Under consideration for publication in J. Functional Programming

Dynamic Witnesses for Static Type Errors∗ (or, Ill-Typed Programs Usually Go Wrong)

ERIC L. SEIDEL, RANJIT JHALA
University of California, San Diego, USA
WESTLEY WEIMER
University of Virginia, USA
(*e-mail:* eseidel@cs.ucsd.edu, jhala@cs.ucsd.edu, weimer@virginia.edu)

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 synthesize witness values that make the program go wrong. We prove that our procedure synthesizes general witnesses in that if a witness is found, then for all inhabited input types, there exist values that can make the function go wrong. Second, we show how to extend this procedure to produce a reduction graph that can be used to interactively visualize and debug witness executions. Third, we evaluate the coverage of our approach on two data sets comprising over 4,500 ill-typed student programs. Our technique is able to generate witnesses for around 85% of the programs, our reduction graph yields small counterexamples for over 80% of the witnesses, and a simple heuristic allows us to use witnesses to locate the source of type errors with around 70% accuracy. Finally, we evaluate whether our witnesses help students understand and fix type errors, and find that students presented with our witnesses show a greater understanding of type errors than those presented with a standard error message.

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:

```
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

∗ This work was supported by NSF grants CCF-1422471, CCF-1223850, CCF-1218344, CCF-1116289, CCF-0954024, Air Force grant FA8750-15-2-0075, and a gift from Microsoft Research.

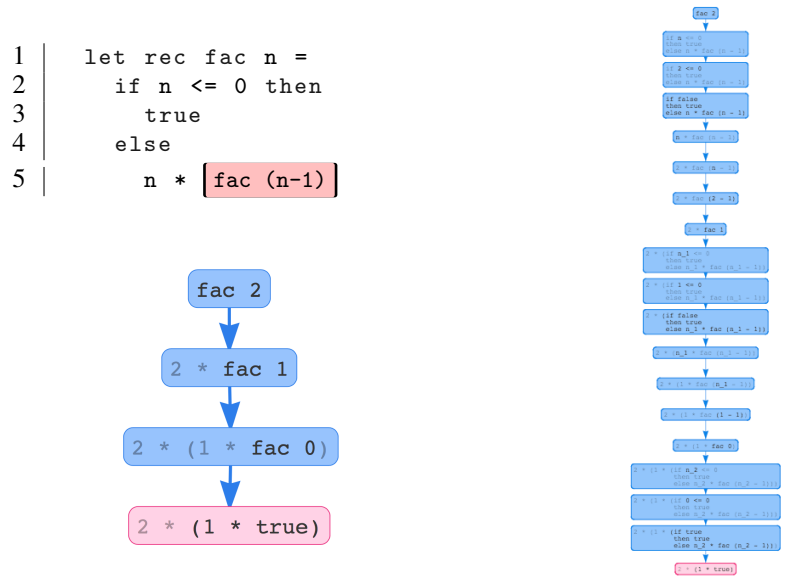


Fig. 1. (top-left) An ill-typed `fac` function highlighting the error location reported by OCaml. (bottom-left) Dynamically witnessing the type error in `fac`, showing only function call-return pairs. (right) The same trace, fully expanded to show each small-step reduction in the computation.

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 (Lerner *et al.*, 2007; Chen & Erwig, 2014; Zhang & Myers, 2014; Pavlinovic *et al.*, 2014), but an accurate source location 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). 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 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 (§ 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 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 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 (§ 5). The first is the UW data set (Lerner *et al.*, 2007) comprising 291 ill-typed programs. The second is a new UCSD data set, comprising 4,509 ill-typed programs. We show that for both data sets, 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 (§ 5.6). 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.

Contributions Relative to Prior Publications This paper extends our ICFP '16 paper of the same name (Seidel *et al.*, 2016), focusing on the experimental evaluation. First, in § 5.3 we investigate the student programs for which we were unable to synthesize a witness. We group the failures into five categories, give representative examples, and suggest ways to improve our feedback in these cases. Interestingly, we find that in the majority of these failed cases, the programs do not actually admit a witness in our semantics. Second, in § 5.7 we attempt to use our witnesses to localize type errors with a simple heuristic. We treat the stuck term as a sink for typing constraints, and the values it contains as sources of constraints. We can then predict that either the stuck term or one of the terms that *produced* a value it contains is likely at fault for the error. We compare our localizations to OCAML and two state-of-the-art type error localization tools, and find that we are competitive with the state of the art. Finally, we have also extended § 5.6 with an analysis of the statistical significance of our user study results.

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 (Runciman *et al.*, 2008). A hole — written $v[\alpha]$ — 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 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* 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 `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)
```

[1] With standard heuristics (Claessen & Hughes, 2000) to favor small values.

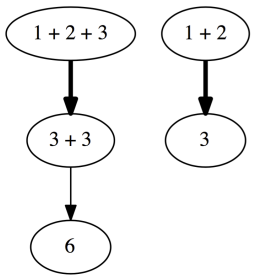


Fig. 2. The reduction graph for `1+2+3`, highlighting the edges produced by reducing `1+2+3` to `3+3`.

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.

2.2 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 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 § 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 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 (Felleisen *et al.*, 2009). Detailed proofs for the theorems in this section can be found in Appendix A.

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 **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 $v[\alpha]$, 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 $\texttt{node}[\alpha]\ v[\alpha]\ \texttt{leaf}[\alpha]\ \texttt{leaf}[\alpha]$ 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 (Runciman *et al.*, 2008)). 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,

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

Fig. 3. Syntax of λ^H

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

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

Fig. 4. Narrowing values

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 $v[\alpha]$). 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 $\langle \mathbf{e}_i, \sigma_i, \theta_i \rangle \hookrightarrow$

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.
$\mathsf{gen}(t_1 \times t_2, \theta)$	$\doteq$	$\langle \mathsf{gen}(t_1, \theta), \mathsf{gen}(t_2, \theta) \rangle$	
$\mathsf{gen}(\texttt{tree}\ t, \theta)$	$\doteq$	tr	non-det.
$\mathsf{gen}(\texttt{fun}, \theta)$	$\doteq$	$\lambda x.v[\alpha]$	v, α are fresh
$\mathsf{gen}(\alpha, \theta)$	$\doteq$	$v[\alpha]$	v is fresh

Fig. 5. Generating values

$\langle \mathbf{e}_{i+1}, \sigma_{i+1}, \theta_{i+1} \rangle$. 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 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.

3.3 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-term f, if $\langle f\ v[\alpha], \varnothing, \varnothing \rangle \hookrightarrow^{\tau} \langle \texttt{stuck}, \sigma, \theta \rangle$, then for every (inhabited[2]) 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 witnesses found by evaluation are indeed maximally general.

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

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

$$\textsc{Plus-G}\ \frac{\begin{array}{c}\langle n_1, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta) \\ \langle n_2, \sigma'', \theta'' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma', \theta') \\ n = n_1 + n_2\end{array}}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle C[n], \sigma'', \theta'' \rangle} \qquad \textsc{Plus-B1}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_1, \mathtt{int}, \sigma, \theta)}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle}$$

$$\textsc{Plus-B2}\ \frac{\langle \mathtt{stuck}, \sigma', \theta' \rangle = \mathsf{narrow}(v_2, \mathtt{int}, \sigma, \theta)}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma', \theta' \rangle} \qquad \textsc{If-G1}\ \frac{\langle \mathtt{true}, \sigma', \theta' \rangle = \mathsf{narrow}(v, \mathtt{bool}, \sigma, \theta)}{\langle C[\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2], \sigma, \theta \rangle \hookrightarrow \langle C[e_1], \sigma', \theta' \rangle}$$

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

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

$$\textsc{Leaf-G}\ \frac{\alpha \text{ is fresh}}{\langle C[\mathtt{leaf}], \sigma, \theta \rangle \hookrightarrow \langle C[\mathtt{leaf}[\alpha]], \sigma, \theta \rangle} \qquad \textsc{Node-G}\ \frac{\begin{array}{c}t = \mathsf{ty}(v_1) \\ \langle v_2', \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1) \\ \langle v_3', \sigma_3, \theta_3 \rangle = \mathsf{narrow}(v_3, \mathtt{tree}\ t, \sigma_2, \theta_2)\end{array}}{\langle C[\mathtt{node}\ v_1\ v_2\ v_3], \sigma, \theta \rangle \hookrightarrow \langle C[\mathtt{node}[t]\ v_1\ v_2'\ v_3'], \sigma_3, \theta_3 \rangle}$$

$$\textsc{Node-B1}\ \frac{\begin{array}{c}t = \mathsf{ty}(v_1) \\ \langle \mathtt{stuck}, \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1)\end{array}}{\langle C[\mathtt{node}\ v_1\ v_2\ v_3], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle} \qquad \textsc{Node-B2}\ \frac{\begin{array}{c}t = \mathsf{ty}(v_1) \\ \langle v_2', \sigma_2, \theta_2 \rangle = \mathsf{narrow}(v_2, \mathtt{tree}\ t, \sigma_1, \theta_1) \\ \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle = \mathsf{narrow}(v_3, \mathtt{tree}\ t, \sigma_2, \theta_2)\end{array}}{\langle C[\mathtt{node}\ v_1\ v_2\ v_3], \sigma, \theta \rangle \hookrightarrow \langle \mathtt{stuck}, \sigma_3, \theta_3 \rangle}$$

$$\textsc{Case-G1}\ \frac{\alpha \text{ is fresh} \qquad \langle \mathtt{leaf}[t], \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v, \mathtt{tree}\ \alpha, \sigma, \theta)}{\langle C\left[\mathtt{case}\ v\ \mathtt{of} \begin{cases} \mathtt{leaf} \to e_1 \\ \mathtt{node}\ x_1\ x_2\ x_3 \to e_2 \end{cases}\right], \sigma, \theta \rangle \hookrightarrow \langle C[e_1], \sigma_1, \theta_1 \rangle}$$

$$\textsc{Case-G2}\ \frac{\alpha \text{ is fresh} \qquad \langle \mathtt{node}[t]\ v_1\ v_2\ v_3, \sigma_1, \theta_1 \rangle = \mathsf{narrow}(v_1, \mathtt{tree}\ \alpha, \sigma, \theta)}{\langle C\left[\mathtt{case}\ v\ \mathtt{of} \begin{cases} \mathtt{leaf} \to e_1 \\ \mathtt{node}\ x_1\ x_2\ x_3 \to e_2 \end{cases}\right], \sigma, \theta \rangle \hookrightarrow \langle C[e_2[v_1/x_1][v_2/x_2][v_3/x_3]], \sigma_1, \theta_1 \rangle}$$

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

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

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

Fig. 6. Evaluation relation for λ^H

The Type of a Value The *dynamic type* of a value v is defined as a function $\mathsf{ty}(v)$ 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,

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

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

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

instead of $\texttt{int} \to \texttt{int}$. The types of tuples are obtained directly from their values, and the types of (polymorphic) trees from their labels. Note that the evaluation relation in Figure 6 guarantees that tree *values* will be annotated with their type. For nodes the type can be taken from the type of its value v_1, but for leaves the evaluation relation creates a new type hole α (this corresponds to polymorphic instantiation in a typechecker).

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 $\mathsf{ty}(v) \sim t$, 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 $t \preceq \mathsf{ty}(v)$, *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 unify, 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 $v[\alpha]$.

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 $v[\alpha]$. 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 $\rho_\tau(f) \preceq \sigma(v[\alpha])$, 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

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

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

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

$\langle f\ v, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \mathtt{stuck}, \sigma, \theta \rangle$ 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 $\tau = \langle f\ v[\alpha], \varnothing, \varnothing \rangle, \ldots, \langle \mathtt{stuck}, \sigma, \theta \rangle$. 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.

□

3.4 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

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

Fig. 8. Generating a saturated application.

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 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[3] 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) via the procedure

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

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

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 $e\ v_1[\alpha_1] \ldots v_n[\alpha_n]$ 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 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}.

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

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

Fig. 9. Generating witnesses.

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 1 (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 $w_1 : t_1\ \ldots\ w_n : t_n$ such that $\langle e\ w_1 \ldots w_n, \varnothing, \varnothing \rangle \hookrightarrow^* \langle \mathtt{stuck}, \sigma', \theta' \rangle$.

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'. □

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 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 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 transitions.

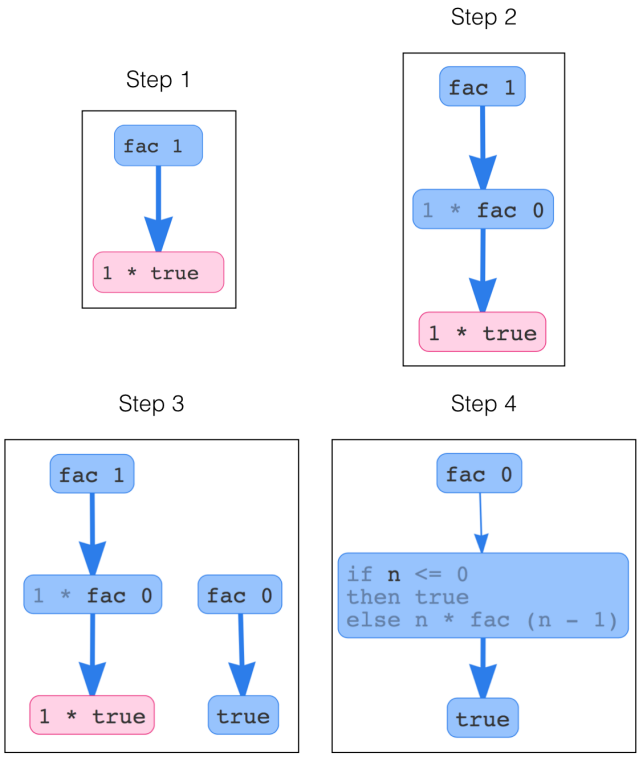


Fig. 10. A sequence of interactions with the trace of `fac 1`. The stuck term is red, in each node the redex is highlighted. Thick arrows denote a multi-step transition, thin arrows denote a single-step transition. We start in step 1. In step 2 we jump forward from the witness to the next function call. 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`.

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.

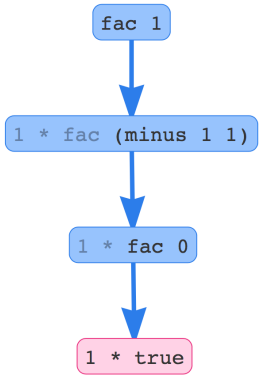


Fig. 11. Jump-compressed trace of `fac 1` with subtraction implemented as a function call.

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;
- 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 reduction 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 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 a purely functional subset of OCaml — with polymorphic types and records, but no modules, objects, or polymorphic variants — in a tool called NanoMaly. [4] 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.

[4] `https://github.com/ucsd-progsys/nanomaly`

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

1. ***Witness Coverage*** (§ 5.2, 5.3) How many ill-typed programs *admit* witnesses?
2. ***Witness Complexity*** (§ 5.4) How *complex* are the traces produced by the witnesses?
3. ***Witness Utility*** (§ 5.5, 5.6) How *helpful* are the witnesses in debugging type errors?
4. ***Witness-based Blame*** (§ 5.7) Can witnesses be used to *locate* the source of an error?

In the sequel we present our experimental methodology (§ 5.1) 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 (Loncaric *et al.*, 2016) and SHERRLOC (Zhang & Myers, 2014).

5.1 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 undergraduate Programming Languages (CSE 130) course at UC San Diego. We recorded each interaction with the OCAML top-level system over the course of the first three assignments (IRB #140608), from which we extracted 4,509 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 (Lerner *et al.*, 2006), 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.

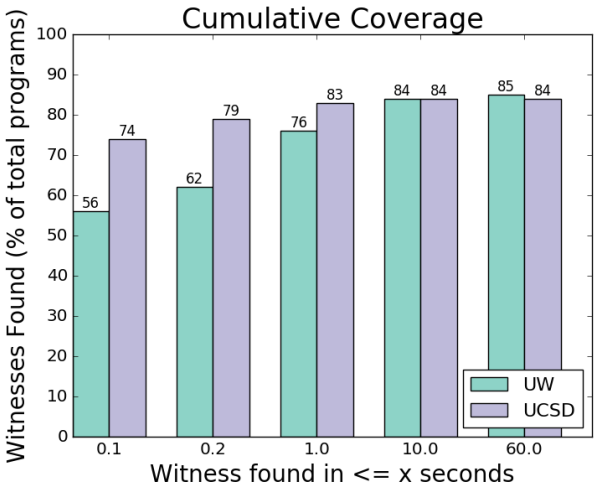


Fig. 12. Results of our coverage testing. Our random search successfully finds witnesses for 76–83% of the programs in under one second, improving to 84–85% in under 10 seconds.

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* (Lempsink, 2009) 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.

5.2 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 12 and 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

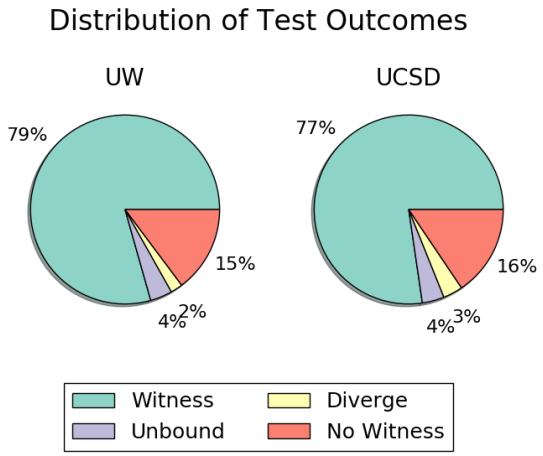


Fig. 13. Distribution of test outcomes. In both datasets we detect actual type errors at least 77% of the time, unbound variables or constructors 4% of the time, and diverging loops 2–3% of the time. For the remaining 15–16% of the programs we are unable to provide any useful feedback.

to a 10 second timeout, we reach 84% coverage, and if we allow a 60 second search, we hit a maximum of 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.

5.3 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 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 14 summarizes the results of our investigation — we note the classes that were based on the random sample with a "*". Note that the percentages referenced in Figure 14 (and in the sequel) are with respect to the total number of programs in the UCSD dataset, not only those were NanoMaLy failed to find a witness.

Ad-hoc Polymorphism We found that for 5% of all programs NanoMaLy got stuck when it tried to compare two holes. OCaml provides polymorphic equality and com-

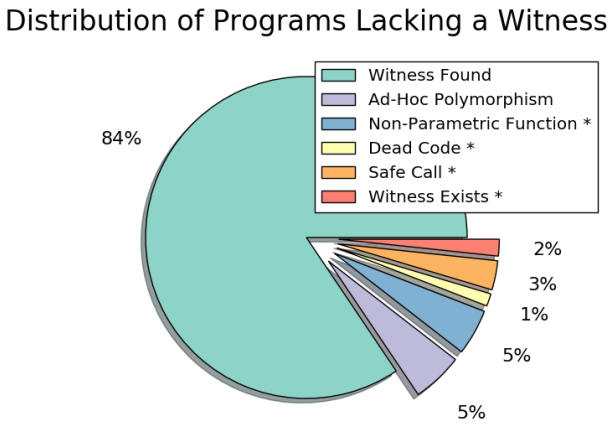


Fig. 14. Results of our investigation into programs where NANOMALY did not produce a witness. A "*" denotes that the percentage is an estimate based on a random sampling of 50 programs.

parison 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 `n <= 0` test in our `fac` example. The `<=` operator is polymorphic, but in this case we can make progress because the literal `0` is not. Suppose, however, we parameterized `fac` by a lower bound, *e.g.*

```
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 all 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; one 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 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.

The `fun` abstraction is also problematic when we have to generate new functions. Consider the following `pipe` function that composes a list of functions.

```
let pipe fs =
  let f a x = a x in
  let base a = a in
  List.fold_left f base fs
```

In the folding function `f`, the student has applied the accumulator `a` to the new function `x`, rather than composing the two. The OCaml compiler again detects a cyclic typing constraint and rejects the program, but NanoMaLy is unable to produce a witness. In this case the issue lies in the fact that the safety of a call to `pipe` is determined by its arguments: the call `pipe [(fun x -> 1)]` is safe, but the call `pipe [(fun x -> 1) ; (fun x -> 2); (fun x -> 3)]` will get stuck when we try to reduce `1 (fun x -> 3)`. Unfortunately, NanoMaLy is unable to synthesize such a witness because of the abstraction to `fun`. Specifically, this abstraction forces our hand inside `gen`; we do not know the what the input and output types of the function should be, so the only safe thing to do is to generate a function that accepts any input and returns a value of a yet-to-be-determined type. Thus, our lenient instantiation of holes prevents us from discovering a witness here.

Dead Code and "Safe" Function Calls * 4% of all 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 (Wheeler, 2014) or may accidentally bring it back to life (Seven, 2014). 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 all 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 majority of programs for which we fail to find a witness do not, in fact, admit a witness under NANOMALY's semantics. 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.

5.4 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.

Results The results of the experiment are summarized in Figure 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.

5.5 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 (Zhang & Myers, 2014). 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,

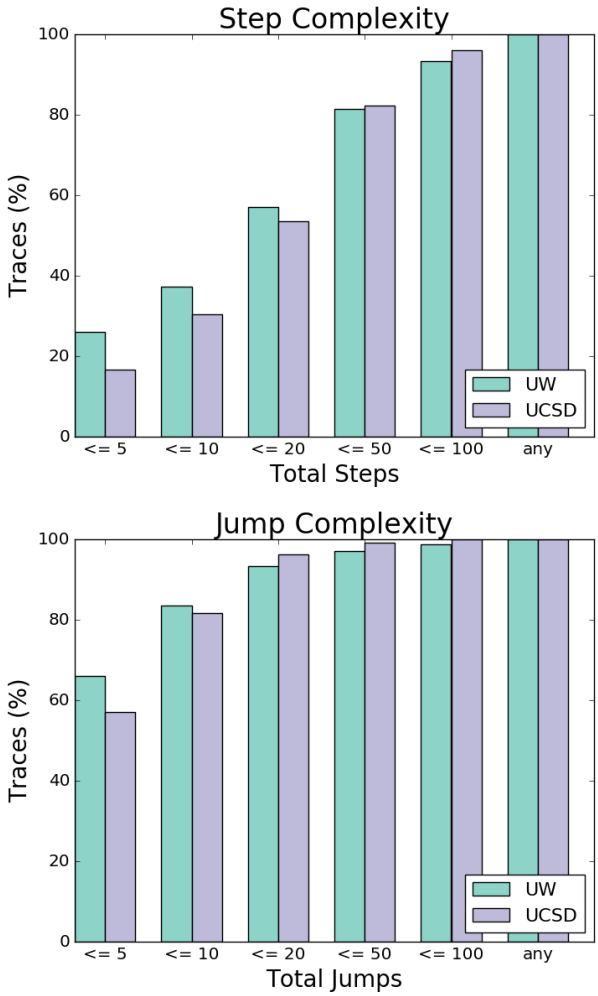


Fig. 15. Complexity of the generated traces. Over 80% of the combined traces have a jump complexity of at most 10, with an average complexity of 7 and a median of 5.

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.

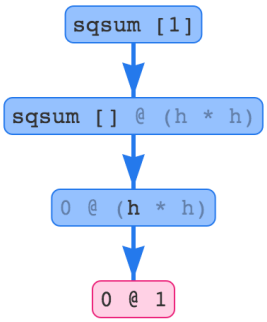
Example: Recursion with Bad Operator The recursive function `sqsum` should square each element of the input list and then compute the sum of the result.

```
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.

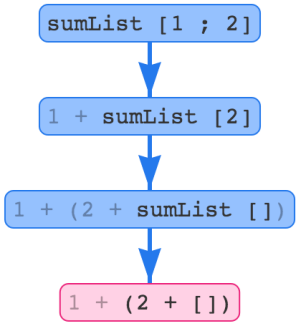
Example: Recursion with Bad Base Case The function `sumList` should add up the elements of its input list.

```
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.

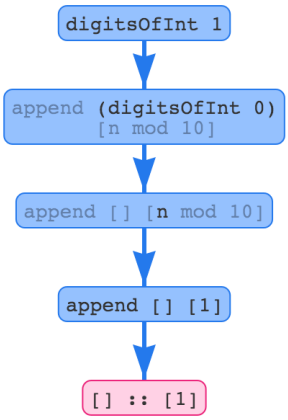
Example: Bad Helper Function that Type-Checks The function `digitsOfInt` should return a list of the digits of the input integer.

```
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.



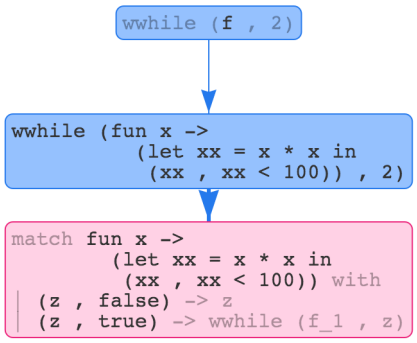
Example: Higher-Order Functions The higher-order function `wwhile` is supposed to emulate a traditional while-loop. It takes a function `f` and repeatedly calls `f` on the first element of its output pair, starting with the initial `b`, till the second element is `false`.

```
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 assumptions about function correctness, NanoMaLy focusses the user's attention on the piece of code that is actually relevant to the error.

5.6 Quantitative Evaluation of Witness Utility

We assigned four problems to the 60 students in the course: the `sumList`, `digitsOfInt`, and `wwhile` programs from § 5.5, 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. [5] As we had more than two annotators assigning nominal ("correct" or "incorrect") ratings we used Fleiss' kappa (Fleiss, 1971) 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 (Mann & Whitney, 1947) 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.

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 (Krippendorff, 2012), kappa values above 0.60 are often called "substantial" agreement (Landis & Koch, 1977). Figure 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 $p < 0.05$.

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. One such metric that is also surely relevant is time-to-completion, *i.e.* a good error report should *quickly* guide the student to a fix. Unfortunately, the in-class exam setting of our study did not admit the collection of timing data.

Furthermore, one might object to our selection of OCAML as the baseline comparison rather than SHERRLOC or MYCROFT, which also claim to produce more accurate error

[5] Measuring IRR is an established practice to account for potential bias among raters. The students were asked to explain the errors in English; judging whether they truly understood the errors involved a surprising amount of subjectivity, and is thus subject to rater bias.

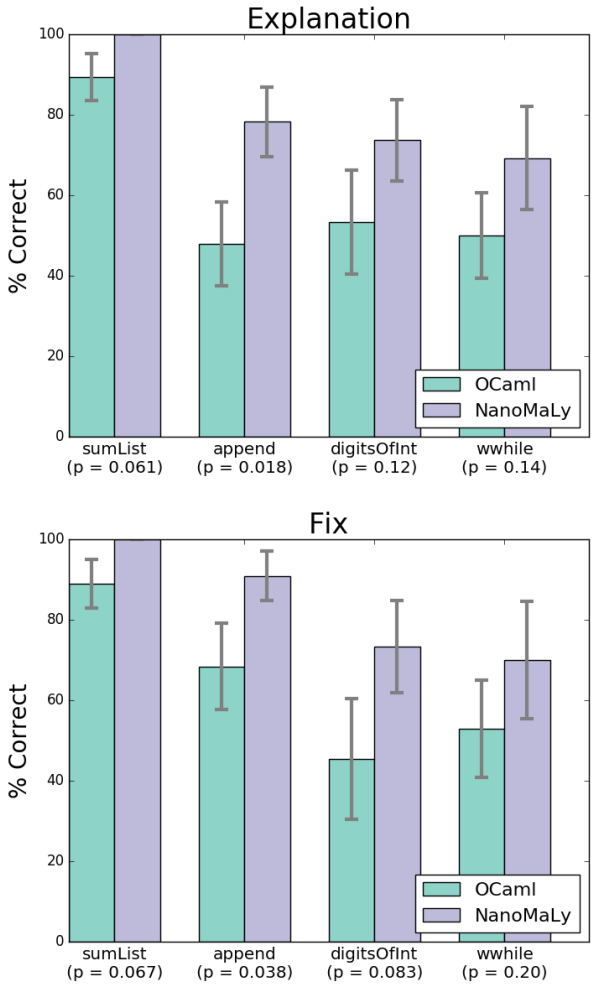


Fig. 16. A classification of students' explanations and fixes for type errors, given either OCaml's error message or NanoMaLy's jump-compressed trace. The students given NanoMaLy's jump-compressed trace consistently scored better ($\geq 10\%$) than those given OCaml's type error. We report the result of a one-sided Mann-Whitney U test for statistical significance in parentheses.

reports. This is indeed a limitation of our study, but we note that SHErrLoc blames the same expression as OCaml in both `wwhile` and `append`, and in `digitsOfInt` both blame the wrong function.

Finally, one might point out that our study investigates the use of NanoMaLy as a *debugging* aid rather than as a *teaching* aid. That is, it may be that NanoMaLy helps students solve their immediate problem, but does not help them build a lasting understanding of the type system. We have not attempted a longitudinal study of the long-term impact of using NanoMaLy, but we agree that it would be an interesting future direction.

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).

5.7 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 § 5.1, 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* (Lempsink, 2009) 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 § 5.5 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.

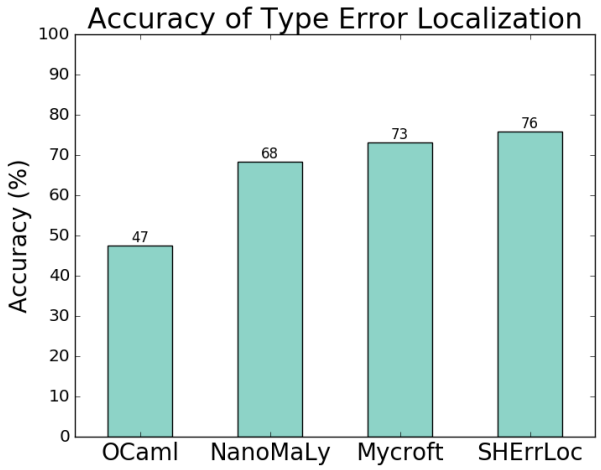


Fig. 17. Accuracy of type error localization. NANOMALY's witness-based predictions outperform OCAML by 21 points, and are competitive with the state-of-the-art tools MYCROFT and SHERRLOC.

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 (Zhang & Myers, 2014) 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 (Loncaric *et al.*, 2016) 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.

Results Figure 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.

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.

5.8 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. In the sequel we discuss a selection of drawbacks, and how we might address them in future work.

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). Our approach does not rely on random generation, we could easily substitute it for dynamic-symbolic execution by extending the evaluation relation to maintain a path condition and replacing the gen function with a call to the constraint solver. 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 (Perera *et al.*, 2012), 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 § 5.3 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 § 5.3, 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 overloads not only functions but also numeric literals, as well as strings and lists if one enables the respective language extensions. 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.

Traversal Bias A common problem with typecheckers is that the order in which the typechecker traverses the abstract syntax tree *biases* it in favor of blaming expressions that are seen later (McAdam, 1998). This usually takes the form of a left-to-right bias with respect to the source code (terms that appear later *textually* are more likely to be blamed), but in our case the bias is with respect to the execution trace.

Incorporating our notion of type sources from § 5.7 into the visualization, *e.g.* by including those reductions in the initial visualization, may help alleviate our bias in a similar manner to McAdam's proposal. Hage and Heeren (2009) offer another solution that allows the compiler author to selectively control the bias, and thus produce better errors, by prioritizing typing constraints. Unfortunately, due to NANOMALY's dynamic nature, providing this sort of control would likely require selectively changing the order of evalution; while sound for the pure subset of OCAML that we address, this could nonetheless confuse newcomers to the language even more.

Side Effects While our implementation does not currently support side effects, we believe it would not be difficult to add support. The search procedure is easily extended to support mutation by maintaining a store in the evaluation relation, the main complexity would be in extending the trace visualization to demonstrate mutation. Incorporating mutation directly into our reduction-based visualization would be difficult, as mutation would have non-local effects on the expressions and may be difficult for students to follow. Instead, we could follow the example of PYTHON TUTOR (Guo, 2013) and provide a separate visualization of the mutable store, with references visualized as constant pointers to changing objects.

6 Related Work

Localizing and Repairing Type Errors Many groups have explored techniques to improve the error locations reported by static type checkers. The traditional Damas-Milner type inference algorithm (Damas & Milner, 1982) reports the first program location where a type mismatch is discovered (subject to the traversal strategy (Lee & Yi, 1998)). As a result the error can be reported far away from its source (McAdam, 1998) without enough information to guide the user. Type-error slicing (Haack & Wells, 2003; Schilling, 2011; Rahli *et al.*, 2015; Sagonas *et al.*, 2013; Gast, 2004; Neubauer & Thiemann, 2003) recognizes this flaw and instead produces a slice of the program containing *all* program locations that are connected to the type error. Though the program slice must contain the source of the error, it can suffer from the opposite problem of providing *too much* information, motivating recent work in ranking the candidate locations. Zhang *et al.* (2014; 2015) present an algorithm for identifying the most likely culprit using Bayesian reasoning. Pavlinovic *et al.* (2014; 2015) translate the localization problem to a MaxSMT optimization problem, using compiler-provided weights to rank the possible sources. Loncaric *et al.* (2016) improve the scalability of Pavlinovic *et al.* by reusing the existing type checker as a theory solver in the Nelson-Oppen (1979) style, thus requiring only a MaxSAT solver.

In addition to localizing the error, Lerner *et al.* (2007) attempt to suggest a fix by replacing expressions (or removing them entirely) with alternatives based on the surrounding program context. Chen & Erwig (2014) 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 *et al.*, who search for changes at the value-level, Chen & Erwig search at the type-level and are thus complete due the finite universe of types used in the program.

In contrast to these approaches, we do not attempt to localize or fix the type error. Instead we try to explain it to the user using a dynamic witness that demonstrates how the 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.

Improving Error Messages The content and quality of the error messages themselves has also been studied extensively. Marceau *et al.* (2011b; 2011a) study the effectiveness of error messages in novice environments and present suggestions for improving their quality and consistency. Hage & Heeren (2006) identify a variety of general heuristics to improve the quality of type error messages, based on their teaching experience. Charguéraud (2014) presents a tabular format for type errors that can provide multiple explanations in a compact form. Heeren *et al.* (2003), Christiansen (2014), and Serrano & Hage (2016) provide methods for library authors to specialize type errors with domain-specific knowledge. The difference with our work is more pronounced here as we do not attempt to improve the quality of the error message, instead we search for a witness to the error and explain it with the resulting execution trace.

Running Ill-Typed Programs Vytiniotis *et al.* (2012) 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.* (2011) 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 (Runciman *et al.*, 2008; Naylor & Runciman, 2007; Lindblad, 2007), 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 (Claessen & Hughes, 2000; Csallner & Smaragdakis, 2004; Pacheco *et al.*, 2007) 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 (Godefroid *et al.*, 2005; Cadar *et al.*, 2008; Tillmann & de Halleux, 2008) 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 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 (Chamarthi *et al.*, 2011; Nguyen & Van Horn, 2015; Seidel *et al.*, 2015).

Program Exploration Flanagan *et al.* (1996) 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.

Clements *et al.* (2001) present a reduction-based visualization of program execution similar to NANOMALY's, though their interaction model is closer to that of a traditional step-debugger, limited to taking single step forwards or backwards. In contrast, NANOMALY first presents an overview of the whole computation, and then allows the user to focus in on the interesting reductions.

Perera *et al.* (2012) 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 *et al.* focus 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.

Acknowledgments

We thank Ethan Chan, Matthew Chan and Timothy Nguyen for assisting with our user study, and we thank the anonymous reviewers and Matthias Felleisen for their insightful feedback on earlier drafts of this paper.

References

Bayne, Michael, Cook, Richard, & Ernst, Michael D. (2011). Always-available static and dynamic feedback. *Pages 521–530 of: Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: ACM.

Cadar, Cristian, Dunbar, Daniel, & Engler, Dawson. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Pages 209–224 of: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. Berkeley, CA, USA: USENIX Association.

Chamarthi, Harsh Raju, Dillinger, Peter C., Kaufmann, Matt, & Manolios, Panagiotis. (2011). Integrating testing and interactive theorem proving. *Pages 4–19 of: Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Applications*. ACL2 '11.

Charguéraud, Arthur. (2014). Improving type error messages in ocaml. *Pages 80–97 of:* Proceedings *of the ML Family/OCaml Users and Developers Workshops*. Electronic Proceedings in Theoretical Computer Science, vol. 198. Open Publishing Association.

Chen, Sheng, & Erwig, Martin. (2014). Counter-factual typing for debugging type errors. *Pages 583–594 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Christiansen, David Raymond. (2014). Reflect on your mistakes! lightweight domain-specific error messages. *Preproceedings of the 15th Symposium on Trends in Functional Programming*.

Claessen, Koen, & Hughes, John. (2000). QuickCheck: A lightweight tool for random testing of haskell programs. *Pages 268–279 of: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM.

Clements, John, Flatt, Matthew, & Felleisen, Matthias. (2001). Modeling an algebraic stepper. *Pages 320–334 of: Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

Csallner, Christoph, & Smaragdakis, Yannis. (2004). JCrasher: an automatic robustness tester for java. *Softw. pract. exp.*, **34**(11), 1025–1050.

Damas, Luis, & Milner, Robin. (1982). Principal type-schemes for functional programs. *Pages 207–212 of: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. New York, NY, USA: ACM.

Felleisen, Matthias, Findler, Robert Bruce, & Flatt, Matthew. (2009). *Semantics engineering with PLT redex*. 1st edn. The MIT Press.

Flanagan, Cormac, Flatt, Matthew, Krishnamurthi, Shriram, Weirich, Stephanie, & Felleisen, Matthias. (1996). Catching bugs in the web of program invariants. *Pages 23–32 of: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI '96. New York, NY, USA: ACM.

Fleiss, Joseph L. (1971). Measuring nominal scale agreement among many raters. *Psychol. bull.*, **76**(5), 378.

Gast, Holger. (2004). Explaining ML type errors by data flows. *Pages 72–89 of: Implementation and Application of Functional Languages*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Godefroid, Patrice, Klarlund, Nils, & Sen, Koushik. (2005). DART: Directed automated random testing. *Pages 213–223 of: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. New York, NY, USA: ACM.

Guo, Philip J. (2013). Online Python Tutor: Embeddable web-based program visualization for CS education. *Pages 579–584 of: Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. New York, NY, USA: ACM.

Haack, Christian, & Wells, J B. (2003). Type error slicing in implicitly typed Higher-Order languages. *Pages 284–301 of: Programming Languages and Systems*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Hage, Jurriaan, & Heeren, Bastiaan. (2006). Heuristics for type error discovery and recovery. *Pages 199–216 of: Implementation and Application of Functional Languages*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Hage, Jurriaan, & Heeren, Bastiaan. (2009). Strategies for solving constraints in type and effect systems. *Electron. notes theor. comput. sci.*, **236**(2 Apr.), 163–183.

Heeren, Bastiaan, Hage, Jurriaan, & Swierstra, S Doaitse. (2003). Scripting the type inference process. *Pages 3–13 of: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, vol. 38. ACM.

Krippendorff, K. (2012). *Content analysis: An introduction to its methodology*. SAGE Publications.

Landis, J R, & Koch, G G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, **33**(1), 159–174.

Lee, Oukseh, & Yi, Kwangkeun. (1998). Proofs about a folklore let-polymorphic type inference algorithm. *Acm trans. program. lang. syst.*, **20**(4), 707–723.

Lempsink, Eelco. (2009). *Generic type-safe diff and patch for families of datatypes*. M.Phil. thesis, Universiteit Utrecht.

Lerner, Benjamin, Grossman, Dan, & Chambers, Craig. (2006). Seminal: Searching for ML type-error messages. *Pages 63–73 of: Proceedings of the 2006 Workshop on ML*. ML '06. New York, NY, USA: ACM.

Lerner, Benjamin S, Flower, Matthew, Grossman, Dan, & Chambers, Craig. (2007). Searching for type-error messages. *Pages 425–434 of: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. New York, NY, USA: ACM.

Lindblad, Fredrik. (2007). Property directed generation of First-Order test data. *Pages 105–123 of:* Morazán, Marco T (ed), *Proceedings of the Eighth Symposium on Trends in Functional Programming*. TFP '07, vol. 8.

Loncaric, Calvin, Chandra, Satish, Schlesinger, Cole, & Sridharan, Manu. (2016). A practical framework for type inference error explanation. *Pages 781–799 of: Proceedings of the 2016 ACM*

SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM.

Mann, H B, & Whitney, D R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Ann. math. stat.*, **18**(1), 50–60.

Marceau, Guillaume, Fisler, Kathi, & Krishnamurthi, Shriram. (2011a). Measuring the effectiveness of error messages designed for novice programmers. *Pages 499–504 of: Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. New York, NY, USA: ACM.

Marceau, Guillaume, Fisler, Kathi, & Krishnamurthi, Shriram. (2011b). Mind your language: On novices' interactions with error messages. *Pages 3–18 of: Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2011. New York, NY, USA: ACM.

McAdam, Bruce J. (1998). On the unification of substitutions in type inference. *Pages 137–152 of:* Hammond, Kevin, Davie, Tony, & Clack, Chris (eds), *Implementation of Functional Languages*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Naylor, M, & Runciman, Colin. (2007). Finding inputs that reach a target expression. *Pages 133–142 of: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM '07.

Nelson, Greg, & Oppen, Derek C. (1979). Simplification by cooperating decision procedures. *Acm trans. program. lang. syst.*, **1**(2), 245–257.

Neubauer, Matthias, & Thiemann, Peter. (2003). Discriminative sum types locate the source of type errors. *Pages 15–26 of: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM.

Nguyen, Phúc C, & Van Horn, David. (2015). Relatively complete counterexamples for higher-order programs. *Pages 446–456 of: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. New York, NY, USA: ACM.

Pacheco, Carlos, Lahiri, Shuvendu K, Ernst, Michael D, & Ball, Thomas. (2007). Feedback-Directed random test generation. *Pages 75–84 of: 29th International Conference on Software Engineering*. ICSE '07.

Pavlinovic, Zvonimir, King, Tim, & Wies, Thomas. (2014). Finding minimum type error sources. *Pages 525–542 of: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. New York, NY, USA: ACM.

Pavlinovic, Zvonimir, King, Tim, & Wies, Thomas. (2015). Practical SMT-based type error localization. *Pages 412–423 of: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: ACM.

Perera, Roly, Acar, Umut A, Cheney, James, & Levy, Paul Blain. (2012). Functional programs that explain their work. *Pages 365–376 of: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. New York, NY, USA: ACM.

Rahli, Vincent, Wells, Joe, Pirie, John, & Kamareddine, Fairouz. (2015). Skalpel: A type error slicer for standard ML. *Electron. notes theor. comput. sci.*, **312**(24 Apr.), 197–213.

Runciman, Colin, Naylor, Matthew, & Lindblad, Fredrik. (2008). Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. *Pages 37–48 of: Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. New York, NY, USA: ACM.

Sagonas, Konstantinos, Silva, Josep, & Tamarit, Salvador. (2013). Precise explanation of success typing errors. *Pages 33–42 of: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*. PEPM '13. New York, NY, USA: ACM.

Schilling, Thomas. (2011). Constraint-Free type error slicing. *Pages 1–16 of: Trends in Functional Programming*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Seidel, Eric L., Vazou, Niki, & Jhala, Ranjit. (2015). Type targeted testing. *Pages 812–836 of: Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems*. ESOP '15. New York, NY, USA: Springer-Verlag New York, Inc.

Seidel, Eric L, Jhala, Ranjit, & Weimer, Westley. (2016). Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *Pages 228–242 of: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP '16. ACM.

Serrano, Alejandro, & Hage, Jurriaan. (2016). Type error diagnosis for embedded DSLs by Two-Stage specialized type rules. *Pages 672–698 of: Programming Languages and Systems*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Seven, Doug. 2014 (17 Apr.). *Knightmare: A DevOps cautionary tale*. `https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/`. Accessed: 2017-4-24.

Tillmann, Nikolai, & de Halleux, Jonathan. (2008). Pex–White box test generation for .NET. *Pages 134–153 of:* Beckert, Bernhard, & Hähnle, Reiner (eds), *Tests and Proofs*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Vytiniotis, Dimitrios, Peyton Jones, Simon, & Magalhães, José Pedro. (2012). Equality proofs and deferred type errors: A compiler pearl. *Pages 341–352 of: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. New York, NY, USA: ACM.

Wheeler, David A. 2014 (23 Nov.). *The apple goto fail vulnerability: lessons learned*. `https://www.dwheeler.com/essays/apple-goto-fail.html`. Accessed: 2017-4-24.

Zhang, Danfeng, & Myers, Andrew C. (2014). Toward general diagnosis of static errors. *Pages 569–581 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Zhang, Danfeng, Myers, Andrew C, Vytiniotis, Dimitrios, & Peyton-Jones, Simon. (2015). Diagnosing type errors with class. *Pages 12–21 of: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. New York, NY, USA: ACM.

A Proofs for Section 3

Proof of Lemma 3
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 $\theta(\alpha) \preceq \sigma(\nu)$, then $\theta'(\alpha) \preceq \sigma'(\nu)$.

We can immediately discharge all of the E-*-BAD rules (except for NODE-B1) 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 $\mathsf{narrow}(\nu[\alpha], t, \sigma, \theta)$ and $\mathsf{narrow}(n, \texttt{int}, \sigma, \theta)$ cases. The $\mathsf{narrow}(n, \texttt{int}, \sigma, \theta)$ case is trivial as it does not change σ or θ. In the $\mathsf{narrow}(\nu[\alpha], t, \sigma, \theta)$ we will either find that $\nu \in \sigma$ 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 unify, thus in the $\nu[\in]\sigma$ we cannot actually refine either ν or α and thus the refinement is preserved. When we extend σ with a binding for ν, the call to unify 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 E-IF-GOOD{1,2}: Similar to PLUS-G.

Case APP-G: Similar to PLUS-G.

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

Case NODE-G: We narrow v_2 and v_3 to `tree` t, so we must consider three cases of narrow.

$\mathsf{narrow}(\nu[\alpha], t, \sigma, \theta)$**:** Similar to PLUS-G.

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

$\mathsf{narrow}(\texttt{node}[t_1]\ v_1\ v_2\ v_3, \texttt{tree}\ t_2, \sigma, \theta)$**:** Same as $\texttt{leaf}[t_1]$.

Case E-CASE-GOOD{1,2}: Similar to PLUS-G.

Case CASE-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$(n, \texttt{int}, \sigma, \theta)$ 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 $t' = \texttt{int}$. Let s be any concrete type that is incompatible with `int` and $\theta_s = \theta_{i-1}[\alpha \mapsto s]$, narrow$(\nu[\alpha], \texttt{int}, \sigma_{i-1}, \theta_s]) = \langle \texttt{stuck}, \sigma_{i-1}, \theta_s \rangle$.

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

Case E-IF-GOOD{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 LEAF-G: This case cannot apply as it does not update θ.

Case NODE-G: Here we narrow v_2 and v_3 to `tree` t, so we must consider three cases of narrow.

narrow$(\nu[\alpha], t, \sigma, \theta)$**:** Similar to PLUS-G.

narrow$(\texttt{leaf}[t_1], \texttt{tree}\ t_2, \sigma, \theta)$**:** 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$(\nu[\alpha], \texttt{int}, \sigma_{i-1}, \theta_s]) = \langle \texttt{stuck}, \sigma_{i-1}, \theta_s \rangle$.

narrow$(\texttt{node}[t_1]\ v_1\ v_2\ v_3, \texttt{tree}\ t_2, \sigma, \theta)$**:** Same as `leaf`$[t_1]$.

Case NODE-B1: This case cannot apply as narrow does not update θ whe it returns `stuck`.

Case NODE-B2: Similar to NODE-G.

Case E-CASE-GOOD{1,2}: Here we narrow v to `tree` α, so we must consider three cases of narrow.

narrow$(\nu[\alpha], t, \sigma, \theta)$**:** Similar to PLUS-G.

narrow$(\texttt{leaf}[t_1], \texttt{tree}\ t_2, \sigma, \theta)$**:** 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$(\texttt{node}[t_1]\ v_1\ v_2\ v_3, \texttt{tree}\ t_2, \sigma, \theta)$**:** Same as `leaf`$[t_1]$.

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

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

narrow$(\nu[\alpha], t, \sigma, \theta)$**:** 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 CASE-PAIR-B: This case cannot apply as narrow does not update θ whe it returns `stuck`.

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

□

B 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:

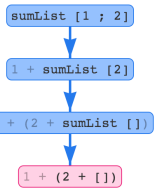
i. We start off with `fac 1`.

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 v0, ..., vn such that:

    - x is equal to v0
    - x' is equal to vn
    - for each i between 0 and n-2, we have
        (f vi) equals (vi+1, true)
    - (f vn-1) equals (vn, 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
```

```
wwhile (f , 2)
```

```
wwhile (fun x ->
          (let xx = x * x in
           (xx , xx < 100)) , 2)
```

```
match fun x ->
        (let xx = x * x in
         (xx , xx < 100)) with
  (z , false) -> z
  (z , true) -> wwhile (f_1 , z)
```

(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.

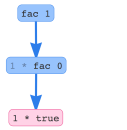
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

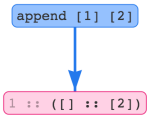
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

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 ) ;;
```



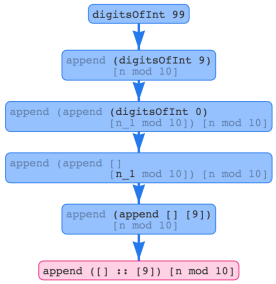

```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h ::  t ::  ys

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.