Finding Bugs in Liquid Haskell

Anish Tondwalkar Rolph Recto
Westley Weimer
Univerity of Virginia
{tondwalkar,rjr7je,weimer}@virginia.edu

Ranjit Jhala
University of California, San Diego
jhala@cs.ucsd.edu

Abstract

Dependent types provide strong guarantees but can be hard to program, admitting mistakes in the implementation as well as the specification. We present algorithms for resolving verification failures by both finding bugs in implementations and also completing annotations in the Liquid Haskell refinement type framework. We present a fault localization algorithm for finding likely bug locations when verification failure stems from a bug in the implementation. We use the type checker as an oracle to search for a set of minimal unsatisfiable type constraints that map to possible bug locations. Conversely, we present an algorithm based on Craig interpolation to discover predicates that allow the type checker to verify programs that would otherwise be deemed unsafe due to inadequate type annotations. We evaluate our algorithms on an indicative benchmark of Haskell programs with Liquid Haskell type annotations. Our fault localization algorithm localizes more bugs than the vanilla Liquid Haskell type checker while still returning a small number of false positives. Our predicate discovery algorithm infers refinements types for large classes of benchmark programs, including all those that admit bounded constraint unrolling. In addition, the design of our algorithms allows them to be effectively extended to other typing systems.

1. Introduction

Automatic type inference for strongly statically typed functional programming languages such as OCaml and Haskell has been a boon for developers, allowing them to catch large classes of errors at compile time [11, 19]. The success of standard type systems has led to significant research on more expressive notions of typing, including type qualifiers [3, 17], and dependent and refinement types [16, 27, 30].

Refinement type systems, which decorate base types with predicates that encode correctness properties such as preconditions and postconditions, are especially promising. While types that depend on expression values are undecidable in general, restricted constraint languages and recent advances in constraint solving (*e.g.*, [6]) have made such systems useful and feasible in practice [23, 27, 28]. These constraint languages restrict the inferred types to those drawn from some *abstract domain*. Refinement type systems allow stronger safety guarantees [1, 9, 30] that are potentially useful for many applications, including security [1], and program synthesis [22]. Unfortunately, the disadvantage of using these expressive type and type inference systems is that annotated program implementations often fail to type check.

When a program does not type check, there are two possibilities: either the implementation is buggy (*i.e.* does not match the specificaiton), or the implementation is actually correct but the type annotations or the prover's abstract domain must be extended to verify correctness. In the first case, the developer must localize the bug and address the issue. Unfortunately, the error locations reported by more complicated type systems are often far from the locations that humans would change to fix those bugs [4]. Previous approaches have been successful at fault localization for classical Hindley-Milner style type checking [18, 21, 32]. However, such techniques do not apply to more sophisticated systems, for which type error localization remains difficult.

In the second case, the type-checker may fail to verify a correct implementation because the abstract domain is insufficiently large for it to infer all of the intermediate types. In this case, the developer must either provide a richer abstract domain or manually annotate necessary intermediate types. This complication arises because refinement type inference is undecidable in general, and thus the type checker cannot usually fill in missing annotations. Providing such annotations has been viewed as a major burden on developers and is often listed as a significant barrier to entry for formal verification systems [8].

To address both possibilities, we introduce two algorithms. Our *fault localization* algorithm uses the type checker (i.e. constraint solver) as an oracle in a search procedure for a minimal unsatisfiable constraint set, whose constraints map to likely locations of the bug in the implementation. We exploit properties of constraints and employ the delta debugging algorithm [31], originally designed to find regressions in codebase changes, to make our approach practical.

Our *predicate discovery* algorithm uses Craig interpolation [5] to simultaneously provide additional intermediate types as well as to expand the abstract domain. Instead of restricting an abstract domain to make type inference decidable, we approximate infinite recursion using bounded unrolling (§ 4.4). In the case that the type constraints admit bounded unrolling, our algorithm finds all intermediate types necessary for program verification.

DOI: http://dx.doi.org/10.1145/(to come)

Our algorithms have several advantages over previous approaches. Since our fault localization algorithm uses the type checker as an oracle, it does not depend on type system internals. In addition, our predicate discovery algorithm is general over Horn constraints over a theory that admits interpolation. Our algorithms thus have wide applicability for a variety of constraint-based typing systems.

We evaluate our algorithms on programs with type annotations from the Liquid Haskell (LH) framework [23, 27]. Our experiments show that our fault localization algorithm is much more accurate at localizing bugs than the Liquid Haskell type checker. In addition, our predicate discovery algorithm successfully infers type refinements and finds predicates which extend the abstract domain, allowing correct implementations to be verified.

We present the following contributions:

- A novel fault localization algorithm for constraint-based type systems. We search for a minimal unsatisfiable constraint set using the type checker as guidance. We exploit the structure of LH constraint sets to optimize our search procedure and find more bug locations.
- A novel predicate discovery algorithm for constraint-based type systems that allows the type checker to verify additional correct implementations. The algorithm "unrolls" typing constraints and computes Craig interpolants that are the predicates of our newly-inferred types. We exploit an orthogonality of approximations to integrate the our results with Liquid Haskell checking.
- An implementation and empirical evaluation of our algorithms. Over a set of benchmarks we find that our fault localization algorithm outperforms the state-of-the-art Liquid Haskell type checker, correctly localizing more than twice as many bugs while returning a modest amount of false positives. Our predicate discovery algorithm finds correct predicates and type annotations, successfully verifying a large class of implementations in time comparable to standard Liquid Haskell type checking.

The undecidability of dependent type inference challenges the adoption of program verification at the type system level. Liquid Types provide an elegant solution to this problem by restricting types to be inferred to an abstract domain. However, it can be difficult programmers to verify Liquid Types implementations and annotations: our two algorithms lower the barrier to entry and thus help bring the benefits of constraint-based type systems to a wider audience.

2. Overview

In this section we review the Liquid Haskell type system, then highlight key features of our algorithms and present motivating examples demonstrating their effectiveness. Formal details are given in (§ 3) and (§ 4).

2.1 Liquid Haskell

Liquid Haskell is a framework for annotating Haskell programs with *refinement types*, which are types decorated with predicates. The predicates are in the language of a decidable logic (quantifier-free logic of linear arithmetic and uninterpreted functions), allowing the use of an SMT solver for decidable type checking. Liquid Haskell comes equipped with a default *abstract domain*, predicate templates that can be filled in with program variables. These are

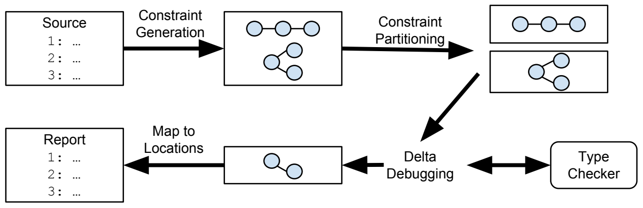

Figure 1. Fault localization algorithm architecture. We use the type checker as an oracle for a search procedure to find a minimal unsatisfiable constraint set.

useful in practice for verifying common operations but are not sufficient to prove all programs correct. Liquid Haskell uses the Liquid Types [23] framework to infer refinement types, which greatly reduces the annotation burden for users.

Liquid Types is built atop a Hindley-Milner style typing system: after an H-M oracle determines the "shapes" of the types of program expressions, the Liquid Types constraint solver attempts to find a solution to the fresh type variables (called κ variables) introduced. A solution maps each variable to a conjunction of predicates. These predicates come from a set of *qualifiers*, taken from our abstract domain.

The Liquid Types solver finds the strongest — most specific — solution for the κ variables by starting with the conjunction of every possible instantiated qualifier (those filled in with variables) and repeatedly weakening the solution until no constraints fail or no solution is possible.

2.2 Fault Localization Algorithm Preview

Figure 1 diagrams the architecture of our fault localization algorithm. We assume an unsatisfiable constraint set generated from a buggy Haskell program with Liquid Haskell annotations. We then partition the constraint set, and for each relevant partition we run the delta debugging minimization algorithm, using the type checker as an oracle to check the (un)satisfiability of constraint sets. After computing minimal unsatisfiable sets from each relevant partition, we map these to locations, which are then returned for inspection.

The intuition for why the constraints in a minimal unsatisfiable set map to likely bug locations is as follows:

1. *A bug captured at the type checking level can be seen as an inconsistency.* In Liquid Haskell, the inconsistency might be a parameter requiring an `Int` with a nonzero value when the actual argument is provably always `0`. The non-zero requirement is encoded in one constraint, and the information that the actual argument is always `0` is encoded in another. Together, these two constraints encode (or witness) the inconsistency in the program.
2. *The locations reported to the user should be minimal to prevent implicating spurious program locations as faults.* If some constraint in an unsatisfiable set can be removed and the set is still unsatisfiable, it is unlikely to be relevant to the explanation of the unsatisfiability of that set.
3. *Minimal explanations implicate relevant locations.* A minimal unsatisfiable constraint set has no irrelevant constraints: all associated program locations are necessary to explain the bug.

Consider this Haskell program with Liquid Haskell types:

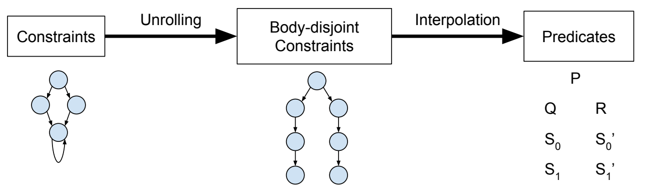

Figure 2. Predicate discovery architecture diagram. We find an interpolant for every node the unrolled induced graph.

```haskell
{-@ div2 :: Int -> { v:Int | v > 0 } -> Int @-}
div2 n d = n `div` d

nonzero d = 0

f n d = n `div2` (nonzero d) + n `div2` (nonzero d)
```

Note that the function `div2` requires a positive second argument, but the `nonzero` function, although evocatively named, does not actually provide a non-zero value (indeed, it always returns zero).

A Liquid Haskell constraint set $\{c_1, \ldots, c_8\}$ is generated from this program. The Liquid Haskell type checker returns the locations mapped to the failing constraints $\{c_1, c_4\}$, which correspond only to the application of the function `nonzero` but not its body. By contrast, the minimal constraint set found by our algorithm, $\{c_1, c_6, c_8\}$, corresponds to the positive argument requirement, function call, and return value. This set maps to both the body of `nonzero` and also the call to it. These are the locations we want to bring to the attention of the developer when localizing this bug. We formalize our algorithm for finding them in (§ 3).

2.3 Predicate Discovery Algorithm Preview

Figure 2 diagrams the architecture for our predicate discovery algorithm. We take as input a system of typing constraints that Liquid Haskell cannot solve over its abstract domain — that is, a program we cannot currently prove correct. Our algorithm extends the abstract domain and produces types for intermediate variables such that the program can be verified. Broadly, we first unroll the system of constraints to obtain constraints that do not share dependencies. Second, we use disjunctive interpolation to produce predicates that makes each constraint node well-typed.

There are two cases governing our algorithm:

- *We can solve all possible unrollable constraints.* If the system of constraints admits a solution and also admits finite unrolling to address cycles, our algorithm produces a special predicate for each type. Decoration with these predicates forms exactly the needed refinement types, allowing the program to be verified.
- *We can usefully expand the abstract domain.* In the case that the system is not unrollable, we still produce a special predicate for each type. We then build templates out of these predicates that extend our abstract domain. While these templates alone do not always allow the program to be verified, they are always a subset of a proper solution.

Consider the following Haskell listing:

```haskell
1  inc :: Int -> Int
2  inc x = x + 1
3
4  divTenBy :: Int -> Int
5  divTenBy n = let b = 0 <= n in
6                if b then
7                  let a = inc n
8                  in div 10 a
9                else 1
```

The `divTenBy` procedure above is safe — it never divides by zero — but a novice programmer may not know how to annotate it so that Liquid Haskell can verify that property. Our predicate discovery algorithm learns two useful pieces of information about this program: first, a refinement type for the `inc` function that encodes its behavior; and second, intermediate types for local variables such as `a` that are necessary for verification.

Liquid Haskell generates constraints for this program but cannot verify it over an empty abstract domain. Intuitively, we cannot prove that `div 10 a` is safe on line 8 because we do not have a type for `a` that rules out `0` values. Solving the implicated failing constraints in undecidable in general; unrolling is one approximation that handle this problem in many cases. We unroll the implicated failing constraints and serialize the resulting graph, obtaining:

$$\begin{aligned} & b_1 \leftrightarrow 0 \leq n \wedge b_1 \\ & \qquad \wedge (b_2 \leftrightarrow 0 \leq n \wedge b_2 \wedge a = n + 1) \\ & \qquad \wedge (b_3 \leftrightarrow 0 \leq n \wedge b_3 \wedge v = n + 1) \\ & \qquad \wedge v = 0 \end{aligned}$$

Note that $b_1 \ldots b_3$, refer to the variable `b` at different program locations (cf. SSA form renumbering). This predicate encodes necessary conditions for program correctness, but it is difficult to manually extract refinement types for program variables from this predicate form [14]. We use Craig interpolation [5] to automatically extract refinement types:

$$\begin{aligned} & b_1 \leftrightarrow 0 \leq n \wedge b_1 & \\ & \qquad \wedge (b_2 \leftrightarrow 0 \leq n \wedge b_2 \wedge a = n + 1) & \texttt{true} \\ & \qquad \wedge (b_3 \leftrightarrow 0 \leq n \wedge b_3 \wedge v = n + 1) & \texttt{true} \\ & \qquad \wedge v = 0 & !(\texttt{v <= 0}) \end{aligned}$$

Craig interpolation returns the predicate `!(v<=0)`, which is a valid type for the variable `a` (which is always strictly greater than zero in the running program). This allows Liquid Haskell to verify the safety of the program.

3. Fault Localization for Constraint-Based Type Systems

Next, we introduce and describe our procedure `minimize` for implicating relevant source locations in programs that are not well-typed with respect to a constraint-based type system. Note that we operate on induced constraints rather than program text or syntax trees. While it would be possible to treat the latter as primary (*e.g.*, finding a minimal subsequence of the program text that fails to typecheck and reporting it as an explanation), such approaches may not capture the semantics of the program and can thus lead to poor explanations (*e.g.*, `"A"+0` may be an ill-typed textual subsequence of many programs, but is unlikely to explain the original problem).

In the rest of this section, we start with a high-level overview of the delta debugging algorithm (§ 3.1). Next, we show how delta debugging can be adapted to the problem of minimizing constraints and finding source locations that pertain to a type error (§ 3.2). Finally, we describe an optimization motivated by the structure of the constraints further improves the accuracy of the minimization procedure (§ 3.3).

3.1 Background: Delta Debugging

A naïve minimization algorithm might enumerate all subsets of the constraints, requiring time exponential in the number of constraints. Since the output of our algorithm is consumed by human developers, we desire rapid feedback, even if it implicates a few extra locations (constraints), as otherwise localization will not be used in practice.

Minimal subsets. The *Delta Debugging* algorithm of Zeller was originally proposed to find the cause of a regression — *i.e.* a bug introduced to the working program — after a codebase changes [31]. Delta debugging takes as input a set of code changes and a black box notion of what it means for a set to be *interesting* (*e.g.* applying those changes results in failed regression tests). Delta debugging efficiently finds a *one-minimal* interesting subset of changes: a set of changes that is interesting, but becomes uninteresting if any single element is removed. Next, we describe the requirements and algorithm of delta debugging and show how to adapt it to efficiently minimize the unsatisfiable constraints.

Tests and Configurations. A *configuration* Δ (*e.g.* diff, patch, commit, *etc.*) is a set of *changes* $\{\delta_1, \dots, \delta_n\}$ to a code base. A `test` function, which formalizes program correctness or a test suite for that program, takes any subset $\Delta' \subseteq \Delta$ of the configuration and returns

$$\texttt{test}(\Delta') = \begin{cases} \checkmark & \text{if } \Delta' \text{ “passes”} \\ \times & \text{if } \Delta' \text{ “fails”} \\ ? & \text{otherwise} \end{cases}$$

Requirements. Delta debugging requires certain properties of configurations and tests to make its search efficient. A configuration Δ is *interesting* if it fails `test`. Δ is *consistent* if for all its subsets `test` returns either ✓ or $\times$. Δ is *monotonic* if any superset of an interesting subset of Δ is also interesting. Δ is *unambiguous* if for two of its subsets that fail, their intersection also fails. This limits attention to one configuration: for example, if Δ_1 produces a failure and Δ_2 produces a failure, some common element(s) $\Delta_1 \cap \Delta_2$ must actually be responsible for that failure.[1]

Delta Debugging. Given a `test` function and a configuration Δ that is interesting, consistent, monotonic and unambiguous, the Delta debugging algorithm returns a *one-minimal* failing configuration Δ' such that

$$\begin{aligned} \text{test}(\Delta') &\neq \checkmark \\ \forall c \in \Delta'.\, \text{test}(\Delta' \setminus \{c\}) &\neq \times \end{aligned}$$

A one-minimal subset can be computed via polynomial (in the size of Δ) queries to `test` [31, Alg. 1].

3.2 General Constraint Minimization

From Rondon *et al.* [23], we can assume the existence of an oracle `solve` and an abstract domain $\mathbb{Q}$ such that, given a set of constraints C, `solve` returns either `SAT` if the constraints are ($\mathbb{Q}$-)*satisfiable* or `UNSAT` otherwise.

By suitably defining the notions of *configuration* and *test*, we can find minimal unsatisfiable constraint sets over general constraint-based languages in a sound manner.

Configurations and Tests. The *configuration* is the constraint set $C = \{c_1, \dots, c_n\}$ generated from the (buggy) program. The `test` oracle is defined as:

$$\texttt{test}(C') = \begin{cases} \checkmark & \text{if } \texttt{solve}(C') = \texttt{SAT} \\ \times & \text{if } \texttt{solve}(C') = \texttt{UNSAT} \end{cases}$$

A *minimal unsatisfiable constraint set* C is one such that: (1) $\texttt{solve}(C) = \texttt{UNSAT}$, and (2) $\forall C' \subset C.\, \texttt{solve}(C') = \texttt{SAT}$. That is, C is minimally unsatisfiable if it is unsatisfiable and any proper subset is satisfiable.

Delta Debugging Requirements. We can instantiate delta debugging with the above notion of configuration and test, as the typing constraints and `solve` oracle obey most of the Delta debugging requirements directly. Constraint configurations are *interesting* since we only consider unsafe programs; they are *consistent* since the oracle returns either `SAT` or `UNSAT`. That constraint sets are *monotonic* is a corollary of Theorem 2 in Rondon *et al.* [23]. Intuitively, adding more constraints only *weakens* the possible solution, which in turn ensures that a failing constraint will continue to fail. Although delta debugging can only guarantee that its output is one-minimal, the monotonicity of Liquid Haskell constraint configurations guarantees that its output is minimal.

Unfortunately, the LH constraint sets are *not* always unambiguous. There can be two independent failure causes within a constraint set. We analyze a concrete example in (§ 5.6).

Algorithm. Our delta-debugging based constraint minimization (and hence, fault localization) algorithm is shown in Algorithm 1. Given an unsatisfiable constraint set, we find a minimal unsatisfiable subset via Delta debugging (Line 2) and return all associated program locations (Line 3). This algorithm works for general constraint-based type systems.

Algorithm 1 Constraint Minimization Fault Localization

```
Require: $C$ is an unsatisfiable constraint set
Require: consLoc returns the program locations associated with
    a constraint
1: procedure MINIMIZE($C$)
2:     $C' \leftarrow \texttt{DeltaDebug}(C, \emptyset)$
3:     $locs \leftarrow \bigcup \text{map}(\texttt{consLoc}, C')$
4:     return $locs$
5: end procedure
```

3.3 Partitioned Constraint Minimization

As discussed above, since Liquid Haskell constraints are not *unambiguous*, there can be multiple causes of failure, and Delta debugging can only return one of these (*i.e.* as a minimal unsatisfiable set). We heuristically address this problem by developing a *partitioned* minimization algorithm `minimizeWCC`, in which *independent* constraints are separated, increasing our chances of localizing the bug.

Formula and Constraint Variables. We say a κ variable *appears in* a formula f, written $\kappa \in f$ if $f \equiv f_1 \wedge \kappa(\bar{y}) \wedge f_2$ for some (sub-) formulas f_1 and f_2 and logical variables $\bar{y}$. That is, $\kappa \in f$ if an instantiation of κ is a conjunct of f. We say that a constraint $c = f \Rightarrow f'$ *reads* a variable κ, written $\mathsf{Reads}(\kappa, c)$, if $\kappa \in f$. Dually, we say c *writes* κ, written $\mathsf{Writes}(\kappa, c)$, if $\kappa \in f'$.

Constraint Dependencies. The structure of these constraints induces a binary dependency relation between constraints. We say a constraint c' *depends on* c, written $c \rightsquigarrow c'$ if, there exists a κ such

[1] To see that unambiguity means that there is only one cause, consider two elements $a, b \in C$ that are separate causes of failure. Now assume $a \in A$ and $b \in B$ and $a, b \notin A \cap B$ for some $A, B \subseteq C$. Then $\text{test}(A) = \times$ and $\text{test}(B) = \times$ but $\text{test}(A \cap B) \neq \times$.

that $\mathsf{Writes}(\kappa, c)$ and $\mathsf{Reads}(\kappa, c')$. We write $c \rightsquigarrow^* c'$ if c and c' are related by the *reflexive*, *symmetric* and *transitive* closure of the depends-on relationship.

Constraint Partitions. As $\rightsquigarrow^*$ is an equivalence relation, we can partition (or quotient) constraints C into equivalence classes with respect to it. Let $\mathtt{WCC}(C)$ denote the above partitioning of C. We can compute this efficiently as the *weakly connected components* of the undirected graph corresponding to the symmetric closure of the *depends-on* relationship.

Partitioned Minimization. Each such partition is *independent* of the others. That is, the whole set is satisfiable iff every partition is satisfiable. This also means that if any constraint c in some partition fails, any other constraint that is relevant to the failure of c is in that partition. Consequently, to localize faults it suffices to minimize each *relevant* partition individually and union the results. *Relevant* partitions are those that have originally failing constraints from the input unsatisfiable constraint set for the buggy program. The procedure to localize bugs within partitions is formalized as `minimizeWCC` (Algorithm 2).

Algorithm 2 Partitioned Constraint Minimization

```
Require: $C$ is an unsatisfiable constraint set
Require: $F \subseteq C$ is the set of failing constraints in $C$
Require: consLoc returns the program locations associated with
    a constraint
Require: $\mathtt{WCC}(C)$ returns the dependency partitions of $C$
Require: $\mathtt{RelWCC}(C) = \{p \mid p \in \mathtt{WCC}(C) \wedge \exists c \in p.\, c \in F\}$
1: procedure MINIMIZEWCC($C$)
2:     $Cs' \leftarrow \{p' \mid p \in \mathtt{RelWCC}(C) \wedge \mathtt{solve}(p) = \mathtt{UNSAT} \wedge p' = \mathtt{DeltaDebug}(p, \emptyset)\}$
3:     $locs \leftarrow \bigcup\bigcup \mathrm{map}(\mathtt{consLoc}, Cs')$
4:     return $locs$
5: end procedure
```

Benefits of Partitioning. Without this partitioning, a direct use of Delta Debugging can only find a single minimal unsatisfiable subset — one cause of failure — but may be many subsets in practice. In contrast, partitioned minimization can find multiple such sets (one for each relevant partition), increasing the likelihood of implicating the correct bug location. An example of this is detailed in (§ 5.6), where `minimizeWCC` localizes a bug that `minimize` does not.

4. Predicate Discovery for Constraint-Based Type Systems

Here we introduce an algorithm that allows additional correct programs to be verified by constraint-based type systems. We do this by inferring correct refinements for intermediate variables as well as by computing a rich abstract domain. We take as input a set of Horn clause constraints that cannot be verified with respect to a default abstract domain (such as $\mathbb{Q}$ from Liquid Haskell).

If that constraint set happens to admit bounded unrolling, our algorithm always finds a solution that admits verification. In this case, our algorithm is correct as well as complete relative to our underlying theory $\mathbb{Q}$. If the system does not admit bounded unrolling, we still produce a solution that is likely useful, in the sense that it is an improper subset of a complete answer. The intuition here is similar to that of classic k-bounded model checking.

In the rest of this section, we first describe three necessary preliminary formalisms. We formalize the input to our algorithm (§ 4.1), the signature of predicate discovery (§ 4.2), and the critical substep of tree interpolation (§ 4.3). Our algorithm proper begins with a bounded unrolling (§ 4.4), in which we produce a system of independent constraints from the initial constraints. We then employ a variant of disjunctive interpolation (§ 4.5) to mine predicates that can be used as either as refinements or qualifiers.

4.1 Constraint Typing Formalism

We represent input programs in constraint-based languages as 2-tuples (Δ, Σ), where Δ is a set of unrefined types, and $\Sigma : \Delta \rightarrow 2^T + \bot$, assigns to each unrefined type a set of predicates from our underlying theory or $\bot$. The interpretation of $\bot$ is that the user has not provided a type annotation for that program element; by contrast, the empty set of predicates means that the program element is unconstrained.

A constraint-based type system, such as Liquid Haskell, can verify the correctness of programs (Δ, Σ) over an abstract domain $\mathbb{Q}$ by first inferring type annotations and then checking them. We use ι to denote *type inference*, with $\iota(\Delta, \Sigma, \mathbb{Q}) \mapsto \Xi$, where $\Xi : \Delta \rightarrow 2^T$. Note that $\Delta \rightarrow 2^T$ is Σ with $\bot$ (which represented no annotation) removed from the range; that is, ι must assign a type annotation to every program element. We use σ to denote *dependent type checking*, with $\sigma(\Delta, \Xi) : \{\mathsf{SAFE}, \mathsf{UNSAFE}\}$. We use L to denote the combined type checker, which includes both type inference and type checking: $L(\Delta, \Sigma, \mathbb{Q}) : \{\mathsf{SAFE}, \mathsf{UNSAFE}\}$. In this formulation, L takes a partially-annotated program and an abstract domain, over which it solves the system of induced constraints, deciding if the program is safe.

The heart of Liquid Haskell type checking is finding solutions of annotations mapped to $\bot$ in Σ — that is, inferring types where no user annotation exists. LH uses κ variables, existentially quantified uninterpreted function symbols, to represent those not-yet-inferred types [23].

4.2 Predicate Discovery Formalism

Our predicate discovery algorithm D runs before type inference, with $D(\Delta, \Sigma) \mapsto \langle \mathbb{Q}, \Xi \rangle$. Given unrefined types Δ and an annotation partial mapping Σ, it returns predicates that can be used to extend the abstract domain as well as candidate types for intermediate variables.

Our discovery algorithm operates on the graph G induced by the constraint dependency relation $\rightsquigarrow$ introduced in § 3.3. Formally, $G = (\Delta, \rightsquigarrow)$. While general constraint type inference is undecidable, in the special case where the graph is acylcic (a DAG or tree), a useful approximation applies. We thus also consider constraint trees that result from a finite flattening (or unrolling) process applied to such graphs.

In mathematical logic, Craig interpolation extracts the relationship between jointly unsatisfiable formulas in terms of symbols that are common to them. Formally, Craig's theorem [5] holds that $\forall A, B.\ (A \wedge B \rightarrow \text{false}) \rightarrow \exists I.\ (A \rightarrow I) \wedge (B \wedge I \rightarrow \text{false}) \wedge \mathcal{L}(I) \subset \mathcal{L}(A) \cap \mathcal{L}(B)$. Here A and B are two jointly unsatisfiable formulas, I is their interpolant, and I uses only terms common to them.

4.3 Tree interpolation

Tree interpolation is a generalization of standard interpolation (*e.g.* [2]) that applies to a single logical formula from a language that contains nested terms and connectives. Intuitively, the abstract syntax tree of the formula is the "tree" in tree interpolation. Tree interpolation is defined as a higher-order function takes as input an abstract syntax tree (V, E) and a function $V \to T$ that assigns to each vertex a formula, and returns a tree interpolant. A tree interpolant is a function $I : V \to T$ from vertices to predicates such that

- $I(\text{root}) = \text{false}$
- $\forall v \in V.\ (v \wedge \bigwedge_{w_i \in E^*(v)} I(w_i) \vdash I(v))$
- $\forall v \in V.\ (\mathcal{L}(I(v)) \subseteq \mathcal{L}(E^*(v)) \cup \mathcal{L}(V - E^*(v)))$

where E^* is the transitive closure of E.

4.4 Constraint Graph Unrolling

We now describe our procedure `unroll` which takes as input a set of constraints C (with induced graph G) and an unrolling depth k. This produces a new set of constraints C' (with induced graph G' that is a tree) such that any solution to C' — an assignment of values to program variables that proves our program unsafe — is also a solution to C. The tree structure of G' means that the constraints C' are body-disjoint [24, § 4.1], that is, each κ variable occurs at most once in each constraint and is read from by at most one constraint in the language of § 3.3.

We take advantage of the insight that at least one vertex of every edge has to be a κ variable, and thus we can always break cycles by altering only κ variable nodes. We obtain G' from G by a depth-first traversal of the graph in which we track the number of times a given node has been visited. A recursive traversal ends when visiting a node for the $k + 1$th time if the unrolling depth is k. After the traversal, the resulting tree is alpha-renamed so that each κ variable node has a unique label. The pseudocode for the traversal is shown in Algorithm 3; initially `visited` maps each node to 0 and we start v corresponding to the failing constraint. We elide a description of the alpha renaming for brevity.

Algorithm 3 Bounded Unrolling

```
Require: $k$ is a finite unrolling depth
Require: visited is a mapping from nodes to visit counts
Require: $v$ is the current vertex
1: procedure UNROLL($k$, visited, $v$)
2:     if visited($v$) $> k$ then
3:         return $\emptyset$
4:     end if
5:     visited$'$ $\leftarrow$ visited[$v \mapsto$ visited($v$) + 1]
6:     edges$'$ $\leftarrow \bigcup_{v' \in v.edges}$ unroll($k$, visited$'$, $v'$)
7:     return $\{v$ with $edges =$ edges$'\}$
8: end procedure
```

As a worked example, consider the following constraints:

```
          v = 10 => K1(v)
          v = 20 => K1(v)
           K1(v) => K2(v)
           K1(v) => K3(v)
K2(x) /\ K3(y) => x = y
```

The first two constraints mean that either the value 10 or the value 20 can flow into `K1(v)` for any `v`. The next two constraints mean that `K1(v)` can flow into either `K2(v)` or `K3(v)` for any `v`. The final constraint requires that if `K2` writes to `x` and `K3` writes to `y`, then `x` must equal `y`. Note that this is unsafe, since `x` and `y` can independently each be either 10 or 20 (transitively).

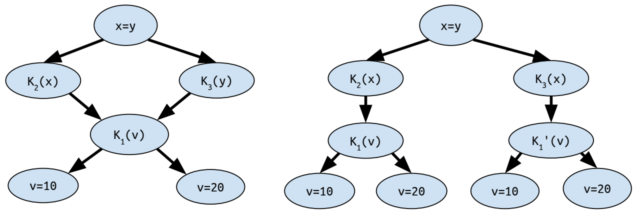

Figure 3. A constraint graph (left) and its corresponding unrolled and renamed constraint tree (right).

The induced graph associated with these constraints does not form a tree. Our unrolling produces a tree of constraints that captures the semantics of the original, modulo recursion. Figure 3 shows the induced graph and resulting tree.

Finally, we say a constraint graph is *k-bounded* if $\forall k' > k$ the k'-bounded unrolling of the graph is equal to the k-bounded unrolling. Our overall predicate discovery algorithm correctly infers all types for constraint sets with solvable k-bounded constraint graphs. Once the constraint graph has been unrolled to a tree we apply tree interpolation to discover predicates and intermediate types.

4.5 Constraint Tree Interpolation

This stage of our algorithm takes as input an unrolled constraint tree. It produces $\mathbb{Q}$, a set of predicates that extend the abstract domain, and $\Xi : (\Delta \to 2^T)$, a mapping from intermediate variables to type annotations. If the unrolled constraint tree is k-bounded then $\sigma(\Delta, \Xi) = \text{SAFE}$ — the type annotations Ξ allow dependent type checking to verify the program. Otherwise, $\mathbb{Q}$ produces an improper subset of an abstract domain over which the program can be proved safe using a refinement type checker.

The key insight is that two relevant approximations are orthogonal: solving over the abstract domain and the qualifiers given by interpolation with k-unrolling. That is, in some cases Liquid Haskell can prove a partially-annotated program correct on its own. In other cases, including those in which the partially-annotated program's constraint graph is k-bounded, the type annotations Ξ provided by interpolation can prove the program correct. Finally, in all other cases, the set of predicates $\mathbb{Q}$ returned by interpolation produce an improper subset of an abstract domain over which the program can be proved correct.

At high level, our algorithm takes the unrolled constraint graph (from § 4.4) and transforms it before applying tree interpolation. We then construct our final answer from a conditional transformation of the result of tree interpolation. We map each input constraint to a rewritten constraint that replaces each existentially quantified κ variable with a universally quantified boolean a_k. This is analogous to falsifying an implication in theorem proving: to check if $H \implies G$ is false it suffices to examine $H \wedge \neg G$. The existentially quantified κ variables in the goal G are replaced by negated universally quantified variables (cf. $\neg\exists P = \forall\neg P$).

The pseudocode for our algorithm is shown in Algorithm 4. Our algorithm iteratively considers a sequence of trees $((V_0, E_0) \ldots (V_{h-1}, E_{h-1}))$ where the nodes are annotated with are predicates (via the A functions). The root node corresponds to an invariant that we should be able to prove given types for all of the variables in the programs — intuitively, the root node corresponds to the goal. When tree G_i is considered, the leaf nodes (h_0 on line 7) correspond to the information available

Algorithm 4 Disjunctive Interpolation

```
Require: $G_0 = (V_0, E_0)$ is the input unrolled tree
Require: $A_0 : V \to T$ gives each vertex's formula
 1: procedure DISJUNCTIVEINTERPOLATION($G_0, A_0$)
 2:     $h \leftarrow \text{height}(G)$
 3:     $\mathbb{Q} \leftarrow \emptyset$
 4:     $\Xi \leftarrow \emptyset$
 5:     for $i \leftarrow 1 \dots h - 1$ do
 6:         $I \leftarrow \texttt{TreeInterpolate}(G_{i-1}, A_{i-1})$
 7:         $h_0 \leftarrow \text{nodesOfHeight}(G_{i-1}, 0)$
 8:         $V_i \leftarrow V_{i-1} \setminus h_0$
 9:         $h_1 \leftarrow \text{nodesOfHeight}(G_{i-1}, 1)$
10:         $E_i \leftarrow \{(v_1, v_2) \in E_{i-1} \mid v_2 \notin h_0\}$
11:         $G_i \leftarrow (V_i, E_i)$
12:         $A_i \leftarrow$
            $\lambda v. \begin{cases} A_{i-1}(v) \wedge \bigvee_{l \in E_{i-1}(v)} \texttt{prep}(I(l)) & \text{if } v \in h_1 \\ A_{i-1}(v) & \text{else} \end{cases}$
13:         $\mathbb{Q} \leftarrow \mathbb{Q} \cup \bigcup_{v \in h_0} \texttt{prep}(I(v))$
14:         $\Xi \leftarrow \Xi \cup \bigcup_{v \in h_0} (v, \texttt{prep}(I(v)))$
15:     end for
16:     return $\langle \mathbb{Q}, \Xi \rangle$
17: end procedure
```

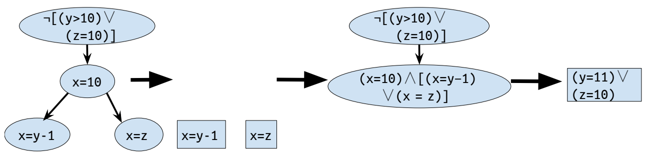

Figure 4. Worked example of Disjunctive Interpolation

after interpolating $i - 1$ times.[2] Each tree is constructed from the previous tree (line 11) in the sequence by performing tree interpolation on the leaves and then folding the result into the annotations for parents (h_1 on line 9) of those leaves (line 12). In particular, the predicate annotation of each leaf parent node in tree i is equal to the conjunction of that node's annotation in tree $i - 1$ with the disjunction of the interpolants from its leaf children in tree $i - 1$ (line 12, first case). The leaves are not carried over to the next tree (lines 8 and 10), which thus has its height reduced by one. The final result of this sequence is a single-node tree. All of the interpolants generated make up the returned $\mathbb{Q}$. In addition, we return Ξ such that $\Xi(\kappa)$ is the interpolant generated from leaf node κ (which occurs at some point in the sequence). The `prep` function maps predicates of the form $\neg a_k \vee I_k$ to I_k. The result of interpolation $I(l)$ will always contain $\neg a_k$ by construction [24, p. 8], and we remove it to produce an invariant that does not contain any auxiliary boolean variables — that is, an interpolant in terms of only our program variables.

Given the unrolled constraint tree (V, E) from from § 4.4, we produce A_0, the initial mapping from vertices to annotations, as follows. A_0 of the root node is the concrete failing invariant that we ultimately wish to prove. Since the tree is induced from the constraints, for other vertices v, $A_0(v)$ consists of the additional invariants stipulated across that vertex in the constraints.

Consider the worked example in Figure 4. The leftmost tree represents G_0, the input. The annotations (A_0) are written inside each vertex. Our algorithm begins by considering the two leaf nodes, annotated with $x = y - 1$ and $x = z$. Bearing in mind that the root node holds the invariant that needs to be proved $\neg(y > 10 \wedge z = 10)$, the interpolants for these two nodes are $x = y - 1$ and $x = z$ (shown in offset rectangles). That is, inter interpolants are trivial because all of the information the leaves encode is needed to prove the root. The next tree G_1 is created from G_0 by removing the leaf nodes and replacing the parent node annotation with its old annotation conjuncted with the disjunction of its children. In the next iteration, that new node is the only leaf node. Its interpolant is $y = 11 \vee z = 10$ — this interpolant encodes all of the information of the children of the tree using the language of only the root. The interpolation at each step encodes all of the information of the subtrees removed level by level in the previous steps.

For unrolled constraint trees that are k-bounded, our algorithm produces Ξ such that $\sigma(\Delta, \Xi) =$ SAFE. That is, the type annotations Ξ are exactly sufficient for the dependent type checker to verify the input program. Even if the tree is not k-bounded, the output $\mathbb{Q}$ is an improper subset of an abstract domain over which the program can be proved. We demonstrate empirically that these domain extensions are useful (§ 5.4).

The correctness of our algorithm follows from the guarantees of tree invariants (see § 4.3) as well as the way in which we construct new node annotations (A_i on line 12) by conjoining tree interpolants of child nodes. Intuitively, this works because of a critical property of interpolants in general: when I is an interpolant of $A \wedge B \to$ false, $A \to I$ and $A \wedge I \to$ false. Our algorithm maintains the invariant that the constraints induced by each tree in sequence are unsatisfiable (i.e., there is no assignment of values to program variables that is unsafe). In addition, we maintain the invariant that the predicate assigned to each node is implied by the conjunction of the predicates of its children in the previous graph: $\bigwedge_{c \in E_{i-1}(v)} A_{i-1}(c) \to A_i(v)$. This holds exactly because the interpolants (in line 12) are implied by the predicates of the leaf nodes from which they were derived. This is equivalent to Theorem 1 of Rümmer *et al.* [24].[3]

5. Evaluation

We implemented our proposed algorithms and evaluated their performance by testing them on Haskell programs with Liquid Haskell type annotations. We ran fault localization experiments on a Lenovo ThinkPad with a Core i5 processor at 3.10 GHz and predicate discovery experiments on a Lenovo Z70-80 with a Core i7 at 2.40GHz.

We consider the following research questions:

- RQ1 — Does our fault localization algorithm provide better accuracy than vanilla LH typechecking?
- RQ2 — How does the speed of our fault localization algorithm compare to that of vanilla LH typechecking?
- RQ3 — Does our predicate discovery algorithm infer all off the needed intermediate types for k-bounded systems of constraints?
- RQ4 — How does our predicate discovery algorithm fare for constraint systems that don't have this property?

[2] Recall that in the case where our constraint graph is not k-bounded, this information may be imperfect.

[3] Our transformation can be viewed as an adaptation Rümmer *et al.*'s with an additional negation at the top of the constraint tree because there is an additional concrete type that our κ variables must prove. The additional concrete type corresponds to the right hand side of the failing constraint.

5.1 Benchmark Selection

Because programs with Liquid Haskell type annotations are not yet common, we constructed a set of microbenchmarks to evaluate both of our algorithms. For fault localization, we modified 28 programs with Liquid Haskell type annotations. These programs were drawn from Liquid Haskell's test suite, found on Github. For these programs, we modified either the constraint set generated or the source code itself. For predicate discovery, we considered two sets of tests: one in which 9 programs have k-bounded constraint graphs, and one in which 4 do not. However, each program may admit multiple safety properties, and thus these 13 programs yield 44 microbenchmarks. Some are from the Liquid Haskell test suite while others were written for this evaluation.[4] The programs range in size from a few lines to over 200 lines. The programs implemented traditional data structures or algorithms (*e.g.*, red-black tree, AVL trees, quick sort, merge sort, arithmetic kernels). The Liquid Haskell annotations for each program encode the total functional correctness of each data structure or algorithm. A conforming implementation that passes the constraint-based type checking thus comes with significant correctness guarantees.

Fault Localization. To evaluate fault localization with respect to latent bugs, we seeded one defect in each program. Each seeded bug was introduced via a syntactic change. The bugs were seeded using local mutation operators common in mutation analysis (cf. [20]), such as replacing one arithmetic operation with another. Such mutations have been previously shown to be indicative of developer mistakes [13] and potentially as difficult to locate as natural human bugs [15].

To measure the correctness of fault localization, we require a ground truth notion of which lines should be considered when debugging the fault. One set of ground truth bug locations is induced by fault seeding: for any file, the location of the bug is just the location that was mutated. In addition, an expert[5] annotated a second set of ground truth locations based on manual inspection, without information about the fault seeding process. The two ground truths widely agree with each other, except that the expert is more lenient, sometimes giving several possible bug locations. This is because the human-judged "cause" of a bug is often ambiguous. The expert also annotated the difficulty of each benchmark. The labels "easy", "med", and "hard" subjectively indicate the distance between the locations mapped to the constraints that fail during type checking and the actual defect cause.

Given a ground truth annotation and the output of an algorithm, we measure accuracy using standard notions from statistics and information retrieval. Every location reported by the algorithm that is in the ground truth set is a true positive. Every location reported by the algorithm that is not is a false positive. Every location in the ground truth but not reported by the algorithm is a false negative.

Predicate Discovery. To evaluate predicate discovery, we constructed a set of benchmarks by considering Liquid Haskell programs that had been manually annotated and removing critical annotation components. The particular sort of information removed varies with the research question considered as well as the specific test case. Our predicate discovery algorithm is successful when Liquid Haskell can verify the result.

[4] Our benchmarks and results are available at *url removed for double-blind submission, contact PC chair*

[5] The expert is one of the co-authors of this paper.

	Ground Truth	True Positives	False Positives (t=0, 1, 2)
`minimizeWCC`	seeded	18	25 / 14 / 9
`minimize`	seeded	17	25 / 10 / 6
Vanilla	seeded	8	20 / 7 / 1
`minimizeWCC`	expert	21	24 / 13 / 9
`minimize`	expert	20	22 / 10 / 6
Vanilla	expert	10	19 / 6 / 1

Table 1. Fault localization accuracy. "False Positives (t=n)" counts the number of files with false positives (out of 28) for tolerance level t, where a file must have at least t false positives to count as having any. "True Positives" counts the number of files with true positives. The "Ground Truth" column indicates which of the two ground truth sets is used.

5.2 RQ1 — Fault Localization Accuracy

We measured the accuracy of our fault localization algorithm (in terms of true positives and false positives) with respect to two ground truth sets. We consider three algorithms: `minimizeWCC`, specialized to Horn Clause-based constraint type systems such as Liquid Haskell; `minimize`, applicable to any constraint-based type system; and the vanilla error reporting by Liquid Haskell,[6] a baseline representing the state-of-the-art.

The results are summarized in Table 1. Our fault localization algorithms find many more bug locations relative to both ground truths compared to vanilla reporting. Whereas vanilla reporting finds about a third of the bug locations, `minimizeWCC` and `minimize` both localize almost two-thirds of the defects each for both ground truths.

Vanilla reporting does return fewer false postives than either of the proposed algorithms at all tolerance levels. This is to be expected, since our algorithms return locations that are in some sense inconsistent with each other (since they are mapped to minimal unsatisfiable sets), not just locations from failing constraints. This allows our algorithms to find failure causes more often, but it also increases the number of false positives. However, while the number of false positives for both proposed algorithms is higher than vanilla reporting, the alarms themselves are quite reasonable in practice. For example, 11 of the 25 false positives from `minimizeWCC` were single spurious locations (as indicated by the difference between the $t = 1$ and $t = 0$ columns in Table 1, since a tolerance $t = 1$ "forgives" only a single false location), which do not represent a significant developer burden. Strikingly, for the "hard" benchmarks, vanilla reporting only found 1 of 9 bug locations (relative to the expert's ground truth) where our algorithms each found 6 out of 9.

5.3 RQ2 — Fault Localization Efficiency

Fault localization for constraint-based type systems operates at compile time. It is important for our algorithms to be efficient (i.e., have reasonable runtimes) because they are meant to be compile-time tools to help developers either find bugs in their program or to prove its correctness. As a result, localization must run rapidly at compile-time.

[6] Vanilla reporting returns the locations mapped to failing constraints for an unsatisfiable constraint set.

The mean running time for both fault localization algorithms is quite reasonable: 3.73 seconds for `minimizeWCC`, and 3.35 for `minimize`. Runtimes range from 0.004 seconds to 43.64 seconds for `minimizeWCC`, and from 0.12 seconds to 27.88 seconds for `minimize`. Note that runtimes do not necessarily correlate with program sizes: both algorithms processed the largest benchmark, red black trees, at around 6 seconds each. These numbers indicate reasonable scaling for larger programs: the time taken relates to the relevant partitions of the constraint graph, not the program overall.

5.4 RQ3 — Predicate Discovery Type Inference

We evaluated our algorithm's ability to discover types for intermediate variables enabling the verification of programs. We considered 21 microbenchmarks — programs that had previously been annotated with Liquid Haskell qualifiers sufficient to verify various correctness properties. We erased all of the qualifiers and applied our algorithm, noting the fraction of unannotated programs that could be proved correct. In all 21 cases our algorithm discovered types for intermediate variables that allowed verification — note that this included instances where Liquid Haskell alone would not have been able to verify the program. This is as expected, since our algorithm is correct by construction on such k-bounded instances. Our algorithm took a median of 0.1 seconds and a maximum of 7.1 seconds.

5.5 RQ4 — Predicate Discovery for Abstract Domains

We evaluated our algorithm's ability to expand abstract domains, yielding an improper subset a domain sufficient to prove correctness. We considered 23 microbenchmarks. These included programs that were not k-bounded for any k. In addition, we also evaluated on programs that were k-bounded, but for which we only permitted our algorithm $i < k$ unrolling. We ran our algorithm on each benchmark, noting the resulting domain expansion. In each case we determined if the domain expansion was sufficient to prove correctness (by running Liquid Haskell with the new $\mathbb{Q}$). In 22 of 23 cases our algorithm produces a sufficiently-rich abstract domain. The one failing case involved an recursive (*i.e.* not k-bounded for any k) constraint in an implementation of merge sort; in this case the domain expansion was missing only one qualifier (out of circa 100). Our algorithm took a median of 0.1 seconds and a maximum of 7.2 seconds.

5.6 Analysis

As expected, the set of bug locations that our `minimizeWCC` algorithm found is a strict superset of the set found by our general `minimize` algorithm. In particular, `minimizeWCC` found the bug location in every common file, plus the bug location in `pointDist`. The originally failing constraints (call them A and B) in `pointDist` are respectively the only vertices in their connected components and are both singleton minimal unsatisfiable sets. Constraint B is mapped to the bug location, while algorithm `minimize` returns A and thus does not report the bug location. By contrast, `minimizeWCC` returns both and thus implicates the correct location.

On the other hand, vanilla reporting found one bug location in `Record0` that the other algorithms did not. Let the failing constraints be A and B with the bug location mapped to A. `minimize` algorithm returns $\{B, C\}$ as a minimal unsatisfiable set, which does not implicate the correct location. However, since A and B are in the same weakly connected component, `minimizeWCC` returns the same locations and thus also fails to implicate the bug. (In this case even though A and B both fail, they do not constitute a minimal unsatisfiable set.) This example shows that `minimizeWCC` resolves some, but not all, cases of ambiguous constraint sets.

Our predicate discovery algorithm performed very well in practice. Its completeness is guaranteed in the k-bounded case, and in each of our 21 such benchmarks it found a set of intermediate types sufficient for verification. However, in the non-k-bounded cases we investigated it was very effective as well, learning abstract domain expansions that admitted the automatic verification of 22 out of 23 such benchmarks (even finding all but one necessary qualifier in the last case). While in theory our discovery algorithm need not apply to non-k-bounded cases, in practice it often does, especially in cases where some initial annotations are available. We view these results are very promising.

6. Related Work

There is a large literature on fault localization for languages with constraint-based (Hindley-Milner) type systems [7, 10, 25, 29]. We discuss two especially relevant prior approaches.

The SEMINAL tool by Lerner *et al.* [18] uses the OCaml type checker as an oracle in a search procedure to find well-typed programs that are syntactically similar to an input program that fails to type check, which are then used to construct helpful error messages. Our algorithm likewise uses the type checker as an oracle, but works on the set of constraints generated from the input program instead. Since the space of constraints is much smaller than the space of possible edits for a program, our algorithm can be more efficient than the SEMINAL tool without sacrificing the ability to find bugs.

Zhang and Myers [32] induce a labeled directed graph from a set of Hindley-Milner typing constraints and use Bayesian inference methods to analyze the graph and find likely bug locations. Our algorithm similarly constructs a graph from a set of Liquid Haskell typing constraints. While the approach is are effective for Hindley-Milner type systems, it is not clear how to extend it to more expressive type systems.

There is also a significant literature on predicate discovery [12]. Unno and Kobayashi [26] describe a procedure for inferring dependent (intersection) types using interpolants. Rümmer *et al.* [24] describes the theory of disjunctive interpolation in great detail. We show how to extend disjunctive interpolation to account for potentially recursive refinement typing constraints in order to automatically synthesize refinements for recursive, polymorphic and higher-order programs manipulating sophisticated data structures.

7. Conclusion

Refinement type systems hold out the promise of greater safety and correctness guarantees at compile-time. Unfortunately, using such rich type systems can be difficult, both because of bugs in program implementations and also because of bugs in program specifications.

We present a *fault localization* algorithm for the Liquid Haskell type system. Our algorithm uses the type checker as an oracle to find a minimal unsatisfiable constraint set, which is then mapped to a set of possible bug locations. We also present a *predicate discovery* algorithm that uses k-bounded unrolling and Craig interpolation to learn intermediate types as well as abstract domain expansions. These annotations allow implementations to be proved correct.

We evaluated our algorithms on benchmarks of Haskell programs with and without type annotations. Our fault localization algorithm produces minimal false positives (almost half of which are a single spurious location) and is efficient enough to be used at compile-time. It is much more effective at fault localization than the Liquid Haskell type checker, localizing twice as many bugs overall and finding six times times more "hard" bugs than the type checker. Our predicate discovery algorithm is correct by construction on k-bounded instances, finding annotations that admit program verification. Together, our two algorithms significantly reduce the barrier to entry for using refinement types systems.

References

[1] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45.

[2] R. Blanc, A. Gupta, L. Kovcs, and B. Kragl. Tree interpolation in Vampire. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *Lecture Notes in Computer Science*, pages 173–181. Springer Berlin Heidelberg, 2013.

[3] B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming Languages and Systems*, pages 264–278, 2006.

[4] A. Chlipala. An introduction to programming and proving with dependent types in Coq. *J. Formalized Reasoning*, 3(2):1–93, 2010.

[5] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22 (3):269–285, 1957.

[6] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[7] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*, pages 500–517, 2001.

[9] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277.

[10] H. Gast. Explaining ml type errors by data flows. In *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2005.

[11] C. V. Hall, K. Hammond, S. L. P. Jones, and P. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, Jan. 2004. ISSN 0362-1340. doi: 10.1145/982962.964021. URL http://doi.acm.org/10.1145/982962.964021.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[14] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. URL http://arxiv.org/abs/1011.1589.

[15] J. C. Knight and P. Ammann. An experimental evaluation of simple methods for seeding program errors. In *International Conference on Software Engineering*, pages 337–342, 1985.

[16] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34. ISSN 0164-0925.

[17] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symposium*, pages 218–234, 2005.

[18] B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *Proceedings of the 2006 workshop on ML*, pages 63–73. ACM, 2006.

[19] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[20] A. J. Offutt and K. N. King. A Fortran 77 interpreter for mutation analysis. *SIGPLAN Not.*, 22(7):177–188, July 1987. ISSN 0362-1340.

[21] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *International Conference on Object Oriented Programming Systems Languages & Applications*, pages 525–542, 2014.

[22] N. Polikarpova and A. Solar-Lezama. Program synthesis from polymorphic refinement types. URL http://arxiv.org/abs/1510.08419.

[23] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[24] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *Computer Aided Verification*, pages 347–363, 2013.

[25] F. Tip and T. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):5–55, 2001.

[26] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 277–288, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-568-0. doi: 10.1145/1599410.1599445. URL http://doi.acm.org/10.1145/1599410.1599445.

[27] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282. ACM, 2014.

[28] Visual Studio. Using SAL annotations to reduce C/C++ code defects. Technical report, Microsoft Developer Network, 2015. URL https://msdn.microsoft.com/en-us/library/ms182032.aspx.

[29] M. Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–43. ACM, 1986.

[30] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 249–257. ISBN 0-89791-987-4.

[31] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering–ESEC/FSE99*, pages 253–267. Springer, 1999.

[32] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *ACM SIGPLAN Notices*, volume 49, pages 569–581. ACM, 2014.