Liquid Types: Type Refinement via Predicate Abstraction

Patrick Maxim Rondon and Ranjit Jhala
Department of Computer Science and Engineering
University of California, San Diego

September 18, 2007

Abstract

Predicate abstraction and ML type inference are two well-known program analyses with very complementary strengths. The former is a technique for inferring precise, local, path-sensitive properties of base data values like integers but which is thwarted by complex data types and higher-order functions. The latter elegantly captures coarse properties of recursive data types and higher order functions. We present *Liquid Types*, a system that synergistically combines the complementary strengths of predicate abstraction and ML type inference to yield an algorithm for statically inferring properties well beyond the scope of either technique. We have implemented liquid type inference for λ_{L}, an extension of the λ-calculus with recursive types and ML style polymorphism. We present examples showing how liquid types can be used to statically prove the absence of divide-by-zero errors, out-of-bounds array access errors, and pattern match errors, with little to no user annotation.

1 Introduction

Recent years have seen significant progress on the vital problem of statically verifying safety properties of software. One fruitful line of research is based on a form of abstract interpretation [5] called *predicate abstraction*[19]. Predicate abstraction has proven to be especially effective for the path-sensitive verification of imperative programs [13, 3, 23, 32], and forms the algorithmic core of industrial-strength tools like SLAM [2]. The success of predicate abstraction stems from its ability to approximate infinitely many program states using using boolean combinations of a finite set of atomic predicates over program variables. This approximation enables the automatic synthesis of loop invariants and procedure summaries required for verification. However, predicate abstraction is most effective for control-dominated properties and runs aground when the verification requires precise reasoning about unbounded heap data such as lists and trees. To describe and reason about such structures, one requires complex predicates which have proven to be difficult to specify and algorithmically analyze.

A second line of research, which has deftly sidestepped these problems, proposes to enrich *type systems* using *refinements* or *dependent types*, allowing them to express sophisticated specifications in a manner that permits static verification. These approaches allow the modular construction of software via interface specifications that capture deep invariants of the modules [1], have been used to to prove array accesses safe statically, eliminating costly run-time checks [30], and have been used to specify and verify rich properties of data structures [17, 31]. The strength of this approach is that the type systems ensure that unbounded structures are used in a disciplined manner and use this discipline to provide an effective and elegant way to reason about the structures. Unfortunately, a significant barrier to the widespread adoption of such systems is that they require programmers to provide detailed type specifications, a difficult task that is only made harder by increasing the richness of the specification mechanism.

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a technique that synergistically combines predicate abstraction with the discipline imposed by the ML type system to yield an algorithm for inferring dependent types that statically prove a variety of safety properties.

The main insight behind liquid types is that predicate abstraction and the ML type system have complementary strengths – the former is capable of precise, path-sensitive, and local reasoning about base data values like integers, while the latter excels at capturing coarser facts about complex structures. As a motivating example, consider the following ML program:

$$
\begin{array}{l}
\texttt{let}\ abs\ =\ \lambda x.\texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ 0 - x\ \texttt{in} \\
\texttt{let}\ g\ =\ \lambda x.\lambda y.x + (abs\ y)\texttt{in} \\
\texttt{letrec}\ fold\ f\ b\ l\ = \\
\quad \texttt{match}\ l\ \texttt{with[]} \rightarrow b \\
\quad \mid h\texttt{::}t \rightarrow fold\ f\ (f\ b\ h)\ t\ \texttt{in} \\
fold\ g\ 0\ l
\end{array}
$$

Predicate abstraction alone is thwarted by use of lists, while ML type inference alone only proves that the expression has type `int`. Using the logical qualifier $\mathbb{Q} = \{0 \leq \text{v}\}$, our liquid type system can infer that the expression has the dependent type $\{\text{v}\!:\!\texttt{int} \mid 0 \leq \text{v}\}$, *i.e.,*the curried application to *fold* returns a non-negative integer!

To do so, first, our system uses predicate abstraction to infer that *abs* always returns a non-negative integer. Next, using predicate abstraction in tandem with the type inferred for *abs*, our system infers that when g is called with a non-negative first argument, its output is non-negative. Finally, our system instantiates the type variables α and β in the polymorphic type $\forall\alpha, \beta.(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ \texttt{list} \rightarrow \alpha$ inferred for *fold*, with $\{\text{v}\!:\!\texttt{int} \mid 0 \leq \text{v}\}$ and `int` respectively to infer that the result of the application $(fold\ g\ 0\ l)$ is non-negative. Thus, liquid types combine two powerful, complementary techniques for reasoning about programs. They blend the coarse but global properties inferrable by ML type inference (*e.g.*, the polymorphic type of *fold*), with the precise but local properties inferred by predicate abstraction (*e.g.*, the liquid type abstractly summarizing the behavior of *abs*) to yield a fact well beyond the ability of either technique.

In our system, type checking and inference are decidable due to two restrictions (Section 2). First, we use a conservative but decidable notion of subtyping, where we reduce the subtyping of arbitrary dependent types to a set of implication checks over base types, which are deemed to hold iff an *embedding* of the implication into a decidable logic yields a valid formula in the logic. Second, our system is *parameterized* by a set of logical qualifiers $\mathbb{Q}$ which are boolean expressions (predicates) over program variables. We say a dependent type is *liquid* if its refinements are *conjunctions* of predicates from $\mathbb{Q}$. Our rules stipulate that, in any valid type derivation, the types of certain expressions, such as λ-abstractions and if-then-else expressions, must be *liquid*. Thus, inference becomes decidable, as the space of possible types is bounded.

We have used these restrictions to design a simple constraint-based algorithm that seamlessly combines ML type inference with predicate abstraction to infer dependent types. In our system, an expression has a valid liquid type derivation only if it has a valid ML type derivation, and the dependent type of every subexpression is a *refinement* of its ML type. We exploit this fact to design the following three-step algorithm for dependent type inference. First, our algorithm invokes Hindley-Milner [6] to infer types for each subexpression and the necessary type generalization and instantiation annotations. Second, our algorithm uses the computed ML types to assign to each subexpression a *template*, a complex type with the same structure as the computed ML type, but which has *liquid type variables* representing the unknown type refinements. We use the syntax-directed liquid typing rules to generate a system of *constraints* that capture the subtyping relationships between the templates that must be met for a liquid type derivation to exist. Third, our algorithm uses the subtyping rules to split the complex template constraints into simple constraints over the liquid type variables, and then solves these simple constraints using a fixpoint computation inspired by predicate abstraction.

Of course, there may be safe programs which cannot be well-typed in our system — due either to an inappropriate choice of $\mathbb{Q}$ or the conservativeness of our notion of subtyping. In the former case, one can use the readable results of the inference to add more qualifiers, and in the latter case we can use the results of the inference to insert a minimal set of run-time checks in order to enforce safety in a hybrid manner [11].

We formalize our system using a core language λ_{L}, an extension of the λ-calculus with several base types, recursive types, and ML-style polymorphism. We first describe the syntax of λ_{L} and formalize the type checking rules (Section 2). Next, we describe our constraint-based inference algorithm (Section 3) and show a detailed example illustrating how we combine predicate abstraction and types to eliminate array bounds checks (Section 4). Next, we extend the formalism to polymorphic recursive datatypes such as lists (Section 5)

e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	`c`	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	`if` e `then` e `else` e	if-then-else
	$\mid$	`let` $x\ =\ e$ `in` e	let-binding
	$\mid$	`letrec` $f\ =\ \lambda x.e$ `in` e	letrec-binding
	$\mid$	$[\Lambda\alpha]e$	type-abstraction
	$\mid$	$[\tau]e$	type-instantiation
Q	$::=$		*Liquid Refinements*
	$\mid$	$true$	true
	$\mid$	q	logical qualifier in $\mathbb{Q}$
	$\mid$	$Q \wedge Q$	conjunction of qualifiers
B	$::=$		*Base Types:*
	$\mid$	`int`	base type of integers
	$\mid$	`bool`	base type of booleans
$\mathbb{T}(\mathbb{B})$	$::=$		*Type Skeletons:*
	$\mid$	$\{\mathrm{v} : B \mid \mathbb{B}\}$	base
	$\mid$	$x : \mathbb{T}(\mathbb{B}) \rightarrow \mathbb{T}(\mathbb{B})$	function
	$\mid$	α	type variable
$\mathbb{S}(\mathbb{B})$	$::=$		*Type Schema Skeletons:*
	$\mid$	$\mathbb{T}(\mathbb{B})$	monotype
	$\mid$	$\forall\alpha.\mathbb{S}(\mathbb{B})$	polytype
τ, σ	$::=$	$\mathbb{T}(true), \mathbb{S}(true)$	*Types, Schemas*
T, S	$::=$	$\mathbb{T}(E), \mathbb{S}(E)$	*Dep. Types, Schemas*
$\hat{T}, \hat{S}$	$::=$	$\mathbb{T}(Q), \mathbb{S}(Q)$	*Liquid Types, Schemas*

Figure 1: **Syntax**

and show how liquid types can infer refinements over the data in the list as well as properties such as the size of the list. We have implemented the inference algorithm for λ_{L}, and we describe some simple experiments using our system. In our experience, the liquid type restriction ensures that the dependent types inferred by our system are extremely readable and can provide useful documentation for API functions. Moreover, to extend the capabilities of the type system and prove more properties statically, the programmer need only enrich the set of predicates and not worry about devising new inference rules. We illustrate this (Section 6) by showing how liquid types can be used to infer properties of programs using recursive data types in order to statically prove the absence of divide-by-zero errors, out-of-bounds array access errors, and pattern match errors, with little to no user annotations.

2 Liquid Type Checking

We now formalize the syntax and static semantics of our language, λ_{L}, a variant of the λ-calculus with ML-style polymorphism extended with liquid types.

Syntax. The syntax of expressions and types for λ_{L} is summarized in Figure 1. λ_{L} expressions include variables, special constants which include integers, arithmetic operators and other primitive operations described below, λ-abstractions, and function applications. In addition to the above, λ_{L} includes an `if-then-else` expression, `let`, and `letrec` expressions, as these are common idioms that the liquid type inference algorithm exploits to generate precise types.

2.1 Types and Schemas

We first describe the different kinds of types in our system, then present the type judgments and derivation rules, and finally relate the static type system with the operational semantics. λ_{L} has a system of *refined* base types, *dependent* function types, and ML-style polymorphism via type variables that are universally quantified at the outermost level to yield polymorphic type schemas.

We use B to denote base types such as `bool` (booleans) or `int` (integers). These types are fairly limited, and cannot, for example, describe an expression as having a positive value or being bounded by some constant. Therefore, following [11, 1], λ_{L} allows *base refinements* of the form $\{\mathrm{v} : B \mid e\}$, where e is a boolean expression (or *predicate*) constraining the variable v. The base refinement predicate specifies the set of values c of the base type B such that the predicate $[c/\mathrm{v}]e$ evaluates to true. Thus, $\{\mathrm{v} : \texttt{int} \mid \mathrm{v} > 0\}$ specifies the set of positive integers, and $\{\mathrm{v} : \texttt{int} \mid \mathrm{v} \leq n\}$ specifies the set of integers whose value is less than or equal to the value of the variable n. Any base type B can be viewed as an abbreviation for $\{\mathrm{v} : B \mid \mathit{true}\}$. Similarly, we use $\{e\}$ as an abbreviation for $\{\mathrm{v} : B \mid e\}$ when the base type B is clear from the context.

We use the base refinements to build up *dependent function types*, written $x : T_1 \rightarrow T_2$ (following [11, 1]). Here, T_1 is the domain type of the function, and the formal parameter x may appear in the base refinements of the range type T_2. For example, $x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \{\mathrm{v} : \texttt{int} \mid (x \leq \mathrm{v}) \wedge (\mathrm{v} \leq y)\}$ denotes the type of a function that takes two (curried) integer arguments x, y and returns a value that is between x and y.

Liquid Types. The types and schemas of our language can be classified into three kinds. The first kind are *ML Types*, written τ for types and σ for schemas. Here, all the base refinement predicates are *true*. These types and schemas are exactly the usual base, function, and variable types of ML with the formal parameters explicit in the function types.

The second kind are the full *Dependent Types*, written T for types and S for schemas, where the base refinement predicates can be arbitrary expressions.

The third kind are *Liquid Types*, written $\hat{T}$ for types and $\hat{S}$ for schemas. Let $\mathbb{Q}$ be a *fixed, finite set* of boolean expressions (predicates) that we call the set of *logical qualifiers*. A *liquid type over* $\mathbb{Q}$ is a dependent type whose base refinement predicates are conjunctions of logical qualifiers in $\mathbb{Q}$. For example, the function *abs* described in Section 1 has the liquid type $x : \texttt{int} \rightarrow \{\mathrm{v} : \texttt{int} \mid 0 \leq \mathrm{v}\}$ over the set of logical qualifiers $\mathbb{Q} = \{0 \leq \mathrm{v}\}$.

Constants. As in [11], the basic units of computation are the constants `c` built into λ_{L}, each of which has a dependent type $\mathit{ty}(\texttt{c})$ that precisely captures the semantics of the constants. These include *basic constants*, corresponding to integers and boolean values, and *primitive functions*, which encode various operations. The set of constants of λ_{L} include:

$$\begin{array}{rl}
\texttt{true}: & \{\mathrm{v} : \texttt{bool} \mid \mathrm{v}\} \\
\texttt{false}: & \{\mathrm{v} : \texttt{bool} \mid \texttt{not}\ \mathrm{v}\} \\
\Leftrightarrow: & x : \texttt{bool} \rightarrow y : \texttt{bool} \rightarrow \{\mathrm{v} : \texttt{bool} \mid \mathrm{v} \Leftrightarrow (x \Leftrightarrow y)\} \\
\texttt{3}: & \{\mathrm{v} : \texttt{int} \mid \mathrm{v} = 3\} \\
=: & x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \{\mathrm{v} : \texttt{bool} \mid \mathrm{v} \Leftrightarrow (x = y)\} \\
+: & x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \{\mathrm{v} : \texttt{int} \mid \mathrm{v} = x + y\} \\
/: & x : \texttt{int} \rightarrow y : \{\mathrm{v} : \texttt{int} \mid \mathrm{v} \neq 0\} \rightarrow \{\mathrm{v} : \texttt{int} \mid \mathrm{v} = x/y\} \\
\texttt{len}: & \texttt{intarray} \rightarrow \{\mathrm{v} : \texttt{int} \mid 0 \leq \mathrm{v}\} \\
\texttt{sub}: & a : \texttt{intarray} \rightarrow i : \{\mathrm{v} : \texttt{int} \mid (0 \leq \mathrm{v}) \wedge (\mathrm{v} < \texttt{len}\ a)\} \rightarrow \texttt{int} \\
\texttt{error}: & \{\mathrm{v} : \texttt{int} \mid \mathit{false}\} \rightarrow \texttt{int}
\end{array}$$

The types of some constants are defined in terms of themselves (*e.g.*, "+"). This does not cause problems as the dynamic semantics of refinement predicates is defined in terms of the operational semantics (as in [11]), and the static semantics is defined via a sound overapproximation of the dynamic semantics. For clarity, we will use infix notation for constants like +. The constant for /, the integer division operator, stipulates that the second argument, used as the denominator, is non-zero. There is a special base type that encodes integer arrays in λ_{L}. The length of an array value is obtained using `len`. To access the elements of the array, we use `sub`, which takes as input an array a and an index i that must be within the bounds of a, *i.e.,*non-negative, and less than the length of the array. Finally, `error` (similar to `assert false`) is used to specify expressions that must never be evaluated; one can model `assert p` as: `if` $\neg$`p` `then` `error` 0 `else` 0.

2.2 Judgements and Rules

Next, we present the key ingredients of the type system — the typing judgements and derivation rules, which are summarized in Figure 2. A *type environment* Γ is a map from variables x to type schemas S. A *guard environment* G is a boolean expression that captures constraints about the conditions under which an expression is evaluated, *e.g.*, the if-branches under which the expression is evaluated. It encodes within the type system a notion of "intra-procedural path-sensitivity" that is vital for finding precise dependent types.

There are three kinds of judgements. A judgement $\Gamma \vdash S$ states that the dependent type schema S is *well-formed* under the type environment Γ. Intuitively, a type is well-formed if its base refinements are boolean expressions which refer only to variables which are in scope. A judgement $\Gamma, G \vdash S_1 <: S_2$ states that the dependent type schema S_2 *subsumes* the schema S_1 under the type environment Γ and guard environment G. A judgement $\Gamma, G \vdash e : S$ states that, over the set of logical qualifiers $\mathbb{Q}$, the expression e has the type schema S under the type environment Γ and guard environment G.

The syntax-directed derivation rules for the judgements are similar to those for other dependent type systems such as [11]. We assume that variables are bound at most once in any type environment; in other words, we assume that variables are α-renamed to ensure that substitutions (such as in [$\hat{T}$-App]) avoid capture.

The Liquid Type Restriction. The critical difference between the rules for liquid type checking and other systems is that we require that certain kinds of expressions have liquid types, that is, types where the refinements are over conjunctions of predicates from the finite set of logical qualifiers $\mathbb{Q}$. In particular, the rules for λ-abstraction ([$\hat{T}$-Abs]), if-then-else ([$\hat{T}$-If]), let ([$\hat{T}$-Let], letrec[$\hat{T}$-Letrec]), and polymorphic instantiation ([$\hat{T}$-Inst]) stipulate that the types of the respective expressions be liquid. The intuitive reason for this restriction is that these are the key points at which appropriate dependent types must be *inferred*. By forcing the types to be liquid, we bound the space of possible solutions, thereby making inference decidable. Put another way, these points are analogous to the function boundaries, control-flow join and loop headers of imperative first-order programs, at which analyses for such programs typically perform abstraction.

Polymorphism. To handle polymorphism, our type system incorporates type generalization and instantiation annotations. Note that the generalization and instantiation annotations are over ML type variables α and monomorphic ML types τ (and not dependent types). While we have explicated the annotations for ease of exposition, we note that they can be reconstructed via a standard type inference algorithm [6], a fact we exploit for liquid type inference. The rule [$\hat{T}$-Inst] allows a type schema to be instantiated with an arbitrary liquid type $\hat{T}$, but we show in Section 3 that, for the entire expression to type check, the instantiated type $\hat{T}$ must have the same *shape* as τ, the monomorphic ML type used for instantiation. In other words, replacing all refinements in $\hat{T}$ with *true* yields τ.

2.3 Decidable, Conservative Subtyping

As in [11], we use predicates as the base refinements. As shown in Figure 2, checking that one type is a subtype of another reduces to (a set of) subtype checks on base refinements, which further reduces to checking if the refinement predicate for the subtype *implies* the predicate for the supertype. Our goal is a purely static technique for proving safety. Therefore, instead of using an *exact*, and therefore undecidable, notion of subtyping, we use the following conservative, but decidable, implication check. Let EUFA be the decidable logic [27] of *equality*, *uninterpreted functions*, and *linear arithmetic*. The subtyping relation

$$\Gamma, G \vdash \{\mathrm{v} : B \mid e_1\} <: \{\mathrm{v} : B \mid e_2\}$$

holds iff the formula

$$(\wedge \{[\![[x/\mathrm{v}]e]\!] \mid \Gamma(x) = \{\mathrm{v}{:}\cdot \mid e\}\} \wedge [\![G]\!] \wedge [\![e_1]\!]) \Rightarrow [\![e_2]\!]$$

is *valid* in EUFA. We write $[\![e]\!]$ for the *embedding* of the expression e into terms of the logic by encoding expressions corresponding to integers, addition, multiplication and division by constant integers, equality, inequality and disequality with corresponding terms in the logic, and encoding *all* other constructs, including λ-abstractions and applications, with *uninterpreted* function terms. Note that we use the mapped refinement predicates from the type environment that do not appear under a function type to *strengthen* the antecedent of the implication. It is easy to check that this embedding is conservative, *i.e.,*the validity of the embedded implication implies the the standard, weaker, exact requirement for subtyping of refined types [11].

For example, the subtyping relation:

$$[x \mapsto \texttt{int};\ y \mapsto \{\mathrm{v}\!:\!\texttt{int} \mid 0 \leq \mathrm{v}\}], \neg(x > 0) \vdash \{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} = y - x\} <: \{\mathrm{v}\!:\!\texttt{int} \mid 0 \leq \mathrm{v}\}$$

holds in our system because the implication

$$((0 \leq y) \wedge \neg(x > 0) \wedge (\mathrm{v} = y - x)) \Rightarrow (0 \leq \mathrm{v})$$

is valid in EUFA.

2.4 Soundness of Liquid Type Checking

Let $\leadsto$ describe the single evaluation step relation for λ_{L} expressions and $\leadsto^*$ describe the reflexive, transitive closure of $\leadsto$. The rules describing $\leadsto$ are standard and omitted for brevity. We can prove that (assuming the constants have appropriate static and dynamic semantics) the type checking rules are sound, in that if there is a valid type derivation for an expression, then we have the usual type preservation and progress guarantees.

Theorem 1 [Type Safety] *Let $\mathbb{Q}$ be a set of logical qualifiers. If there exists a derivation $\emptyset, \mathit{true} \vdash e : S$ over $\mathbb{Q}$ then: (1) if $e \leadsto e'$ then there exists a derivation $\emptyset, \mathit{true} \vdash e' : S$ over $\mathbb{Q}$, and, (2) either $e \leadsto e'$ for some e', or e is a λ-abstraction or a constant.*

The proofs of the above follow from induction on the derivation. Note also that the existence of a liquid type derivation implies the existence of a more general dependent type derivation via the rules of [11]. Thus, for any expression e containing / (resp. `sub`, `error`) if there is a derivation $\emptyset, \mathit{true} \vdash e : T$ over any set of qualifiers $\mathbb{Q}$, then we are guaranteed that no divide-by-zero (resp. out-of-bounds array accesses, assertion failures) occur at run-time during the course of evaluating e.

Liquid Type Checking $\boxed{\Gamma, G \vdash e : S}$

$$\frac{\Gamma, G \vdash e : S_1 \qquad \Gamma, G \vdash S_1 <: S_2 \qquad \Gamma \vdash S_2}{\Gamma, G \vdash e : S_2}\ [\hat{T}\text{-Sub}]$$

$$\frac{\Gamma(x) = \{\mathrm{v} : B \mid e\}}{\Gamma, G \vdash x : \{\mathrm{v} : B \mid \mathrm{v} = x\}}\ [\hat{T}\text{-Var}] \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma, G \vdash x : \Gamma(x)}\ [\hat{T}\text{-Var}]$$

$$\frac{}{\Gamma, G \vdash \mathtt{c} : ty(\mathtt{c})}\ [\hat{T}\text{-Const}]$$

$$\frac{\Gamma[x \mapsto \hat{T}], G \vdash e_1 : \hat{T}_1}{\Gamma, G \vdash (\lambda x.e_1) : (x{:}\hat{T} \to \hat{T}_1)}\ [\hat{T}\text{-Fun}]$$

$$\frac{\Gamma, G \vdash e_1 : (x{:}T_1 \to T) \qquad \Gamma, G \vdash e_2 : T_1}{\Gamma, G \vdash e_1\ e_2 : [e_2/x]T}\ [\hat{T}\text{-App}]$$

$$\frac{\Gamma, G \vdash e_1 : \mathtt{bool} \qquad \Gamma, G \wedge e_1 \vdash e_2 : \hat{T} \qquad \Gamma, G \wedge \neg e_1 \vdash e_3 : \hat{T}}{\Gamma, G \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \hat{T}}\ [\hat{T}\text{-If}]$$

$$\frac{\Gamma, G \vdash e_1 : S_1 \quad \Gamma[x \mapsto S_1], G \vdash e_2 : \hat{T}_2}{\Gamma, G \vdash \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \hat{T}_2}\ [\hat{T}\text{-Let}]$$

$$\frac{\Gamma[x \mapsto \hat{S}_1], G \vdash e_1 : \hat{S}_1 \quad \Gamma[x \mapsto \hat{S}_1], G \vdash e_2 : \hat{T}_2}{\Gamma, G \vdash \mathtt{letrec}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \hat{T}_2}\ [\hat{T}\text{-Letrec}]$$

$$\frac{\Gamma, G \vdash e : S \qquad \alpha \notin \Gamma}{\Gamma \vdash [\Lambda\alpha]e : \forall\alpha.S}\ [\hat{T}\text{-Gen}] \qquad \frac{\Gamma, G \vdash e : \forall\alpha.S}{\Gamma, G \vdash [\hat{T}]e : [\hat{T}/\alpha]S}\ [\hat{T}\text{-Inst}]$$

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\Gamma[\mathrm{v} \mapsto B] \vdash e : \mathtt{bool}}{\Gamma \vdash \{\mathrm{v}{:}B \mid e\}}\ [\text{WF-Base}] \qquad \frac{}{\Gamma \vdash \alpha}\ [\text{WF-Var}]$$

$$\frac{\Gamma[x \mapsto T_1] \vdash T_2}{\Gamma \vdash x{:}T_1 \to T_2}\ [\text{WF-Fun}] \qquad \frac{\Gamma \vdash S}{\Gamma \vdash \forall\alpha.S}\ [\text{WF-Poly}]$$

Implication $\boxed{\Gamma, G \vdash e_1 \Rightarrow e_2}$

$$\frac{\mathsf{Valid}(\wedge\{[\![[x/\mathrm{v}]e]\!] \mid \Gamma(x) = \{\mathrm{v} : \cdot \mid e\}\} \wedge [\![G]\!] \wedge [\![e_1]\!] \Rightarrow [\![e_2]\!])}{(\Gamma \wedge G \wedge e_1) \Rightarrow e_2}\ [\text{Imp}]$$

Subtyping $\boxed{\Gamma, G \vdash S_1 <: S_2}$

$$\frac{(\Gamma \wedge G \wedge e_1) \Rightarrow e_2}{\Gamma, G \vdash \{\mathrm{v}{:}B \mid e_1\} <: \{\mathrm{v}{:}B \mid e_2\}}\ [<:\text{-Base}]$$

$$\frac{\Gamma, G \vdash T_2' <: T_1' \qquad \Gamma[x \mapsto T_2'], G \vdash T_1'' <: T_2''}{\Gamma, G \vdash x{:}T_1' \to T_1'' <: x{:}T_2' \to T_2''}\ [<:\text{-Fun}]$$

$$\frac{}{\Gamma, G \vdash \alpha <: \alpha}\ [<:\text{-Var}] \qquad \frac{\Gamma, G \vdash S_1 <: S_2}{\Gamma, G \vdash \forall\alpha.S_1 <: \forall\alpha.S_2}\ [<:\text{-Poly}]$$

Figure 2: **Rules for Liquid Type Checking**

3 Liquid Type Inference

The goal of our work is to statically prove the safety of dynamic asserts by demonstrating the existence of a liquid type derivation, which by Theorem 1 proves that the asserts do not fail. Instead of requiring programmers to annotate expressions with appropriate dependent types, we show how to *infer* dependent liquid types for each subexpression, from which a liquid type derivation can be constructed, if such types exist.

Our type inference algorithm exploits the tight connection between ML types and their refinements (Section 3.1) to generate templates for the unknown liquid types of subexpressions and subtyping constraints (Section 3.2) over the templates that capture the subtyping relationships that must hold for a liquid type derivation to exist (Section 3.3). Finally, we use the finite set of logical qualifiers $\mathbb{Q}$ to solve the constraints using a technique inspired by predicate abstraction, thereby determining whether the expression can be well-typed over $\mathbb{Q}$, whilst also inferring readable types for all subexpressions.

3.1 The Shape of Liquid Types: ML Types

Our type inference algorithm is based on the observation that the dependent types and type derivations for each subexpression are *refinements* of their ML types.

Shapes. The function Shape, shown in Figure 3.3, maps arbitrary dependent types (schemas) to ML types (schemas). Intuitively, $\mathsf{Shape}(S)$, called the *shape* of the dependent type schema S, is the ML type schema obtained by replacing all refinement predicates with *true*. We lift the function Shape to type environments Γ by defining $\mathsf{Shape}(\Gamma)(x) = \mathsf{Shape}(\Gamma(x))$.

ML Type Derivations. The judgment $\Gamma \vdash e : \sigma$, states that, under the type environment Γ that maps variables to ML type schemas, the expression e has the ML type schema σ. The derivation rules for such judgments are standard [6] and are omitted for brevity. We observe that if there is a valid liquid type derivation for an expression, then there is an ML type derivation for the expression which has the same structure.

Proposition 1 *[Derivation Projection] For all liquid type environments Γ, guard environments G, expressions e, dependent type schemas S, if $\Gamma, G \vdash e : S$ then $\mathsf{Shape}(\Gamma) \vdash e : \mathsf{Shape}(S)$.*

The proof follows by straightforward induction on the liquid type derivation (notice that the liquid type derivation rules are "refinements" of the ML type derivation rules).

The algorithmic importance of this observation is that the ML types can be inferred automatically using classical techniques, which simultaneously infer appropriate type schemas for each expression and also "insert" suitable type generalization $[\Lambda\alpha]e$ and instantiation $[\tau]e$ annotations. Thus, the problem of inferring appropriate dependent types for all subexpressions can be reduced to the problem of determining how the ML types for the expressions need to be refined in order to find a valid liquid type derivation.

3.2 Constraints and Assignments

We now describe the elements that constitute our constraints and the properties of their solutions.

Variables with Pending Substitutions. Let $\mathbb{K}$ be a set of *liquid type variables*, used to represent unknown type refinement predicates. We define *sequences of pending substitutions* θ as

$$\theta \quad ::= \quad \epsilon \mid [e/x]\theta$$

As shown in Figure 2, for some expressions like function application (rule $[\hat{T}\text{-App}]$) we need to *substitute* all occurrences of the formal argument x in the output type of e_1 with the actual expression e_2 passed in at the application. When generating the constraints, the output type of e_1 is unknown and is represented by a template containing liquid type variables. Suppose that it is of the form $\{\mathrm{v} : B \mid \kappa\}$, where κ is a liquid type variable. In this case, we will write the type of the application $e_1\ e_2$ as $\{\mathrm{v} : B \mid [e_2/x] \cdot \kappa\}$, where $[e_2/x] \cdot \kappa$ is a variable with a *pending* substitution [24].

Templates. A *template* is a dependent type schema where some of the refinement predicates are replaced with liquid type variables with pending substitutions. Templates can be described via the grammar

$$F \quad ::= \quad \mathbb{S}(E \cup \theta \cdot \mathbb{K})$$

A *template environment* is a map Γ from variables to templates. We use templates to represent liquid type schemas with (partially) unknown base refinements.

To ensure that our constraints only contain templates of the form defined above, we use a function, Push, which distributes substitutions over $\rightarrow$, so that substitutions are applied only at the leaves of the template; *e.g.*,

$$\mathsf{Push}(\theta \cdot (\{\kappa_1\} \rightarrow \{\kappa_2\})) = \{\theta \cdot \kappa_1\} \rightarrow \{\theta \cdot \kappa_2\}$$

Constraints. Our inference algorithm utilizes two kinds of constraints over templates. *1. Well-formedness* constraints ensure that the liquid types inferred for each expression are over variables that are in scope at that expression. These constraints are of the form $\Gamma \vdash F$, where Γ is template environment, and F is a template. *2. Subtyping* constraints ensure that the liquid types inferred for each expression can be combined to yield a valid type derivation by ensuring that the subsumption relationship holds at the relevant points. These constraints are of the form $\Gamma, G \vdash F_1 <: F_2$ where Γ is a template environment, G is a guard environment, *i.e.,*a predicate, and F_1 and F_2 are two templates with the same shape ($\mathsf{Shape}(F_1) = \mathsf{Shape}(F_2)$).

Liquid Type Assignments. A *Liquid Type Assignment* A over a set of logical qualifiers $\mathbb{Q}$ is a map from liquid type variables to conjunctions of predicates from $\mathbb{Q}$. Assignments can be lifted to templates F, to obtain dependent type schemas $A(F)$, by replacing each liquid type variable κ with $A(\kappa)$ and then *applying the pending substitutions.* We lift assignments to template environments Γ by defining $A(\Gamma)(x) = A(\Gamma(x))$.

Constraint Satisfaction. An assignment A *satisfies* a well-formedness constraint $\Gamma \vdash F$ if $\mathsf{Shape}(\Gamma) \vdash A(F)$; we use Shape because only the ML types (and not the refinements) are required to determine well-formedness. An assignment A *satisfies* a subtyping constraint $\Gamma, G \vdash F_1 <: F_2$ if $A(\Gamma), G \vdash A(F_1) <: A(F_2)$.

3.3 Constraint Generation

Next, we describe the algorithm that generates constraints over templates by traversing the expression in the syntax-directed manner of a type checker, generating fresh templates for unknown types and constraints that capture the relationships between the types of various subexpressions and requirements for well-formedness. The generated constraints are such that they have a solution iff the expression has a valid liquid type derivation.

Algorithm Cons. Our constraint generation algorithm Cons, shown in Figure 3, takes as input a template environment Γ, a guard environment G and an expression e that we wish to find a type for and returns as output a pair of a type template F, which corresponds to the unknown type of e, and a set of constraints C that must be satisfied in order for e to type check.

To understand how Cons works, notice that the expressions of λ_{L} can be split into two classes — those whose types are immediately derivable from the environment and the types of subexpressions, and those whose types are not. We describe Cons by showing how it handles each of these two classes of expressions.

1. Expressions with Derivable Types. The first class of expressions are variables, constants, function applications, let-bindings and polymorphic generalizations, whose types can be immediately derived from the types of subexpressions or the environment. For such expressions, Cons recursively computes templates and constraints for the subexpressions and appropriately combines them to form the template and constraints for the expression.

For example, let us consider the case of $\mathsf{Cons}(\Gamma, G, e_1\ e_2)$. First, Cons is called to obtain the templates and constraints for the subexpressions e_1 and e_2. If a valid ML type derivation exists, then e_1 must be a function type with some formal x. The template returned for the application is the result of "pushing" the (pending) substitution of x with the actual argument e_2 into the "leaves" of the template corresponding to the return type of e_1. The constraints returned for the application are the union of the constraints for the subexpressions, and a subtyping constraint ensuring that the argument e_2 is a subtype of the input type of e_1.

```
$\mathsf{Cons}(\Gamma, G, e) =$
  match $e$ with
    | $x \longrightarrow$
        if $\mathsf{Shape}(\Gamma) \vdash e : B$ then $(\{\mathrm{v}\!:\!B \mid \mathrm{v} = x\}, \emptyset)$
        else $(\Gamma(x), \emptyset)$
    | $\mathsf{c} \longrightarrow$
        $(ty(\mathsf{c}), \emptyset)$
    | $\lambda x.e_1$
        when $\mathsf{Shape}(\Gamma) \vdash e : \tau \longrightarrow$
        let $(x\!:\!F \rightarrow F_1) = \mathsf{Fresh}(\tau)$ in
        let $(C_1, F_1') = \mathsf{Cons}(\Gamma[x \mapsto F], G, e_1)$ in
        $(x\!:\!F \rightarrow F_1, C_1 \cup \{\Gamma \vdash x\!:\!F \rightarrow F_1\} \cup$
            $\{\Gamma[x \mapsto F], G \vdash F_1' <: F_1\})$
    | $e_1\ e_2 \longrightarrow$
        let $(x\!:\!F_2'' \rightarrow F', C_1) = \mathsf{Cons}(\Gamma, G, e_1)$ in
        let $(F_2', C_2) = \mathsf{Cons}(\Gamma, G, e_2)$ in
        $([e_2/x]F', C_1 \cup C_2 \cup \{\Gamma, G \vdash F_2' <: F_2''\})$
    | if $e_1$ then $e_2$ else $e_3$
        when $\mathsf{Shape}(\Gamma) \vdash e : \tau \longrightarrow$
        let $F = \mathsf{Fresh}(\tau)$ in
        let $(\_, C_1) = \mathsf{Cons}(\Gamma, G, e_1)$ in
        let $(F_2, C_2) = \mathsf{Cons}(\Gamma, G \wedge e_1, e_2)$ in
        let $(F_3, C_3) = \mathsf{Cons}(\Gamma, G \wedge \neg e_1, e_3)$ in
        $(F, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup$
            $\{\Gamma, G \wedge e_1 \vdash F_2 <: F\} \cup$
            $\{\Gamma, G \wedge \neg e_1 \vdash F_3 <: F\})$
    | let $x = e_1$ in $e_2$
        when $\mathsf{Shape}(\Gamma) \vdash e : \tau \longrightarrow$
        let $(F_1', C_1) = \mathsf{Cons}(\Gamma, G, e_1)$ in
        let $(F_2', C_2) = \mathsf{Cons}(\Gamma[x \mapsto F_1'], G, e_2)$ in
        let $F = \mathsf{Fresh}(\tau)$ in
        $(F, C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup$
            $\{\Gamma[x \mapsto F_1'], G \vdash F_2' <: F\})$
    | letrec $x = e_1$ in $e_2$
        when $\mathsf{Shape}(\Gamma)[x \mapsto \sigma_1] \vdash e_1 : \sigma_1,$
               $\mathsf{Shape}(\Gamma) \vdash e : \tau \longrightarrow$
        let $F_1 = \mathsf{Fresh}(\sigma_1)$ in
        let $(F_1', C_1) = \mathsf{Cons}(\Gamma[x \mapsto F_1], G, e_1)$ in
        let $(F_2', C_2) = \mathsf{Cons}(\Gamma[x \mapsto F_1], G, e_2)$ in
        let $F = \mathsf{Fresh}(\tau)$ in
        $(F, C_1 \cup C_2 \cup \{\Gamma \vdash F_1\} \cup \{\Gamma \vdash F\} \cup$
            $\{\Gamma[x \mapsto F_1], G \vdash F_1' <: F_1\})$
            $\{\Gamma[x \mapsto F_1], G \vdash F_2' <: F\} \cup$
    | $[\Lambda\alpha]e \longrightarrow$
        let $(F', C) = \mathsf{Cons}(\Gamma, G, e)$ in
        $(\forall\alpha.F', C)$
    | $[\tau]e$
        when $\mathsf{Shape}(\Gamma) \vdash e : \forall\alpha.\sigma \longrightarrow$
        let $F = \mathsf{Fresh}(\tau)$ in
        let $(\forall\alpha.F', C) = \mathsf{Cons}(\Gamma, G, e)$ in
        $([F/\alpha]F', C \cup \{\Gamma \vdash F\})$
```

Figure 3: **Constraint Generation:** Cons maps a triple (Γ, G, e) to a pair (F, C) of a template and a set of constraints, such that if $\mathsf{Shape}(\Gamma) \vdash e : \sigma$ then $\mathsf{Shape}(F) = \sigma$.

2. Expressions with Liquid Types. The second class are expressions whose types *cannot* be derived as above, as the subsumption rule is required to perform some kind of "over-approximation" of their concrete semantics. These include λ-abstractions, if-then-else expressions, let bodies, letrec bindings and bodies, and polymorphic instantiations. The key restriction in our type system that enables inference is that, for such expressions, the dependent types must be liquid, *i.e.,*the refinement predicates must be conjunctions of logical qualifier predicates from the finite set $\mathbb{Q}$ (cf. rules [$\hat{T}$-Let], [$\hat{T}$-Letrec], [$\hat{T}$-Fun], [$\hat{T}$-If], [$\hat{T}$-Inst] of Figure 2).

To generate the template and constraints for these expressions, we exploit the connection with ML types. Suppose that there is a solution A such that $A(\Gamma), G \vdash e : S$. Then, by Proposition 1, $\mathsf{Shape}(A(\Gamma)) \vdash e : \mathsf{Shape}(S)$. As $\mathsf{Shape}(A(\Gamma))$ is the same as $\mathsf{Shape}(\Gamma)$, we can conclude that any valid dependent type for e must have the same shape as the ML type for e inferred under the type environment $\mathsf{Shape}(\Gamma)$. Thus, for expressions that must have liquid types, Cons uses the ML type of the expression (obtained via the **when** clauses) to generate a template over fresh liquid type variables that has the same shape as the ML type. To do so, Cons invokes the function Fresh, shown in Figure 3.3, which maps ML type schemas σ to templates F over new liquid type variables such that $\mathsf{Shape}(F) = \mathsf{Shape}(\sigma)$, and then Cons generates the appropriate constraints on this fresh template.

For example, let us consider the case of $\mathsf{Cons}(\Gamma, G,$ `if` e_1 `then` e_2 `else` $e_3)$. First, a fresh template is generated using the ML type of the expression. Next, Cons recursively generates templates and constraints for the `then` and `else` sub-expressions. Note that for the `then` (resp. `else`) sub-expression, the guard environment is conjoined with e_1 (resp. $\neg e_1$) as in the type derivation rule ([$\hat{T}$-If] from Figure 2). The template returned for the whole expression is the fresh template. The constraints returned are the union of those for the subexpressions, together with subtyping constraints forcing the templates for the `then` and

$\mathsf{Shape}(t) =$
match t **with**
$\mid \{\mathrm{v}\!:\!B \mid \_\} \longrightarrow B$
$\mid (x\!:\!t_1 \to t_2) \longrightarrow (x\!:\!\mathsf{Shape}(t_1) \to \mathsf{Shape}(t_2))$
$\mid \alpha \longrightarrow \alpha$
$\mid \forall\alpha.t_1 \longrightarrow \forall\alpha.\mathsf{Shape}(t_1)$

$\mathsf{Fresh}(\sigma) =$
match σ **with**
$\mid \{\mathrm{v}\!:\!B \mid \_\} \longrightarrow \{\mathrm{v}\!:\!B \mid$ fresh liquid type variable $\kappa\}$
$\mid (x\!:\!\tau_1 \to \tau_2) \longrightarrow (x\!:\!\mathsf{Fresh}(\tau_1) \to \mathsf{Fresh}(\tau_2))$
$\mid \alpha \longrightarrow \alpha$
$\mid \forall\alpha.\sigma_1 \longrightarrow \forall\alpha.\mathsf{Fresh}(\sigma_1)$

Figure 4: Shape maps dependent type schemas and templates to ML types. Fresh maps type schemas σ to templates F with fresh liquid type variables, such that $\mathsf{Shape}(F) = \mathsf{Shape}(\sigma)$.

`else` sub-expressions to be subtypes of the fresh template for the whole expression.

Cons mirrors the type derivation rules and sets up a system of constraints over templates that have a satisfying assignment over $\mathbb{Q}$ iff the expression has a valid type derivation over $\mathbb{Q}$.

Proposition 2 *[**Constraint Generation**] For any set of logical qualifiers $\mathbb{Q}$, template environment Γ, guard environment G and expression e, if $\mathsf{Cons}(\Gamma, G, e) = (F, C)$ then: (1) if A is a liquid type assignment over $\mathbb{Q}$ that satisfies C then $A(\Gamma), G \vdash e : A(F)$ is derivable over $\mathbb{Q}$ and, (2) if C has no satisfying assignment over $\mathbb{Q}$ then there is no assignment A over $\mathbb{Q}$ such that $A(\Gamma), G \vdash e : A(F)$ is derivable over $\mathbb{Q}$.*

Example: Polymorphic Fold (Constraints)

The following code implements a recursive accumulation function akin to the well-known fold — instead of accumulating the result over a list, the function operates over the sequence of integers from 0 up to (but not including) n.

```
let foldn =
  λn.λb.λf.
    letrec loop =
      λi.λc.
        if i < n then loop (i + 1) (f i c) else c  in
    loop 0 b  in
let x₂ = foldn 42 0 (+)
```

ML type inference would assign the function bound to *foldn* the polymorphic type $\forall\alpha.n\!:\!\texttt{int} \to b\!:\!\alpha \to f : (\texttt{int} \to \alpha \to \alpha) \to \alpha$. This function is then applied to sum the integers from 0 to 42.

Constraints. The upper table of Figure 6 shows the constraints generated by Cons on the expression bound to *foldn*. ML type inference deduces the polymorphic type schema shown in row 1. As the expression is a λ-abstraction, Cons generates a fresh template from the ML type (row 1). We omit the trivial subtyping constraints of the form $\cdot, \cdot \vdash \alpha <: \alpha$, such as the one relating the body with the output. For the `letrec` binding, we use the ML type to generate a fresh template, $i : \kappa_7 \to c : \alpha \to \alpha$. We use the environment Γ_a with bindings for *abs*, x_1, n, f and the binding from *loop* with the fresh template, to generate constraints for the body of *loop*. First, we constrain the type computed for $\lambda i.\lambda c.\langle loop\rangle$ to be a subtype of the fresh type generated for the `letrec` binding (row 2). Next, we generate constraints for the body $\langle loop\rangle$. The application *loop* $(i + 1)$ occurs under the guard $(i < n)$, yielding the constraint which forces the type of the argument $\{\mathrm{v} = i + 1\}$ (the result of applying $(+)$ to i and 1), to be a subtype of the input type of *loop* as bound in the environment Γ_a, namely κ_7 (row 3). The application $(f\ i)$ yields a constraint forcing the type of the actual parameter i to be a subtype of κ_5, the formal for f (row 4). Finally, the application (*loop* 0) generates a constraint stipulating that the argument's template, $\{\mathrm{v} = 0\}$, be a subtype of the input template κ_7 bound to *loop* in Γ_a.

To see where the "cycle" due to recursion comes in, observe that by contravariance the constraint on row 2 turns into a "flow" from κ_7 to κ_6 and the constraint on row 3 forces a "flow" from κ_6 (to which i is bound in the environment) to κ_7, which is on the right hand side of the subtyping constraint.

We assume ML type inference has put the instantiation $[\texttt{int}]$*foldn* in the expression bound to x_2. Cons generates a fresh template, κ_{20}, for the instantiated type, and upon substituting this through the schema, the instantiated function gets the template shown in row 6. The two curried applications in $[\texttt{int}]$*foldn* 42 0, yield the template and two constraints on the input templates shown in row 7. Note that the first application introduces a pending substitution θ_1 (*i.e.,*$[42/n]$) into its output template. Finally, the application of $(+)$, yields the template $\theta_1 \cdot \kappa_{20}$ (which gets bound to x_2), and the critical subtyping constraint (row 8) between the argument $(+)$ and the input template of the callee expression (row 7).

3.4 Constraint Solving

Our algorithm for solving the constraints generated by Cons proceeds in two steps. In the first step, we use the subtyping rules to *split* the complex constraints, which may contain function types, into *simple constraints* over variables with pending substitutions. In the second step, we *iteratively refine* a trivial assignment where each liquid type variable is assigned the conjunction of all logical qualifiers until all the simplified constraints are satisfied or we determine that the constraints are not satisfiable.

Splitting. In the first step, we convert all the constraints over complex types into simple constraints over liquid type variables with pending substitutions. This is done by procedure Split, shown in Figure 5, which uses the rules for well-formedness and subtyping, shown in Figure 2, to simplify the constraints.

Simple Constraints. The splitting procedure returns a set of *simple constraints* comprising simple well-formedness constraints of the form $\Gamma \vdash \rho$ and simple implication constraints of the form $\Gamma, G \vdash \rho \Rightarrow \rho'$, where ρ and ρ' are either variables with pending substitutions or boolean expressions (predicates). A liquid type assignment A satisfies a simple well-formedness constraint if $\mathsf{Shape}(\Gamma) \vdash A(\rho) : \texttt{bool}$. A satisfies a simple implication constraint if $(A(\Gamma) \wedge G \wedge A(\rho)) \Rightarrow A(\rho')$ (rule [Imp] in Figure 2). It is easy to see that splitting preserves satisfiability.

Proposition 3 ***[Splitting]*** *For any set of constraints C, a liquid type assignment A satisfies all the constraints* Split(C) *iff A satisfies all the constraints C.*

Iterative Refinement In the second step, we find a satisfying assignment for the constraints using the procedure Solve, shown in Figure 5, which takes a set of simple constraints and a finite set of logical qualifiers $\mathbb{Q}$, returns either an assignment satisfing the constraints or reports that no such assignment exists.

Solve starts with an initial assignment that maps each liquid type variable to the conjunction of *all* the logical qualifiers $\mathbb{Q}$. Solve then iteratively picks a constraint that is not satisfied and refines the solution by dropping the logical qualifiers that prevent the constraint from being satisfied.

- For unsatisfied simple well-formedness constraints of the form: $\Gamma \vdash \theta \cdot \kappa$, we remove from the assignment for κ all the qualifiers q such that the ML type of $\theta \cdot q$ (the result of applying the pending substitutions θ to q) cannot be derived to be `bool` in the corresponding ML type environment.
- For unsatisfied simple implication constraints of the form: $\Gamma, G \vdash \rho \Rightarrow \theta \cdot \kappa$, we remove from the assignment for κ all the logical qualifers q such that the implication $(A(\Gamma) \wedge G \wedge A(\rho)) \Rightarrow \theta \cdot q$ does not hold, *i.e.,*the embedded implication is not valid in EUFA according to the decision procedure.
- For unsatisfied simple implication constraints of the form: $\Gamma, G \vdash \rho \Rightarrow e$, the refinement procedure, and therefore, Solve returns **raise Failure**. Note that the refinement process *weakens* the assignment for each variable. Thus, if at some point an implication constraint of the latter form is not satisfied, it will never be satisfied in the future.

Minimal Assignments. For two assignments A and A', we say that $A \leq A'$ if for all κ, the set of logical qualifiers whose conjunction is $A(\kappa)$ *contains* the set of logical qualifiers whose conjunction is $A'(\kappa)$. It is easy to prove the following invariant about the iterative refinement loop: If A^* is an assignment that satisfies all the constraints, then the current assignment $A \leq A^*$. This invariant, together with the observation that the refinement weakens the solution, yields the following proposition, which states the correctness of our constraint solving algorithm.

Proposition 4 ***[Solving]*** *For any set of simple constraints C and logical qualifiers $\mathbb{Q}$, (1) if C has no satisfying assignment over $\mathbb{Q}$, then* Solve$(C, \mathbb{Q})$ *returns* ***raise Failure****, (2) If C has a satisfying assignment A^* over $\mathbb{Q}$, then* Solve$(C, \mathbb{Q})$ *returns a satisfying assignment A over $\mathbb{Q}$ such that $A \leq A^*$.*

In other words, if the constraints have a solution, then there exists a unique minimal solution (w.r.t. $\leq$), and $\mathsf{Solve}(C)$ finds this solution. The above steps, namely constraint generation, splitting and solving, are combined in the procedure Infer, shown in Figure 5, which takes as input an expression e and a finite set of logical qualifiers $\mathbb{Q}$, and determines whether there exists a valid liquid type derivation over $\mathbb{Q}$ for e.

Theorem 2 [Liquid Type Inference] *For any expression e and set $\mathbb{Q}$ of logical qualifers,* $\mathsf{Infer}(e, \mathbb{Q})$ *returns S iff there exists a derivation for $\emptyset, \mathit{true} \vdash e : S$ over the logical qualifiers $\mathbb{Q}$.*

Moreover, using standard worklist-based techniques, Infer can be implemented so that its running time is linear in $D \times \|\mathbb{Q}\|$ where D is the size of the ML type derivation $\emptyset \vdash e : \sigma$, and $\|\mathbb{Q}\|$ is the number of qualifiers. Of course, D can in the worst case be exponential in $\|e\|$, the size of e, but is rarely so in practice. Infer makes $D \times \|\mathbb{Q}\|$ queries to the EUFA decision procedure, and each query has size $O(\|e\| \times \|\mathbb{Q}\|)$. Though the problem of validity checking in EUFA is NP-Hard, several solvers for this theory exist which are very efficient for queries that arise in program verification.

Hybrid Checking. We note that our approach can be trivially extended to insert run-time checks when static checking fails. Notice that the simple constraints of the form $\Gamma, G \vdash \rho \Rightarrow e$, are *not* used to refine the solution, and are only used to ensure that a valid type derivation exists. We can prove that any assignment A that satisfies all the well-formedness constraints, and constraints of the form $\Gamma, G \vdash \rho \Rightarrow \rho'$, in fact gives a sound overapproximation of the run-time values. Thus, we can ensure safety in a hybrid manner [11] by inserting dynamic assertions at points corresponding to the constraints of the first kind that are not satisfied by A. As the focus of this paper is a technique for static verification, we omit the details.

Example: Polymorphic Fold (Solutions and Inferred Types) We return to the polymorphic fold example from Section 3.3. To infer the liquid types of the polymorphic fold and its application, we solve the constraints in the upper table of Figure 6. When called with the logical qualifiers $\mathbb{Q} = \{0 \leq \mathrm{v};\ \mathrm{v} < n\}$, the procedure Solve returns the following minimal satisfying assignment: $\kappa_4, \kappa_6, \kappa_7, \kappa_{20}$ map to $0 \leq \mathrm{v}$ and κ_5 maps to $(0 \leq \mathrm{v}) \wedge (\mathrm{v} < n)$. The cycle between κ_6 and κ_7 has a fixpoint with both getting $0 \leq \mathrm{v}$. Note that κ_7 (and thus, κ_6) does not get the conjunct $\mathrm{v} < n$ because of the constraint on row 3. κ_5 does, because of the guard environment in row 4. Thus, the liquid type inferred for *foldn* is

$$n{:}\{0 \leq \mathrm{v}\} \rightarrow b{:}\alpha \rightarrow f{:}(i{:}\{(0 \leq \mathrm{v}) \wedge (\mathrm{v} \leq n)\} \rightarrow c{:}\alpha \rightarrow \alpha) \rightarrow \alpha$$

Notice that the function subtyping constraint in row 8 of Figure 6 creates a self-cyclic dependency on κ_{20}. The constraint splits to $\Gamma_b[x \mapsto \kappa_5;\ y \mapsto \theta_1 \cdot \kappa_{20}], \mathit{true} \vdash \{\mathrm{v} = x + y\} <: \theta_1 \cdot \kappa_{20}$, and, as the solution for κ_5 includes the conjunct $0 \leq \mathrm{v}$, the fixpoint for the cycle is that κ_{20} is non-negative (which also satisfies the "base case" constraint in the second last row). Thus, by exploiting the polymorphic structure of the type for *foldn*, liquid type inference infers that x_2 is non-negative.

4 Example: Array Bounds Checking

The code below implements a function that takes a value of type `intarray`, a special base type in λ_{L} used for representing arrays of integers, and uses the accumulator *foldn* to return the sum of the elements of the array.

$$\begin{array}{l} \mathtt{let}\ \mathit{magnitude}\ = \\ \quad \lambda a.\ \mathit{foldn}\ (\mathtt{len}\ a)\ 0 \\ \qquad (\lambda i.\lambda s.\mathtt{let}\ z\ =\ \mathtt{sub}\ a\ i\ \mathtt{in}\ z + s) \end{array}$$

We have already generated and solved the constraints necessary to infer the liquid types of the polymorphic fold used in this example (Sections 3.3 and 3.4). Recall from Section 2.1 that the constants `len` and `sub` are used to obtain the length and elements of arrays.

Constraints. The lower table of Figure 6 shows the constraints generated for the function *magnitude*. We omit the ML types for brevity — they are the templates with all variables with pending substitutions replaced with `int`. We assume that ML type inference has inserted the required type instantiation [`int`]*foldn* for the polymorphic function *foldn*. As the expression bound to *magnitude* is a λ-abstraction, we get the fresh template $a : \mathtt{intarray} \rightarrow \kappa_8$, and generate a subtyping constraint between the template recursively

```
$\mathsf{Split}(c) =$
    $\textbf{match } c \textbf{ with}$
        $\mid (\Gamma \vdash \{\nu : B \mid \rho\}) \longrightarrow \{\Gamma[\nu \mapsto B] \vdash \rho\}$
        $\mid (\Gamma \vdash x : F_1 \to F_2) \longrightarrow$
            $\mathsf{Split}(\Gamma \vdash F_1) \cup \mathsf{Split}(\Gamma[x \mapsto F_1] \vdash F_2)$
        $\mid (\Gamma \vdash \alpha) \longrightarrow \emptyset$
        $\mid (\Gamma \vdash \forall\alpha.F) \longrightarrow \mathsf{Split}(\Gamma \vdash F)$
        $\mid (\Gamma, G \vdash x : F_1' \to F_1'' <: x : F_2' \to F_2'') \longrightarrow$
            $\mathsf{Split}(\Gamma, G \vdash F_2' <: F_1') \cup$
            $\mathsf{Split}(\Gamma[x \mapsto F_2'], G \vdash F_1'' <: F_2'')$
        $\mid (\Gamma, G \vdash \alpha <: \alpha) \longrightarrow \emptyset$
        $\mid (\Gamma, G \vdash \forall\alpha.F_1' <: \forall\alpha.F_2') \longrightarrow \mathsf{Split}(\Gamma, G \vdash F_1' <: F_2')$
        $\mid (\Gamma, G \vdash \{\_ \mid \rho\} <: \{\_ \mid \rho'\}) \longrightarrow \Gamma, G \vdash \rho \Rightarrow \rho'$

$\mathsf{Refine}(A, c) =$
    $\textbf{match } c \textbf{ with}$
        $\mid \Gamma \vdash \theta \cdot \kappa \longrightarrow$
            $\textbf{let } Q' = \{q \mid \mathsf{Shape}(\Gamma) \vdash \theta \cdot q : \texttt{bool}\} \textbf{ in}$
            $A[\kappa \mapsto A(\kappa) \cap Q']$
        $\mid \Gamma, G \vdash \theta_1 \cdot \kappa_1 \Rightarrow \theta_2 \cdot \kappa_2 \longrightarrow$
            $\textbf{let } Q' = \{q \mid (A(\Gamma) \wedge G \wedge \theta_1 \cdot A(\kappa_1)) \Rightarrow \theta_2 \cdot q\} \textbf{ in}$
            $A[\kappa_2 \mapsto A(\kappa_2) \cap Q']$
        $\mid \Gamma, G \vdash e \Rightarrow \theta \cdot \kappa \longrightarrow$
            $\textbf{let } Q' = \{q \mid (A(\Gamma) \wedge G \wedge e) \Rightarrow \theta \cdot q\} \textbf{ in}$
            $A[\kappa \mapsto A(\kappa) \cap Q']$
        $\mid \_ \longrightarrow \textbf{raise Failure}$
```

```
$\mathsf{Solve}(C) =$
    $A := \lambda\kappa.Q$
    $\textbf{while}$ exists $c \in C$ such that not $\mathsf{Sat}(A, c)$ $\textbf{do}$
        $A := \mathsf{Refine}(A, c)$
    $\textbf{return } A$

$\mathsf{Infer}(e, \mathbb{Q}) =$
    $\textbf{let } (F, C) = \mathsf{Cons}(\emptyset, \mathit{true}, e) \textbf{ in}$
    $\textbf{let } A = \mathsf{Solve}(\mathsf{Split}(C), \mathbb{Q}) \textbf{ in}$
    $A(F)$
```

Figure 5: **Liquid Type Inference:** $\mathsf{Infer}(e, \mathbb{Q})$ determines whether there exists a liquid type derivation over logical qualifers $\mathbb{Q}$ for the expression e. Solve maps a set of constraints to the maximal assignment satisfying the constraints. Split maps subtyping constraints to sets of atomic implication constraints.

computed from the body ($\langle \mathit{magnitude} \rangle$, shown in the last row) and the output type κ_8. Descending into the body, for the instantiation $[\texttt{int}]\mathit{foldn}$ we generate a fresh template of type `int` and substitute it for α to get the template shown in row 2. The first application (to `len` a) yields the template corresponding to "pushing" the pending substitution $[(\texttt{len}\ a)/n]$ through the output type of $[\texttt{int}]\mathit{foldn}$, and a subtyping constraint on the input template κ_4 (row 3). The second application (to 0) creates a constraint forcing the argument 0 to be a subtype of the input template $\theta_2 \cdot \kappa_9$ (row 3). Let us next consider the λ-abstraction passed as the final parameter to *foldn*. A fresh template is generated whose output, κ_{12}, is constrained to be a supertype of the body's template κ_{13} (row 4), in the environment extended with the formals i and s. The body, $\langle \mathit{sum} \rangle$, is analyzed with this extended environment. The curried application `sub` a i creates a constraint forcing the type of i to be subtype of the input type of `sub`, thereby creating the constraint that captures the array bounds requirement (row 5). The body of *sum* is a `let` of ML type `int`, and so we generate the fresh template κ_{13}, which is constrained to be a supertype of $s + z$ in the environment extended by binding z to `int`, the template computed for the curried application of `sub` (row 6). Recall that this template, κ_{13}, was used to constrain the output type for $\lambda i.\lambda s.\langle \mathit{sum} \rangle$ (row 4). Finally, the application of the expression from row 3 to $\lambda i.\lambda s.\langle \mathit{sum} \rangle$ yields the body $\langle \mathit{magnitude} \rangle$, with the template shown in row 7 (which is used to constrain the output type for $\lambda a.\langle \mathit{magnitude} \rangle$ in row 1). The application creates a subtyping constraint (row 7) between the *foldn* argument's template (row 4), and the input template of the callee expression (row 3).

Solutions and Inferred Types. Solving the constraints shown in the lower table of Figure 6 over the logical type qualifiers $\mathbb{Q} = \{0 \leq \nu;\ \ \nu < n;\ \ \nu < \texttt{len}\ a\}$ yields the assignment where $\kappa_8, \kappa_9, \kappa_{11}, \kappa_{12}, \kappa_{13}$ get mapped to *true* and κ_{10} gets mapped to $(0 \leq \nu) \wedge (\nu < \texttt{len}\ a)$. The function subtyping constraint in the last row, after splitting, yields $\Gamma_c, \mathit{true} \vdash \theta_2 \cdot \kappa_5 <: \kappa_{10}$. From the previous example, we saw that κ_5 gets the solution $(0 \leq \nu) \wedge (\nu < n)$ which, after applying the pending substitution, yields the solution for κ_{10}.

Thus, due to the binding $i \mapsto \kappa_{10}$ in the environment for the constraint on row 5, the array bounds check constraint is satisfied, thereby proving that this program always accesses the array safely.

As for κ_{20} in Example 2, there is a cyclic dependency involving $\kappa_8, \kappa_{11}, \kappa_{12}, \kappa_{13}$, and κ_9, the variable introduced due to the polymorphic instantiation, thus, all of these solve to *true* in the minimal fixpoint, due to the constraint in row 6 (z may be any integer). If instead, z was bound to *abs* (`sub` $a\ i$), our system would have found that the minimal solution for all of $\kappa_8, \kappa_{11}, \kappa_{12}, \kappa_{13}$ was $0 \leq \mathrm{v}$, thereby showing *magnitude* would return a non-negative integer.

	Expression	**Template**	**Constraints**
1	$[\Lambda\alpha]\lambda n.\lambda b.\lambda f.\langle \mathit{foldn}\rangle$	$\forall\alpha.(n\!:\!\kappa_4 \rightarrow b\!:\!\alpha \rightarrow f\!:\!(\kappa_5 \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha)$	
2	$\lambda i.\lambda c.\langle \mathit{loop}\rangle$	$i\!:\!\kappa_7 \rightarrow c\!:\!\alpha \rightarrow \alpha$	$\Gamma_a, \mathit{true} \vdash i\!:\!\kappa_6 \rightarrow c\!:\!\alpha \rightarrow \alpha <: i\!:\!\kappa_7 \rightarrow c\!:\!\alpha \rightarrow \alpha$
3	$\mathit{loop}\ (i+1)$	$\alpha \rightarrow \alpha$	$\Gamma_a[i \mapsto \kappa_6;\ c \mapsto \alpha], (i < n) \vdash \{\mathrm{v} = i+1\} <: \kappa_7$
4	$(f\ i)$	$\alpha \rightarrow \alpha$	$\Gamma_a[i \mapsto \kappa_6;\ c \mapsto \alpha], (i < n) \vdash \{\mathrm{v} = i\} <: \kappa_5$
5	$\mathit{loop}\ 0$	$\alpha \rightarrow \alpha$	$\Gamma_a, \mathit{true} \vdash \{\mathrm{v} = 0\} <: \kappa_7$
6	$[\mathtt{int}]\mathit{foldn}$	$n\!:\!\kappa_4 \rightarrow b\!:\!\kappa_{20} \rightarrow f\!:\!(\kappa_5 \rightarrow \kappa_{20} \rightarrow \kappa_{20}) \rightarrow \kappa_{20}$	
7	$[\mathtt{int}]\mathit{foldn}\ 42\ 0$	$f\!:\!(\kappa_5 \rightarrow \theta_1 \cdot \kappa_{20} \rightarrow \theta_1 \cdot \kappa_{20}) \rightarrow \theta_1 \cdot \kappa_{20}$	$\Gamma_b, \mathit{true} \vdash \{\mathrm{v} = 42\} <: \kappa_4$, $\Gamma_b, \mathit{true} \vdash \{\mathrm{v} = 0\} <: \theta_1 \cdot \kappa_{20}$
8	$[\mathtt{int}]\mathit{foldn}\ 42\ 0\ (+)$	$\theta_1 \cdot \kappa_{20}$	$\Gamma_b, \mathit{true} \vdash x\!:\!\mathtt{int} \rightarrow y\!:\!\mathtt{int} \rightarrow \{\mathrm{v} = x+y\} <: \kappa_5 \rightarrow \theta_1 \cdot \kappa_{20} \rightarrow \theta_1 \cdot \kappa_{20}$

	Expression	**Template**	**Constraints**
1	$\lambda a.\langle \mathit{magnitude}\rangle$	$a\!:\!\mathtt{intarray} \rightarrow \kappa_8$	$\Gamma_c, \mathit{true} \vdash [\lambda i.\lambda s.\langle \mathit{sum}\rangle/f]\theta_2 \cdot \kappa_9 <: \kappa_8$
2	$[\mathtt{int}]\mathit{foldn}$	$n\!:\!\kappa_4 \rightarrow b\!:\!\kappa_9 \rightarrow f\!:\!(\kappa_5 \rightarrow \kappa_9 \rightarrow \kappa_9) \rightarrow \kappa_9$	
3	$[\mathtt{int}]\mathit{foldn}\ (\mathtt{len}\ a)\ 0$	$f\!:\!(\theta_2 \cdot \kappa_5 \rightarrow \theta_2 \cdot \kappa_9 \rightarrow \theta_2 \cdot \kappa_9) \rightarrow \theta_2 \cdot \kappa_9$	$\Gamma_c, \mathit{true} \vdash \{\mathrm{v} = (\mathtt{len}\ a)\} <: \kappa_4$, $\Gamma_c, \mathit{true} \vdash \{\mathrm{v} = 0\} <: \theta_2 \cdot \kappa_9$
4	$\lambda i.\lambda s.\langle \mathit{sum}\rangle$	$i\!:\!\kappa_{10} \rightarrow s\!:\!\kappa_{11} \rightarrow \kappa_{12}$	$\Gamma_c[i \mapsto \kappa_{10};\ s \mapsto \kappa_{11}], \mathit{true} \vdash \kappa_{13} <: \kappa_{12}$
5	$\mathtt{sub}\ a\ i$	$\mathtt{int}$	$\Gamma_c[i \mapsto \kappa_{10};\ s \mapsto \kappa_{11}], \mathit{true} \vdash \{\mathrm{v} = i\} <: \{0 \leq \mathrm{v} \wedge \mathrm{v} < (\mathtt{len}\ a)\}$
6	$\langle \mathit{sum}\rangle$	κ_{13}	$\Gamma_c[i \mapsto \kappa_{10};\ s \mapsto \kappa_{11};\ z \mapsto \mathtt{int}], \mathit{true} \vdash \{\mathrm{v} = s+z\} <: \kappa_{13}$
7	$\langle \mathit{magnitude}\rangle$	$[\lambda i.\lambda s.\langle \mathit{sum}\rangle/f]\theta_2 \cdot \kappa_9$	$\Gamma_c, \mathit{true} \vdash i\!:\!\kappa_{10} \rightarrow s\!:\!\kappa_{11} \rightarrow \kappa_{12} <: i\!:\!\theta_2 \cdot \kappa_5 \rightarrow c\!:\!\theta_2 \cdot \kappa_9 \rightarrow \theta_2 \cdot \kappa_9$

Figure 6: ML types, templates, and constraints generated for the examples of Sections 3 and 4 (upper and lower table, respectively). We write $\langle \mathit{foldn}\rangle$, $\langle \mathit{loop}\rangle$, $\langle \mathit{magnitude}\rangle$ for the bodies of *foldn*, *loop*, and *magnitude* respectively. For brevity, we write κ for $\{\mathrm{v} : \mathtt{int} \mid \kappa\}$, and $\{e\}$ for $\{\mathrm{v} : \mathtt{int} \mid e\}$. We write Γ_1 for $[\mathit{abs} \mapsto x : \kappa_1 \rightarrow \kappa_2;\ x_1 \mapsto [\mathtt{input}\ ()/x] \cdot \kappa_2]$, Γ_a for $\Gamma_1[n \mapsto \kappa_4;\ b \mapsto \alpha;\ f \mapsto \kappa_5 \rightarrow \alpha \rightarrow \alpha;\ \mathit{loop} \mapsto \kappa_7 \rightarrow \alpha \rightarrow \alpha]$, Γ_b for $\Gamma_1[\mathit{foldn} \mapsto \forall\alpha.(n\!:\!\kappa_4 \rightarrow \alpha \rightarrow (f\!:\!\kappa_5 \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha)]$, and θ_1 for the pending substitution $[42/n]$. We write θ_2 for the pending substitution $[(\mathtt{len}\ a)/n]$, and Γ_c for $\Gamma_b[a \mapsto \mathtt{intarray}]$

5 Liquid Recursive Types

We now show how the type system and inference algorithm described so far can be smoothly extended to reason about recursively-defined datatypes. As we shall demonstrate, in this setting the ML type system and predicate abstraction combine in a truly synergistic manner to enable the automatic inference of program properties that are well beyond the approach of the individual analyses. In the sequel, we focus on recursive list types — it is straightforward (but space-consuming, and not especially edifying) to generalize the technique to full ML style recursive datatypes.

Guarded Polymorphic Lists. We extend λ_{L} with a special type for lists, defined as follows:

$$\begin{array}{lcll} \mathtt{type}\ \alpha\ \mathtt{list} & = & \mathtt{[]} & \mathtt{where}\ g_{\mathtt{[]}} \\ & \mid & \mathtt{::}\ \mathtt{of}\ (x_1\!:\!\alpha, x_2\!:\!\alpha\ \mathtt{list}) & \mathtt{where}\ g_{\mathtt{::}} \end{array}$$

This declaration is almost the same as in ML, except that as for functions, we *name* the parameters passed to the constructors, and, for each constructor, `[]` and `::`, we have a *guard*, a λ_{L} boolean expression over v and the variables used by the constructor, that describes some property of the *constructed* expression in terms of the properties of the expressions corresponding to the variables.

For lists, the guard could relate the *size* of the constructed list to the sizes of x_2:

$$\begin{array}{lcl} g_{\mathtt{[]}} & \stackrel{\triangle}{=} & \mathtt{size}\ v\ =\ 0 \\ g_{\mathtt{::}} & \stackrel{\triangle}{=} & \mathtt{size}\ v\ =\ (1 + \mathtt{size}\ x_2) \end{array}$$

where `size` is a special constant, similar to `len` for the `intarray` type. Figure 7 shows how the language of expressions is extended to handle lists: in addition to expressions `[]` and `::` used to create lists, we have the usual `match` expression to operate on lists.

Liquid Lists. Figure 7 shows how we extend the language of types to incorporate lists. A *dependent list type* is of the form $\{\mathrm{v} : T\ \texttt{list} \mid e\}$ where T is an *element dependent type* describing every *element* of the list, and e is a *list refinement predicate* constraining the value of the list itself. A *liquid list type* is a dependent list type where all the refinement predicates are conjunctions of predicates from the set of logical qualifiers $\mathbb{Q}$. We write $T\ \texttt{list}$ as an abbreviation for $\{\mathrm{v} : T\ \texttt{list} \mid \mathit{true}\}$. Thus, an *ML list type* is a dependent list type where all refinement predicates are *true*.

For example, the type $\{\mathrm{v} : \texttt{int} \mid 0 \leq \mathrm{v}\}\ \ \texttt{list}$ specifes lists of non-negative integers. The type

$$\{\mathrm{v} : \{\mathrm{v}' : \texttt{int} \mid 100 \leq \mathrm{v}'\}\ \texttt{list} \mid 0 < \texttt{size}\ \mathrm{v}\}$$

specifies *non-empty* lists of integers greater than or equal to 100. The fact that the list is non-empty, or rather, not a `[]` value, is implied by the list refinement predicate which implies that the list cannot satisfy the guard $g_{\texttt{[]}}$ for `[]` lists, a property formalized via our type checking rules, described next.

Syntax

$$\begin{array}{lll}
e & ::= \ \ldots & \textit{Expressions:} \\
 & \mid \texttt{[]} & \text{list-empty} \\
 & \mid e\texttt{::}e & \text{list-cons} \\
 & \mid (\texttt{match}\ e\ \texttt{with}\ \texttt{[]}\!\rightarrow e \mid x_1\texttt{::}x_2\!\rightarrow e) & \text{list-match} \\
\mathbb{T}(\mathbb{B}) & ::= \ \ldots & \textit{Skeletons:} \\
 & \mid \{\mathrm{v} : \mathbb{T}(\mathbb{B})\ \texttt{list} \mid \mathbb{B}\} & \text{list type}
\end{array}$$

Subtyping $\boxed{\Gamma, G \vdash S_1 <: S_2}$

$$\frac{\Gamma, G \vdash T_1 <: T_2 \qquad \Gamma, G \vdash e_1 \Rightarrow e_2}{\Gamma, G \vdash \{\mathrm{v} : T_1\ \texttt{list} \mid e_1\} <: \{\mathrm{v} : T_2\ \texttt{list} \mid e_2\}}\ [<:\text{-}\textsc{List}]$$

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\Gamma \vdash S \qquad \Gamma[\mathrm{v} \mapsto S\ \texttt{list}] \vdash e : \texttt{bool}}{\Gamma \vdash \{\mathrm{v} : S\ \texttt{list} \mid e\}}\ [\text{WF-}\textsc{List}]$$

Liquid Type Checking $\boxed{\Gamma, G \vdash e : S}$

$$\frac{}{\Gamma, G \vdash \texttt{[]} : \{\mathrm{v} : \hat{T}\ \texttt{list} \mid g_{\texttt{[]}}\}}\ [\hat{T}\text{-}\textsc{Nil}]$$

$$\frac{\Gamma, G \vdash e_1 : \hat{T} \qquad \Gamma, G \vdash e_2 : \hat{T}\ \texttt{list}}{\Gamma, G \vdash e_1\texttt{::}e_2 : \{\mathrm{v} : \hat{T}\ \texttt{list} \mid [e_1, e_2/x_1, x_2]g_{\texttt{::}}\}}\ [\hat{T}\text{-}\textsc{Cons}]$$

$$\frac{\begin{array}{c}\Gamma, G \vdash e_1 : \{\mathrm{v} : T\ \texttt{list} \mid e\} \\ \Gamma, G \wedge [e_1/\mathrm{v}]g_{\texttt{[]}} \vdash e_2 : \hat{T} \qquad \Gamma[x_1 \mapsto T;\ x_2 \mapsto T\ \texttt{list}], G \wedge [e_1/\mathrm{v}]g_{\texttt{::}} \vdash e_3 : \hat{T}\end{array}}{\Gamma, G \vdash (\texttt{match}\ e_1\ \texttt{with}\ \texttt{[]}\!\rightarrow e_2 \mid x_1\texttt{::}x_2\!\rightarrow e_3) : \hat{T}}\ [\hat{T}\text{-}\textsc{Match}]$$

Figure 7: Rules for Lists

Type Checking. Figure 7 shows the syntax-directed type checking rules for lists. The rule for `[]` specifies that the empty list can have any liquid element type, and the refinement predicate is $g_{\texttt{[]}}$. The rule for `::` stipulates that the element type of the constructed list must be a liquid type such that the "head" element has that type, and the "tail" is a list of elements of that liquid type. The rule ensures that whenever a list is constructed, its element type is liquid, as it must over-approximate (*i.e.,*be the "join" of) the head and tail values. The refinement predicate is the guard predicate with the variables x_1, x_2 from the list type definition substituted with the actual expressions passed to the constructor, analogous to the result of function application, rule $\hat{T}$-APP from Figure 2. The rule for the `match` expression checks the expression being pattern-matched on is a list, and uses the type of the list and the constructor guards to extend the type and guard environments appropriately when checking the individual cases. Again, as this is a "join" point, we require that the type of the entire `match` expression be liquid. Theorem 1 about the soundness of the typechecking rules continues to hold with lists.

Type Inference. As before, we assume that the expression typechecks under the ML type system, and use the results of ML type inference to generate a system of constraints in a syntax-directed manner mimicking the typechecking rules. For brevity, we have omitted the defintions of Shape, Fresh, and Cons, and Push for lists. Observe that we have already given such definitions for a datatype constructor, $\rightarrow$, which is contravariant in its first arugment and covariant in its second. The definitions of these functions applied to covariant lists follow a similar pattern.

Once the constraints are generated and split, solving proceeds exactly as before, via the iterative refinement described in Figure 5. Propositions 3 and 4 and Theorem 2 about the correctness and running time of the inference procedure continue to hold in this setting.

Example. We now show a small example to illustrate how our system works. It is easy to check that, using the set of logical qualifiers $\mathbb{Q} = \{0 \leq \mathrm{v};\ 0 < \mathtt{size}\ \mathrm{v}\}$, our system allows us to derive:

$$\emptyset, \mathit{true} \vdash \mathtt{[]} : \{\mathrm{v} : \{\mathrm{v} : \mathtt{int} \mid 0 \leq \mathrm{v}\}\ \mathtt{list} \mid \mathtt{size}\ \mathrm{v} = 0\}$$

and from this, derive:

$$\begin{aligned} \emptyset, \mathit{true} &\vdash \mathtt{let}\ \mathit{nil} = \mathtt{[]}\ \mathtt{in}\ (1\mathtt{::}\mathit{nil}) : \{\mathrm{v} : \mathtt{int} \mid 0 \leq \mathrm{v}\}\ \mathtt{list} \\ \emptyset, \mathit{true} &\vdash \mathtt{let}\ \mathit{nil} = \mathtt{[]}\ \mathtt{in}\ (1\mathtt{::}\mathit{nil}) : \{\mathrm{v} : \mathtt{int}\ \mathtt{list} \mid 0 < \mathtt{size}\ \mathrm{v}\} \end{aligned}$$

Using the above qualifiers, our rules allow us to check that:

$$\begin{array}{l} \mathtt{let}\ \mathit{nil}\ = \mathtt{[]}\ \mathtt{in} \\ \mathtt{let}\ x\ = 1\mathtt{::}\mathit{nil}\ \mathtt{in} \\ (\mathtt{match}\ x\ \mathtt{with}\ \mathtt{[]} \rightarrow \mathtt{error}\ 0 \mid x_1\mathtt{::}x_2 \rightarrow x_1) \end{array}$$

has the type $\{\mathrm{v} : \mathtt{int} \mid 0 \leq \mathrm{v}\}$. That is, the type system can statically infer that the `[]` pattern is *never matched*, and the result of the expression is non-negative. This is because in the `[]` case, the application `error` 0 is checked under the type environment: $\Gamma \overset{\triangle}{=} [x \mapsto \{\mathrm{v} : \{\mathrm{v} : \mathtt{int} \mid 0 \leq \mathrm{v}\}\ \mathtt{list} \mid 0 < \mathtt{size}\ \mathrm{v}\}]$ and guard environment $G \overset{\triangle}{=} \mathtt{size}\ x = 0$ obtained by substituting x for v in $g_{\mathtt{[]}}$. It is easy to see that in this case, $\Gamma, G \vdash e \Rightarrow \mathit{false}$ and so the application to `error` typechecks, and its result has the type `false` which is a subtype of $\{\mathrm{v} : \mathtt{int} \mid 0 \leq \mathrm{v}\}$. In the `::` case, the x_1 is checked under an environment where its type is bound to the list refinement predicate, and thus can be shown to be non-negative.

A-Normalization. Note that derivations like the one at the end of the previous section fail without the use of *nil* (*i.e.,*if a type constructor like `::` is applied to arbitrary expressions whose types are not "bound" in the environment). However, we can simply sidestep this issue by first converting to *A-Normal Form*[14], where all the intermediate subexpressions get bound to temporary variables, thereby allowing us to infer the strongest possible liquid types.

6 Experimental Results

As a proof of concept, we have implemented type inference for λ_{L}. Our implementation is capable of inferring liquid types given in all worked examples appearing in this paper; we now demonstrate the expressiveness and flexibility of our technique by showing some interesting examples that our implementation can prove safe via liquid type inference. For clarity, we have elided quantifiers $\forall\,\alpha$ from polymorphic types.

Divide By Zero. Conside the integer truncation function "Truncation", shown in Figure 8. Recall that / is a constant of the type $x\!:\!\mathtt{int} \rightarrow y\!:\!\{\mathrm{v} : \mathtt{int} \mid \mathrm{v} \neq 0\} \rightarrow \mathtt{int}$. Using the logical qualifiers $\mathbb{Q} = \{0 \leq \mathrm{v}\}$, our system is able to infer that *abs* from Section 1 has the liquid type $\mathtt{int} \rightarrow \{\mathrm{v}\!:\!\mathtt{int} \mid 0 \leq \mathrm{v}\}$. Thus, both i_a

and n_a have the type $\{\mathrm{v}\!:\!\texttt{int} \mid 0 \leq \mathrm{v}\}$. This, coupled with the guard $\neg(i_a \leq n_a)$ in the else branch allows the system to infer that the i_a passed to the divide function (in the else branch) has the type $\{\mathrm{v} : \texttt{int} \mid 0 < \mathrm{v}\}$, a subtype of the input type, allowing the type system to statically prove that no divide by zero errors occur at runtime.

Array Bounds Violations. Consider the function *bsearch* from [30], shown in the "Binary Search" box of Figure 8. Our system is able to infer that all array accesses in the are safe using just the logical qualifiers $\mathbb{Q} = \{0 \leq \mathrm{v};\ \ \mathrm{v} < \texttt{len}\ a\}$. The generated constraints have the minimal assignment which yields the liquid type: $l\!:\!\{\mathrm{v}\!:\!\texttt{int} \mid 0 \leq \mathrm{v}\} \rightarrow h\!:\!\{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} < (\texttt{len}\ a)\} \rightarrow \texttt{int}$ for the function *look*. Note that the subtyping constraints from the curried application *look* $0\ (\texttt{len}\ a - 1)$ are trivially satisfied by this assignment. It is easy to check (and for the theorem prover to prove) that in an environment where l and h are bound to the types specified above, and where $l \leq h$, the expression bound to m, and therefore m, is also non-negative and less than $\texttt{len}\ a$, thereby meeting the subtyping requirements at the array access applications. Thus at both the recursive call sites inside *look*, the arguments are subtypes of the parameters for *look* in the assignment described above.

In a similar manner, we can statically prove that all the array accesses from the "Dot Product" function *dotprod* of Figure 8 (adapted from [30]) are safe, using the logical qualifiers: $\mathbb{Q} = \{0 \leq \mathrm{v};\ \ \mathrm{v} \leq \texttt{len}\ u;\ \ \mathrm{v} \leq \texttt{len}\ v\}$. The system infers the liquid type: $i : \{\mathrm{v}\!:\!\texttt{int} \mid 0 \leq \mathrm{v}\} \rightarrow n : \{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} \leq (\texttt{len}\ u) \wedge \mathrm{v} \leq (\texttt{len}\ v)\} \rightarrow \texttt{int}$ for the function *loop*. To see why this is a valid fixpoint solution, observe that under the environment where i and n are bounded as specified above, and $\neg(i \geq n)$ (in the else branch): (1) the value i used to access the array has a type that is within bounds, and, (2) $(i+1)$ is non-negative, and n continues to be bounded as above, meeting the subtype requirements at the recursive callsite. The "base" application also meets the requirements as $0 \leq 0$ and the guard in the branch ensures that N gets the liquid type: $\{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} \leq (\texttt{len}\ u) \wedge \mathrm{v} \leq (\texttt{len}\ v)\}$ meeting the requirements of the inferred type for *loop*. If the dot product were computed using an accumulator like *foldn* instead, our system would still be able to prove the accesses safe, using the same set of qualifers, using reasoning similar to that used for *magnitude* in Section 4.

List Data. Next, we show a few examples that illustrate how the liquid type inference algorithm can statically prove properties of programs manipulating recursive data structures. The box "Generate" in Figure 8 shows a function *generate* that, given a parameter n, a base value b, and function f, generates the list $[f^0(b);\ \ldots;\ f^n(b)]$. ML type inference finds that *generate* has the ML type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \texttt{int} \rightarrow \alpha\ \texttt{list}$. Using the qualifier set $\mathbb{Q} = \{0 < \mathrm{v}\}$, we are able to determine that the function *double* has the type $k\!:\!\{\mathrm{v} : \texttt{int} \mid 0 < \mathrm{v}\} \rightarrow \{\mathrm{v} : \texttt{int} \mid 0 < \mathrm{v}\}$, *i.e.,*when applied to a positive number, it returns a positive number. As the "base" parameter 1 is positive, our system infers that that the list generated by *generate double* 1 10 has the type $\{\mathrm{v}\!:\!\texttt{int} \mid 0 < \mathrm{v}\}\ \ \texttt{list}$.

This is effected by automatically instantiating the polymorphic type variable α with a fresh liquid type variable κ. and generating the constraints: $\cdot \vdash \kappa_1 \rightarrow \kappa_2 <: \kappa \rightarrow \kappa$, $\cdot \vdash \{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} = 1\} <: \kappa$, for the curried application to *generate*, where $\kappa_1 \rightarrow \kappa_2$ is the template for *double*, whose body generates the constraint: $[k \mapsto \kappa_1], \mathit{true} \vdash \{\mathrm{v}\!:\!\texttt{int} \mid \mathrm{v} = k + k\} <: \kappa_2$. As the minimal satisfying assignment to these constraints maps $\kappa, \kappa_1, \kappa_2$ to $0 < \mathrm{v}$, the system infers that the output is a list of positive integers.

Uninterpreted Functions. Next, we show an example illustrating how uninterpreted functions combine with polymorphism to allow us to statically prove properties that may at first blush seem only to be within the grasp of dynamic checking. Consider the *mapfilter* function shown in "Map Filter" of Figure 8. λ_{L} can encode the polymorphic option type in a manner akin to lists. *mapfilter* has the type: $(\alpha \rightarrow \beta\ \texttt{option}) \rightarrow \alpha\ \ \texttt{list} \rightarrow \beta\ \ \texttt{list}$. Using the logical qualifier *prime* v, and generating constraints on fresh liquid type variables corresponding to the instantiation of polymorphic type variables, our system infers that xs' has the liquid type $\{\mathrm{v} : \texttt{int} \mid \mathit{prime}\ \mathrm{v}\}\ \texttt{list}$, and so x_1 has the liquid type $\{\mathrm{v} : \texttt{int} \mid \mathit{prime}\ \mathrm{v}\}$, which, by treating applications of *prime* as an applications of uninterpreted functions, as is done in our embedding to the decidable logic, suffices to typecheck the program. Thus, we prove that the `error` in the else branch is never called! Of course, the system has not proved that x_1 is a prime integer, merely that applications *prime* x_1 always evaluate to *true*. We note that our system would not work for the usual filter function as we currently prohibit refinements of values whose base type is a polymorphic type variable.

List Sizes. The box "Append" in Figure 8 shows the *append* function on two lists. Using only the logical qualifier: $\mathbb{Q} = \{\texttt{size}\ \mathrm{v} = \texttt{size}\ l + \texttt{size}\ m\}$, our system infers that *append* has the liquid type $l\!:\!\alpha\ \texttt{list} \rightarrow$

```
let $trunc = \lambda n.\ \lambda i.$                                  $\boxed{\textbf{Truncation}}$
   let $i_a = abs\ i$ in
   let $n_a = abs\ n$ in
   if $i_a \leq n_a$ then $i$ else $n_a * (i/i_a)$
```

```
let $bsearch = \lambda k.\lambda a.$                                $\boxed{\textbf{Binary Search}}$
   letrec $look = \lambda l.\lambda h.$
      if $l \leq h$ then
         let $m = l + ((h - l)/2)$ in
         if (sub $a\ m) = k$ then $m$ else
            if (sub $a\ m) < k$ then $look\ l\ (m - 1)$
            else $look\ (m + 1)\ h$
      else $(-1)$
   in $look\ 0$ ((len $a) - 1)$
```

```
let $dotprod = \lambda u.\lambda v.$                                $\boxed{\textbf{Dot Product}}$
   letrec $loop = \lambda i.\lambda n.\lambda s.$
      if $n \leq i$ then $s$
      else $loop\ (i + 1)\ n\ (s + ($(sub $u\ i) *$ (sub $v\ i$))) in
   let $N =$ if len $u <$ len $v$ then (len $u$) else (len $v$) in
   $loop\ 0\ N\ 0$
```

```
letrec $generate = \lambda f.\lambda b.\lambda n.$                         $\boxed{\textbf{Generate}}$
   if $n = 0$ then $b$::[]
   else let $h = f\ b$ in $h$::($generate\ (n - 1)\ f\ h$) in
let $double = \lambda k.k + k$ in
   $generate\ double\ 1\ 10$
```

```
letrec $mapfilter = \lambda f.\lambda l.$                            $\boxed{\textbf{MapFilter}}$
   match $l$ with [] $\rightarrow$ []
   | $(x_1$::$x_2) \rightarrow$
      match $f\ h$ with None $\rightarrow mapfilter\ f\ x_2$
      | Some $x \rightarrow x$::($mapfilter\ f\ x_2$) in
...
let $prime = \lambda x.(*$ tricky primality test$*)$ in
...
let $xs' =$
   $mapfilter$ ($\lambda x.$ if $prime\ x$ then Some $x$ else None) $xs$
in ...
match $xs'$ with [] $\rightarrow$ ...
| $x_1$::$x_2 \rightarrow$ if $prime\ x_1$ then ...else error 0
```

```
letrec $append = \lambda l.\lambda m.$                                $\boxed{\textbf{Append}}$
   match $l$ with [] $\rightarrow m$
   | $x_1$::$x_2 \rightarrow (x_1$::($append\ x_2\ m$))
```

```
let $pow2 = \lambda n.$                                           $\boxed{\textbf{Power 2}}$
   let $x = generate\ double\ 1\ n$
   (match $x$ with [] $\rightarrow$ error 0 | $x_1$::$x_2 \rightarrow x_1$)
```

Figure 8: Liquid Type Examples

$m{:}\alpha$ list $\rightarrow L(l, m)$ where $L(l, m)$ is an abbreviation for $\{$v$:\alpha$ list | size v = size l + size $m\}$.

To derive this type, our system infers that both branches of the match return values of the type $L(l, m)$.

This trivially holds for the `[]` case, which is is evaluated under a guard environment strengthened with $[l/\mathrm{v}]g_{\texttt{[]}}$. In the $x_1\texttt{::}x_2$ case, notice that the guard environment contains contains the predicate $[l/\mathrm{v}]g_{\texttt{::}}$, which is $\texttt{size}\ l = \texttt{size}\ x_2 + 1$. In the fixpoint solution, the system uses the type environment assumption that the return value of *append* is $L(m, n)$ to infer that the expression *append* x_2 m has the type $\{\mathrm{v}\!:\!\alpha\ \texttt{list} \mid \texttt{size}\ \mathrm{v} = \texttt{size}\ x_2 + \texttt{size}\ m\}$. This, coupled with the guard predicate $g_{\texttt{::}}$, allows the system to infer that the type of $x_1\texttt{::}(\mathit{append}\ x_2\ m)$ is $\{\mathrm{v}\!:\!\alpha\ \texttt{list} \mid \texttt{size}\ \mathrm{v} = \texttt{size}\ x_2 + \texttt{size}\ m + 1\}$. As this occurs in the case where l matches with $x_1\texttt{::}x_2$, the predicate $[l/\mathrm{v}]g_{\texttt{::}}$ in the guard environment allows the system to infer in that environment, $\{\texttt{size}\ \mathrm{v} = \texttt{size}\ x_2 + \texttt{size}\ m + 1\}$ is a subtype of $\{\texttt{size}\ \mathrm{v} = \texttt{size}\ l + \texttt{size}\ m\}$, thereby showing that the inferred type is indeed a fixpoint.

Using similar reasoning, our system infers, using only the logical qualifers $\mathbb{Q} = \{\texttt{size}\ \mathrm{v} \leq \texttt{size}\ l\}$, that *mapfilter* from box "Map Filter" in Figure 8, has the liquid type:

$$(\alpha \rightarrow \beta\ \texttt{option}) \rightarrow \alpha\ \texttt{list} \rightarrow \{\mathrm{v}\!:\!\beta\ \ \texttt{list} \mid \texttt{size}\ \mathrm{v} \leq \texttt{size}\ l\}$$

and a similar type for the usual filter function (here the required refinement is on the list, not a polymorphic type variable). Note that, by constraining the output value's size, our system avoids the need for existentially quantified types [30].

Pattern Match Errors. Finally, we note that liquid types can be used to statically prove the redundancy of certain cases of match expressions. Using only the logical qualifiers $\mathbb{Q} = \{0 < \texttt{size}\ \mathrm{v}\}$ our system infers that the function *generate* in the box "Generate", has the liquid type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \texttt{int} \rightarrow \{\mathrm{v}\!:\!\alpha\ \texttt{list} \mid 0 < \texttt{size}\ \mathrm{v}\}$. Consider the function *pow2* in the box "Power 2" in Figure 8. Our system uses the liquid type inferred for *generate*, to infer that x has the liquid type $\{\mathrm{v}\!:\!\texttt{int list} \mid 0 < \texttt{size}\ \mathrm{v}\}$, *i.e.,*is not an empty list. Using reasoning similar to the example from Section 5, our system is able to typecheck *pow2* by showing that `error` is never called.

7 Related Work

The first component of our approach is predicate abstraction [19]which has its roots in early work on axiomatic semantics [15, 22] and predicate transformers [9]. While it is very effective for control-dominated software, it is less effective for automated reasoning about complex data and higher-order control flow, as the decision procedures that it critically relies on [27] are limited in their ability to reason about such structures.

The second component of our approach is ideas from constraint-based program analysis. One can view the ML type inference algorithm, from which we draw inspiration, as an instance of such an analysis. Our work focuses on full inference for a particular instantiation of a type system with subtyping parameterized by a constraint language; more general approaches are studied in [29, 28]. The problem of type inference for a particular Hindley-Milner-style type system with subtyping is also investigated by several authors, including [25, 18, 12]. Our work also draws inspiration from the discipline of type qualifiers [10, 16] that refine types with a lattice of built-in and programmer-specified annotations. Liquid types extend qualifers by assigning them semantics via predicates interpreted in a logic of arithmetic, equality and uninterpreted functions, and our inference algorithm combines value flow (via the subtyping constraints) with information drawn from guards and assignments. The idea of assigning semantics to qualifiers has been proposed recently [4], but with the intention of checking the soundness of programmer-specified rules for qualifier derivations. Our approach is complementary in that the rules themselves are fixed, but allow for the use of arbitrary guard and value binding information in their derivation. This leads to a more powerful analysis (we believe none of the examples in our paper could be proven by safe by the approach of [4]), but a slower one as the decision procedure is integrated with type inference.

The notion of type refinements was introduced in [17] with refinements limited to restrictions on the structure of algebraic datatypes, for which inference is decidable. DML(C) [31] extends ML with dependent types over a constraint domain C; type checking is shown to be decidable modulo the decidability of the domain, but inference is still undecidable. Many other authors have looked at the problem of checking types described via refinement predicates over different logics [20], and the interaction of refinements and effects [8, 26]. Liquid types can be viewed as a controlled way to extend the language of types using simple predicates over a decidable logic, such that both checking and inference remain decidable. Our notion of variables with pending substitutions is inspired by a construct from [24], which presents a technique to reconstruct the dependent type of an expression that captures its *exact* semantics (analogous to strongest postconditions for imperative languages). The technique works in a restricted setting without recursive types or polymorphism. Moreover, the reconstructed types are terms containing existentially quantified variables (due to variables that are not in scope), and the `fix` operator (used to handle recursion), which make static reasoning problematic.

Conclusions and Future Work. We have shown how to combine predicate abstraction and ML type inference to obtain a simple and efficient dependent type inference algorithm capable of inferring properties that are well beyond the capabilities of predicate abstraction, which requires significantly fewer and simpler annotations than previous dependent type systems. The only manual annotation required by our system in order to prove a variety of safety properties are simple atomic inequalities that can be derived either via syntactic methods [13, 7], or using a proof-theoretic approach [21]. We defer the task of adapting these techniques to our setting to future work. We would like to implement the algorithm for OCaml — which will require us to extend the system to include some reasoning about imperative features. Finally, we would like apply these ideas to check C++, Java and C# programs that make heavy use of generic data structures.

References

[1] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

[2] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[3] S. Chaki, J. Ouaknine, K. Yorav, and E.M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC*, 2003.

[4] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI*. ACM, 2005.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[7] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI*, pages 310–324, 2003.

[8] R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP*, 2000.

[9] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] D. Evans. Static detection of dynamic memory errors. In *PLDI*, 1996.

[11] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[12] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, 1999.

[13] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*. ACM, 2002.

[14] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[15] R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[16] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*. ACM, 1999.

[17] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[18] Y.C. Fuh and P. Mishra. Type inference with subtypes. In *ESOP*, 1988.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, LNCS 1254, pages 72–83. Springer, 1997.

[20] S. Hayashi. Logic of refinement types. In *TYPES*, pages 108–126, 1993.

[21] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*. ACM, 2004.

[22] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[23] H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, pages 397–412, 2005.

[24] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, 2007.

[25] P. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *POPL*, Albequerque, New Mexico, 1992.

[26] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP*, pages 213–225, 2003.

[27] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[28] F. Pottier. Simplifying subtyping constraints. In *ICFP*, New York, NY, USA, 1996. ACM Press.

[29] M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types. In *FOOL*, 1997.

[30] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[31] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

[32] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.