Dynamic Witnesses for Static Type Errors

(or, ill-typed programs *usually* go wrong)

Abstract

Static type errors are a common stumbling block for newcomers to typed functional languages. We present a *dynamic* approach to explaining type errors by generating counterexample witness inputs that illustrate *how* an ill-typed program goes wrong. First, given an ill-typed function, we symbolically execute the body to dynamically synthesize witness values that can make the program go wrong. We prove that our procedure synthesizes *general witnesses* in that if a witness is found, then *for all* input types, there exist inhabitants that can make the function go wrong. Second, we show how to extend the above procedure to produce a *reduction graph* that can be used to interactively visualize and debug witness executions. Third, we evaluate our approach on two data sets comprising over 4,500 ill-typed student programs. Our technique is able to generate witnesses for 88% of the programs, and our reduction graph yields small counterexamples for 81% of the witnesses.

1. Introduction

Type errors are a common stumbling block for students trying to learn typed functional languages like OCaml and Haskell. Consider the ill-typed `fac` function on the left in Figure 1. The function returns `true` in the base case (instead of `1`), and so OCaml responds with the error message:

```
File "fac.ml", line 5, characters 8-17:
Error: This expression has type bool
       but an expression was expected
       of type int
```

This message makes perfect sense to an expert who is familiar with the language and has a good mental model of how the type system works. However, it may perplex a novice who has yet to develop such a mental model. To make matters worse, unification-based type inference algorithms often report errors far removed from their source. This further increases the novice's confusion and can actively mislead them to focus their investigation on an irrelevant piece of code. Much recent work has focused on analyzing unification constraints to properly *localize* a type error [2, 8, 13, 18], but an accurate source location still does not explain *why* the program is wrong.

In this paper 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 1, which shows how the recursive calls reduce to a configuration where the program "goes wrong" — *i.e.* the `int` value `1` is to be multiplied with the `bool` value `true`. 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 surprisingly tricky when we cannot rely on static type information. In particular, we must avoid the trap of *spurious* inputs that cause irrelevant problems 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; thereby 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* input types, there exist inhabitants that can make the function go wrong.

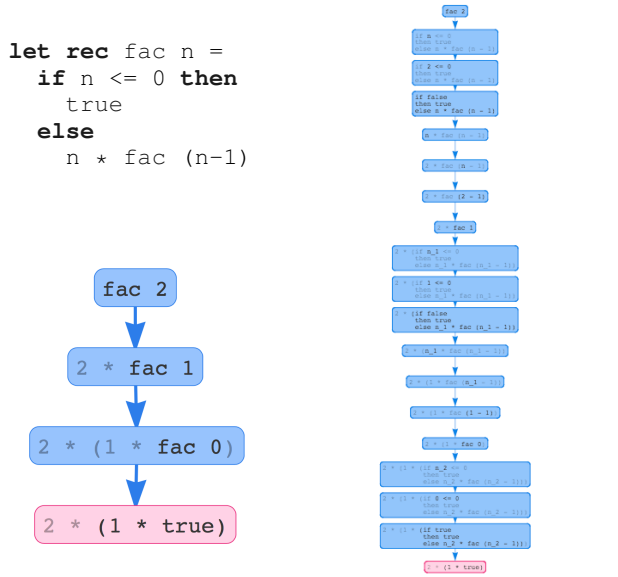


Figure 1. (TL) An ill-typed `fac` function; (BL) Dynamically witnessing the type error in `fac`, showing only function calls; (R) The same trace, fully expanded.

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 (§ 4). 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 visu-

alize 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 big steps to the next function call or return, for example the trace in the bottom-left of Figure 1 is a jump-compressed trace. Jump-compressed traces are similar to stack traces in that both show a sequence of function calls that lead to a crash, but the jump-compressed trace also shows 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 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 (§ 5). The first is the UW data set [8] comprising 284 ill-typed programs. The second is a new AUTHORS data set, comprising 4,407 ill-typed programs. We show that for both data sets, our technique is able to generate witnesses for nearly 90% 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 57% of the programs, and at most 10 steps for 81% of the programs.

Our results show that in the vast majority of cases, (novices') ill-typed programs *do* go wrong. This, in turn, opens the door to a novel dynamic way to explain, understand, and appreciate the benefits of static typing.

2. 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.

2.1 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 `fac` function from Figure 1. What *types* of inputs should we test `fac` with? Values of type `int` are fair game, but values of type, say, `string` or `int list` will cause the program to go wrong in an *irrelevant* manner. Concretely, we want to avoid testing `fac` with any type other than `int` because any other type would cause `fac` to get stuck immediately in the `n <= 0` test.

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 [15]. A hole — written $\nu[\alpha]$ — is a *placeholder* for a value ν of some unknown type α. We leave all inputs as uninstantiated holes until they are demanded by the program, *e.g.* due to a primitive operation like the <= test.

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 `fac`, given a hole as input we will narrow the hole to an `int` upon reaching the <= test. At this point we choose a random `int`[1] for the instantiation and concrete execution takes over entirely, leading us to the expected crash in the multiplication.

Witness Generality We show in § 3.3 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* type you might assign to the function, there exist inhabitants 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 `fac` takes a single argument instead of two (or none)? It is clear, syntactically, that `fac` 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 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.

[1] With standard heuristics to favor small values.

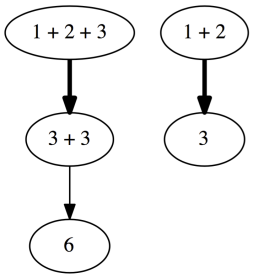


Figure 2. The reduction graph for `1+2+3`. The two edges produced by the transition from `1+2+3` to `3+3` are highlighted.

2.2 Visualizing Witnesses

We have shown 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 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 an uninteresting sub-computation.

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 § 4 — 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 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. 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 (§ 3.1). Second, we develop a (non-deterministic) operational semantics

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & \mathbf{e} & ::= & e \mid \texttt{stuck} \\
& e & ::= & v \mid x \mid e\ e \mid e + e \\
& & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid & \langle e, e\rangle \mid \texttt{case}\ e\ \texttt{of}\ \langle x, x\rangle \to e \\
& & \mid & \texttt{node}\ e\ e\ e \mid \texttt{leaf} \\
& & \mid & \texttt{case}\ e\ \texttt{of} \begin{cases} \texttt{leaf} \to e \\ \texttt{node}\ x\ x\ x \to e \end{cases} \\
\textbf{\textit{Values}} & v & ::= & n \mid b \mid \lambda x.e \mid \nu[\alpha] \mid tr \\
& tr & ::= & \texttt{node}[t]\ v\ v\ v \mid \texttt{leaf}[t] \\
\textbf{\textit{Integers}} & n & ::= & 0, 1, -1, \ldots \\
\textbf{\textit{Booleans}} & b & ::= & \texttt{true} \mid \texttt{false} \\
\textbf{\textit{Types}} & t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{fun} \\
& & \mid & t \times t \mid \texttt{tree}\ t \mid \alpha \\
\textbf{\textit{Substitutions}} & \sigma & ::= & \varnothing \mid \sigma\,[\nu[\alpha] \mapsto v] \\
& \theta & ::= & \varnothing \mid \theta\,[\alpha \mapsto t] \\
\textbf{\textit{Contexts}} & C & ::= & \bullet \mid C\ e \mid v\ C \\
& & \mid & C + e \mid v + C \\
& & \mid & \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid & \langle C, e\rangle \mid \langle v, C\rangle \\
& & \mid & \texttt{case}\ C\ \texttt{of}\ \langle x, x\rangle \to e \\
& & \mid & \texttt{node}\ C\ e\ e \\
& & \mid & \texttt{node}\ v\ C\ e \\
& & \mid & \texttt{node}\ v\ v\ C \\
& & \mid & \texttt{case}\ C\ \texttt{of} \begin{cases} \texttt{leaf} \to e \\ \texttt{node}\ x\ x\ x \to e \end{cases}
\end{array}
$$

Figure 3. Syntax of λ^H

for ill-typed programs that precisely defines the notion of a *witness* (§ 3.2). 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" (§ 3.3). Finally, we refine the operational semantics into a *search procedure* that returns concrete (general) witnesses for ill-typed programs § (3.4). We have formalized and tested our semantics and generality theorem in PLT-REDEX [5].

3.1 Syntax

Figure 3 describes the syntax of λ^H, a simple lambda calculus with integers, booleans, pairs, and binary trees. As we are specifically interested in programs that *do* go wrong, we include an explicit `stuck` term in our syntax. We write $\mathbf{e}$ 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 be equipping our term language with a notion of a *hole*, written $\nu[\alpha]$, which represents an *unconstrained* value ν 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 $\texttt{node}[\alpha]\ \nu[\alpha]\ \texttt{leaf}[\alpha]\ \texttt{leaf}[\alpha]$ 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 [15]). 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.

3.2 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: called *narrow* (narrow) and *generate* (gen), that "dynamically" perform type-checking and hole-filling respectively.

Narrowing Types The procedure

$$\mathsf{narrow} : v \times t \times \sigma \times \theta \to \langle v \cup \mathtt{stuck}, \sigma, \theta \rangle$$

defined in Figure 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. 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 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 $\nu[\alpha]$). When generating a `tree` t we must take care to ensure the resulting tree is well-typed. For a polymorphic type `tree` α or $\alpha_1 \times \alpha_2$ we will place holes in the generated value; they will be lazily filled in later, on demand.

Steps and Traces Figure 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.

Recursion Our semantics lacks a built-in `fix` construct for defining recursive function, 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.

3.3 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 $\langle f\ \nu[\alpha], \varnothing, \varnothing \rangle \hookrightarrow^* \langle \mathtt{stuck}, \sigma, \theta \rangle$, 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-term f, if $\langle f\ \nu[\alpha], \varnothing, \varnothing \rangle \hookrightarrow^\tau \langle \mathtt{stuck}, \sigma, \theta \rangle$, then for every type t there exists a value v of type t such that $\langle f\ v, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \mathtt{stuck}, \sigma', \theta' \rangle$.*

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 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 $\mathsf{typeof}(v)$ shown in Figure 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,

$$\mathsf{typeof}(\lambda x.x + 1) = \mathtt{fun}$$

instead of `int` $\to$ `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 $f\ \nu[\alpha]$ 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 $\langle f\ \nu[\alpha], \varnothing, \varnothing \rangle \hookrightarrow^\tau \langle \mathbf{e}, \sigma, \theta \rangle$, 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 $\mathsf{typeof}(v) \sim t$, 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 $t \preceq \mathsf{typeof}(v)$, *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 $\tau \doteq \langle f\ \nu[\alpha], \varnothing, \varnothing \rangle, \ldots$ and 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 $\theta \neq \theta'$. Note that only narrow can change θ, and it can only do so via unify, which can only extend θ. ■

$$
\begin{array}{lcl}
\mathsf{narrow} & : & v \times t \times \sigma \times \theta \to \langle v \cup \texttt{stuck}, \sigma, \theta \rangle \\
\mathsf{narrow}(\nu[\alpha], t, \sigma, \theta) & \doteq & \begin{cases} \langle v, \sigma, \theta' \rangle & \text{if } v = \sigma(\nu[\alpha]), \theta' = \mathsf{unify}(\{\alpha, t, \mathsf{typeof}(v)\}, \theta) \\ \langle \texttt{stuck}, \sigma, \theta \rangle & \text{if } v = \sigma(\nu[\alpha]) \\ \langle v, \sigma\,[\nu[\alpha] \mapsto v], \theta' \rangle & \text{if } \theta' = \mathsf{unify}(\{\alpha, t\}, \theta), v = \mathsf{gen}(t, \theta') \end{cases} \\
\mathsf{narrow}(n, \texttt{int}, \sigma, \theta) & \doteq & \langle n, \sigma, \theta \rangle \\
\mathsf{narrow}(b, \texttt{bool}, \sigma, \theta) & \doteq & \langle b, \sigma, \theta \rangle \\
\mathsf{narrow}(\lambda x.e, \texttt{fun}, \sigma, \theta) & \doteq & \langle \lambda x.e, \sigma, \theta \rangle \\
\mathsf{narrow}(\langle v_1, v_2 \rangle, t_1 \times t_2, \sigma, \theta) & \doteq & \langle \langle v_1, v_2 \rangle, \sigma, \theta'' \rangle, \text{if } \theta' = \mathsf{unify}(\{\mathsf{typeof}(v_1), t_1\}, \theta), \theta'' = \mathsf{unify}(\{\mathsf{typeof}(v_2), t_2\}, \theta') \\
\mathsf{narrow}(\texttt{leaf}[t_1], \texttt{tree}\ t_2, \sigma, \theta) & \doteq & \langle \texttt{leaf}[t_1], \sigma, \theta' \rangle, \text{if } \theta' = \mathsf{unify}(\{t_1, t_2\}, \theta) \\
\mathsf{narrow}(\texttt{node}[t_1]\ v_1\ v_2\ v_3, \texttt{tree}\ t_2, \sigma, \theta) & \doteq & \langle \texttt{node}[t_1]\ v_1\ v_2\ v_3, \sigma, \theta' \rangle, \text{if } \theta' = \mathsf{unify}(\{t_1, t_2\}, \theta) \\
\mathsf{narrow}(v, t, \sigma, \theta) & \doteq & \langle \texttt{stuck}, \sigma, \theta \rangle
\end{array}
$$

Figure 4. Narrowing values

$$
\begin{array}{lcll}
\mathsf{gen} & : & t \times \theta \to v & \\
\mathsf{gen}(\alpha, \theta) & \doteq & \mathsf{gen}(\theta(\alpha), \theta), & \text{if } \alpha \in dom(\theta) \\
\mathsf{gen}(\texttt{int}, \theta) & \doteq & n, & \text{non-deterministic} \\
\mathsf{gen}(\texttt{bool}, \theta) & \doteq & b, & \text{non-deterministic} \\
\mathsf{gen}(t_1 \times t_2, \theta) & \doteq & \langle \mathsf{gen}(t_1, \theta), \mathsf{gen}(t_2, \theta) \rangle & \\
\mathsf{gen}(\texttt{tree}\ t, \theta) & \doteq & tr, & \text{non-deterministic} \\
\mathsf{gen}(\texttt{fun}, \theta) & \doteq & \lambda x.\nu[\alpha], & \nu \text{ and } \alpha \text{ are fresh} \\
\mathsf{gen}(\alpha, \theta) & \doteq & \nu[\alpha], & \nu \text{ is fresh}
\end{array}
$$

Figure 5. Generating values

Second, we show that at each step of evaluation, the partial input type of f is a refinement of the instantiation of $\nu[\alpha]$.

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

Proof. By induction on τ. In the base case $\tau = \langle f\ \nu[\alpha], \varnothing, \varnothing \rangle$ and α is trivially a refinement of $\nu[\alpha]$. 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 $\rho_\tau(f) \preceq \sigma(\nu[\alpha])$, then $\rho_{\tau'}(f) \preceq \sigma'(\nu[\alpha])$. ■

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* $\langle f\ \nu[\alpha], \varnothing, \varnothing \rangle \hookrightarrow^\tau \langle e, \sigma, \theta \rangle$ *and* $t \nsim \rho_\tau(f)$, *then there exists a* v *such that* $\mathsf{typeof}(v) = t$ *and* $\langle f\ v, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma', \theta' \rangle$ *in at most* k *steps, where* k *is the length of* τ.

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

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

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`.

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 $\rho_{\tau_{i-1}}(f) \preceq \sigma_{i-1}(\nu[\alpha])$ and thus $\alpha' \in \sigma_{i-1}(\nu[\alpha])$. Let

$$\begin{array}{rcl} u & = & \mathsf{gen}(s, \theta) \\ v & = & \sigma_{i-1}(\nu[\alpha])\,[u/\nu'[\alpha']]\,[s/\alpha'] \end{array}$$

$\langle f\ v, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \texttt{stuck}, \sigma, \theta \rangle$ 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 $\tau = \langle f\ \nu[\alpha], \varnothing, \varnothing \rangle, \ldots, \langle \texttt{stuck}, \sigma, \theta \rangle$. The value $v = \sigma(\nu[\alpha])$ demonstrates that $f\ v$ gets stuck.

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

■

3.4 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.

In order to convert the semantics into a *procedure* for finding witnesses, we must address two challenges. 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 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. First, a search bound k which is used to define the *number* of traces to explore. (We assume, without loss of generality, that all traces are finite.) Second, the target expression e that contains the type error, and may be a curried function of multiple arguments. The witness generation procedure returns as output 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) via the procedure

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

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

$$\text{E-Plus-Good}\ \frac{\langle n_1, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta) \quad \langle n_2, \sigma'', \theta'' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma', \theta') \quad n = n_1 + n_2}{\langle C\,[v_1 + v_2]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[n]\,, \sigma'', \theta'' \rangle}$$

$$\text{E-Plus-Bad1}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta)}{\langle C\,[v_1 + v_2]\,, \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\text{E-Plus-Bad2}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma, \theta)}{\langle C\,[v_1 + v_2]\,, \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\text{E-If-Good1}\ \frac{\langle \mathtt{true}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C\,[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[e_1]\,, \sigma', \theta' \rangle}$$

$$\text{E-If-Good2}\ \frac{\langle \mathtt{false}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C\,[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[e_2]\,, \sigma', \theta' \rangle}$$

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

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

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

$$\text{E-Leaf-Good}\ \frac{\alpha \text{ is fresh}}{\langle C\,[\mathtt{leaf}]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[\mathtt{leaf}[\alpha]]\,, \sigma, \theta \rangle}$$

$$\text{E-Node-Good}\ \frac{t = \mathsf{typeof}(v_1) \quad \langle v_2', \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1) \quad \langle v_3', \sigma_3, \theta_3 \rangle = \mathsf{narrow}(v_3, \mathtt{tree}\ t, \sigma_2, \theta_2)}{\langle C\,[\mathtt{node}\ v_1\ v_2\ v_3]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[\mathtt{node}[t]\ v_1\ v_2'\ v_3']\,, \sigma_3, \theta_3 \rangle}$$

$$\text{E-Node-Bad1}\ \frac{t = \mathsf{typeof}(v_1) \quad \langle \mathtt{stuck}, \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1)}{\langle C\,[\mathtt{node}\ v_1\ v_2\ v_3]\,, \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle}$$

$$\text{E-Node-Bad2}\ \frac{t = \mathsf{typeof}(v_1) \quad \langle v_2', \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1) \quad \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle = \mathsf{narrow}(v_3, \mathtt{tree}\ t, \sigma_2, \theta_2)}{\langle C\,[\mathtt{node}\ v_1\ v_2\ v_3]\,, \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle}$$

$$\text{E-Case-Good1}\ \frac{\alpha \text{ is fresh}, \langle \mathtt{leaf}[t], \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \mathtt{tree}\ \alpha, \sigma, \theta)}{\langle C \left[\mathtt{case}\ v\ \mathtt{of} \begin{cases} \mathtt{leaf} \to e_1 \\ \mathtt{node}\ x_1\ x_2\ x_3 \to e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle C\,[e_1]\,, \sigma_1, \theta_1 \rangle}$$

$$\text{E-Case-Good2}\ \frac{\alpha \text{ is fresh}, \langle \mathtt{node}[t]\ v_1\ v_2\ v_3, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v_1, \mathtt{tree}\ \alpha, \sigma, \theta)}{\langle C \left[\mathtt{case}\ v\ \mathtt{of} \begin{cases} \mathtt{leaf} \to e_1 \\ \mathtt{node}\ x_1\ x_2\ x_3 \to e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle C\,[e_2\,[v_1/x_1]\,[v_2/x_2]\,[v_3/x_3]]\,, \sigma_1, \theta_1 \rangle}$$

$$\text{E-Case-Bad}\ \frac{\alpha \text{ is fresh}, \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \mathtt{tree}\ \alpha, \sigma, \theta)}{\langle C \left[\mathtt{case}\ v\ \mathtt{of} \begin{cases} \mathtt{leaf} \to e_1 \\ \mathtt{node}\ x_1\ x_2\ x_3 \to e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle}$$

$$\text{E-Case-Pair-Good}\ \frac{\alpha_1, \alpha_2 \text{ are fresh}, \langle \langle v_1, v_2 \rangle, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C\,[\mathtt{case}\ v\ \mathtt{of}\ \langle x_1, x_2 \rangle \to e]\,, \sigma, \theta \rangle \hookrightarrow \langle C\,[e\,[v_1/x_1]\,[v_2/x_2]]\,, \sigma_1, \theta_1 \rangle}$$

$$\text{E-Case-Pair-Bad}\ \frac{\alpha_1, \alpha_2 \text{ are fresh}, \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C\,[\mathtt{case}\ v\ \mathtt{of}\ \langle x_1, x_2 \rangle \to e]\,, \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_1, \theta_1 \rangle}$$

Figure 6. Evaluation relation for λ^H

$$
\begin{array}{lcl}
\mathsf{typeof}(n) & \doteq & \texttt{int} \\
\mathsf{typeof}(b) & \doteq & \texttt{bool} \\
\mathsf{typeof}(\lambda x.e) & \doteq & \texttt{fun} \\
\mathsf{typeof}(\langle v_1, v_2 \rangle) & \doteq & \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2) \\
\mathsf{typeof}(\texttt{leaf}[t]) & \doteq & \texttt{tree}\ t \\
\mathsf{typeof}(\texttt{node}[t]\ v_1\ v_2\ v_3) & \doteq & \texttt{tree}\ t \\
\mathsf{typeof}(\nu[\alpha]) & \doteq & \alpha
\end{array}
$$

Figure 7. The *dynamic type* of a value.

$$
\begin{array}{lcll}
\mathsf{Saturate} & : & e \to e & \\
\mathsf{Saturate}(e) & = & \textbf{case}\ \mathsf{eval}(e)\ \textbf{of} & \\
\quad \langle \lambda x.e, \sigma, \theta \rangle, \ldots & \to & \mathsf{Saturate}(e\ \nu[\alpha]) & (\nu, \alpha \text{ are fresh}) \\
\quad \_ & \to & e &
\end{array}
$$

Figure 8. Generating a saturated application.

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

Figure 9. Generating witnesses.

Due to the non-determinism introduced by gen, a call $\mathsf{eval}(e)$ 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 8, that takes as input an expression e and produces a *saturated* expression of the form $e\ \nu_1[\alpha_1] \ldots \nu_n[\alpha_n]$ 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 9 summarizes the overall implementation of our search for witnesses with the procedure $\mathsf{GenWitness}(k,\ e)$, 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 $\mathsf{Saturate}(e)$ 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*

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

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' to be the uncurried version of f that takes all of its arguments as a single nested pair, and then apply Theorem 1 to f'. ■

4. 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 an interactive visualization of program executions. First, we extend our semantics (§ 4.1) to record each reduction step in a *trace*, producing a *reduction graph* alongside the witness. Then we describe a set of common *interactive debugging* steps that can be expressed as simple traversals over the reduction graph (§ 4.2), yielding an interactive debugger that allows the user to effectively visualize *how* the program goes (wrong).

4.1 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:

$$G ::= \bullet \mid e \rightsquigarrow e; G$$

Tracing Semantics We extend the transition relation (§ 3.2) 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 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 $\langle e, \sigma, \theta, G \rangle \hookrightarrow \langle e', \sigma', \theta', e \rightsquigarrow e'; G \rangle$.

4.2 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 — and the set of *commands* available to user and what they do.

Visualization State A *visualization state* 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 10. The *initial state* is the graph containing a single edge linking the initial and final expressions.

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

- StepForward, StepBackward: show the result of a single step forward or backward respectively,
- JumpForward, JumpBackward: show the result of taking multiple steps (a *"big"* step) up to the first function call, or return, forward or backward respectively,
- StepInto: show the result of stepping into a function call in a sub-term, isolating it in a new thread, and
- StepOver: show the result of skipping over a function call in a sub-term.

Jump Compression A *jump compressed* trace is one whose edges are limited to forward or backward jumps. In our experience, jump compression abstracts many details of the computation that are often uninteresting or irrelevant to the explanation. In particular, jump compressed traces hide low-level operations and summarize

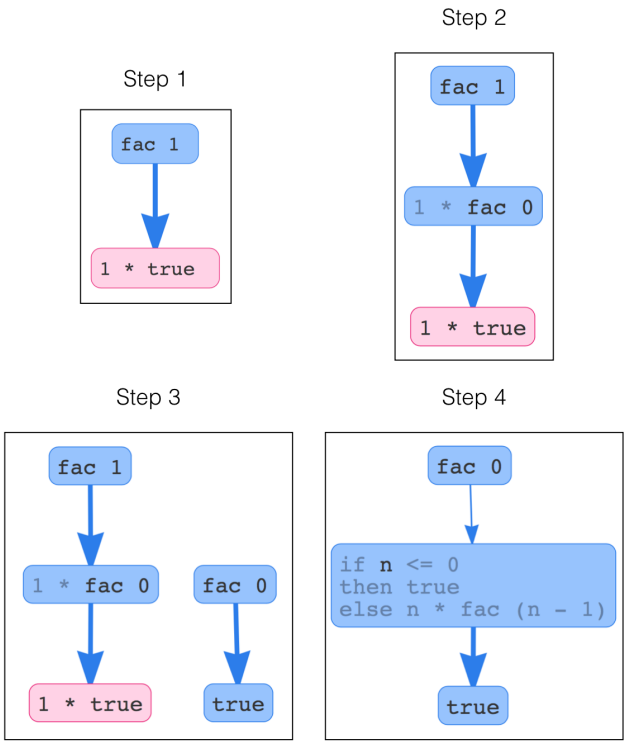


Figure 10. A sequence of interactions with the trace of `fac 1`. The stuck term is red, in each node the redex is highlighted. Thick arrows denote a multi-step transition, thin arrows denote a single-step transition. We start in step 1. In step 2 we jump forward from the witness to the next function call. In step 3 we step into the recursive `fac 0` call, which spawns a new "thread" of execution. In step 4 we take a single step forward from `fac 0`.

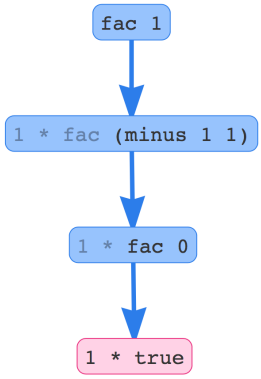


Figure 11. Jump-compressed trace of `fac 1` with subtraction implemented as a function call.

function calls as call-return pairs, see Figure 11 for a variant of `fac` that implements the subtraction as a function call instead of a primitive. Once users have identified interesting call-return pairs, they can step into those calls and proceed with more fine-grained steps. Note that jump compressed traces are not quite the same as stack-traces as they show *all* function calls, including those that returned successfully.

5. Evaluation

We have implemented a prototype of our search procedure and trace visualization for the purely functional subset of OCaml, in a tool called NanoMaly. 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 two questions we seek to answer with our evaluation:

1. ***Witness Coverage*** How many ill-typed programs can we find witnesses for?
2. ***Witness Complexity*** How *complex* are the traces produced by the witnesses?

Benchmarks We answer these 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 undergraduate Programming Languages course at our institution. We recorded each interaction with the OCaml top-level system over the course of the first three assignments (IRB # hidden for blind review), from which we extracted 4,407 distinct, ill-typed OCaml programs. The second dataset — widely used in the literature — comes from a similar course at the University of Washington [7], from which we extracted 284 ill-typed programs. Both datasets include programs that demonstrate a variety of functional programming idioms, including (tail) recursive functions, higher-order functions, polymorphic data types, and expression evaluators.

5.1 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 10 minutes 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 Figure 12. In both datasets our tool was able to find a witness for 83% 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 hit a maximum of 87% coverage. Interestingly, while the vast majority of witnesses corresponded to a type-error, as expected, 3-4% triggered an unbound variable error (even though OCaml reported a type error) and 2-3% triggered an infinite recursion error. For the remaining 12% of programs we were unable to provide any useful feedback as they either completed 1,000 tests successfully, or timed out after 10 minutes. While a more advanced search procedure, *e.g.* dynamic-symbolic execution, could likely trigger more of the type errors, our experiments show that type errors are coarse enough (or that novice programs are *simple* enough) that these techniques are not necessary.

5.2 Trace Complexity

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

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

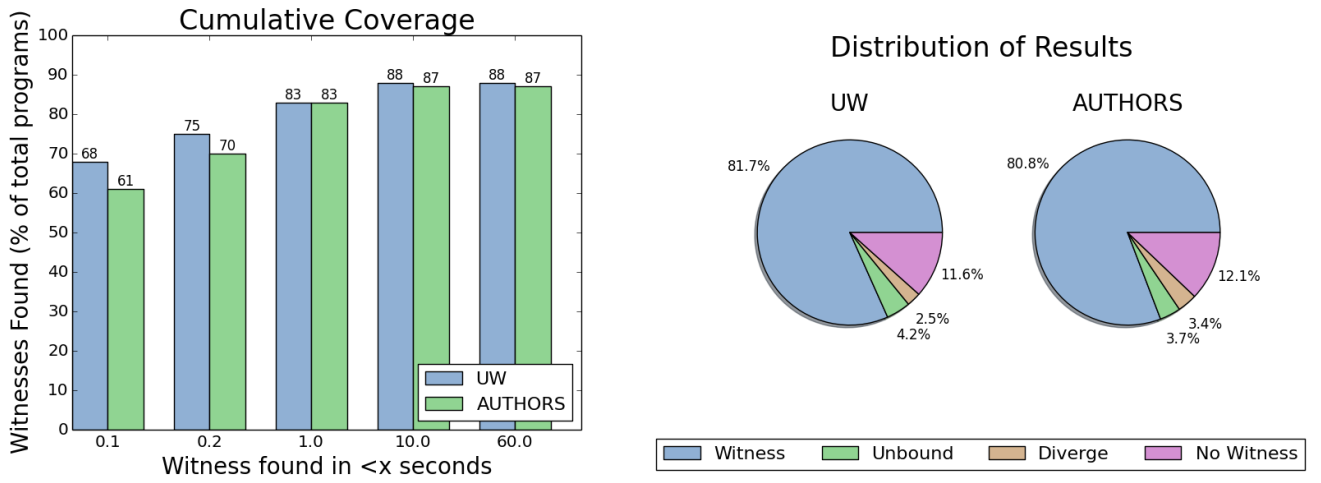


Figure 12. Results of our coverage testing and the distribution of test outcomes. Our random search successfully finds witnesses for 83% of the programs in under one second, improving slightly to 87% in under 10 seconds. In both datasets we detect actual type errors about 82% of the time, unbound variables or constructors 3-4% of the time, and diverging loops 2-3% of the time. For the remaining 11-12% of the programs we are unable to provide any useful feedback.

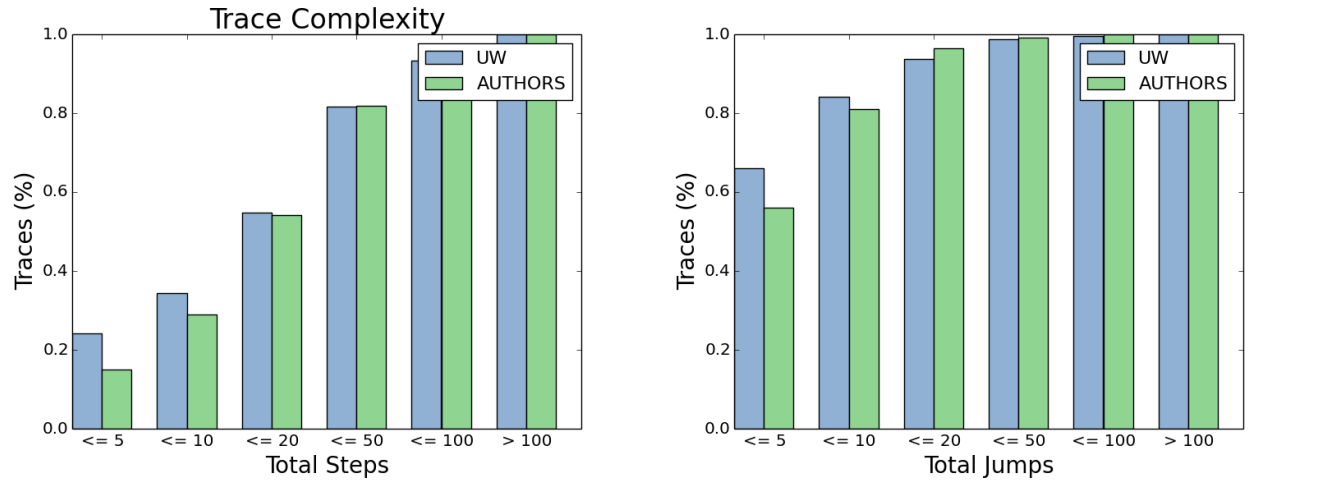


Figure 13. Complexity of the generated traces. 81% of the combined traces have a jump complexity of at most 10, with an average complexity of 7 and a median of 5.

Results The results of the experiment are summarized in Figure 13. The average number of single-step reductions per trace is 31 for the AUTHORS dataset (35 for the UW dataset) with a maximum of 2,745 (660 for UW) and a median of 16 (also 16 for UW). The average number of jumps per trace is 7 (also 7 for UW) with a maximium of 353 (105 for UW) and a median of 4 (also 4 for UW). In both datasets 80% or more traces have at most 10 jumps.

5.3 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, and (2) with traces that have a low average complexity of 7 jumps.

There are, of course, drawbacks to our approach. Three that stand out are: (1) coverage limits due to random generation, (2) the inability to handle certain instances of infinite types, and (3) dealing with an explosion in the size of generated traces.

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 the novice programs tend to be simple.

Infinite Types Our implementation does check for infinite types inside narrow, but there are some degenerate cases where it is unable to detect them. Consider, the following buggy `replicate`

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

This code produces a nested list (with `n` levels of nesting) containing a single copy of `x`, instead of a list with `n` copies of `x`. OCaml detects a cyclic `'a = 'a list` constraint in the recursive call and throws a type error, whereas NanoMaLy happily produces the nested list. Strictly speaking, this function itself cannot "go wrong", the program would not get stuck until a *client* attempted to use the result expecting a flat list. But this is not very satisfying as `replicate` is clearly to blame. Furthermore, in our experience, infinite-type errors are often difficult to debug (and to explain to novices), so better support for this scenario would be useful.

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 [14], would likely help alleviate this issue, allowing users to highlight a confusing value and ask: "Where did this come from?"

5.4 The Advantage Of Traces

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 [18]. In particular, we illustrate, using a series of examples drawn from student programs in the AUTHORS 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 (underlined) and SHErrLoc (in bold),[2] and (4) the jump-compressed trace produced by NanoMaLy.

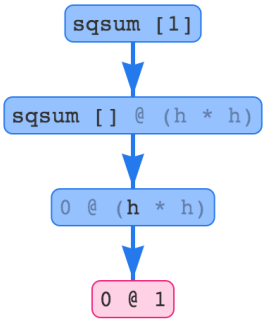
Example: Recursion with Bad Operator The recursive function `sqsum` should square each element of the input list and then compute the sum of the result.

```
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 `+` to compute the sum. Both OCaml and SHErrLoc blame the *wrong location*, namely the recursive call `sqsum t` with the message

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

NanoMaLy produces the following trace showing how the evaluation of `sqsum [1]` gets stuck:



The figure 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.

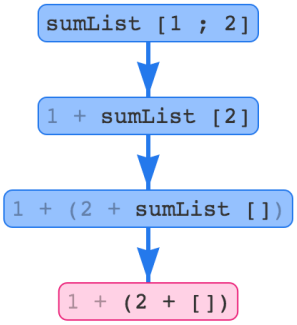
Example: Recursion with Bad Base Case The function `sumList` should add up the elements of its input list.

```
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 succinctly summarized by NanoMaLy's trace showing how `sumList [1; 2]` gets stuck at `2 + []`.



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

Example: Bad Helper Function that Type-Checks The function `digitsOfInt` is supposed to return a list of the digits of the input integer.

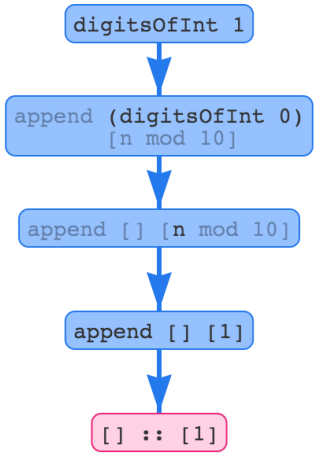
```
1  let append x xs =
2    match xs with
3    | [] -> [x]
4    | _  -> x :: xs
5
6  let rec digitsOfInt n =
7    if n <= 0 then
8      []
9    else
10     append (digitsOfInt (n / 10))
11            [n mod 10]
```

[2] When the locations from OCaml and SHErrLoc overlap, we just underline the relevant code.

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
```

NanoMaLy, in contrast, makes no asummptions about `append` and produces a trace that illustrates the true error on line 4, by highlighting the conflict in consing a list onto a list of integers.



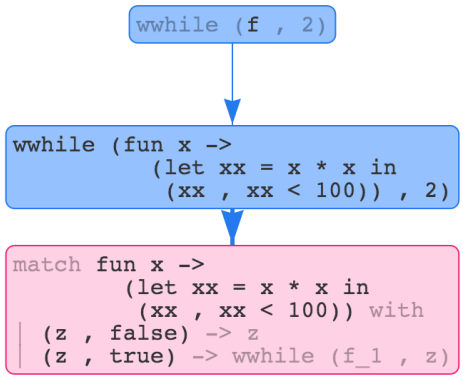
Example: Higher-Order Functions The higher-order function `wwhile` is supposed to emulate a traditional while-loop. It takes a function `f` and repeatedly calls `f` on the first element of its output pair, starting with the initial value `b`, until the second element is `false`.

```
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)
```

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

6. Related Work

In this section we connect our work to related efforts on type errors, testing, and program exploration.

Diagnosing and Repairing Type Errors It is well known that unification-based type inference procedures can produce poor error messages, and in particular, can misidentify the *source* of the type error. Thus, many groups have explored techniques to pinpoint the true source of the error and recommend fixes.

Lerner *et al.* [8] attempt to localize the type error and suggest a fix by replacing expressions (or removing them entirely) with alternatives based on the surrounding program context. Chen and Erwig [2] use a variational type system to allow for the possibility of changing an expression's type, and search for an expression whose type can be changed such that type inference would succeed. In contrast to Lerner, who searches for changes at the value-level, by searching at the type level Chen's search is complete due the finite universe of types in a program. Neubauer and Thiemann [11] 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. Zhang and Myers [18] present an algorithm for identifying the most likely culprit in a system of unsatisfiable constraints (*e.g.* type equalities), based on Bayesian reasoning. Pavlinovic *et al.* [13] translate the localization problem to a MaxSMT optimization problem, using compiler-provided weights to rank the possible sources.

In contrast to these approaches, we do not attempt to localize or fix the type error. Instead we try to explain it to the novice user using a dynamic witness that demonstrates how their program is not just ill-typed but truly wrong. In addition, allowing users to run their program (even knowing that it is wrong) enables experimentation and the use of debuggers to step through the program and investigate its evolution.

Running Ill-Typed Programs Vytiniotis *et al.* [17] 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.

Testing NanoMaLy is at its heart a test generator, and 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 [9, 10, 15], 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 [3, 4, 12] 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 [1, 6, 16] 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, and produces high coverage for our domain of novice programs.

Program Exploration Perera *et al.* [14] 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. Perera's system focuses on backward exploration; in contrast, our visualization supports forward *and* backward exploration, though our backward steps are more limited than Perera's. Specifically, we do not support selecting a value and inserting the intermediate terms that preceded it while ignoring unrelated computation steps; this would be interesting future work.

References

[1] 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, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855756.

[2] 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, pages 583–594, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535863.

[3] 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, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266.

[4] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, Sept. 2004. ISSN 1097-024X. doi: 10.1002/spe.602.

[5] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. The MIT Press, 2009.

[6] 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, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065036.

[7] B. Lerner, D. Grossman, and C. Chambers. Seminal: Searching for ML Type-error Messages. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 63–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: 10.1145/1159876.1159887.

[8] 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, pages 425–434, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250783.

[9] F. Lindblad. Property Directed Generation of First-Order Test Data. In M. T. Morazn, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007*, volume 8 of *Trends in Functional Programming*, pages 105–123. Intellect, 2007. ISBN 978-1-84150-196-3.

[10] M. Naylor and C. Runciman. Finding Inputs that Reach a Target Expression. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, 2007. SCAM 2007*, SCAM '07, pages 133–142, Sept. 2007. doi: 10.1109/SCAM.2007.30.

[11] 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, pages 15–26, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944708.

[12] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 75–84, May 2007. doi: 10.1109/ICSE.2007.37.

[13] 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, pages 525–542, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660230.

[14] 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, pages 365–376, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364579.

[15] 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, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411292.

[16] N. Tillmann and J. d. Halleux. Pex - White Box Test Generation for .NET. In B. Beckert and R. Hhnle, editors, *Tests and Proofs*, number 4966 in Lecture Notes in Computer Science, pages 134–153. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79123-2 978-3-540-79124-9. URL http://link.springer.com/chapter/10.1007/978-3-540-79124-9_10.

[17] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhes. Equality Proofs and Deferred Type Errors: A Compiler Pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364554.

[18] 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, pages 569–581, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535870.