) *[2 pts]* Fix the `append` program.

(c) *[2 pts]* Why is the `digitsOfInt` program not well-typed?

(d) *[2 pts]* Fix the `digitsOfInt` program.

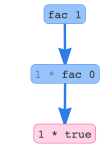
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

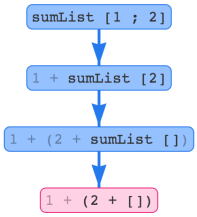
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

iii. When we return from `fac 0`, we are left with `1 * true`, indicating a program error: we cannot multiply an `int` with a `bool`.

```
let rec fac n =
    if n <= 0 then
        true
    else
        n * fac (n - 1)

assert( fac 1 = 1 ) ;;
```



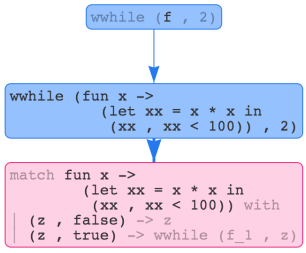

```
(* "sumlist xs" returns the sum of the
    integer elements of "xs" *)
let rec sumList xs = match xs with
    | []       -> []
    | y ::  ys -> y + sumList ys

assert( sumList [1;2] = 3 );;

Error encountered because
    'a list
is incompatible with
    int
```



(e) *[2 pts]* Why is the `sumList` program not well-typed?

(f) *[2 pts]* Fix the `sumList` program.

```
(* "wwhile (f, x)" returns x' where there exist
    values v_0, ..., v_n such that:

    - x is equal to v_0
    - x' is equal to v_n
    - for each i between 0 and n-2, we have
        (f v_i) equals (v_{i+1}, true)
    - (f v_{n-1}) equals (v_n, false) *)

let f x =
    let xx = x * x in
    (xx, (xx < 100))

let rec wwhile (f,b) =
    match f with
    | (z, false) -> z
    | (z, true)  -> wwhile (f, z)

assert( wwhile (f, 2) = 256 ) ;;

Error encountered because
    'a -> 'b
is incompatible with
    'c * 'd
```



(g) *[2 pts]* Why is the `wwhile` program not well-typed?

(h) *[2 pts]* Fix the `wwhile` program.

B.2 Version B

8 Debugging and Functional Programming (16 points)

Consider these OCaml programs that *do not type-check* and their corresponding error messages (including the implicated code, shown underlined). Each has comments detailing what the program *should* do as well as sample invocations that *should* type-check.

```
(* "sumlist xs" returns the sum of the
    integer elements of "xs" *)
let rec sumList xs = match xs with
    | []       -> []
    | y ::  ys -> y + sumList ys

assert( sumList [1;2] = 3 );;

This expression has type
    'a list
but an expression was expected of type
    int
```

```
(* "wwhile (f, x)" returns x' where there exist
    values v_0, ..., v_n such that:

    - x is equal to v_0
    - x' is equal to v_n
    - for each i between 0 and n-2, we have
        (f v_i) equals (v_{i+1}, true)
    - (f v_{n-1}) equals (v_n, false) *)

let f x =
    let xx = x * x in
    (xx, (xx < 100))

let rec wwhile (f,b) =
    match f with
    | (z, false) -> z
    | (z, true)  -> wwhile (f, z)

assert( wwhile (f, 2) = 256 ) ;;

This expression has type
    int -> int * bool
but an expression was expected of type
    'a * bool
```

(a) *[2 pts]* Why is the `sumList` program not well-typed?

(b) *[2 pts]* Fix the `sumList` program.

(c) *[2 pts]* Why is the `wwhile` program not well-typed?

(d) *[2 pts]* Fix the `wwhile` program.

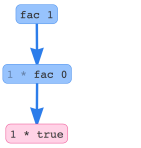
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

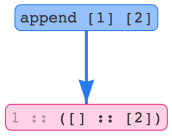
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

iii. When we return from `fac 0`, we are left with `1 * true`, indicating a program error: we cannot multiply an `int` with a `bool`.

```
let rec fac n =
    if n <= 0 then
        true
    else
        n * fac (n - 1)

assert( fac 1 = 1 ) ;;
```



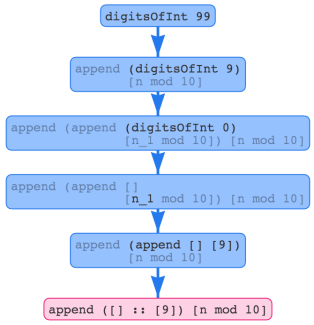

```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h :: t :: ys

assert( append [1] [2] = [1;2] ) ;;

Error encountered because
    int
is incompatible with
    int list
```



(e) *[2 pts]* Why is the `append` program not well-typed?

(f) *[2 pts]* Fix the `append` program.

```
(* "digitsOfInt n" returns "[]" if "n" is
    not positive, and otherwise returns the
    list of digits of "n" in the order in
    which they appear in "n".  *)
let rec append x xs =
    match xs with
    | [] -> [x]
    | _  -> x ::  xs

let rec digitsOfInt n =
    if n <= 0 then
       []
    else
       append (digitsOfInt (n/10))
              [n mod 10]

assert( digitsOfInt 99 = [9;9] ) ;;

Error encountered because
    'a list
is incompatible with
    int
```



(g) *[2 pts]* Why is the `digitsOfInt` program not well-typed?

(h) *[2 pts]* Fix the `digitsOfInt` program.

Appendix C

NATE User Study

C.1 Version A

CS4610 Exam 3 UVa ID: _____

6. Debugging, Opsems, Types (18 points)

Consider these shown Reason programs that *do not type-check*; the code implicated by the type checker will be highlighted and underlined. Each has English comments explaining what the program *should* do, as well as assertions that *should* type check *and* succeed.

(a)
```
/* "sepConcat sep [s1;s2;s3]" should insert "sep" between "s1", "s2", and "s3", and
     concatentate the result. */
/* Recall that List.fold_left takes a function, an accumulator, and a list as input */
let rec sepConcat = fun sep sl =>
  switch sl {
  | [] => ""
  | [h, ...t] =>
    let f = fun a x => a ^ (sep ^ x);
    let base = [];
    List.fold_left f base sl
  };

assert (sepConcat "," ["foo", "bar", "baz"] == "foo,bar,baz");
```

i. (3 points) Why is `sepConcat` not well-typed?

ii. (3 points) Describe how you would fix the code so that `sepConcat` works correctly.

(b)
```
/* "padZero xs ys" returns a pair "(xs', ys')" where the shorter of "xs" and "ys" has
     been left-padded by zeros until both lists have equal length. */
let rec clone = fun x n =>
  if (n <= 0) {
    []
  } else {
    [x, ...clone x (n - 1)]
  };

let padZero = fun l1 l2 => {
  let n = List.length l1 - List.length l2;
  if (n < 0) {
    (clone 0 ((-1) * n) @ l1, l2)
  } else {
    (l1, [clone 0 n, ...l2])
  }
};

assert (padZero [1, 2] [1] == ([1, 2], [0, 1]));
```

i. (3 points) Why is `padZero` not well-typed?

ii. (3 points) Describe how you would fix the code so that `padZero` works correctly.

(c)
```
/* "mulByDigit d [n1;n2;n3]" should multiply the "big integer" "[n1;n2;n3]"
   by the single digit "d". */
let rec mulByDigit = fun d n =>
  switch (List.rev n) {
  | [] => []
  | [h, ...t] => [mulByDigit d t, (h * d) mod 10]
  };

assert (mulByDigit 4 [2, 5] == [1, 0, 0]);
```

i. (3 points) Why is `mulByDigit` not well-typed?

ii. (3 points) Describe how you would fix the code so that `mulByDigit` works correctly.

C.2 Version B

CS4610 Exam 3 UVa ID: _____

6. Debugging, Opsems, Types (18 points)

Consider these shown Reason programs that *do not type-check*; the code implicated by the type checker will be highlighted and underlined. Each has English comments explaining what the program *should* do, as well as assertions that *should* type check *and* succeed.

(a)
```
/* "sepConcat sep [s1;s2;s3]" should insert "sep" between "s1", "s2", and "s3", and
     concatentate the result. */
/* Recall that List.fold_left takes a function, an accumulator, and a list as input */
let rec sepConcat = fun sep sl =>
  switch sl {
  | [] => ""
  | [h, ...t] =>
    let f = fun a x => a ^ (sep ^ x);
    let base = [];
    List.fold_left f base sl
  };

assert (sepConcat "," ["foo", "bar", "baz"] == "foo,bar,baz");
```

i. (3 points) Why is `sepConcat` not well-typed?

ii. (3 points) Describe how you would fix the code so that `sepConcat` works correctly.

(b)
```
/* "padZero xs ys" returns a pair "(xs', ys')" where the shorter of "xs" and "ys" has
     been left-padded by zeros until both lists have equal length. */
let rec clone = fun x n =>
  if (n <= 0) {
    []
  } else {
    [x, ...clone x (n - 1)]
  };

let padZero = fun l1 l2 => {
  let n = List.length l1 - List.length l2;
  if (n < 0) {
    (clone 0 ((-1) * n) @ l1, l2)
  } else {
    (l1, [clone 0 n, ...l2])
  }
};

assert (padZero [1, 2] [1] == ([1, 2], [0, 1]));
```

i. (3 points) Why is `padZero` not well-typed?

ii. (3 points) Describe how you would fix the code so that `padZero` works correctly.

CS4610 Exam 3 UVa ID: _____

(c)
```
/* "mulByDigit d [n1;n2;n3]" should multiply the "big integer" "[n1;n2;n3]"
   by the single digit "d". */
let rec mulByDigit = fun d n =>
  switch (List.rev n) {
  | [] => []
  | [h, ...t] => [mulByDigit d t, (h * d) mod 10]
  };

assert (mulByDigit 4 [2, 5] == [1, 0, 0]);
```

i. (3 points) Why is `mulByDigit` not well-typed?

ii. (3 points) Describe how you would fix the code so that `mulByDigit` works correctly.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, 2006. DOI: 10.1109/PRDC.2006.18.

[2] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION 2007, 2007. DOI: 10.1109/TAIC.PART.2007.13.

[3] G. Allen, F. Löffler, E. Schnetter, and E. L. Seidel. Component specification in the cactus framework: the cactus configuration language. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, GRID '10. IEEE, 2010. DOI: 10.1109/GRID.2010.5698008.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2014. arXiv: 1409.0473 [cs.CL].

[5] M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11. ACM, 2011. DOI: 10.1145/1985793.1985864.

[6] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.*, 2(1-4), 1993. DOI: 10.1145/176454.176460.

[7] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), 2011. DOI: 10.1145/1890028.1890031.

[8] K. L. Bernstein and E. W. Stark. Debugging Type Errors. Technical report, State University of New York at Stony Brook, 1995.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[10] P. Bielik, V. Raychev, and M. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33rd International Conference on Machine Learning*, ICML '16, 2016.

[11] E. Brady. Idris, a general-purpose dependently typed programming language: design and implementation. *J. Funct. Programming*, 23(05), 2013. DOI: 10.1017/S095679681300018X.

[12] L. Breiman. Random forests. *Mach. Learn.*, 45(1), 2001. DOI: 10.1023/A:1010933404324.

[13] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, Berkeley, CA, USA. USENIX Association, 2008.

[15] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. *Electronic Proceedings in Theoretical Computer Science*, 70, 2011. DOI: 10.4204/EPTCS.70.1.

[16] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, 2002. DOI: 10.1109/DSN.2002.1029005.

[17] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014. DOI: 10.1145/2535838.2535863.

[18] S. Chen and M. Erwig. Guided type debugging. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, Lecture Notes in Computer Science. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-07151-0_3.

[19] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01. ACM, 2001. DOI: 10.1145/507635.507659.

[20] D. R. Christiansen. Reflect on your mistakes! lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.

[21] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00. ACM, 2000. DOI: 10.1145/351240.351266.

[22] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, New York, NY, USA. ACM, 2005. DOI: 10.1145/1062455.1062488.

[23] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77. ACM, 1977. DOI: 10.1145/512950.512973.

[24] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Softw. Pract. Exp.*, 34(11), 2004. DOI: 10.1002/spe.602.

[25] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82. ACM, 1982. DOI: 10.1145/582153.582176.

[26] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), 1977. DOI: 10.1145/359636.359712.

[27] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1), 1996. DOI: 10.1016/0167-6423(95)00007-0.

[28] J. Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, New York, NY, USA. ACM, 2007. DOI: 10.1145/1292597.1292602.

[29] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt free ivory. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015. DOI: 10.1145/2804302.2804318.

[30] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[31] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, volume 31 of *PLDI '96*. ACM, 1996. DOI: 10.1145/249069.231387.

[32] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychol. Bull.*, 76(5), 1971. DOI: 10.1037/h0031619.

[33] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10. ACM, 2010. DOI: 10.1145/1882291.1882315.

[34] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science. Springer, 2004. DOI: 10.1007/11431664_5.

[35] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05. ACM, 2005. DOI: 10.1145/1065010.1065036.

[36] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. DRAW: a recurrent neural network for image generation, 2015. arXiv: 1502.04623 [cs.CV].

[37] C. Haack and J. B. Wells. Type error slicing in implicitly typed Higher-Order languages. In *Programming Languages and Systems*, Lecture Notes in Computer Science. Springer, 2003. DOI: 10.1007/3-540-36575-3_20.

[38] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science. Springer, 2006. DOI: 10.1007/978-3-540-74130-5_12.

[39] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intell. Syst.*, 24(2), 2009. DOI: 10.1109/MIS.2009.36.

[40] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer New York, 2009. DOI: 10.1007/978-0-387-84858-7.

[41] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2003. DOI: 10.1145/944705.944707.

[42] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, New York, NY, USA. ACM, 1998. DOI: 10.1145/268946.268976.

[43] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA. IEEE Press, 2012.

[44] R. Hindley. The principal Type-Scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146, 1969. DOI: 10.2307/1995158.

[45] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02. ACM, 2002. DOI: 10.1145/581339.581397.

[46] S. Joosten, K. Van Den Berg, and G. Van Der Hoeven. Teaching functional programming to first-year students. *J. Funct. Programming*, 3(01), 1993. DOI: 10.1017/S0956796800000599.

[47] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 46. ACM, 2011. DOI: 10.1145/1993316.1993550.

[48] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, New York, NY, USA. ACM, 2009. DOI: 10.1145/1542476.1542510.

[49] W. L. Khoo, E. L. Seidel, and Z. Zhu. Designing a virtual environment to evaluate multimodal sensors for assisting the visually impaired. In *Computers Helping People with Special Needs*, ICCHP '12. Springer, 2012. DOI: 10.1007/978-3-642-31534-3_84.

[50] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization, 2014. arXiv: 1412.6980 [cs.LG].

[51] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016. ACM, 2016. DOI: 10.1145/2931037.2931051.

[52] S. B. Kotsiantis. Supervised machine learning: a review of classification techniques. *Informatica*, 31(3), 2007.

[53] T. Kremenek and D. Engler. Z-Ranking: using statistical analysis to counter the impact of static analysis approximations. In R. Cousot, editor, *Static Analysis*. Volume 2694, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2003. DOI: 10.1007/3-540-44898-5_16.

[54] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 2012.

[55] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1), 1977.

[56] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Qual J*, 21(3), 2013. DOI: 10.1007/s11219-013-9208-0.

[57] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4), 1998. DOI: 10.1145/291891.291892.

[58] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, volume 35. ACM, 1999. DOI: 10.1145/331960.331977.

[59] E. Lempsink. *Generic type-safe diff and patch for families of datatypes*. Master's thesis, Universiteit Utrecht, 2009.

[60] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. ACM, 2007. DOI: 10.1145/1250734.1250783.

[61] B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ML type-error messages. In *Proceedings of the 2006 Workshop on ML*, ML '06. ACM, 2006. DOI: 10.1145/1159876.1159887.

[62] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), 2009. DOI: 10.1145/1538788.1538814.

[63] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06. ACM, 2006. DOI: 10.1145/1140335.1140356.

[64] F. Lindblad. Property directed generation of First-Order test data. In M. T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming*, volume 8 of *TFP '07*, 2007.

[65] C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan. A practical framework for type inference error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016. DOI: 10.1145/2983990.2983994.

[66] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, 18(1), 1947. DOI: 10.1214/aoms/1177730491.

[67] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11. ACM, 2011. DOI: 10.1145/1953163.1953308.

[68] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011. ACM, 2011. DOI: 10.1145/2048237.2048241.

[69] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, Lecture Notes in Computer Science. Springer, 1998. DOI: 10.1007/3-540-48515-5_9.

[70] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17(3), 1978. DOI: 10.1016/0022-0000(78)90014-4.

[71] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4), 2000. DOI: 10.1145/363516.363526.

[72] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010.

[73] M. Naylor and C. Runciman. Finding inputs that reach a target expression. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, 2007. DOI: 10.1109/SCAM.2007.30.

[74] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2), 1979. DOI: 10.1145/357073.357079.

[75] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03. ACM, 2003. DOI: 10.1145/944705.944708.

[76] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[77] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[78] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed random test generation. In *29th International Conference on Software Engineering*, ICSE '07, 2007. DOI: 10.1109/ICSE.2007.37.

[79] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14. ACM, 2014. DOI: 10.1145/2660193.2660230.

[80] Z. Pavlinovic, T. King, and T. Wies. Practical SMT-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015. DOI: 10.1145/2784731.2784765.

[81] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. ACM, 2012. DOI: 10.1145/2364527.2364579.

[82] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. Your proof fails? testing helps to find the reason. In B. K. Aichernig and C. A. Furia, editors, *Tests and Proofs*, Lecture Notes in Computer Science. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-41135-4_8.

[83] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1), 2003. DOI: 10.1145/596980.596983.

[84] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[85] V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical report HW-MACS-TR-0079, Herriot Watt University, 2010.

[86] V. Rahli, J. Wells, J. Pirie, and F. Kamareddine. Skalpel: a type error slicer for standard ML. *Electron. Notes Theor. Comput. Sci.*, 312, 2015. DOI: 10.1016/j.entcs.2015.04.012.

[87] V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016. DOI: 10.1145/2983990.2984041.

[88] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15. ACM, 2015. DOI: 10.1145/2676726.2677009.

[89] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 49. ACM, 2014. DOI: 10.1145/2594291.2594321.

[90] J. A. Robinson. A Machine-Oriented logic based on the resolution principle. *J. ACM*, 12(1), 1965. DOI: 10.1145/321250.321253.

[91] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, New York, NY, USA. ACM, 2008. DOI: 10.1145/1375581.1375602.

[92] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08. ACM, 2008. DOI: 10.1145/1411286.1411292.

[93] K. Sagonas, J. Silva, and S. Tamarit. Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13. ACM, 2013. DOI: 10.1145/2426890.2426897.

[94] T. Schilling. Constraint-Free type error slicing. In *Trends in Functional Programming*, Lecture Notes in Computer Science. Springer, 2011. DOI: 10.1007/978-3-642-32037-8_1.

[95] E. L. Seidel. Metadata management in scientific computing. *JOCSE*, 3(2), 2012.

[96] E. L. Seidel, G. Allen, S. Brandt, F. Löffler, and E. Schnetter. Simplifying complex software assembly: the component retrieval language and implementation. In *Proceedings of the 2010 TeraGrid Conference*, TG '10. ACM, 2010. DOI: 10.1145/1838574.1838592.

[97] E. L. Seidel and R. Jhala. A Collection of Novice Interactions with the OCaml Top-Level System, 2017. DOI: 10.5281/zenodo.806813.

[98] E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16. ACM, 2016. DOI: 10.1145/2951913.2951915.

[99] E. L. Seidel, R. Jhala, and W. Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *In submission to J. Funct. Programming*, 2017.

[100] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *In submission to OOPSLA '17*, 2017.

[101] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *Proceedings of the 24th European Symposium on Programming*, ESOP '15. Springer, 2015. DOI: 10.1007/978-3-662-46669-8_33.

[102] A. Serrano and J. Hage. Type error diagnosis for embedded DSLs by Two-Stage specialized type rules. In *Programming Languages and Systems*, Lecture Notes in Computer Science. Springer, 2016. DOI: 10.1007/978-3-662-49498-1_26.

[103] D. Seven. Knightmare: a DevOps cautionary tale. https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/, 2014. Accessed: 2017-4-24.

[104] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Information Security Technology for Applications*. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-29615-4_16.

[105] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, New York, NY, USA. ACM, 2011. DOI: 10.1145/2034773.2034811.

[106] N. Tillmann and J. d. Halleux. Pex–White box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, Lecture Notes in Computer Science. Springer, 2008. DOI: 10.1007/978-3-540-79124-9_10.

[107] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1), 2001. DOI: 10.1145/366378.366379.

[108] N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14. ACM, 2014. DOI: 10.1145/2633357.2633366.

[109] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14. ACM, 2014. DOI: 10.1145/2628136.2628161.

[110] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. ACM, 2012. DOI: 10.1145/2364527.2364554.

[111] M. Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86. ACM, 1986. DOI: 10.1145/512644.512648.

[112] D. A. Wheeler. The apple goto fail vulnerability: lessons learned. https://www.dwheeler.com/essays/apple-goto-fail.html, 2014. Accessed: 2017-4-24.

[113] W. E. Wong and V. Debroy. A survey of software fault localization. Technical report UTDCS-45-09, University of Texas at Dallas, 2009.

[114] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, New York, NY, USA. ACM, 1998. DOI: 10.1145/277650.277732.

[115] J. Yang. Explaining type errors by finding the source of a type conflict. In *Selected Papers from the 1st Scottish Functional Programming Workshop*, SFP '99, Exeter, UK. Intellect Books, 1999.

[116] J. Yang and G. Michaelson. A visualisation of polymorphic type checking. *J. Funct. Programming*, 10(01), 2000.

[117] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3), 2013. DOI: 10.1145/2491509.2491513.

[118] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM Press, 2014. DOI: 10.1145/2535838.2535870.

[119] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015. ACM, 2015. DOI: 10.1145/2737924.2738009.