

Dynamic Witnesses for Static Type Errors

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

Eric L. Seidel Ranjit Jhala

UC San Diego
{eseidel,rjhala}@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 dynamically synthesize witness values that can make the program go wrong. We prove that our procedure synthesizes *general witnesses* in that if a witness is found, then *for all* 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 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.

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:

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

This message makes perfect sense to an expert who is familiar with the language and has a good mental model of how the type system works. However, it may perplex a novice who has yet to develop

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

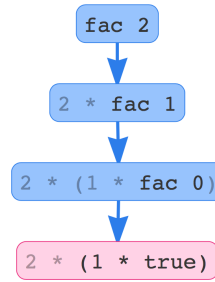


Figure 1. (top-left) An ill-typed `fac` function; (bottom-left) Dynamically witnessing the type error in `fac`, showing only function calls; (right) The same trace, fully expanded.

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 [3, 16, 24, 34], but an accurate source location still does not explain *why* the program is wrong.

In this paper we propose a new approach that explains static type errors by *dynamically* witnessing how an ill-typed program goes wrong. We have developed NANOMALY, an interactive tool that uses the source of the ill-typed function to automatically synthesize the result on the bottom-left in Figure 1, which shows how the recursive calls reduce to a configuration where the program “goes wrong” — *i.e.* the `int` value 1 is to be multiplied with the `bool` value `true`. We achieve this via three concrete contributions.

1. Finding Witnesses Our first contribution is an algorithm for searching for *witnesses* to type errors, *i.e.* inputs that cause a program to go wrong (§ 3). This problem is surprisingly tricky when we cannot rely on static type information. In particular, we must avoid the trap of *spurious* inputs that cause irrelevant problems that would be avoided by picking values of a different, relevant type. We solve this problem by developing a novel operational semantics that

combines evaluation and type inference. We execute the program with *holes* — values whose type is unknown — as the inputs. A hole remains abstract until the evaluation context tells us what type it must have, for example the parameters to an addition operation must both be integers. Our semantics *conservatively* instantiates holes with concrete values; thereby dynamically inferring the type of the input until the program goes wrong. We prove that our procedure synthesizes *general witnesses*, which means, intuitively, that if a witness is found for a given ill-typed function, then, *for all* (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 big steps to the next function call or return, for example the trace in the bottom-left of Figure 1 is a jump-compressed trace. Jump-compressed traces are similar to stack traces in that both show a sequence of function calls that lead to a crash, but the jump-compressed trace also shows the return values of successful calls, which can be useful in understanding why a particular path was taken.

3. Evaluating Witnesses Of course, the problem of finding witnesses is undecidable in 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 [16] comprising 284 ill-typed programs. The second is a new UCSD data set, comprising 4,407 ill-typed programs. We show that for both data sets, our technique is able to generate witnesses for nearly 90% of the programs, in under a second in the vast majority of cases. Furthermore, we show that a simple interactive strategy yields compact counterexample traces with at most 5 steps for 57% of the programs, and at most 10 steps for 81% of the programs.

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

2. Overview

We start with an overview of our approach to explaining (static) type errors using *witnesses* that (dynamically) show how the program

goes wrong. We illustrate why generating suitable inputs to functions is tricky in the absence of type information. Then we describe our solution to the problem and highlight the similarity to static type inference. Finally, we demonstrate our visualization of the synthesized witnesses.

2.1 Generating Witnesses

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

Problem: Which inputs are bad? One approach is to randomly generate input values and use them to execute the program until we find one that causes the program to go wrong. Unfortunately, this approach quickly runs aground. Recall the erroneous `fac` function from Figure 1. What *types* of inputs should we test `fac` with? Values of type `int` are fair game, but values of type, say, `string` or `int list` will cause the program to go wrong in an *irrelevant* manner. Concretely, we want to avoid testing `fac` with any type other than `int` because any other type would cause `fac` to get stuck immediately in the `n <= 0` test.

Solution: Don’t generate inputs until forced. Our solution is to avoid generating a concrete value for the input at all, until we can be sure of its type. The intuition is that we want to be as lenient as possible in our tests, so we make no assumptions about types until it becomes clear from the context what type an input must have. This is actually quite similar in spirit to type inference.

To defer input generation, we borrow the notion of a “hole” from SmallCheck [28]. A hole — written $\nu[\alpha]$ — is a *placeholder* for a value ν of some unknown type α . We leave all inputs as uninstantiated holes until they are demanded by the program, *e.g.* due to a primitive operation like the `<=` test.

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

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

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

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

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 `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

¹ With standard heuristics [5] to favor small values.

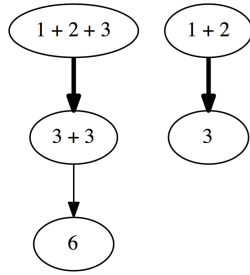


Figure 2. The reduction graph for $1+2+3$. The two edges produced by the transition from $1+2+3$ to $3+3$ are highlighted.

“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 `incAllByOne` takes a single argument instead of two (or none)? It is clear, syntactically, that `fac` takes *at least* one argument, but in a higher-order language with currying, syntax can be deceiving. Consider the following definition:

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

Is `incAllByOne` a function? If so, how many arguments does it take? The OCAML compiler deduces that `incAllByOne` takes a single argument because the *type* of `List.map` says it takes two arguments, and it is partially applied to `(+ 1)`. As we are dealing with ill-typed programs we do not have the luxury of typing information.

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

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 [8].

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 v 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 $\nu[\alpha]$, which represents an *unconstrained* value ν that may be replaced with *any* value of an unknown type α . Intuitively, the type holes α can be viewed as type variables that we will *not* generalize over. A *normalized* value is one that is not a hole, but which may internally contain holes. For example `node[α] $\nu[\alpha]$ leaf[α] leaf[α]` 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 [28]). We track how the holes are incrementally filled in, by using value (resp. type) *substitutions* σ (resp. θ) that map value (resp. type) holes to values (resp. types). The substitutions let us ensure that we consistently instantiate each hole with the same (partially defined) value or type, regardless of the multiple contexts in which the hole appears. This ensures we can report a concrete (and general) witness for any (dynamically) discovered type errors.

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

Expressions	$e ::= e \mid \text{stuck}$
	$e ::= v \mid x \mid ee \mid e + e$
	$\quad \mid \text{if } e \text{ then } e \text{ else } e$
	$\quad \mid \langle e, e \rangle \mid \text{case } e \text{ of } \langle x, x \rangle \rightarrow e$
	$\quad \mid \text{node } eee \mid \text{leaf}$
	$\quad \mid \text{case } e \text{ of } \begin{cases} \text{leaf} \rightarrow e \\ \text{node } xxx \rightarrow e \end{cases}$
Values	$v ::= n \mid b \mid \lambda x. e \mid \nu[\alpha] \mid tr$
	$tr ::= \text{node}[t] vvv \mid \text{leaf}[t]$
Integers	$n ::= 0, 1, -1, \dots$
Booleans	$b ::= \text{true} \mid \text{false}$
Types	$t ::= \text{bool} \mid \text{int} \mid \text{fun}$
	$\quad \mid t \times t \mid \text{tree } t \mid \alpha$
Substitutions	$\sigma ::= \emptyset \mid \sigma[\nu[\alpha] \mapsto v]$
	$\theta ::= \emptyset \mid \theta[\alpha \mapsto t]$
Contexts	$C ::= \bullet \mid Ce \mid vC$
	$\quad \mid C + e \mid v + C$
	$\quad \mid \text{if } C \text{ then } e \text{ else } e$
	$\quad \mid \langle C, e \rangle \mid \langle v, C \rangle$
	$\quad \mid \text{case } C \text{ of } \langle x, x \rangle \rightarrow e$
	$\quad \mid \text{node } Cee$
	$\quad \mid \text{node } vCe$
	$\quad \mid \text{node } vvC$
	$\quad \mid \text{case } C \text{ of } \begin{cases} \text{leaf} \rightarrow e \\ \text{node } xxx \rightarrow e \end{cases}$

Figure 3. Syntax of λ^H

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

$$\text{narrow} : v \times t \times \sigma \times \theta \rightarrow \langle v \cup \text{stuck}, \sigma, \theta \rangle$$

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 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) $\text{gen}(t, \theta)$ 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 α) it yields a fresh hole constrained to have type α (denoted by $\nu[\alpha]$). When generating a

tree t we must take care to ensure the resulting tree is well-typed. For a polymorphic type $\text{tree } \alpha$ 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 $\langle e, \sigma, \theta \rangle$ of an expression e or the stuck term stuck , a value substitution σ , and a type substitution θ . We write $\langle e, \sigma, \theta \rangle \hookrightarrow \langle e', \sigma', \theta' \rangle$ if the state $\langle e, \sigma, \theta \rangle$ transitions in a *single step* to $\langle e', \sigma', \theta' \rangle$. A (finite) *trace* τ is a sequence of configurations $\langle e_0, \sigma_0, \theta_0 \rangle, \dots, \langle e_n, \sigma_n, \theta_n \rangle$ such that $\forall 0 \leq i < n$, we have $\langle e_i, \sigma_i, \theta_i \rangle \hookrightarrow \langle e_{i+1}, \sigma_{i+1}, \theta_{i+1} \rangle$. We write $\langle e, \sigma, \theta \rangle \hookrightarrow^\tau \langle e', \sigma', \theta' \rangle$ if τ is a trace of the form $\langle e, \sigma, \theta \rangle, \dots, \langle e', \sigma', \theta' \rangle$. We write $\langle e, \sigma, \theta \rangle \hookrightarrow^* \langle e', \sigma', \theta' \rangle$ if $\langle e, \sigma, \theta \rangle \hookrightarrow^\tau \langle e', \sigma', \theta' \rangle$ 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 $\langle f \nu[\alpha], \emptyset, \emptyset \rangle \hookrightarrow^* \langle \text{stuck}, \sigma, \theta \rangle$, then *for every* concrete type t , we can find a value v of type t such that $f v$ goes wrong.

Theorem 1. [Witness Generality] *For any lambda-term f , if $\langle f \nu[\alpha], \emptyset, \emptyset \rangle \hookrightarrow^\tau \langle \text{stuck}, \sigma, \theta \rangle$, then for every (inhabited²) type t there exists a value v of type t such that $\langle f v, \emptyset, \emptyset \rangle \hookrightarrow^* \langle \text{stuck}, \sigma', \theta' \rangle$.*

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.

The Type of a Value The *dynamic type* of a value v is defined as a function $\text{typeof}(v)$ shown in Figure 7. The types of primitive values are defined in the natural manner. The types of functions are *approximated*, which is all that is needed to ensure an application does not get stuck. For example,

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

instead of `int` \rightarrow `int`. The types of (polymorphic) trees are obtained from the labels on their values, and the types of tuples directly from their values.

Dynamic Type Inference We can think of the evaluation of $f \nu[\alpha]$ as synthesizing a partial instantiation of α , and thus *dynamically inferring* a (partial) type for f ’s input. We can extract this type from an evaluation trace by applying the final type substitution to

²All types in our formalism are inhabited, but in a larger language like OCAML this may not be the case.

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

Figure 4. Narrowing values

gen	:	$t \times \theta \rightarrow v$	
gen(α, θ)	\doteq	gen($\theta(\alpha), \theta$)	if $\alpha \in \text{dom}(\theta)$
gen(int, θ)	\doteq	n	non-det.
gen(bool, θ)	\doteq	b	non-det.
gen($t_1 \times t_2, \theta$)	\doteq	$\langle \text{gen}(t_1, \theta), \text{gen}(t_2, \theta) \rangle$	
gen(tree t, θ)	\doteq	tr	non-det.
gen(fun, θ)	\doteq	$\lambda x.\nu[\alpha]$	ν, α are fresh
gen(α, θ)	\doteq	$\nu[\alpha]$	ν is fresh

Figure 5. Generating values

α . Formally, we say that if $\langle f \nu[\alpha], \emptyset, \emptyset \rangle \hookrightarrow^\tau \langle e, \sigma, \theta \rangle$, 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 $\text{typeof}(v) \sim t$, 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 $s \preceq t$, 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 $t \preceq \text{typeof}(v)$, 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 $\tau \doteq \langle f \nu[\alpha], \emptyset, \emptyset \rangle, \dots, \langle e, \sigma, \theta \rangle$ 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 $\theta' = \theta[\alpha_1 \mapsto t_1] \dots [\alpha_n \mapsto t_n]$.

Proof. By case analysis on the evaluation rules. α does not change, so if the partial input types differ then $\theta \neq \theta'$. Note that only narrow can change θ , 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 $\nu[\alpha]$.

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

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

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

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

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

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

be the shortest prefix of τ such that $\rho_{\tau_i}(f) \not\sim t$. 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 $\rho_{\tau_i}(f) \not\sim t$. 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 $\theta_i = \theta_{i-1}[\alpha_1 \mapsto t_1] \dots [\alpha_n \mapsto t_n]$. 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

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

Finally, by Lemma 3 we know that $\rho_{\tau_{i-1}}(f) \preceq \sigma_{i-1}(\nu[\alpha])$ and thus $\alpha' \in \sigma_{i-1}(\nu[\alpha])$. Let

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

$\langle f v, \emptyset, \emptyset \rangle \hookrightarrow^* \langle \text{stuck}, \sigma, \theta \rangle$ in i steps. ■

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

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

Case $s \not\sim t$: By Lemma 4, we can derive a v from τ 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

Evaluation

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

$$\begin{array}{c}
\text{E-PLUS-GOOD} \frac{\langle n_1, \sigma', \theta' \rangle = \text{narrows}(v_1, \text{int}, \sigma, \theta) \quad \langle n_2, \sigma'', \theta'' \rangle = \text{narrows}(v_2, \text{int}, \sigma', \theta') \quad n = n_1 + n_2}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle C[n], \sigma'', \theta'' \rangle} \quad \text{E-PLUS-BAD1} \frac{\langle \text{stuck}, \sigma', \theta' \rangle = \text{narrows}(v_1, \text{int}, \sigma, \theta)}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma', \theta' \rangle} \\
\\
\text{E-PLUS-BAD2} \frac{\langle \text{stuck}, \sigma', \theta' \rangle = \text{narrows}(v_2, \text{int}, \sigma, \theta)}{\langle C[v_1 + v_2], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma', \theta' \rangle} \quad \text{E-IF-GOOD1} \frac{\langle \text{true}, \sigma', \theta' \rangle = \text{narrows}(v, \text{bool}, \sigma, \theta)}{\langle C[\text{if } v \text{ then } e_1 \text{ else } e_2], \sigma, \theta \rangle \hookrightarrow \langle C[e_1], \sigma', \theta' \rangle} \\
\\
\text{E-IF-GOOD2} \frac{\langle \text{false}, \sigma', \theta' \rangle = \text{narrows}(v, \text{bool}, \sigma, \theta)}{\langle C[\text{if } v \text{ then } e_1 \text{ else } e_2], \sigma, \theta \rangle \hookrightarrow \langle C[e_2], \sigma', \theta' \rangle} \quad \text{E-IF-BAD} \frac{\langle \text{stuck}, \sigma', \theta' \rangle = \text{narrows}(v, \text{bool}, \sigma, \theta)}{\langle C[\text{if } v \text{ then } e_1 \text{ else } e_2], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma', \theta' \rangle} \\
\\
\text{E-APP-GOOD} \frac{\langle \lambda x. e, \sigma', \theta' \rangle = \text{narrows}(v_1, \text{fun}, \sigma, \theta)}{\langle C[v_1 \ v_2], \sigma, \theta \rangle \hookrightarrow \langle C[e[v_2/x]], \sigma', \theta' \rangle} \quad \text{E-APP-BAD} \frac{\langle \text{stuck}, \sigma', \theta' \rangle = \text{narrows}(v_1, \text{fun}, \sigma, \theta)}{\langle C[v_1 \ v_2], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma', \theta' \rangle} \\
\\
\text{E-LEAF-GOOD} \frac{\alpha \text{ is fresh}}{\langle C[\text{leaf}], \sigma, \theta \rangle \hookrightarrow \langle C[\text{leaf}[\alpha]], \sigma, \theta \rangle} \quad \text{E-NODE-GOOD} \frac{t = \text{typeof}(v_1) \quad \langle v'_2, \sigma_2, \theta_2 \rangle = \text{narrows}(v_2, \text{tree } t, \sigma_1, \theta_1) \quad \langle v'_3, \sigma_3, \theta_3 \rangle = \text{narrows}(v_3, \text{tree } t, \sigma_2, \theta_2)}{\langle C[\text{node } v_1 \ v_2 \ v_3], \sigma, \theta \rangle \hookrightarrow \langle C[\text{node}[t] \ v_1 \ v'_2 \ v'_3], \sigma_3, \theta_3 \rangle} \\
\\
\text{E-NODE-BAD1} \frac{t = \text{typeof}(v_1) \quad \langle \text{stuck}, \sigma_2, \theta_2 \rangle = \text{narrows}(v_2, \text{tree } t, \sigma_1, \theta_1)}{\langle C[\text{node } v_1 \ v_2 \ v_3], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma_3, \theta_3 \rangle} \quad \text{E-NODE-BAD2} \frac{t = \text{typeof}(v_1) \quad \langle v'_2, \sigma_2, \theta_2 \rangle = \text{narrows}(v_2, \text{tree } t, \sigma_1, \theta_1) \quad \langle \text{stuck}, \sigma_3, \theta_3 \rangle = \text{narrows}(v_3, \text{tree } t, \sigma_2, \theta_2)}{\langle C[\text{node } v_1 \ v_2 \ v_3], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma_3, \theta_3 \rangle} \\
\\
\text{E-CASE-GOOD1} \frac{\alpha \text{ is fresh} \quad \langle \text{leaf}[t], \sigma_1, \theta_1 \rangle = \text{narrows}(v, \text{tree } \alpha, \sigma, \theta)}{\langle C \left[\text{case } v \text{ of } \begin{cases} \text{leaf} \rightarrow e_1 \\ \text{node } x_1 \ x_2 \ x_3 \rightarrow e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle C[e_1], \sigma_1, \theta_1 \rangle} \\
\\
\text{E-CASE-GOOD2} \frac{\alpha \text{ is fresh} \quad \langle \text{node}[t] \ v_1 \ v_2 \ v_3, \sigma_1, \theta_1 \rangle = \text{narrows}(v_1, \text{tree } \alpha, \sigma, \theta)}{\langle C \left[\text{case } v \text{ of } \begin{cases} \text{leaf} \rightarrow e_1 \\ \text{node } x_1 \ x_2 \ x_3 \rightarrow e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle C[e_2[v_1/x_1][v_2/x_2][v_3/x_3]], \sigma_1, \theta_1 \rangle} \\
\\
\text{E-CASE-BAD} \frac{\alpha \text{ is fresh} \quad \langle \text{stuck}, \sigma_1, \theta_1 \rangle = \text{narrows}(v, \text{tree } \alpha, \sigma, \theta)}{\langle C \left[\text{case } v \text{ of } \begin{cases} \text{leaf} \rightarrow e_1 \\ \text{node } x_1 \ x_2 \ x_3 \rightarrow e_2 \end{cases} \right], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma_1, \theta_1 \rangle} \\
\\
\text{E-CASE-PAIR-GOOD} \frac{\alpha_1, \alpha_2 \text{ are fresh} \quad \langle \langle v_1, v_2 \rangle, \sigma_1, \theta_1 \rangle = \text{narrows}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C[\text{case } v \text{ of } \langle x_1, x_2 \rangle \rightarrow e], \sigma, \theta \rangle \hookrightarrow \langle C[e[v_1/x_1][v_2/x_2]], \sigma_1, \theta_1 \rangle} \\
\\
\text{E-CASE-PAIR-BAD} \frac{\alpha_1, \alpha_2 \text{ are fresh} \quad \langle \text{stuck}, \sigma_1, \theta_1 \rangle = \text{narrows}(v, \alpha_1 \times \alpha_2, \sigma, \theta)}{\langle C[\text{case } v \text{ of } \langle x_1, x_2 \rangle \rightarrow e], \sigma, \theta \rangle \hookrightarrow \langle \text{stuck}, \sigma_1, \theta_1 \rangle}
\end{array}$$

Figure 6. Evaluation relation for λ^H

$\text{typeof}(n)$	\doteq	int
$\text{typeof}(b)$	\doteq	bool
$\text{typeof}(\lambda x.e)$	\doteq	fun
$\text{typeof}(\langle v_1, v_2 \rangle)$	\doteq	$\text{typeof}(v_1) \times \text{typeof}(v_2)$
$\text{typeof}(\text{leaf}[t])$	\doteq	$\text{tree } t$
$\text{typeof}(\text{node}[t] \ v_1 \ v_2 \ v_3)$	\doteq	$\text{tree } t$
$\text{typeof}(\nu[\alpha])$	\doteq	α

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

Saturate	$:$	$e \rightarrow e$
$\text{Saturate}(e)$	$=$	$\text{case eval}(e) \text{ of}$
$\langle \lambda x.e, \sigma, \theta \rangle, \dots$	\rightarrow	$\text{Saturate}(e \ \nu[\alpha]) \quad (\nu, \alpha \text{ are fresh})$
$-$	\rightarrow	e

Figure 8. Generating a saturated application.

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 explore³ 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 $e \ v_1 \dots v_n$. 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

$$\text{eval} : e \rightarrow \langle v \cup \text{stuck}, \sigma, \theta \rangle^*$$

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

Currying We address the issue of currying by defining a procedure `Saturate(e)`, defined in Figure 8, that takes as input an expression e and produces a *saturated* expression of the form $e \ \nu_1[\alpha_1] \dots \nu_n[\alpha_n]$ that *does not* evaluate to a lambda. This is achieved with a simple loop that keeps adding holes to the target application until evaluating the term yields a non-lambda value.

Generating Witnesses Finally, Figure 9 summarizes the overall implementation of our search for witnesses with the procedure `GenWitness(k, e)`, which takes as input a bound k and the target expression e , and returns a list of witness expressions $e \ v_1 \dots v_n$ that demonstrate how the input program gets stuck. The search proceeds as follows.

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

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

GenWitness	$:$	$\text{Nat} \times e \rightarrow e^*$
$\text{GenWitness}(n, e)$	$=$	$\{\sigma(e_{\text{sat}}) \mid \sigma \in \Sigma\}$
where		
e_{sat}	$=$	$\text{Saturate}(e)$ (1)
res	$=$	$\text{take}(n, \text{eval}(e_{\text{sat}}))$ (2)
Σ	$=$	$\{\sigma \mid \langle \text{stuck}, \sigma, \theta \rangle \in \text{res}\}$ (3)

Figure 9. Generating witnesses.

We obtain the following corollary of Theorem 1:

Corollary. [Witness Generation] If

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

then for all types $t_1 \dots t_n$ there exist values $w_1 \dots w_n$ such that $\langle e \ w_1 \dots w_n, \emptyset, \emptyset \rangle \hookrightarrow^ \langle \text{stuck}, \sigma', \theta' \rangle$.*

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

4. Explaining Type Errors With Traces

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

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

4.1 Tracing Semantics

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

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

Tracing Semantics We extend the transition relation (§ 3.2) to collect the set of edges corresponding to the reduction graph. Concretely, we extend the operational semantics to a relation of the form $\langle e, \sigma, \theta, G \rangle \hookrightarrow \langle e', \sigma', \theta', G' \rangle$ where G' collects the edges of the transition.

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

4.2 Interactive Debugging

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

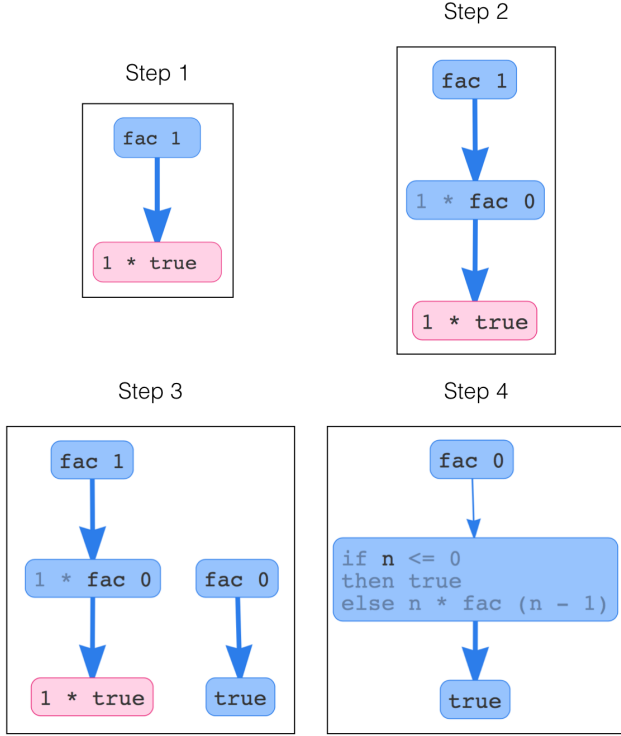


Figure 10. A sequence of interactions with the trace of `fac 1`. The stuck term is red, in each node the redex is highlighted. Thick arrows denote a multi-step transition, thin arrows denote a single-step transition. We start in step 1. In step 2 we jump forward from the witness to the next function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

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

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

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

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

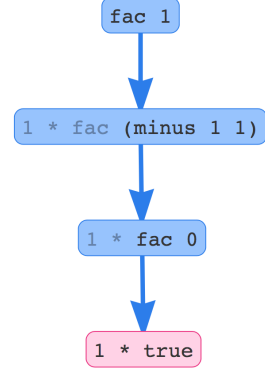


Figure 11. Jump-compressed trace of `fac 1` with subtraction implemented as a function call.

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. We treat explicit type signatures, *e.g.* `(x : int)`, as primitive operations that narrow the type of the wrapped value. In our implementation we instantiated `gen` with a simple random generation of values, which we will show suffices for the majority of type errors.

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

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

Benchmarks We answer these questions on two sets of ill-typed programs, *i.e.* programs that were rejected by the OCAML compiler because of a type error. The first dataset comes from the Spring 2014 undergraduate Programming Languages course at 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 [15], 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 data types, and expression evaluators.

5.1 Witness Coverage

We ran our search algorithm on each program for 1,000 iterations, with the entry point set to the function that OCAML had identified as containing a type error. Due to the possibility of non-termination we set a timeout of 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, while the vast majority of witnesses corresponded to a type-error, as expected, 3–4% triggered an unbound variable error (even though OCAML reported a type error) and 2–3% triggered an infinite recursion error. For the remaining 12% of programs we were unable to provide any useful feedback as they either completed 1,000 tests successfully, or timed out after one minute. While a more advanced search procedure, *e.g.* dynamic-symbolic execution, could likely trigger more of the type errors, our experiments show that type errors are coarse enough (or that novice programs are *simple* enough) that these techniques are not necessary.

5.2 Trace Complexity

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

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

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 The Advantage of Traces

Next, we present a *qualitative* evaluation that compares the explanations provided by NANOMALY’s dynamic witnesses with the static reports produced by the OCAML compiler and SHERRLOC, a state-of-the-art fault localization approach [34]. 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 by 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 Operator The recursive function `sqsum` should square each element of the input list and then compute the sum of the result.

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

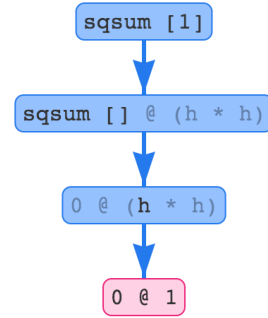
Unfortunately the student has used the list-append operator `@` instead of `+` to compute the sum. Both OCAML and SHERRLOC blame

⁴When the locations from OCAML and SHERRLOC overlap, we just underline the relevant code.

the *wrong location*, namely the recursive call `sqsum t` with the message

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

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



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

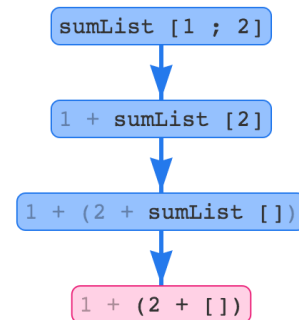
Example: Recursion with Bad Base Case The function `sumList` should add up the elements of its input list.

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

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

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

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



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

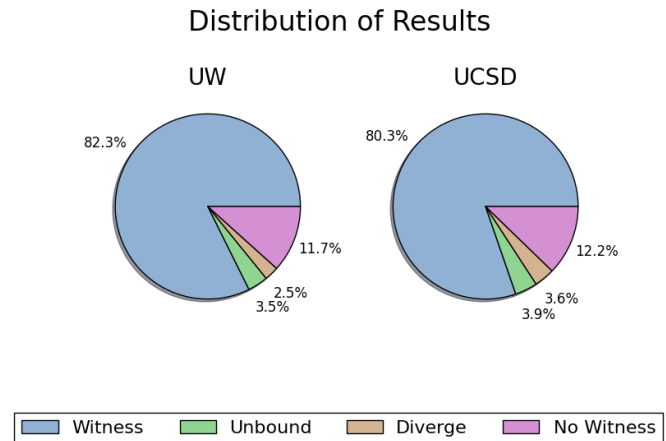
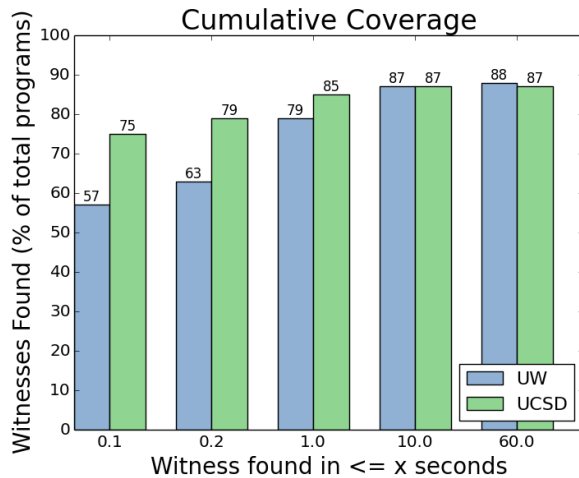


Figure 12. Results of our coverage testing and the distribution of test outcomes. Our random search successfully finds witnesses for 79–85% of the programs in under one second, improving slightly to 87% in under 10 seconds. In both datasets we detect actual type errors about 82% of the time, unbound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

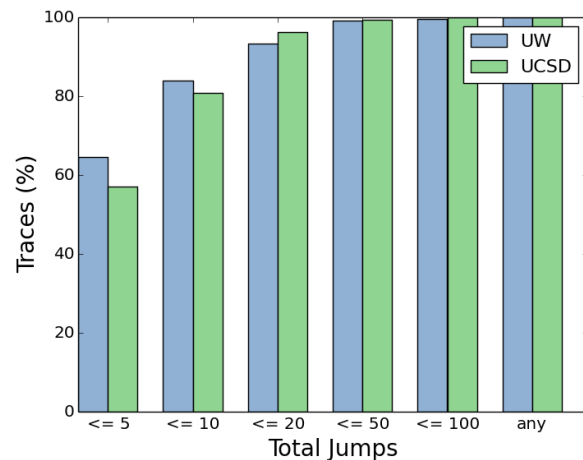
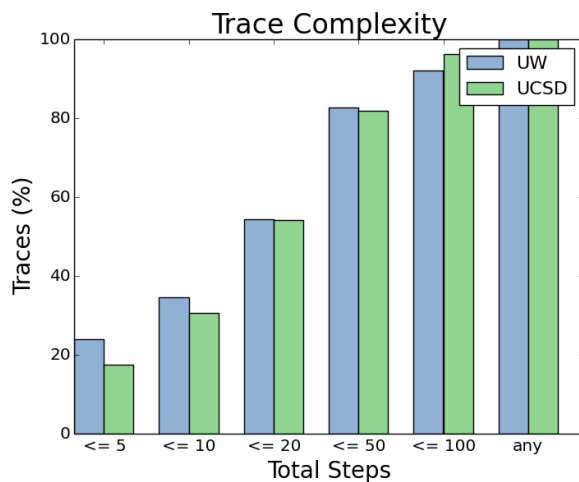


Figure 13. Complexity of the generated traces. 81% of the combined traces have a jump complexity of at most 10, with an average complexity of 7 and a median of 5.

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

```

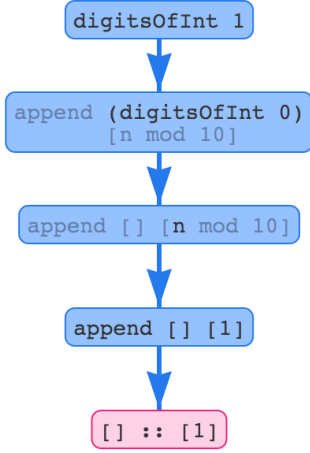
1  let append x xs =
2    match xs with
3    | [] -> [x]
4    | _  -> x :: xs
5
6  let rec digitsOfInt n =
7    if n <= 0 then
8      []
9    else
10     append (digitsOfInt (n / 10))
11            [n mod 10]
```

Unfortunately, the student’s `append` function *conses* an element onto a list instead of appending two lists. Though incorrect, `append`

still type-checks and thus OCAML and SHERRLOC blame the *use-site* on line 10.

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

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



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

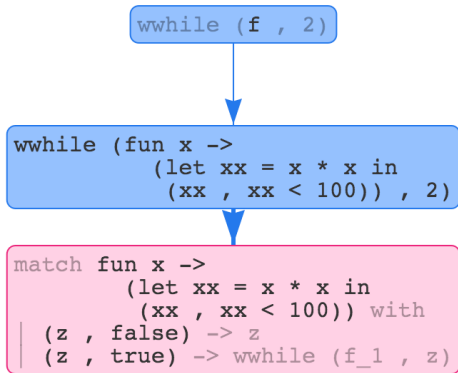
```

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

Unfortunately, the student has forgotten to *apply* `f` at all on line 2, and just matches it directly against a pair. This faulty definition of `wwhile` still typechecks however, and is assumed to be correct by and both OCAML and SHERRLOC which blame the use-site on line 10.

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

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



By highlighting conflicting values (*i.e.* the source and sink of the problem) and not making assumption about function correctness,

NANOMALY focusses the user's attention on the piece of code that is actually relevant to the error.

5.4 Discussion

To summarize, our experiments demonstrate that NANOMALY finds witnesses to type errors: (1) with high coverage in a timespan amenable to compile-time analysis, and (2) with traces that have a low average complexity of 7 jumps.

There are, of course, drawbacks to our approach. 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 NANOMALY may not be able to trigger it. Exhaustive testing and dynamic-symbolic execution can address this short-coming by performing an exhaustive search for inputs (*resp.* paths through the program). As our experiments show, however, novice programs do not appear to require more advanced search techniques, likely because the novice programs tend to be simple.

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

```

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

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

Trace Explosion Though the average complexity of our generated traces is low in terms of jumps, there are some extreme outliers. We cannot reasonably expect a novice user to explore a trace containing 50+ terms and draw a conclusion about which pieces contributed to the bug in their program. Enhancing our visualization to slice out program paths relevant to specific values [26], 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 `fac` example. `<=` is polymorphic in OCAML, but in this case we can make progress because the literal `0` is not. If we parameterized `fac` by a lower bound, *e.g.*

```

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

and called `fac` with two holes, we would get stuck at the $n \leq m$ 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

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

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

The traditional Damas-Milner type inference algorithm [7] reports the first program location where a type mismatch is discovered (subject to the traversal strategy [14]). As a result the error can be reported far away from its source [20] without enough information to guide the user. Type-error slicing [9, 11, 22, 27, 29, 30] 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. [34, 35] present an algorithm for identifying the most likely culprit using Bayesian reasoning. Pavlinovic et al. [24, 25] 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. [16] attempt to suggest a fix by replacing expressions (or removing them entirely) with alternatives based on the surrounding program context. Chen and Erwig [3] 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 et al. search at the type-level and are thus complete due to 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 novice user using a dynamic witness that demonstrates how their program is not just ill-typed but truly wrong. In addition, allowing users to run their program (even knowing that it is wrong) enables experimentation and the use of debuggers to step through the program and investigate its evolution.

Improving Error Messages The content and quality of the error messages themselves has also been studied extensively. Marceau et al. [18, 19] study the effectiveness of error messages in novice environments and present suggestions for improving their quality and consistency. Hage and Heeren [12] identify a variety of general heuristics to improve the quality of type error messages, based on their teaching experience. Heeren et al. [13], Christiansen [4], and Serrano and Hage [31] 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. [33] 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 NANOMALY is at its heart a test generator, and builds on a rich line of work. Our use of holes to represent unknown values is inspired by the work of Runciman, Naylor, and Lindblad [17, 21, 28], 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 [5, 6, 23] 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, 10, 32] 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.

Program Exploration Perera et al. [26] 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 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 the anonymous reviewers for their insightful and enthusiastic feedback.

References

- [1] M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 521–530, New York, NY, USA, 21 May 2011. ACM. ISBN 9781450304450. doi: [10.1145/1985793.1985864](https://doi.org/10.1145/1985793.1985864).

- [2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
- [3] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM. ISBN 9781450325448. doi: [10.1145/2535838.2535863](https://doi.org/10.1145/2535838.2535863).
- [4] D. R. Christiansen. Reflect on your mistakes! lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.
- [5] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 9781581132021. doi: [10.1145/351240.351266](https://doi.org/10.1145/351240.351266).
- [6] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Softw. Pract. Exp.*, 34(11):1025–1050, 1 Sept. 2004. ISSN 0038-0644. doi: [10.1002/spe.602](https://doi.org/10.1002/spe.602).
- [7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 9780897910651. doi: [10.1145/582153.582176](https://doi.org/10.1145/582153.582176).
- [8] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 9780262062756.
- [9] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 72–89. Springer Berlin Heidelberg, 8 Sept. 2004. ISBN 9783540260943, 9783540320388. doi: [10.1007/11431664_5](https://doi.org/10.1007/11431664_5).
- [10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 9781595930569. doi: [10.1145/1065010.1065036](https://doi.org/10.1145/1065010.1065036).
- [11] C. Haack and J. B. Wells. Type error slicing in implicitly typed Higher-Order languages. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 284–301. Springer Berlin Heidelberg, 7 Apr. 2003. ISBN 9783540008866, 9783540365754. doi: [10.1007/3-540-36575-3_20](https://doi.org/10.1007/3-540-36575-3_20).
- [12] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 199–216. Springer Berlin Heidelberg, 4 Sept. 2006. ISBN 9783540741299, 9783540741305. doi: [10.1007/978-3-540-74130-5_12](https://doi.org/10.1007/978-3-540-74130-5_12).
- [13] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, volume 38, pages 3–13. ACM, 25 Aug. 2003. ISBN 9781581137569. doi: [10.1145/944705.944707](https://doi.org/10.1145/944705.944707).
- [14] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, July 1998. ISSN 0164-0925. doi: [10.1145/291891.291892](https://doi.org/10.1145/291891.291892).
- [15] B. Lerner, D. Grossman, and C. Chambers. Seminal: Searching for ML type-error messages. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 63–73, New York, NY, USA, 2006. ACM. ISBN 9781595934833. doi: [10.1145/1159876.1159887](https://doi.org/10.1145/1159876.1159887).
- [16] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM. ISBN 9781595936332. doi: [10.1145/1250734.1250783](https://doi.org/10.1145/1250734.1250783).
- [17] F. Lindblad. Property directed generation of First-Order test data. In M. T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming*, volume 8 of *TFP '07*, pages 105–123, 2007. ISBN 9781841501963.
- [18] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM. ISBN 9781450305006. doi: [10.1145/1953163.1953308](https://doi.org/10.1145/1953163.1953308).
- [19] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18, New York, NY, USA, 2011. ACM. ISBN 9781450309417. doi: [10.1145/2048237.2048241](https://doi.org/10.1145/2048237.2048241).
- [20] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, Lecture Notes in Computer Science, pages 137–152. Springer Berlin Heidelberg, 9 Sept. 1998. ISBN 9783540662297, 9783540485155. doi: [10.1007/3-540-48515-5_9](https://doi.org/10.1007/3-540-48515-5_9).
- [21] M. Naylor and C. Runciman. Finding inputs that reach a target expression. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 133–142, 2007. doi: [10.1109/SCAM.2007.30](https://doi.org/10.1109/SCAM.2007.30).
- [22] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 15–26, New York, NY, USA, 2003. ACM. ISBN 9781581137569. doi: [10.1145/944705.944708](https://doi.org/10.1145/944705.944708).
- [23] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed random test generation. In *29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007. doi: [10.1109/ICSE.2007.37](https://doi.org/10.1109/ICSE.2007.37).
- [24] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM. ISBN 9781450325851. doi: [10.1145/2660193.2660230](https://doi.org/10.1145/2660193.2660230).
- [25] Z. Pavlinovic, T. King, and T. Wies. Practical SMT-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 412–423, New York, NY, USA, 2015. ACM. ISBN 9781450336697. doi: [10.1145/2784731.2784765](https://doi.org/10.1145/2784731.2784765).
- [26] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 365–376, New York, NY, USA, 2012. ACM. ISBN 9781450310543. doi: [10.1145/2364527.2364579](https://doi.org/10.1145/2364527.2364579).
- [27] V. Rahlh, J. Wells, J. Pirie, and F. Kamareddine. Skapel: A type error slicer for standard ML. *Electron. Notes Theor. Comput. Sci.*, 312: 197–213, 24 Apr. 2015. ISSN 1571-0661. doi: [10.1016/j.entcs.2015.04.012](https://doi.org/10.1016/j.entcs.2015.04.012).
- [28] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 9781605580647. doi: [10.1145/1411286.1411292](https://doi.org/10.1145/1411286.1411292).
- [29] K. Sagonas, J. Silva, and S. Tamarit. Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 33–42, New York, NY, USA, 2013. ACM. ISBN 9781450318426. doi: [10.1145/2426890.2426897](https://doi.org/10.1145/2426890.2426897).
- [30] T. Schilling. Constraint-Free type error slicing. In *Trends in Functional Programming*, Lecture Notes in Computer Science, pages 1–16. Springer Berlin Heidelberg, 16 May 2011. ISBN 9783642320361, 9783642320378. doi: [10.1007/978-3-642-32037-8_1](https://doi.org/10.1007/978-3-642-32037-8_1).
- [31] A. Serrano and J. Hage. Type error diagnosis for embedded DSLs by Two-Stage specialized type rules. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 672–698. Springer Berlin Heidelberg, 3 Apr. 2016. ISBN 9783662494974, 9783662494981. doi: [10.1007/978-3-662-49498-1_26](https://doi.org/10.1007/978-3-662-49498-1_26).

- [32] N. Tillmann and J. de Halleux. Pex–White box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, Lecture Notes in Computer Science, pages 134–153. Springer Berlin Heidelberg, 2008. ISBN 9783540791232. doi: [10.1007/978-3-540-79124-9_10](https://doi.org/10.1007/978-3-540-79124-9_10).
- [33] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 9781450310543. doi: [10.1145/2364527.2364554](https://doi.org/10.1145/2364527.2364554).
- [34] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 569–581, New York, NY, USA, 2014. ACM. ISBN 9781450325448. doi: [10.1145/2535838.2535870](https://doi.org/10.1145/2535838.2535870).
- [35] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 12–21, New York, NY, USA, 2015. ACM. ISBN 9781450334686. doi: [10.1145/2737924.2738009](https://doi.org/10.1145/2737924.2738009).

A. Proofs for Section 3.2

Proof of Lemma 3. By induction on τ . In the base case $\tau = \langle f \nu[\alpha], \emptyset, \emptyset \rangle$ 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- \ast -BAD rules (except for E-NODE-BAD1) as the calls to narrow return stuck. An examination of narrow shows that if narrow returns stuck then σ and θ are unchanged.

Case E-PLUS-GOOD: We narrow v_1 and v_2 to `int`, so we must consider the $\text{narrow}(\nu[\alpha], t, \sigma, \theta)$ and $\text{narrow}(n, \text{int}, \sigma, \theta)$ cases. The $\text{narrow}(n, \text{int}, \sigma, \theta)$ case is trivial as it does not change σ or θ . In the $\text{narrow}(\nu[\alpha], t, \sigma, \theta)$ 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 $\nu[\alpha]$ 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 E-PLUS-GOOD.

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

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

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

$\text{narrow}(\nu[\alpha], t, \sigma, \theta)$: Similar to E-PLUS-GOOD.

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

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

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

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

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

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

be the shortest prefix of τ such that $\rho_{\tau_i}(f) \approx t$. 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 $\rho_{\tau_i}(f) \approx t$. 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

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

Case E-PLUS-GOOD: Here we narrow v_1 and v_2 to `int`, so the first case of narrow must apply ($\text{narrow}(n, \text{int}, \sigma, \theta)$ 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 $t' = \text{int}$. Let s be any concrete type that is incompatible with `int` and $\theta_s = \theta_{i-1} [\alpha \mapsto s]$, $\text{narrow}(\nu[\alpha], \text{int}, \sigma_{i-1}, \theta_s) = \langle \text{stuck}, \sigma_{i-1}, \theta_s \rangle$.

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 E-PLUS-GOOD.

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

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

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

$\text{narrow}(\nu[\alpha], t, \sigma, \theta)$: Similar to E-PLUS-GOOD.

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

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

Case E-NODE-BAD1: This case cannot apply as narrow does not update θ when it returns stuck.

Case E-NODE-BAD2: Similar to E-NODE-GOOD.

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

$\text{narrow}(\nu[\alpha], t, \sigma, \theta)$: Similar to E-PLUS-GOOD.

$\text{narrow}(\text{leaf}[t_1], \text{tree } t_2, \sigma, \theta)$: 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.

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

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

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

$\text{narrow}(\nu[\alpha], t, \sigma, \theta)$: Similar to E-PLUS-GOOD.

$\text{narrow}(\langle v_1, v_2 \rangle, t_1 \times t_2, \sigma, \theta)$: 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 E-CASE-PAIR-BAD: This case cannot apply as narrow does not update θ when 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 $u = \text{gen}(s, \theta)$ and $v = \sigma_{i-1}(\nu) [u/\nu'[\alpha']] [s/\alpha']$, $\langle f v, \emptyset, \emptyset \rangle \hookrightarrow^* \langle \text{stuck}, \sigma, \theta \rangle$ in i steps. ■