Finding and Fixing Bugs in Liquid Haskell

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Science)

by

Anish Tondwalkar

May 2016

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 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 LiquidHaskell 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.

Approval Sheet

This thesis is submitted in partial fulfillment of the

requirements for the degree of

Master of Science (Computer Science)

Anish Tondwalkar

This thesis has been read and approved by the Examining Committee:

Thesis Adviser

Committee Chair

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

May 2016

Contents

List of Figures

List of Tables

Chapter 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 [12, 22]. The success of standard type systems has led to significant research on more expressive notions of typing, including type qualifiers [4, 19], and dependent and refinement types [33, 18, 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.*, [7]) have made such systems useful and feasible in practice [26, 31, 30]. These constraint languages restrict the inferred types to those drawn from some *abstract domain*. Refinement type systems allow stronger safety guarantees [33, 10, 2] that are potentially useful for many applications, including security [2], and program synthesis [25]. 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 specification), 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 [5]. Previous approaches have been successful at fault localization for classical Hindley-Milner style type checking [20, 35, 24]. 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 [9].

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 [34], originally designed to find regressions in codebase changes, to make our approach practical.

Our *predicate discovery* algorithm uses disjunctive interpolation [27], a general-

ization of Craig interpolation [6], to automatically expand the abstract domain used by the constraint solver by inferring predicate templates that serve as the refinement types of program expressions. Instead of restricting an abstract domain to make type inference decidable, we approximate infinite recursion using a bounded unrolling algorithm (Section 5.4). In the case that the type constraints admit a k-bounded unrolling (as described in Chapter 5), our algorithm finds all intermediate types necessary for program verification.

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 framework [26, 30]. 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 Liquid Haskell 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 typechecking.

- 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 without the empirical input of the LiquidHaskell type checker

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 for 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.

Chapter 2 provides background on the Liquid Haskell programming language, and the Liquid Types framework. Chapter 3 gives an overview of the work couched in the language of Liquid Types, and contains short previews of the two algorithms described in this work. Chapter 4 describes in detail our fault localization algorithm, in the case of Liquid Haskell, as well as a more general case with weakened assumptions. Chapter 5 describes our predicate discovery algorithm, the regime in which is it also a complete type inference algorithm, and its applicability to Liquid Haskell. Chapter

6 presents and experimental evaluation supporting our claims and evaluating our effectiveness We discuss related work in Chapter 7 and future work and conclusions in Chapter 8.

Chapter 2

Background: 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. In this work, we will write Liquid Haskell, in body text for the name of the programming language and the system, but LiquidHaskell in small capitals for the typechecking algorithm.

Liquid Haskell comes equipped with a default *abstract domain*, predicate templates that can be filled in with program variables. This default abstract domain is useful for verifying common operations in practice — indeed it is determined empirically, based on its ability to verify a suite of benchmark programs — but is not sufficient to prove all programs correct.

Liquid Haskell uses the Liquid Types [26] framework to infer refinement types, which greatly reduces the annotation burden for users. The syntax of Liquid Haskell refinement type annotations is described in Figure 2.1, while simplified inference rules for subtyping constraint generation is given in Figure 2.2. Liquid Types subtyping

$$
\begin{array}{rrcl}
\textbf{\textit{Basic Types}} & b & ::= & \alpha \mid x : \tau \to \tau \mid C\ \bar{\tau}\ \bar{r} \mid \tau\ \tau \\
\textbf{\textit{Types}} & \tau & ::= & \{v : b \mid r\} \mid Cl\ \bar{\tau} \\
\textbf{\textit{Abstract Refinements}} & \pi & ::= & \forall\, \langle p \colon \tau \rangle\, .\pi \mid \tau \\
\textbf{\textit{Type Schemata}} & \sigma & ::= & \forall \alpha.\sigma \mid \pi \\
\textbf{\textit{Refinements}} & r & ::= & (ar, cr) \\
\textbf{\textit{Abstract Refinements}} & ar & ::= & [] \mid p\ \bar{e}, ar \\
\textbf{\textit{Concrete Refinements}} & cr & ::= & k[e/x] \mid pr \mid cr \wedge cr \\
\textbf{\textit{Predicates}} & pr & ::= & true \mid false \mid \bigwedge\ \bar{pr} \mid \bigvee\ \bar{pr} \mid \neg pr \\
 & & \mid & pr \Rightarrow pr \mid pr \Leftrightarrow pr \mid e \mid e\,[=|\neq|>|<|\geq|\leq]\,e \\
\textbf{\textit{Expressions}} & e & ::= & c \mid n \mid x \mid c\ \bar{e} \mid \text{if } pr \text{ then } e \text{ else } e \\
 & & \mid & e\,[+ \mid - \mid * \mid / \mid \%]\,e
\end{array}
$$

Figure 2.1: Syntax of Liquid Haskell

constraints have the general form

$$\Gamma \vdash \{\nu : B \mid e_1\} <: \{\nu : B \mid e_2\}$$

where Γ is an *environment*, and e_1, e_2 are both expressions — either predicative type variables or formulae from the refinement logic. An environment is a list of bindings, which have the general form $x : \{\nu : \tau \mid e\}$, where τ is a base type, and e is an expression. Here, $\tau_1 <: \tau_2$ is the subtyping relation, read "τ_1 is a subtype of τ_2."

Intuitively, a refinement type like the one above specifies the set of all values within the base type that satisfy some expression. For example, $\{\nu : \text{Int} \mid \nu \geq 0\}$ represents nonnegative integers. A predicate variable κ, with optional pending substitutions θ (usually written with right juxtaposition as in $\kappa\theta$) represents an unknown refinement type; the pending substitution θ intuitively represents the arguments in a function application. For example, $\{\nu : \text{Int} \mid \kappa[y/x - 1]\}$ can denote the refinement type of the

output of a function given an argument $x - 1$ for a formal parameter y. Intuitively, subtyping constraints capture requirements that program expressions must satisfy.

For example, a subtyping constraint of the form

$$\ldots \vdash \{\nu : \text{Int} \mid \kappa\} <: \{\nu : \text{Int} \mid \nu \neq 0\}$$

encodes the constraint that a variable with the refinement κ must be nonzero. This may be used to statically prevent a division-by-zero error, by enforcing this safety property on the second argument of a call to the `div` function.

Liquid Types is built atop a Hindley-Milner style typing system: after an Hindley-Milner oracle determines the "shapes" of the types of program expressions, the Liquid Types constraint solver attempts to find a solution to the fresh predicate variables (the κ variables) introduced. A solution maps each variable to a conjunction of predicates. These predicates come from a set of *qualifiers* — predicate templates that can be filled in program variables — drawn from our abstract domain. The Liquid Types solver finds the strongest — most specific — solution for the predicate 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. If this is the case, the LiquidHaskell algorithm returns the Horn queries (as defined in Chapter 5) that were unable to be satisfied.

$$\frac{(x, \{v : b \mid e\}) \in \Gamma}{\Gamma \vdash x \uparrow \{v : b \mid e \wedge x = v\}; \emptyset} \qquad \frac{(x, \sigma \in \Gamma) \quad \sigma \neq \{v : b \mid e\}}{\Gamma \vdash x \uparrow \sigma; \emptyset}$$

$$\frac{}{\Gamma \vdash c \uparrow \{v : ty(c) \mid v = c\}; \emptyset}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 \uparrow \tau_1; C_1 & \Gamma \vdash e_2 \downarrow \tau_x; C_2 \\ (\tau_1', C_p) = freshPreds(\Gamma, \tau_1) & x : \tau_x \rightarrow \tau = \tau_1'\end{array}}{\Gamma \vdash e_1\ e_2 \uparrow \tau[e_2/x]; (C_1, C_2, C_p)}$$

$$\frac{\Gamma \vdash e \uparrow \sigma; C}{\Gamma \vdash [\Lambda\alpha]e \uparrow \forall\alpha.\sigma; C}$$

$$\frac{\Gamma \vdash e \uparrow \sigma; C}{\Gamma \vdash Tick\ t\ e \uparrow \sigma; C}$$

Figure 2.2: Simplified Liquid Types constraint generation

Chapter 3

Overview of Proposed Algorithms

This chapter gives an overview of the work. We then highlight key features of our algorithms and present motivating examples demonstrating their effectiveness. Section 3.1 gives an overview of our fault localization algorithm, along with some intuition as to how and why it works, along with a worked example. Section 3.2 gives an overview of our predicate discovery algorithm, demonstrating bits of the pipeline on representative code fragments.

Formal details of these algorithms previewed here are given in Chapter 4 and Chapter 5, respectively.

3.1 Fault Localization Algorithm Preview

Figure 3.1 diagrams the architecture of our fault localization algorithm. We take as input 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.* For example, in the process of Hindley-Milner typchecking, we might encounter the inconsistency that function requiring an `Int` parameter when its actual argument is a `String`. 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. *Minimal inconsistent sets explain bugs.* 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.

It is not feasible to enumerate all constraint subsets to find a minimal unsatisfiable subset. As a result, we exploit properties common to constraint-based type systems to

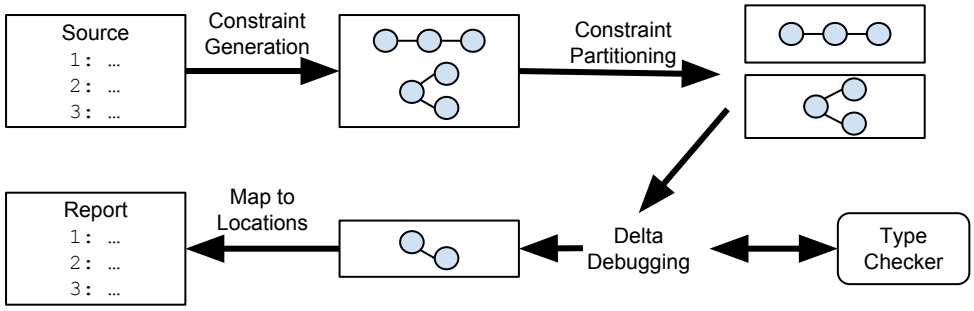

Figure 3.1: Fault localization algorithm architecture. We use the type checker as an oracle for a search procedure to find a minimal unsatisfiable constraint set.

adopt the automated delta debugging algorithm, which efficiently finds subsets [34]. We outline the properties needed for delta debugging to work, and how our algorithm fulfills those requirements, in Chapter 4.

Consider the following program with Liquid Haskell types:

```
1 div2 :: Int -> { v:Int | 0 < v } -> Int
2 div2 n d = n `div` d
3
4 nonzero d = 0
5
6 example n d = n `div2` (nonzero d)
7             + 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 is generated from this program. The Liquid Haskell type checker returns the locations mapped to the failing constraints, which correspond only to the application of the function `nonzero` but not its body. By contrast, the minimal constraint set found by our algorithm corresponds to the positive argument

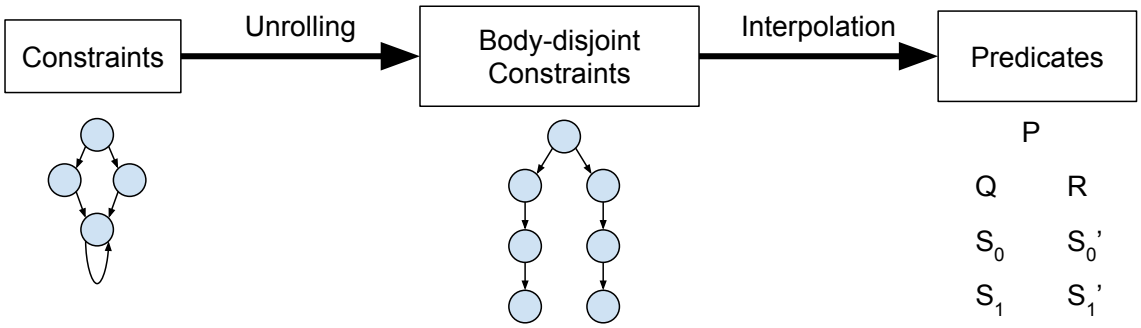

Figure 3.2: Predicate discovery architecture diagram. We find an interpolant for every node the unrolled induced graph.

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 Chapter 4.

3.2 Predicate Discovery Algorithm Preview

Our predicate discovery algorithm takes as input a set of typing constraints generated from a Haskell program annotated with refinements that correspond to safety properties that functions must satisfy. Our algorithm converts these constraints into constrained Horn clauses, from which an bounded query tree is generated. Our algorithm then transforms this tree into a set of interpolation problems, and using an interpolation oracle we compute interpolants that can be transformed into predicate templates that constitute an abstract domain over which the program can be verified.

Consider the following Haskell program:

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
```

```
3   where
4     go k
5       | k <= 0    = 0
6       | otherwise = let s = go (k-1) in s + k
```

This program, which computes the sum of the first k natural numbers, is required to produce output that is not less than its input per the refinement type specification. LiquidHaskell requires as input qualifiers (*i.e.* predicate templates) to verify that safety property. Our discovery algorithm can compute a set of qualifiers $\{\lambda\nu.\nu \geq 0,\ \lambda\nu.\lambda k.\nu \geq k\}$. The Liquid Types constraint solver can then use this abstract domain to infer that the output of `sum` has the refinement type $\{\nu : \text{Int} \mid \nu \geq \text{k} \wedge \nu \geq 0\}$, which indeed conforms to the refinement annotation — the postcondition.

Figure 3.2 diagrams the architecture for our predicate discovery algorithm. We take as input a system of subtyping 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. This happens at least when our system of constraints admits k-bounded unrolling, as detailed in Chapter 5.

- *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 (possibly improper) subset of one that does.

Consider the following Haskell listing:

```
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
8                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 of a program is undecidable in general, so, inspired by bounded model-checking, we introduce the unrolling approximation in Chapter 5.

$$\Gamma; x : \kappa_x \vdash \{v = x + 1\} <: \kappa_1$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\} \vdash \kappa_n <: \kappa_x$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\}; a : \kappa_1[x := n] \vdash Int <: \kappa_{let}$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\} \vdash \kappa_{let} <: \kappa_{if}$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{False}\} \vdash \{v : Int \mid v = 1\} <: \kappa_{if}$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\} \vdash \kappa_{if} <: \kappa_{iflet}$$

$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\}; a : \kappa_1[x := n] \vdash \kappa_n <: Int$$
$$\Gamma; x : \kappa_x \vdash \kappa_x <: Int$$
$$\Gamma; x : \kappa_x \vdash \{v : Int \mid v = 1\} <: Int$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\}; a : \kappa_1[x := n] \vdash \kappa_1[x := n] <: \{v : Int \mid \neg(v = 0)\}$$

Figure 3.3: List of constraints for `divTenBy`

$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\}; a : \kappa_1[x := n] \vdash \kappa_1[x := n] <: \{v : Int \mid \neg(v = 0)\}$$
$$\Gamma; x : \kappa_x \vdash \{v = x + 1\} <: \kappa_1$$
$$\Gamma; n : \kappa_n; b : \{\text{Prop } v \leftrightarrow 0 \leq n\}; b : \{v = \text{True}\} \vdash \kappa_n <: \kappa_x$$

Figure 3.4: Constraints in the relevant weakly connected component

This program gives us the Horn clause constraints shown in Figure 3.3. Note that Γ contains the bindings provided by the standard library for functions such as `div`, `(+)`, and `(<=)`. In particular, we want to focus on those separated out for consideration in Figure 3.4. The first constraint of this set, c, is the one that fails when classic LH typechecking is performed over an empty abstract domain, and the remaining constraints of the set are those upon which it depends. (That is, are in the weakly connected component as c)

These constraints correspond to the intermediate type annotations needed to show

that we do not perform division by zero.

We unroll the implicated failing constraints and serialize the resulting graph, obtaining the following failure derivation:

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

This predicate is unsatisfiable because there exists no unsafe assignment of values to variables. Note that $b_1 \ldots b_3$, refer to the variable `b` at different program locations, much like renumbering in conversion to SSA form, and that we use line breaks here to show where we've brought in a dependent constraint. This predicate encodes the necessary conditions for program correctness, but it is difficult to manually extract refinement types for program variables from such predicate form [16]. We use Craig interpolation [6] to automatically extract refinement types:

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

Performing Craig interpolation at each point, we see the information needed at that

point to prove the rest of the failure derivation. At several point it just gives us the interpolant `true`, indicating that points before that one encode only dead constraints, "dead" in the same sense as "dead code." It 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.

Chapter 4

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. The intuition behind our `minimize` algorithm is threefold. First, constraints capture inconsistencies: we analyze the induced constraints rather than the text of the program. Second, minimal inconsistent sets explain bugs: we find a *minimal unsatisfiable constraint set* using the existing type checker as an oracle. Third, minimal explanations implicate relevant locations: we map those minimal constraints back to program locations.

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 in Section 4.1. 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 in Section 4.2. Finally, we describe an optimization motivated by the structure of the constraints further improves the accuracy of the minimization procedure in Section 4.3).

4.1 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 [34]. 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.

To do so soundly, we detail the delta debugging algorithm and its four key assumptions (presence, consistency, monotonicity, and unambiguity) and then indicate

how our use of it satisfies each of those assumptions.

We thus make a design decision between placing additional requirements on the constraints (and finding a minimal set of special items efficiently) or relaxing our requirements on the output (and finding an almost-minimal set of general items efficiently); for general fault localization we adopt the latter approach.

While assuming additional structure about the constraints (e.g., that they come equipped with a total ordering related to implication) could allow for a more efficient determination of minimal subsets (e.g., analogous to binary search's efficient membership testing given the assumption that the input is sorted), such additional structure may be not be provided by many constraint-based typing systems.

Tests and Configurations. A *configuration* Δ (*e.g.* diff, patch, commit, *etc.*) is a set of *changes* $\{\delta_1, \ldots, \delta_n\}$ 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:

- ***Interesting:*** A configuration Δ is *interesting* if it fails the test. Delta debug-

ging assumes that the input configuration Δ is interesting:

$$\begin{aligned} \text{test}(\Delta) &= \times \quad \text{(the original input fails/is interesting)} \\ \text{test}(\emptyset) &= \checkmark \quad \text{(removing everything does not fail/is uninteresting)} \end{aligned}$$

- ***Consistent:*** A configuration Δ is *consistent* if

$$\forall \Delta' \subseteq \Delta. \texttt{test}(\Delta') = \checkmark \vee \texttt{test}(\Delta') = \times$$

That is, in a consistent configuration, any subset of code changes either fails or passes the `test`.

- ***Monotonic:*** A configuration Δ is *monotonic* if

$$\forall \Delta'' \subseteq \Delta' \subseteq \Delta. \text{ if } \texttt{test}(\Delta'') = \times \text{ then } \texttt{test}(\Delta') = \times$$

That is, in a monotonic configuration, any superset of a failing (or interesting) set is also failing (or interesting).

- ***Unambiguous:*** A configuration Δ is *unambiguous* if

$$\forall \Delta_1, \Delta_2 \subseteq \Delta. \text{test}(\Delta_1) = \times \wedge \text{test}(\Delta_2) = \times \implies \text{test}(\Delta_1 \cap \Delta_2) = \times$$

This condition limits attention to one failure-inducing 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. 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 $\text{test}(A \cap B) \neq \times$.

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

$$\begin{aligned} \text{test}(\Delta') &\neq \checkmark \quad (\Delta' \text{ is a failing configuration}) \\ \forall c \in \Delta'.\, \text{test}(\Delta' \setminus \{c\}) &\neq \times \quad (\Delta' \text{ is one-minimal}) \end{aligned}$$

A one-minimal subset can be computed via polynomial (in the size of Δ) queries to `test` [34, Alg. 1].

Algorithm 1 Delta Debugging

```
procedure DELTADEBUG(Δ, r)
    (Δ1, Δ2) ← split_in_half(Δ)
    if | Δ |= 1 then                                   ▷ Min-Unsat set is Δ
        return Δ
    else if test(Δ1 ∪ r) = × then                      ▷ Min-set in Δ1
        return DeltaDebug(Δ1, r)
    else if test(Δ2 ∪ r) = × then                      ▷ Min-set in Δ2
        return DeltaDebug(Δ2, r)
    else                                               ▷ Min-set in Δ1 ∪ Δ2
        return DeltaDebug(Δ1, r ∪ Δ2) ∪ DeltaDebug(Δ2, r ∪ Δ1)
    end if
end procedure
```

4.2 General Constraint Minimization

From Rondon *et al.* [26], 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 `SAFE`

if the constraints are ($\mathbb{Q}$-)*satisfiable* or `UNSAFE` 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 $C = \{c_1, \ldots, c_n\}$ generated from the (buggy) program. The `test` oracle is defined as:

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

A *minimal unsatisfiable constraint set* C is one such that: (1) $\texttt{solve}(C) = \texttt{UNSAFE}$, and (2) $\forall C' \subset C.\, \texttt{solve}(C') = \texttt{SAFE}$. 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 `SAFE` or `UNSAFE`. That constraint sets are *monotonic* is a corollary of Theorem 2 in Rondon *et al.* [26]. 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 Chapter 6.6).

Algorithm. Our delta-debugging based constraint minimization (and hence, fault localization) algorithm is shown in Algorithm 2. 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 2 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' ← DeltaDebug(C, ∅)
3:     locs ← ⋃ map(consLoc, C')
4:     return locs
5: end procedure
```

4.3 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 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 $\texttt{WCC}(C)$ 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 3).

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 Chapter 6.6, where `minimizeWCC` localizes a bug that `minimize` does not.

Algorithm 3 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: $\texttt{WCC}(C)$ returns the dependency partitions of C
Require: $\texttt{RelWCC}(C) = \{p \mid p \in \texttt{WCC}(C) \ \wedge \ \exists c \in p.\, c \in F\}$

1: **procedure** MINIMIZEWCC(C)
2: $Cs' \leftarrow \{p' \mid p \in \texttt{RelWCC}(C) \ \wedge \ \texttt{solve}(p) = \texttt{UNSAFE} \ \wedge \ p' = \texttt{DeltaDebug}(p, \emptyset)\}$
3: $locs \leftarrow \bigcup\bigcup \text{map}(\texttt{consLoc}, Cs')$
4: **return** *locs*
5: **end procedure**

Chapter 5

Automatically Synthesizing Qualifiers

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.

LiquidHaskell fails to verify some programs because it lacks the correct set of (user provided) qualifiers that are needed to synthesize appropriate refinement types.

We infer qualifiers by transforming typing constraints into a Horn clauses, on which we perform a bounded unrolling, yielding logical formulas that are *interpolated* to yield suitable qualifiers, and hence refinement types.

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.

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.

We present our algorithm in four stages. First, Liquid type constraints are generated from the input annotated Liquid Haskell program. Second, we transform those Liquid type constraints into a system of Horn clauses. Third, because recursive Horn constraints cannot generally be solved, we approximate the problem via k-bounded unrolling the cyclic system of constraints into a tree-structured formula. Fourth, we compute a disjunctive interpolant (conceptually, an explanation of why the refinement type annotations are not violated) for the tree-structured formula. We then use the interpolant to map κ-vars to formulas that make the original Liquid Types constraints valid.

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 refinement logic.

5.1 Refinement Types

Refinement types encode invariants by decorating types with predicates that constrain the values described by that type. We represent input programs as 3-tuples (Δ, Σ, Θ), where Δ is a set of type variables, $\Sigma : \Delta \rightarrow 2^T + \bot$, assigns to each type variable a set of predicates from our underlying theory or $\bot$, and Θ is a set of Horn constraints.

The interpretation of $\perp$ 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 refinement type system, such as Liquid Haskell, can verify the correctness of programs (Δ, Σ, Θ) over an abstract domain $\mathbb{Q}$ by first inferring type annotations and then checking them. We use ι to denote *type inference*, with $\iota(\Delta, \Sigma, \Theta, \mathbb{Q}) \mapsto \Xi$, where $\Xi : \Delta \to 2^T$. We use σ to denote *refinement type checking*, with $\sigma(\Delta, \Xi, \Theta) : \{\text{SAFE}, \text{UNSAFE}\}$. In this formulation, $L(\Delta, \Sigma, \Theta, \mathbb{Q}) : \{\text{SAFE}, \text{UNSAFE}\}$, formed by the composition of the two, takes a partially-annotated program and an abstract domain, over which it solves the system of induced constraints, deciding if the program is safe or unsafe.

Our predicate discovery algorithm D runs before type inference, with $D(\Delta, \Sigma, \Theta) \mapsto \Xi$. Given unrefined types Δ and an annotation partial mapping Σ, it returns an assignment of predicates to type variables that satisfy our system of constraints Θ. That is, it takes an unrolled constraint tree and 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.

5.2 Liquid Haskell to Liquid Type Constraints

The first stage of our algorithm uses the standard LiquidHaskell constraint generator [26, Fig. 4] to generate Liquid Type constraints from an input partially-annotated

LiquidHaskell source program. See Chapter 2 for a description of Liquid Type constraints.

Running Example. We illustrate all stages of our algorithm with the function `sum`:

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0    = 0
6       | otherwise = let s = go (k-1) in s + k
```

This program, which computes the sum of the first `k` natural numbers, is required to produce output that is not less than its input (per the refinement type specification). LiquidHaskell requires user-provided qualifiers (predicate templates) to verify that safety property.

The following relevant Liquid Types constraints are generated from `sum`[1]:

$$\mathtt{k} : \kappa_k, \mathtt{k} \leq 0 \vdash \{\nu = 0\} <: \kappa_s \tag{5.1}$$

$$\mathtt{k} : \kappa_k, \neg(\mathtt{k} \leq 0), \mathtt{s} : \kappa_s[\mathtt{k}/\mathtt{k}-1] \vdash \{\nu = \mathtt{s} + \mathtt{k}\} <: \kappa_s \tag{5.2}$$

$$\emptyset \vdash \mathtt{Int} <: \kappa_k \tag{5.3}$$

$$\mathtt{k} : \kappa_k \vdash \kappa_s <: \{\nu \geq \mathtt{k}\} \tag{5.4}$$

Intuitively, κ_k and κ_s denote the *unknown* refinements of the input and output type of `go`. Constraints 5.1 and 5.2 encode that the two branches of the conditional expression must both be subtypes of the ultimate output type.

[1] We abbreviate $\{\nu : \tau \mid e\}$ below as $\{e\}$, where $\tau = \text{Int}$.

Constraint 5.3 and 5.4 encode the pre-condition and post-condition on κ_k and κ_s given by the refinement type specification on `sum`. Note that `s` is the output of `go` with argument `k-1`; as denoted by the substitution in the constraint 5.2. Also note that 5.2 is *cyclic* as κ_s appears on *both* sides of the implication.

5.3 Liquid Type Constraints to Horn Clauses

The second stage of our algorithm transforms a Liquid Types subtyping constraint into a Horn clause with pending substitutions. We make use of the Liquid Types notion of pending substitutions [26, Sec. 4.1], which correspond to the arguments passed to a possibly-recursive function. Our transformation is recursively defined on the structure of the Liquid Type constraints and retains these pending substitutions so that they can be applied.

Horn Clauses A *Horn Clause* is of the form $B \rightarrow H$ with a *body* B and a *head* that belong to the grammar:

$$\begin{array}{lrcl} \textbf{Atom} & A & ::= & \kappa\theta \mid e \\ \textbf{Body} & B & ::= & A \wedge B \mid \mathit{true} \\ \textbf{Head} & H & ::= & A \end{array}$$

An atom is either an uninterpreted symbol κ with pending substitutions θ, or an expression from an underlying (for our purposes, SMT-decidable) theory. A Horn clause is an implication from a body to a head, where the head is an atom, and the body is a conjunction of atoms.

Encoding Constraints We now define a function $\mathsf{Enc}(\cdot)$ that transforms a Liquid Type constraint into a constrained Horn clause with pending substitutions. Sub-

typing constraints have the general form $\Gamma \vdash \{\nu : B \mid e_1\} <: \{\nu : B \mid e_2\}$, where Γ is an *environment* and e_1, e_2 are both expressions, which can either be κ-vars or formulae from the refinement logic. An environment is a list of bindings. Bindings have the general form $x : \{\nu : \tau \mid e\}$, where e is an expression in our predicate calculus. We define our $\mathsf{Enc}(\cdot)$ function which encodes constraints as follows:

$$
\begin{aligned}
\mathsf{Enc}(\Gamma \vdash T_1 <: T_2) &= \mathsf{Enc}(\Gamma) \wedge \mathsf{Enc}(T_1) \rightarrow \mathsf{Enc}(T_2) \\
\mathsf{Enc}(\Gamma) &= \bigwedge_{b \in \Gamma} \mathsf{Enc}(b) \\
\mathsf{Enc}(x : \{\nu : \tau \mid e\}) &= \mathsf{EncExpr}(e, x) \\
\mathsf{Enc}(\{\nu : \tau \mid e\}) &= \mathsf{EncExpr}(e, \nu) \\
\mathsf{EncExpr}(e, x) &= \begin{cases} \kappa(x)\theta & e = \kappa\theta \\ e[\nu/x] & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that the environment is encoded as a conjunction of recursively-encoded bindings and guard predicates. Bindings are encoded as expressions. If the expression is an uninterpreted symbol variable known as a κ-variable then we encode it as a call to a new undefined predicate. Otherwise the expression is augmented with a substitution that captures the constrained value of the variable.

Example. The Liquid Type constraints for `sum` are transformed into the following set of Horn clauses[2]:

[2]Note that we assume $\kappa_k = \texttt{true}$, which means that `sum` can take any integer as an argument; we elide this part of the transformation below for brevity.

$$\kappa_k(k) \wedge \mathtt{k} \leq 0 \wedge \nu = 0 \rightarrow \kappa_s(\nu) \quad (5.1)$$

$$\kappa_k(k) \wedge \mathtt{k} > 0 \wedge \kappa_s(\mathtt{s})[\mathtt{k}/\mathtt{k}-1] \wedge \nu = \mathtt{s} + \mathtt{k} \rightarrow \kappa_s(\nu) \quad (5.2)$$

$$\mathit{true} \rightarrow \kappa_k(\nu) \quad (5.3)$$

$$\kappa_s(\nu) \rightarrow \nu \geq \mathtt{k} \quad (5.4)$$

Note the one-to-one mapping between constraints and Horn clauses as well as the pending substitution in (2), now applied to Horn clause logical formulae rather than Liquid Type constraints. Finally, note that in each case we transform κ-variables into predicate symbols (we retain the same names for clarity).

Horn Queries A Horn clause $B \rightarrow H$ is a *query* if H is not a κ-var. In this domain, a query enforces the safety property given in a refinement type annotation. Note that the safety specification ensures that encoding always produces a query: for example (5.4), enforces a lower bound the result of `sum`.

5.4 Unrolling Cyclic Horn Clauses

The transformation above converts Liquid subtyping constraints into constrained Horn clauses (CHC) with pending substitutions. Solving for a set of CHCs amounts to finding an interpretation (*i.e.* predicates) for the predicate symbols that make all of the clauses in the set valid. In this domain, the predicate symbols are κ-vars, and a solution to a κ-var corresponds to the inferred refinement type of a program expression. In the `sum` example, the solution to κ_s corresponds to the inferred refinement type of `sum` that satisfies the postcondition given in the refinement type annotation.

There is an extensive literature on interpolation-based methods to solve a set of CHCs. However, we cannot directly use such methods to find solutions for κ-vars

because of the presence of recursive constraints. To make such methods applicable, we introduce a *bounded query function* that constructs a labelled tree-structured formula that corresponds to a disjunctive interpolation problem (see §5.5). This formula is a failure derivation for a Horn query: its satisfiability corresponds to the violation of the safety property encoded by the query. Because Liquid Types constraints can be possibly recursive, and the corresponding problem of solving for recursive CHCs is undecidable in general [27], we create a recursion-free approximation by finitely "unrolling" a set of CHCs with pending substitutions.

Some preliminaries are needed before we can define the bounded query function.

Dependencies Let HC be a set of Horn clauses. The *clauses defining* κ, written HC_κ are the set of clauses in HC whose heads are of the form $\kappa\theta$. We write $\mathsf{Dep}(\cdot)$ for the *set of* κ*-variables* on which an atom, head, or body depends as defined as:

$$\begin{aligned}
\mathsf{Dep}(\kappa(x)\theta) &= \{\kappa\} \cup \bigcup_{B \to \kappa\theta \in HC} \mathsf{Dep}(B) \\
\mathsf{Dep}(e) &= \emptyset \\
\mathsf{Dep}(A \wedge B) &= \mathsf{Dep}(A) \cup \mathsf{Dep}(B)
\end{aligned}$$

Recursive Dependencies A variable κ is recursive if it depends upon itself, *i.e.* $\kappa \in \mathsf{Dep}(\kappa)$. A body B is *recursive* if it contains a recursive κ. Let NHC_κ be the subset of bodies in HC_κ that are not recursive.

Bounded Dependencies The D-bounded dependencies of κ are defined as

$$\mathsf{BDep}(\kappa, i, D) = \begin{cases} HC_\kappa & \text{if } i < D \\ NHC_\kappa & \text{otherwise} \end{cases}$$

Labeled Tree dd a comment to this line We represent the bounded query as a labeled AND-OR tree. Given a set of *labels* L,

$$\begin{aligned} \text{AndOr} ::= &\ \text{And}(\text{Expr}, \{\text{AndOr}\}) \\ &\mid \text{Or}(\{\text{AndOr}\})_L \end{aligned}$$

where Expr is an expression. We will ensure that each label L appears *at most once* in each labeled tree.

Bounded Queries The function $\mathsf{BQ}(B \to H, D)$ constructs a bounded query tree for a query constraint. It makes recursive calls to the function $\mathsf{BQ}(B, m, D)$ (resp. $\mathsf{BQ}(A, m, D)$), which takes as input a body B (resp. atom A) a depth D and an auxiliary map m that tracks, for each κ-variable, the depth that it has already been unfolded to, and returns as output a *labeled* formula corresponding to the body (resp. atom). This function is defined in Figure 5.4

We then rename instances of x in p to avoid capture, indexing each x by the unrolling depth. This entire process is analogous to loop unrolling with translation to SSA form as in bounded model checking: $x^{(n)}$ captures the value of the x after n recursive calls. In what follows, we represent this by the sequence $x, x', x'', \ldots$

Example. If we let $K = 2$, the query for constraint (3) in `sum` is given in Figure 5.2. The branches correspond to disjunctions. Notice that we had to rename `s` to `s'` in some nodes to avoid to avoid capture.

$$
\begin{aligned}
\mathsf{BQ}(B \to H, D) &= \mathsf{And}(\neg H \wedge \mathsf{head}, \mathsf{children}) \\
\mathsf{BQ}(B, m, D) &= \mathsf{And}(\mathsf{head}, \mathsf{children}) \\
\text{where } \mathsf{head} &= \wedge_{b \in \mathrm{BE}(B)} \mathsf{BQ}(b, m, D) \\
\mathsf{children} &= \cup_{b \in \mathrm{BH}(B)} \{\mathsf{BQ}(b, m, D)\} \\
\mathsf{BQ}(\kappa(x)\theta, m, D) &= \mathsf{Or}(\cup_{B \in B_s} \mathsf{BQ}(B\theta[\nu/x], m', D))_L \\
\text{where } m' &= m[\kappa \mapsto m(\kappa) + 1] \\
B_s &= \mathsf{BDep}(\kappa, m(\kappa), D) \\
L &= \text{A fresh label} \\
\mathsf{BQ}(e, m, D) &= e \\
\mathrm{BE}(B_1 \wedge \ldots \wedge B_n) &= \{B_i \mid B_i \text{ is not a } \kappa\text{-var}\} \\
\mathrm{BH}(B_1 \wedge \ldots \wedge B_n) &= \{B_i \mid B_i \text{ is a } \kappa\text{-var}\}
\end{aligned}
$$

Figure 5.1: Bounded Queries

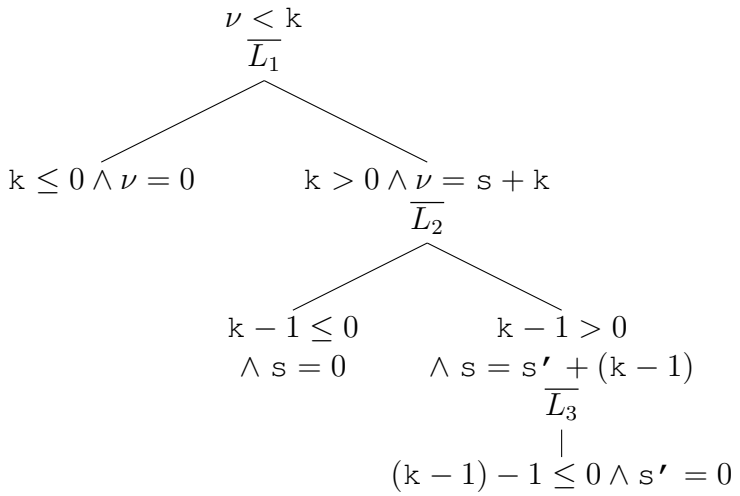

Figure 5.2: The unrolled failure derivation for constraint (3) of `sum` with unrolling depth $D = 2$. Each branch corresponds to a disjunction. The labels for subformulas are shown overlined at the top of each node.

5.5 Interpolation and Solutions to κ-vars

A bounded query tree represents multiple failure derivations for a query constraint, with each failure derivation representing a distinct tree interpolation problem, which is a generalization of Craig interpolation [3]. Tree interpolants can be used to compute solutions to Horn clauses, as in the DUALITY algorithm of [21]. We use tree interpolants here in a similar manner.

The bounded query formulas correspond to a disjunctive interpolation problems [27].

Sub-formulas at a Label Recall that each label L appears at most once in a formula e. Let $e \downarrow L$ be the (unique) subformula of e that appears under label L *i.e.*

$$e \downarrow L = e' \text{ where } (e')_L \text{ occurs in } e$$

The *children* of a label L (in a formula e) are the labels L' that appear in $e \downarrow L$. Let $e[L/g]$ correspond to substituting the subformula of e under L with formula g. For example, in the formula e corresponding to Figure 5.2, $e \downarrow L_1 = \mathtt{k} \leq 0 \wedge \nu = 0$.

Disjunctive Interpolants Formally, given a formula f with location labels that only occur under disjunction and conjunction, a *disjunctive interpolant* I is a mapping from locations to formulas such that

- For each label L with children $L_1, \ldots, L_n$,

$$f[L_1/\mathrm{I}(L_1), \ldots, L_n/\mathrm{I}(L_n)] \downarrow L \models \mathrm{I}(L)$$

[3] The bounded query tree itself corresponds to a disjunctive interpolation problem [27], which is a generalization of a tree interpolation problem in that labels can occur under both conjunction *and* disjunction.

- For each label L,

$$\mathrm{V}(\mathrm{I}(L)) \subseteq \mathrm{V}(f \downarrow L) \cap \mathrm{V}(f[L/\mathrm{true}])$$

  where $\mathrm{V}(e)$ are the free variables in e (cf. [27, Def. 2]).

- If all the labels in f are $L_1, \ldots, L_n$, then

$$f[L_1/\mathrm{I}(L_1), \ldots, L_n/\mathrm{I}(L_n)] \models \mathit{false}$$

An AND-OR tree can be trivially transformed into a tree interpolation problem:

$$\mathrm{toQuery}(\mathsf{And}(h, C)_L) = h \bigwedge_{c_i \in C} \mathrm{toQuery}(c_i)$$

$$\mathrm{toQuery}(\mathsf{Or}(C)_L) = \left(\bigvee_{c_i \in C} \mathrm{toQuery}(c_i) \right)_L$$

We use the disjunctive interpolation algorithm as described in [27] to reduce disjunctive interpolation queries to tree interpolation, and the use a standard tree interpolation engine to compute the disjunctive interpolants.

Given a disjunctive interpolant I, we can compute a set of qualifiers that can be used as a solution to a κ-variable by applying a substitution to an interpolant location that corresponds to the "inverse" of the substitution applied to the location during unrolling. Formally, we define a mapping $\mathsf{Sol}(I, \kappa)$:

$$\mathsf{Sol}(I, \kappa) = \bigcup_{L \in \mathrm{KL}(\kappa)} \mathrm{I}(L)\theta^{-1}[x/\nu]$$

$$\text{where } \theta = \mathsf{SL}(L)$$

$$x = \mathsf{VL}(L)$$

where θ^{-1} is the "inverse" of θ (e.g. if $\theta = [k/k-1]$, then $\theta^{-1} = [k-1/k]$). $\mathsf{SL}(\cdot)$ and $\mathsf{VL}(\cdot)$ are both arguments passed to BQ during unrolling. That is, given $\mathsf{BQ}(\kappa(x)\theta, m, D) = \mathsf{And}(\ldots,\ldots)_L$, $\mathsf{SL}(L) = \theta$ and $\mathsf{VL}(L) = x$.

These formulas returned by sol are exactly the solutions to inferred types Ξ, but we further transform formulas returned by sol before we convert them to qualifiers. First, we split apart conjuncted predicates into their conjuncts. We do this because qualifiers should be "atomic", and since the Liquid Types constraint solver conjuncts predicates computed from qualifiers this presents no loss — in fact, as the example below shows with $\lambda k.k \leq 0$, this technique can be used to filter out irrelevant conjuncts.

To produce qualifiers from these formulas, we simply bind the free variables of the formulas with lambda abstractions. We can then union all of the qualifiers for each κ-var, which gives us an abstract domain $\mathbb{Q}$ that the Liquid Types constraint solver can use to find a solution to the original constraint set.

Example. The AND-OR tree in figure 5.2 yields the interpolant:

$$\mathrm{I}\,(L_1) = \nu \geq \mathtt{k} \wedge \mathtt{k} \leq 0$$

$$\mathrm{I}\,(L_2) = s \geq \mathtt{k} - 1$$

$$\mathrm{I}\,(L_3) = s' = 0$$

From these interpolants we can compute a set of qualifiers

$$\{\lambda k.k \leq 0,\ \lambda\nu.\lambda k.\nu \geq k,\ \lambda\nu.\nu \geq 0,$$
$$\lambda\nu.\lambda k.k \leq\ 0 \vee \nu \geq k \vee \nu \geq 0\}$$

Using this set, the Liquid Types constraint solver can compute that $\kappa_s = \nu \geq \mathtt{k} \wedge \nu \geq 0$ is a valid solution to the original constraint set. This solution corresponds to a valid refinement type of `sum`, and in fact it can replace the refinement annotation as an even stronger postcondition.

However, because our unrolling algorithm is only an approximation for solving recursive constraints, $\mathsf{Sol}(I, \kappa)$ will not always be a solution for κ. Moreover, the Liquid Types constraint solver might fail to find a solution given the qualifiers in sol, since finding the solution to the constraints might require a higher unrolling depth. In that case, we return our expanded abstract domain and prompt the user to provide either additional type annotations or qualifiers, as they would were they using LiquidHaskell. This ensures that our algorithm is sound — that is, it will only return `SAFE` only when it computes a set of qualifiers that can be mapped to a valid refinement type annotation.

By construction, if the failure derivation constructed from the bounded query is

satisfiable, however, the refinement type annotation corresponding to the constraint set is invalid. In this case an interpolant does not exist, and our algorithm returns `UNSAFE`. Furthermore, we can query the interpolation oracle for a counterexample that corresponds to argument values that provide witness to the invalidity of the annotation. For example, if `sum` instead had the refinement annotation

```
1   sum :: k:Int -> { v:Int | v < k }
```

The algorithm determines that the corresponding failure derivation is satisfiable, and can provide a counterexample: $\texttt{k} = 0$, since $\texttt{sum 0} = 0$ and thus $\neg(\texttt{sum 0} < 0)$.

Algorithm 4 Unroll

Require: C is a set of Liquid subtyping constraints
Require: $V :=$ set of κ-vars in C
Require: D is a (finite) unrolling depth
Require: Interpolate is a tree interpolation oracle
Require: Solve is a Liquid Types constraint solver

1: **procedure** UNROLL(C, D)
2: $QS := \lambda\kappa.\emptyset$
3: $i := 0$
4: **while** $i \leq D$ **do**
5: **for** all queries $B \to H \in \mathsf{Enc}(C)$ **do**
6: $q = \mathsf{BQ}(B \to H, i)$
7: $QS := QS \cup$ EXTRACTSOL(q)
8: **end for**
9: $\mathbb{Q} := \bigcup_{\kappa \in V} QS(\kappa)$
10: **if** Solve($C, \mathbb{Q}$) = SAFE **then**
11: **return** SAFE
12: **else**
13: $i := i + 1$
14: **end if**
15: **end while**
16: **return** UNKNOWN
17: **end procedure**
18: **procedure** EXTRACTSOL(q)
19: $ts = \mathrm{T}(q)$
20: $QS := \lambda\kappa.\emptyset$
21: **for** $t \in ts$ **do**
22: $I = \mathsf{Interpolate}(q)$
23: **if** I does not exist **then**
24: **return** UNSAFE
25: **end if**
26: **for** $\kappa \in V$ **do**
27: $QS := QS[\kappa \mapsto QS(\kappa) \cup \mathsf{Sol}(I, \kappa)]$
28: **end for**
29: **end for**
30: **return** QS
31: **end procedure**

Chapter 6

Evaluation

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

We consider the following research questions:

- Section 6.2 — Does our fault localization algorithm provide better accuracy than vanilla LH typechecking?

- Section 6.3 — How does the speed of our fault localization algorithm compare to that of vanilla LH typechecking?

- Section 6.4 — Does our predicate discovery algorithm infer all off the needed intermediate types for k-bounded systems of constraints?

- Section 6.5 — How does our predicate discovery algorithm fare for constraint systems that don't have this property?

6.1 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 programs have k-bounded constraint graphs, and one in which they do not. Some are from the Liquid Haskell test suite while others were written for this evaluation.[1] 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. However, many programs can pass Haskell's standard Hindley-Milner type checking without satisfying the Liquid Haskell constraints and thus have latent bugs.

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. [23]), such as replacing one arithmetic operation with another. Such mutations have been previously shown to be indicative of developer mistakes [14] and potentially as difficult to locate as natural human bugs [17].

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

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 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 (*i.e.*, locations $T = \{t_1, t_2, \dots\}$ that implicate and explain the bug) and the output of an algorithm, (*i.e.*, locations $A = \{a_1, a_2, \dots\}$ should be inspected) we measure accuracy using standard notions from statistics and information retrieval. Every location reported by the algorithm that is in the ground truth set (*i.e.*, $a_i \in T$) 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 (*i.e.* $t_i \notin A$) is a false negative.

Finally, we observe that users tolerate some amount of imprecision in the output of a fault localization algorithm. For example, given that an algorithm outputs two locations, one of which is the bug location and the other is spurious, a user may still count the output of the algorithm as useful [15, 1]. This is especially true when the effort required to rule out a false positive is generally low, as is the case for many lines implicated by our algorithm. To capture this in our evaluation, we measure false positives relative to some tolerance level n: if an algorithm returns fewer than n

spurious bug locations for a file, then the algorithm is not counted as having a false positive for that file.

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.

6.2 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,[2] a baseline representing the state-of-the-art.

The results are summarized in Table 6.1. 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 positives than either of the proposed

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

	Ground Truth	True Positives	False Positives (t=0)	False Positives (t=1)	False Positives (t=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 6.1: Fault localization accuracy. "False Positives (t=n)" counts files with false positives (out of 28) for tolerance level n. "True Positives" counts files with true positives (out of 28) at a given tolerance t. The "Ground Truth" column indicates which of the two ground truth sets is used.

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 false positives 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 6.1, since a tolerance $t = 1$ "forgives" only a single false location), which do not represent a significant developer burden.

Table Table 6.2 breaks down algorithm success per benchmark. We also break down benchmarks by defect category. Our proposed algorithms also never found fewer bug locations than vanilla reporting for any defect category, and in some categories our algorithms found many more bug locations than vanilla reporting. For arithmetic

			True Positive?					
		Defect	Seeded			Expert		
File	Difficulty	Category	v	`m`	`mW`	v	`m`	`mW`
AVLRJ	hard	guard	–	–	–	–	–	–
Eval	easy	var	–	–	–	–	–	–
Evens	med	arith	–	✓	✓	–	✓	✓
FilterAbs	med	bool	✓	✓	✓	✓	✓	✓
GCD	easy	arith	–	✓	✓	–	✓	✓
InsertSort	easy	var	–	✓	✓	–	✓	✓
KmpIO	hard	arith	–	✓	✓	–	✓	✓
KmpVec	hard	var	–	✓	✓	–	✓	✓
ListLen	easy	bool	✓	✓	✓	✓	✓	✓
Maybe	easy	var	–	✓	✓	–	✓	✓
Mergesort	med	guard	–	–	–	–	–	–
MutualRec	easy	arith	✓	✓	✓	✓	✓	✓
NullTerm	hard	arith	–	✓	✓	–	✓	✓
Permutation	hard	list	–	–	–	–	–	–
PointDist	easy	arith	✓	–	✓	✓	–	✓
Poly0	easy	var	–	✓	✓	–	✓	✓
QuickSort	hard	bool	–	✓	✓	–	✓	✓
RBTree	easy	constr	–	–	–	–	–	–
RecQSort	easy	var	✓	✓	✓	✓	✓	✓
Record0	easy	arith	✓	–	–	✓	–	–
RelativeComplete	med	arith	–	✓	✓	–	✓	✓
Repeat	easy	arith	✓	✓	✓	✓	✓	✓
Shuffle	hard	badspec	–	–	–	–	✓	✓
Stacks	hard	arith	–	–	–	–	–	–
Top	hard	arith	–	✓	✓	–	✓	✓
TreeSum	easy	badspec	–	–	–	✓	✓	✓
Vectors	easy	badspec	–	–	–	✓	✓	✓
WBL	easy	arith	✓	✓	✓	✓	✓	✓

Table 6.2: Per-benchmark fault localization success. The "Difficulty" column marks the difficulty of that benchmark as marked by the expert. The "True Positives?" columns indicate whether vanilla, `minimize`, or `minimizeWCC` succeeded for that benchmark with respect to the automatically generated ground truth from seeded bugs ("Seeded") and to the ground truth marked by the expert ("Expert"). The "Defect Category" column lists an abbreviated classification of the bug for that benchmark:

arith	wrong arithmetic expression	badspec	wrong refinement type annotation
bool	wrong boolean expression	constr	wrong constructor
guard	wrong predicate in guard	list	wrong expression involving a list
var	wrong variable in expression		

Algorithm	Min Time	Max Time	Avg Time
`minimizeWCC`	0.004	43.64	3.73
`minimize`	0.120	27.88	3.35

Table 6.3: Time statistics for our proposed algorithms, given in seconds. "Min Time" and "Max Time" respectively record the minimum and maximum time that the algorithm took to process a single benchmark. "Avg Time" is the average time that an algorithm took to process a single benchmark.

expression bugs, `minimize` and `minimizeWCC` found 9 and 10 bugs respectively, while vanilla reporting only found 5. For wrong variable defects, `minimize` and `minimize` found 5 bugs each while vanilla reporting only found 1.

Strikingly, for the benchmarks that the expert marked as "hard", vanilla reporting only found 1 out of 9 bug locations (relative to the expert's ground truth) whereas both `minimizeWCC` and `minimize` found two-thirds (6 out of 9) of bug locations.

6.3 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. For Liquid Haskell, when a program passes the standard Haskell Hindley-Milner type checking but fails the Liquid Haskell constraint-based type checking, fault localization is invoked to report implicated lines to the developer. As a result, localization must run rapidly at compile-time.

Table 6.3 summarizes the running times of our unoptimized prototype. 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.

6.4 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.

6.5 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.

6.6 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.

Chapter 7

Related Work

There is a large literature on fault localization for languages with constraint-based (Hindley-Milner) type systems [8, 11, 28, 32]. We discuss two especially relevant prior approaches.

The SEMINAL tool by Lerner *et al.* [20] 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 fault localization 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.

Pavlinovic *et al.* [24] reduce fault localization into an instance of the MaxSAT problem by generating a set of assertions from the input program that are weighted by a ranking measure provided by the compiler. An SMT solver is used to find a minimum set of error clauses that can be mapped back to possible bug locations.

While the approach is successful in finding bugs in OCaml programs, it is unclear how successfully it could be applied to more sophisticated type systems such as Liquid Haskell. Our algorithm avoids this problem by using the existing constraint set generated for type checking the input program.

Zhang and Myers [35] 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 related to predicate discovery [13].

Bjørner *et al.* [3] review techniques for reducing program verification to Horn clause constraints, and review the state of the art in solving systems of Horn clauses.

Unno and Kobayashi [29] describe a procedure for inferring dependent intersection types using interpolants. Rümmer *et al.* [27] 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.

Chapter 8

Conclusions

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 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.

Bibliography

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Symposium on Principles of Programming Languages*, pages 97–105, 2003.

[2] Jesper Bengtson, Karthikeyan Bhargavan, Cdric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45.

[3] Nikolaj Bjrner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II*, number 9300 in Lecture Notes in Computer Science, pages 24–51. Springer International Publishing. DOI: 10.1007/978-3-319-23534-9_2.

[4] Brian Chin, Shane Markstrum, Todd D. Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming Languages and Systems*, pages 264–278, 2006.

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

[6] 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.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

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

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

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

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

[12] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[13] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, January 2004.

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

[15] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[16] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability.

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

[18] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34.

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

[20] Benjamin Lerner, Dan Grossman, and Craig Chambers. Seminal: searching for ml type-error messages. In *Proceedings of the 2006 workshop on ML*, pages 63–73. ACM, 2006.

[21] Kenneth L McMillan and Andrey Rybalchenko. Solving constrained horn clauses using interpolation. 2012.

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

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

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

[25] Nadia Polikarpova and Armando Solar-Lezama. Program synthesis from polymorphic refinement types.

[26] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[27] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *Computer Aided Verification*, pages 347–363, 2013.

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

[29] Hiroshi Unno and Naoki 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.

[30] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282. ACM, 2014.

[31] Visual Studio. Using SAL annotations to reduce C/C++ code defects. Technical report, Microsoft Developer Network, 2015.

[32] Mitchell 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.

[33] Hongwei Xi and Frank 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.

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

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