Dynamic Witnesses for Static Type Errors

(or, ill-typed programs usually go wrong)

Eric L. Seidel Ranjit Jhala
UC San Diego
{eseidel,jhala}@cs.ucsd.edu

Westley Weimer
University of Virginia
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 the above 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 88% of the programs, and our reduction graph yields small counterexamples for 81% of the witnesses. 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.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; D.3.4 [Programming Languages]: Processors—Debuggers; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages

Keywords debugging, testing, type errors

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 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 [4, 21, 29, 40], 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 NANO MALY, 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
is a user study that compares NANOMALY’s dynamic witnesses against OCAML’s type errors along the dimension of comprehensibility (§ 5.4). 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.

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 \texttt{fac} function from Figure 1. What types of inputs should we test \texttt{fac} with? Values of type \texttt{int} are fair game, but values of type, say, \texttt{string} or \texttt{int list} will cause the program to go wrong in an irrelevant manner. Concretely, we want to avoid testing \texttt{fac} with any type other than \texttt{int} because any other type would cause \texttt{fac} to get stuck immediately in the \texttt{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 [33]. A hole — written \( \nu[a] \) — is a placeholder for a value \( \nu \) of some unknown type \( \alpha \). 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 \texttt{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 \texttt{fst} function:

\[
\text{let } \texttt{fst} \ p = \text{match } \ p \text{ with } (a, b) \to a
\]

The case analysis tells us that \( p \) must be a pair, but it says \texttt{nothing} about the contents of the pair. Thus, upon reaching the case-analysis we would generate a pair containing fresh holes for the \texttt{fst} and \texttt{snd} component. Notice the similarity between instantiation of type variables and instantiation of holes. We can compute an approximate
type for \( \text{fst} \) by approximating the types of the (instantiated) input and output, which would give us:

\[
\text{fst} : (\alpha_1 * \alpha_2) \rightarrow \alpha_1
\]

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 \( \text{fac} \), given a hole as input we will narrow the hole to an \( \text{int} \) upon reaching the \( \leq \) test. At this point we choose a random \( \text{int} \) 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 \( \text{fac} \) takes a single argument instead of two (or none)? It is clear, syntactically, that \( \text{fac} \) takes at least one argument, but in a higher-order language with currying, syntax can be deceiving. Consider the following definition:

\[
\text{let incAllByOne = List.map (+ 1)}
\]

Is \( \text{incAllByOne} \) a function? If so, how many arguments does it take? The OCaml compiler deduces that \( \text{incAllByOne} \) takes a single argument because the type of \( \text{List.map} \) 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 an uninteresting sub-computation.

**Interacting with the Graph** The reduction graph is useful for formulating and executing traversals, but displaying it all at once would quickly become overwhelming. Our interaction begins by displaying a big-step reduction, i.e. the witness followed by the stuck term. The user can then progressively fill in the hidden steps of the computation by selecting a visible term and choosing one of the applicable traversal strategies — described in § 4 — to insert another term into the visualization.

---


---

**Jump-compressed Witnesses** It is rare for the initial state of the visualization to be informative enough to diagnose the error. Rather than abandon the user, we provide a short-cut to expand the witness to a jump-compressed trace, which contains every function call and return step. The jump-compressed trace abstracts the computation as a sequence of call-response pairs, providing a high-level overview of steps taken to reach the crash, and a high level of compression compared to the full trace. For example, the jump-compressed trace in Figure 1 contains 4 nodes compared to the 19 in the fully expanded trace. Our benchmark suite of student programs shows that jump-compression is practical, with an average jump-compressed trace size of 7 nodes and a median of 5.

### 3. Type-Error Witnesses

Next, we formalize the notion of type error witnesses as follows. First, we define a core calculus within which we will work (§ 3.1). Second, we develop a (non-deterministic) operational semantics 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 [9]. Detailed proofs for the theorems in this section can be found in Appendix A.

#### 3.1 Syntax

Figure 3 describes the syntax of \( \lambda^H \), 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 \( \texttt{stuck} \) 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 \( \alpha \). Intuitively, the type holes \( \alpha \) 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 [33]). We track how the holes are incrementally filled in, by using value (resp. type) substitutions \( \sigma \) (resp. \( \theta \)) that map value (resp. type) holes to values (resp. types).

The substitutions let us ensure that we consistently instantiate each hole with the same (partially defined) value or type, regardless of the multiple contexts in which the hole appears. This ensures we can report a concrete (and general) witness for any (dynamically) discovered type errors.

A normalized value substitution is one whose co-domain is comprised of normalized values. In the sequel, we will assume and ensure that all value substitutions are normalized. We ensure additionally that the co-domain of a substitution does not refer to any elements of its domain, i.e. when we extend a substitution with a new binding we apply the substitution to itself.

### 3.2 Semantics

Recall that our goal is to synthesize a value that demonstrates why (and how) a function goes wrong. We accomplish this by combining evaluation with type inference, giving us a form of dynamic type inference. Each primitive evaluation step tells us more about the types of the program values. For example, addition tells us that the addends must be integers, and an if-expression tells us the condition must be a boolean. When a hole appears in such a context, we know what type it must have in order to make progress and can fill it in with a concrete value.

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

#### Narrowing Types

The procedure

\[
\text{narrow} : v \times t \times x \times \theta \to \langle v \cup \text{stuck}, \sigma, \theta \rangle
\]

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 \( \sigma \) 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) \( \text{gen}(t, \theta) \) 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 \( \alpha \)) it yields a fresh hole constrained to have type \( \alpha \) (denoted by \( \nu[\alpha] \)). When generating a \( \text{tree} \) we must take care to ensure the resulting tree is well-typed. For a polymorphic type \( \text{tree} \alpha \) or \( \alpha \cdot \alpha \cdot \alpha \) we will place holes in the generated value; they will be lazily filled in later, on demand.

### 3.3 Generality

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

#### Theorem 1. [Witness Generality]

For any lambda-term \( f \), if \( \langle f, \nu[\alpha], \emptyset, \varnothing \rangle \sim^* \langle \text{stuck}, \sigma, \theta \rangle \), then for every (inhabited)\footnote{All types in our formalism are inhabited, but in a larger language like OCAML this may not be the case.} type \( t \) there exists a value \( v \) of type \( t \) such that \( \langle f, v, \emptyset, \varnothing \rangle \sim^* \langle \text{stuck}, \sigma, \theta \rangle \).

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.
values are defined in the natural manner. The types of functions are

The Type of a Value

Lemma 2. We prove two preservation lemmas. First, we show

preserving type compatibility.

A type \( v \) is compatible with \( t \) if the dynamic type of \( v \) is \( \tau \) is compatible with \( \theta \). Thus, we can use this type from an evaluation trace by applying the final type substitution to \( \alpha \). Formally, we say that if \( \{ f \, v[\alpha], \varnothing, \varnothing \} \rightsquigarrow^* \langle e, \sigma, \theta \rangle \), then the partial input type of \( f \) up to \( \tau \), written \( \rho_r(f) \), 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 \( \text{typeof}(v) \sim t \), that is, if the dynamic type of \( v \) is compatible with \( t \).

Type Refinement A type \( s \) is a refinement of a type \( t \), written \( s \subseteq t \), 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 \( t \leq v \), if \( t \leq \text{typeof}(v) \), i.e., if \( t \) is a refinement of the dynamic type of \( v \).

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

Lemma 2. If \( \tau \equiv \{ f \, v[\alpha], \varnothing, \varnothing \}, \ldots, \langle e, \sigma, \theta \rangle \) and \( \tau' \equiv \langle e', \sigma', \theta' \rangle \) (i.e., \( \tau' \) is a single-step extension of \( \tau \)) and \( \rho_r(f) \neq \rho_r(f) \) then \( \theta' = \theta[\alpha \mapsto t_1 \ldots \alpha_n \mapsto t_n] \).

Proof. By case analysis on the evaluation rules. \( \alpha \) does not change, so if the partial input types differ then \( \theta \neq \theta' \). Note that only narrow can change \( \theta \), via unify, which can only extend \( \theta \).

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 \( \tau \equiv \{ f \, v[\alpha], \varnothing, \varnothing \}, \ldots, \langle e, \sigma, \theta \rangle \), \( \rho_r(f) \leq \sigma(v[\alpha]) \).

Proof. By induction on \( \tau \). In the base case \( \tau = \{ f \, v[\alpha], \varnothing, \varnothing \} \) and \( \alpha \) is trivially a refinement of \( v[\alpha] \). In the inductive case, consider the single-step extension of \( \tau \), \( \tau' \equiv \langle e', \sigma', \theta' \rangle \). We show by case analysis on the evaluation rules that if \( \rho_r(f) \leq \sigma(v[\alpha]) \), then \( \rho_r(f) \leq \sigma'(v[\alpha]) \).

Incompatible Types Are Wrong For all types that are incompatible with the partial input type up to \( \tau \), there exists a value that will cause \( f \) to get stuck in at most \( k \) steps, where \( k \) is the length of \( \tau \).

Lemma 4. For all types \( t \), if \( \{ f \, v[\alpha], \varnothing, \varnothing \} \rightsquigarrow^* \langle e, \sigma, \theta \rangle \) and \( t \sim \rho_r(f) \), then there exists a \( v \) such that \( \text{typeof}(v) = t \) and \( \{ f \, v, \varnothing, \varnothing \} \rightsquigarrow^* \langle \text{stuck}, \sigma', \theta' \rangle \) in at most \( k \) steps, where \( k \) is the length of \( \tau \).

Proof. We can construct \( v \) from \( \tau \) as follows. Let

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

be the shortest prefix of \( \tau \) such that \( \rho_{\tau_{i-1}}(f) \approx \tau_i \). We will show that \( \rho_{\tau_{i-1}}(f) \) must contain some other hole \( \alpha' \) that is instantiated at step \( i \). Furthermore, \( \alpha' \) is instantiated in such a way that \( \rho_{\tau_{i}}(f) \approx \tau_{i+1} \). Finally, we will show that if we had instantiated \( \alpha' \) such that \( \rho_{\tau_{i}}(f) \approx \tau_{i+1} \), the current step would have gotten stuck.

By Lemma 2 we know that \( \theta_i = \theta_{i-1}[\alpha \mapsto t_1] \ldots [\alpha_n \mapsto t_n] \). We will assume, without loss of generality, that \( \theta_i = \theta_{i-1}[\alpha' \mapsto t'] \). Since \( \theta_{i-1} \) and \( \theta_i \) differ only in \( \alpha' \) 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 \( \beta \) be the above type such that \( \rho_{\tau_{i-1}}(f)[s/\alpha'] \). We show by case analysis on the evaluation rules that

\[ \langle e_{i-1}, \sigma_{i-1}, \theta_{i-1}[\alpha' \mapsto s] \rangle \rightsquigarrow^* \langle \text{stuck}, \sigma, \theta \rangle \]

Finally, by Lemma 3 we know that \( \rho_{\tau_{i-1}}(f) \not\leq \sigma_{i-1}(v[\alpha]) \) and thus \( \alpha' \in \sigma_{i-1}(v[\alpha]) \). Let

\[ u = \text{gen}(s, \theta) \]
\[ v = \sigma_{i-1}(v[\alpha]) \{ u/v'[\alpha'] \} \{ s/\alpha' \} \]

\[ \{ f \, v, \varnothing, \varnothing \} \rightsquigarrow^* \langle \text{stuck}, \sigma, \theta \rangle \]
Figure 6. Evaluation relation for $\lambda^H$
**Proof of Theorem 1.** Suppose \( \tau \) witnesses that \( f \) gets stuck, and let \( s = \rho_\tau(f) \). We show that all \( \tau \) types \( t \) have stuck-inducing values by splitting cases on whether \( t \) is compatible with \( s \).

Case \( s \sim t \): Let \( \tau = (f \; \nu[\alpha], \emptyset, \emptyset), \ldots, (\text{stuck}, \sigma, \theta) \). The value \( v = \sigma[\nu[\alpha]] \) demonstrates that \( f \; v \) gets stuck.

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

### 3.4 Search Algorithm

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

In order to convert the semantics into a procedure for finding witnesses, we must address two challenges. First, we must resolve the non-determinism introduced by \( \text{gen} \). 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 \( \text{GenWitness} \) 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 as output a list of (general) witness expressions, each of which is of the form \( e \; v_1 \ldots v_n \). The empty list is returned when no witness can be found after exploring \( k \) traces.

**Modeling Semantics** We resolve the non-determinism in the operational semantics (§ 3.2) via the procedure

\[
\text{eval} : e \to (w \cup \text{stuck}, \sigma, \theta)^* 
\]

Due to the non-determinism introduced by \( \text{gen} \), a call \( \text{eval}(e) \) returns a list of possible results of the form \( (w \cup \text{stuck}, \sigma, \theta) \) such that \( (e, \emptyset, \emptyset) \mapsto^* (w \cup \text{stuck}, \sigma, \theta) \).

**Currying** We address the issue of currying by defining a procedure \( \text{GenWitness} \), defined in Figure 8, that takes as input an expression \( e \) and produces a saturated expression of the form \( e \; \nu[\alpha_1] \ldots \nu[n] \) 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.

---

\(^3\)We assume, without loss of generality, that all traces are finite.
The visualization state looks like a set of linear lists of witnesses to the next function call. In step 3 we step into the recursive call, which spawns a new “thread” of execution. In step 4 we take a single step forward from \( \text{fac} \ 0 \).

**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 \( \langle e, \sigma, \theta, G \rangle \rightarrow \langle e', \sigma', \theta', G' \rangle \) where \( G' \) collects the edges of the transition.

**Collecting Edges** The general recipe for collecting steps-to edges is to record the consequent of each original rule in the trace. That is, each original judgment \( \langle e, \sigma, \theta \rangle \rightarrow \langle e', \sigma', \theta' \rangle \) becomes \( \langle e, \sigma, \theta, G \rangle \rightarrow \langle e', \sigma', \theta', e \rightarrow e', G \rangle \).

**4.2 Interactive Debugging**

Next, we show how to build a visual interactive debugger from the traced semantics, by describing the visualization state — i.e. what the user sees at any given moment — and the set of commands available to user and what they do.

**Visualization State** A visualization state is a directed graph whose vertices are expressions and whose edges are such that each vertex has at most one predecessor and at most one successor. In other words, the visualization state looks like a set of linear lists of expressions as shown in Figure 10. The initial state is the graph containing a single edge linking the initial and final expressions.

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

- **StepForward**, **StepBackward**: show the result of a single step forward or backward respectively.
- **StepInto**: show the result of stepping into a function call in a sub-term, isolating it in a new thread, and
- **StepOver**: show the result of skipping over a function call in a sub-term.

**Jump Compression** A jump compressed trace is one whose edges are limited to forward or backward jumps. In our experience, jump compression abstracts many details of the computation that are often uninteresting or irrelevant to the explanation. In particular, jump compressed traces hide low-level operations and summarize function calls as call-return pairs, see Figure 11 for a variant of \( \text{fac} \) 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 NANOMLY. We treat explicit type signatures, e.g. \((x : \text{int})\), as primitive operations that narrow the type of the wrapped value. In our implementation we instantiated \text{gen} with a simple random generation of values, which we will show suffices for the majority of type errors.

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

1. **Witness Coverage** How many ill-typed programs can we find witnesses for?
2. **Witness Complexity** How complex are the traces produced by the witnesses?
3. **Witness Utility** How helpful (qualitatively and quantitatively) are the witnesses and traces in debugging type errors?

**Benchmarks** 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 4,407 distinct, ill-typed OCaml programs. The second dataset — widely used in the literature — comes from a graduate-level course at the University of Washington [20], 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, polymorphic and algebraic data types.

We answer the third question in two steps. First, we present a qualitative evaluation of NanoMALy’s traces on a selection of programs drawn from the UCSD dataset. Second, we present a quantitative user study of students in the University of Virginia’s Spring 2016 undergraduate Programming Languages (CS 4501) course. As part of an exam, we presented the students with ill-typed OCaml programs and asked them to (1) explain the type error, and (2) fix the type error (IRB #2014090090). For each problem the students were given the ill-typed program and either OCaml’s error message or NanoMALy’s jump-compressed trace.

5.1 Witness Coverage
We ran our search algorithm on each program for 1,000 iterations, with the entry point set to the function that OCaml had identified as containing a type error. Due to the possibility of non-termination we set a timeout of 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 Figure 12. In both datasets our tool was able to find a witness for 83% of the programs in under one second, i.e. fast enough to be integrated as a compile-time check. If we extend our tolerance to a 10 second timeout, we hit a maximum of 87% coverage. Interestingly, the vast majority of witnesses corresponded to a type-error, as with the entry point set to the function that OCaml reported a type error and 2–3% triggered an infinite recursion error. For the remaining 12% of programs we were unable to provide any useful feedback as they either completed 1,000 tests successfully, or timed out after one minute. While a more advanced search procedure, e.g. dynamic-symbolic execution, could likely trigger more of the type errors, our experiments suggest that type errors are coarse enough (or that novice programs are simple enough) that these techniques are not necessary.

5.2 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

Results The results of the experiment are summarized in Figure 13. The average number of single-step reductions per trace is 31 for the UCSD dataset (35 for the UW dataset) with a maximum of 2,745 (986 for UW) and a median of 17 (also 17 for UW). The average number of jumps per trace is 7 (also 7 for UW) with a maximum of 353 (185 for UW) and a median of 4 (also 4 for UW). In both datasets 80% or more traces have at most 10 jumps.

5.3 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 [40]. 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 than blaming a single one, shows the steps necessary to reach the stuck state, and does not assume that a function is correct just because it type-checks. For each example we will present (1) the code, (2) the error message returned OCaml, (3) the error locations returned by OCaml (underlined) and SherrLOC (in bold), 4 and (4) the jump-compressed trace produced by NanoMALy.

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

\[
\begin{align*}
1 & \text{let rec sqsum xs = match xs with} \\
2 & | [] -> 0 \\
3 & | h::t -> (sqsum t) @ (h * h)
\end{align*}
\]

Unfortunately the student has used the list-append operator @ instead of + to compute the sum. 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.

\[
\begin{align*}
1 & \text{let rec sumList xs = match xs with} \\
2 & | [] -> [] \\
3 & | y::ys -> y + sumList ys
\end{align*}
\]

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

4 When the locations from OCaml and SherrLOC overlap, we just underline the relevant code.
Figure 12. Results of our coverage testing and the distribution of test outcomes. Our random search successfully finds witnesses for 79–85% of the programs in under one second, improving to 87% in under 10 seconds. In both datasets we detect actual type errors about 82% of the time, unbound variables or constructors 3–4% of the time, and diverging loops 2–3% of the time. For the remaining 11–12% of the programs we are unable to provide any useful feedback.

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

Figure 14. A classification of students’ explanations and fixes for type errors, given either OCAML’s error or NANOONLY’s jump-compressed trace. The students given NANOONLY’s jump-compressed trace consistently scored better (≥ 10%) than those given OCAML’s type error.
Both of the above are parts of the full story, which is summarized by NANO MALY’s trace showing how `sumList [1; 2]` gets stuck at `2 + []`.

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

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

```
let append x xs =  
  match xs with  
  | [] -> [x]  
  | _ -> x :: xs
```

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.

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

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

```
let _ = wwhile (f, 2)
```

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

**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`.

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

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.

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

```
let _ = wwhile (f, 2)
```

5.4 Quantitative Evaluation of Witness Utility

We assigned four problems to the (`n = 60`) students in the course: the `sumList`, `digitsOfInt`, and `wwhile` programs from § 5.3, as well as the following `append` program:

```
let append x l =  
  match x with  
  | [] -> l  
  | 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 message or NANO MALY’s jump-compressed trace. Due to the nature of an in-class exam, not every student answered every question; we received between 13 and 28 (out of a possible 30) responses for each problem-tool pair.

We then instructed four annotators (one of whom is an author, the other three are teaching assistants at UCSD) to classify the answers as correct or incorrect. We performed an inter-rater reliability (IRR) analysis to determine the degree to which the annotators consistently graded the exams. As we had more than two annotators assigning
nominal ("correct" or "incorrect") ratings we used Fleiss’ kappa [11] 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.

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

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

**Internal** We assigned students randomly to two groups. The first group was given OCAML’s errors for append and digitsOfInt, and NANO MALY’s trace for sumList and while. The second group was given the opposite assignment of errors and traces. This assignment ensured that (1) each student was given OCAML and NANO MALY problems, and (2) each student was given an “easy” and “hard” problem for both OCAML and NANO MALY. 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 is based on students in the process of learning OCAML, and thus may not generalize to all developers. The four programs we used 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 NANO MALY version of while) to 28 (for the OCAML version of sumList), out of a maximum of 30 per program-tool pair. 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 NANO MALY).

**Results** Figure 14 summarizes a single annotator’s results, which show that students given NANO MALY’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 NANO MALY responses were marked correct 10 – 30% more often than the OCAML responses, which suggests that the students who had access to NANO MALY’s traces had a better understanding of the type errors. 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 constitutes strong agreement [17], kappa values above 0.60 are often called “substantial” agreement [18].

5.5 Discussion

To summarize, our experiments demonstrate that NANO MALY finds witnesses to type errors: (1) with high coverage in a timespan amenable to compile-time analysis, (2) with traces that have a low average complexity of 7 jumps, and (3) that are more helpful to novice programmers than traditional type error messages. There are, of course, drawbacks to our approach. Four that stand out are: (1) coverage limits due to random generation, (2) the inability to handle certain instances of infinite types, (3) dealing with an explosion in the size of generated traces, and (4) handling ad-hoc polymorphism.

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

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

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

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

**Trace Explosion** Though the average complexity of our generated traces is low in terms of jumps, there are some extreme outliers. We cannot reasonably expect a novice user to explore a trace containing 50+ terms and draw a conclusion about which pieces contributed to the bug in their program. Enhancing our visualization to slice out program paths relevant to specific values [31], would likely help alleviate this issue, allowing users to highlight a confusing value and ask: “Where did this come from?”

**Ad-hoc Polymorphism** 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. For example, consider the \( n <= 0 \) test in our \( \text{fac} \) example. \( <= \) is polymorphic in OCAML, but in this case we can make progress because the literal 0 is not. If we parameterized \( \text{fac} \) by a lower bound, e.g.

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

and called \( \text{fac} \) with two holes, we would get stuck at the \( n <= m \) test; not because of a type error, but because all we know about \( n \) and \( m \) at that point is that they must have the same (unknown) type.

This issue is uncommon in OCAML (we did not detect a single instance of it across all of our benchmarks), but it would surely be exacerbated by a language like HASKELL, which makes heavy use of overloading. We suspect that dynamic-symbolic execution would allow us to handle ad-hoc polymorphism, but defer a proper treatment to future work.

6. Related Work

**Localizing and Repairing Type Errors** Many groups have explored techniques to pinpoint the true source of errors reported by static type checkers. The traditional Damas-Milner type inference algorithm [8] reports the first program location where a type
mismatch is discovered (subject to the traversal strategy [19]). As a result the error can be reported far away from its source [25] without enough information to guide the user. Type-error slicing [12, 14, 27, 32, 34, 35] 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. [40, 41] present an algorithm for identifying the most likely culprit using Bayesian reasoning. Pavlinovic et al. [29, 30] translate the localization problem to a MaxSMT optimization problem, using compiler-provided weights to rank the possible sources.

In addition to localizing the error, Lerner et al. [21] attempt to suggest a fix by replacing expressions (or removing them entirely) with alternatives based on the surrounding program context. Chen and Erwig [4] 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., we search for changes at the value-level. Chen et al. 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 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. [23, 24] study the effectiveness of error messages in novice environments and present suggestions for improving their quality and consistency. Hage and Heeren [15] identify a variety of general heuristics to improve the quality of type error messages, based on their teaching experience. Heeren et al. [16], Christiansen [5], and Serrano and Hage [37] 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 Vythiotis et al. [39] 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. [1] also provide a semantics for running ill-typed (JAVA) programs, but in contrast 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 NANO MALY 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 [22, 26, 33], 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 genera-

tion [6, 7, 28] 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 [2, 13, 38] combines symbolic execution (to track which path a given input triggers) with concrete execution (to ensure failures are not spurious). The system collects a path condition during execution, which tracks symbolically what conditions must be met to trigger the current path. Upon successfully completing a test run, it negates the path condition and queries a solver for another set of inputs that satisfy the negated path condition, i.e. inputs that will not trigger the same path. Thus, it can prune the search space much faster than techniques based on enumerating values, but is limited by the expressiveness of the underlying solver.

Our operational semantics is amenable to dynamic-symbolic execution, one would just need to collect the path condition and replace our implementation of gen by a call to the solver. We chose to use lazy, random generation instead because it is efficient, and the overhead of an external solver, and produces high coverage for our domain of novice programs.

A function’s type is a theorem about the function’s behavior. Thus, NANO MALY’s witnesses can be viewed as counter-examples, thereby connecting it to work on using test generation to find counter-examples prior to starting a proof [3, 36].

Program Exploration Flanagan et al. [10] 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, NANO MALY generates witnesses and allows the user to explore the resulting dynamic execution. Perera et al. [31] 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


A. Proofs for Section 3

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

We can immediately discharge all of the $E-s-BAD$ rules (except for $E-NODE-BAD$) as the calls to narrow return stuck. An examination of narrow shows that if narrow returns stuck then $\sigma$ and $\theta$ are unchanged.

Case E-PLUS-GOOD: We narrow $v_1$ and $v_2$ to $\mathsf{int}$, so we must consider the narrow($\nu[\alpha], t, \sigma, \theta$) and narrow($n, \mathsf{int}, \sigma, \theta$) cases. The narrow($n, \mathsf{int}, \sigma, \theta$) case is trivial as it does not change $\sigma$ or $\theta$. In the narrow($\nu[\alpha], t, \sigma, \theta$) we will either find that $\nu \in \sigma$ or we will generate a fresh $\mathsf{int}$ and extend $\sigma$. Note that when we extend $\sigma$ we also extend $\theta$ due to the call to unify, thus in the $\nu[\epsilon] \sigma$ we cannot actually refine either $\nu$ or $\alpha$ and thus the refinement is preserved. When we extend $\sigma$ with a binding for $\nu$, the call to unify ensures that we add a compatible binding for $\alpha$ if one was not already in $\theta$, thus the refinement relation must continue to hold.

Case E-IF-GOOD[1,2]: Similar to E-PLUS-GOOD.
Case E-APP-GOOD: Similar to E-PLUS-GOOD.
Case E-LEAF-GOOD: This step cannot change $\sigma$ or $\theta$ thus the refinement relation continues to hold trivially.

Case E-NODE-GOOD: We narrow $v_2$ and $v_3$ to $\mathsf{tree}$, so we must consider three cases of narrow.

Case E-CASE-PAIR-GOOD: Similar to E-PLUS-GOOD.

Proof of Lemma 4. We can construct $v$ from $\tau$ as follows. Let

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

be the shortest prefix of $\tau$ such that $p_{\tau_i}(f) \approx t$. We will show that $p_{\tau_{i+1}}(f)$ must contain some other hole $\alpha'$ that is instantiated at step $i$. Furthermore, $\alpha'$ is instantiated in such a way that $p_{\tau_i}(f) \approx t$. Finally, we will show that if we had instantiated $\alpha'$ such that $p_{\tau_i}(f) \approx t$, the current step would have gotten stuck.

Case E-PLUS-GOOD: Here we narrow $v_1$ and $v_2$ to $\mathsf{int}$, so the first case of narrow must apply (narrow($n, \mathsf{int}, \sigma, \theta$) cannot apply as it does not change $\theta$). In particular, since we extended $\theta_{i-1}$ with $[\alpha' \mapsto t]$ we know that $\alpha' = \alpha$ and $t = \mathsf{int}$. Let $s$ be any concrete type that is incompatible with $\mathsf{int}$ and $\theta_s = \theta_{i-1}[\alpha \mapsto s]$, narrow($\nu[\alpha], \mathsf{int}, \sigma_{i-1}, \theta_s$) = (stuck, $\sigma_{i-1}, \theta_s$).

Case E-PLUS-BAD[1,2]: These cases cannot apply as narrow does not update $\theta$ when it returns stuck.

Case E-IF-GOOD[1,2]: Similar to E-PLUS-GOOD.
Case E-IF-BAD: This case cannot apply as narrow does not update $\theta$ when it returns stuck.
Case E-APP-GOOD: Similar to E-PLUS-GOOD.
Case E-APP-BAD: This case cannot apply as narrow does not update $\theta$ when it returns stuck.
Case E-LEAF-GOOD: This case cannot apply as it does not update $\theta$.

Case E-NODE-GOOD: Here we narrow $v_2$ and $v_3$ to $\mathsf{tree}$, so we must consider three cases of narrow.

Case E-CASE-GOOD: Similar to E-PLUS-GOOD.

Case E-CASE-BAD: This case cannot apply as narrow does not update $\theta$ when it returns stuck.

Case E-APP-BAD: Similar to E-PLUS-GOOD.

Case E-APP-GOOD: Here we narrow $v_2$ and $v_3$ to $\mathsf{tree}$, so we must consider three cases of narrow.

Case E-LEAF-GOOD: This case cannot apply as it does not update $\theta$.

Case E-NODE-GOOD: Here we narrow $v_2$ and $v_3$ to $\mathsf{tree}$, so we must consider three cases of narrow.

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

Case E-CASE-BAD: This case cannot apply as narrow does not update $\theta$ when it returns stuck.

Case E-APP-BAD: This case cannot apply as narrow does not update $\theta$ when it returns stuck.