Liquid Types [*]

Patrick M. Rondon Ming Kawaguchi Ranjit Jhala

University of California, San Diego

{prondon,mwookawa,jhala}@cs.ucsd.edu

Abstract

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system that combines *Hindley-Milner* type inference with *Predicate Abstraction* to automatically infer dependent types precise enough to prove a variety of safety properties. Liquid types allow programmers to reap many of the benefits of dependent types, namely static verification of critical properties and the elimination of expensive run-time checks, without the heavy price of manual annotation. We have implemented liquid type inference in DSOLVE, which takes as input an OCAML program and a set of logical qualifiers and infers dependent types for the expressions in the OCAML program. To demonstrate the utility of our approach, we describe experiments using DSOLVE to statically verify the safety of array accesses on a set of OCAML benchmarks that were previously annotated with dependent types as part of the DML project. We show that when used in conjunction with a fixed set of array bounds checking qualifiers, DSOLVE reduces the amount of manual annotation required for proving safety from 31% of program text to under 1%.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Dependent Types, Hindley-Milner, Predicate Abstraction, Type Inference

1. Introduction

Modern functional programming languages, like ML and Haskell, have many features that dramatically improve programmer productivity and software reliability. Two of the most significant are strong static typing, which detects a host of errors at compile-time, and type inference, which (almost) eliminates the burden of annotating the program with type information, thus delivering the benefits of strong static typing for free.

[*] This work was supported by NSF CAREER grant CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603, and a gift from Microsoft Research.

The utility of these type systems stems from their ability to predict, at compile-time, invariants about the run-time values computed by the program. Unfortunately, classical type systems only capture relatively coarse invariants. For example, the system can express the fact that a variable `i` is of the type `int`, meaning that it is always an integer, but not that it is always an integer within a certain range, say between 1 and 99. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by `i`, or the accessing of an array `a` of size 100 at an index `i`. Instead, the language can only provide a weaker dynamic safety guarantee at the additional cost of high performance overhead.

In an exciting development, several authors have proposed the use of *dependent types* [20] as a mechanism for enhancing the expressivity of type systems [14, 27, 2, 22, 10]. Such a system can express the fact

$$\mathtt{i} :: \{\nu{:}\mathtt{int} \mid 1 \leq \nu \wedge \nu \leq 99\}$$

which is the usual type `int` together with a *refinement* stating that the run-time value of `i` is an always an integer between 1 and 99. Pfenning and Xi devised DML, a practical way to integrate such types into ML, and demonstrated that they could be used to recover static guarantees about the safety of array accesses, while simultaneously making the program significantly faster by eliminating run-time checking overhead [27]. However, these benefits came at the price of automatic inference. In the DML benchmarks, about 31% of the code (or 17% by number of lines) is manual annotations that the typechecker needs to prove safety. We believe that this nontrivial annotation burden has hampered the adoption of dependent types despite their safety and performance benefits.

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system for automatically inferring dependent types precise enough to prove a variety of safety properties, thereby allowing programmers to reap many of the benefits of dependent types without paying the heavy price of manual annotation. The heart of our inference algorithm is a technique for blending Hindley-Milner type inference with *predicate abstraction*, a technique for synthesizing loop invariants for imperative programs that forms the algorithmic core of several software model checkers [3, 16, 4, 29, 17]. Our system takes as input a program and a set of *logical qualifiers* which are simple boolean predicates over the program variables, a special *value variable* ν, and a special placeholder variable $\star$ that can be instantiated with program variables. The system then infers *liquid types*, which are dependent types where the refinement predicates are *conjunctions* of the logical qualifiers.

In our system, type checking and inference are decidable for three reasons (Section 3). 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, each of which is deemed to hold if and only if an *embedding* of the implication into a decidable logic yields a valid formula in the logic. Second, 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. Third, in any valid type derivation, the types of certain expressions, such as λ-abstractions, if-then-else expressions, and recursive functions must be liquid. Thus, inference becomes decidable, as the space of possible types is bounded. We use these features to design a three-step algorithm for dependent type inference (Section 4).

Step 1: Hindley-Milner Type Inference: First, our algorithm invokes Hindley-Milner [7] to infer types for each subexpression and the necessary type generalization and instantiation annotations. Next, our algorithm uses the computed ML types to assign to each subexpression a *template*, a dependent type with the same structure as the inferred ML type, but which has *liquid type variables* representing the unknown type refinements.

Step 2: Liquid Constraint Generation: Second, 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.

Step 3: Liquid Constraint Solving: 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 [1, 15].

Of course, there may be safe programs which cannot be well-typed in our system due either to an inappropriate choice of qualifiers or the conservativeness of our notion of subtyping. In the former case, we can use the readable results of the inference to manually 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 [22, 10].

To validate the utility of our technique, we have built DSOLVE, which infers liquid types for OCAML programs. While liquid types can be used to statically prove a variety of properties [24], in this paper we focus on the canonical problem of proving the safety of array accesses. We use a diverse set of challenging benchmarks taken from the DML project to demonstrate that DSOLVE, together with a simple set of *array bounds checking* qualifiers, can prove safety completely automatically for many programs (Section 5). For the few programs where these bounds checking qualifiers are insufficient, the programmer typically only needs to specify one or two extra qualifiers. Even in these rare cases, the dependent types inferred by DSOLVE using only the bounds checking qualifiers help the programmer to rapidly identify the relevant extra qualifiers. We show that, over all the benchmarks, DSOLVE reduces the manual annotation required to prove safety from 31% of program text (or 17% by number of lines) to under 1%. Finally, we describe a case study where DSOLVE was able to pinpoint an error in an open-source OCAML bit vector library implementation, in a function that contained an explicit (but insufficient) safety check.

2. Overview

We begin with an overview of our algorithm for inferring dependent types using a set of logical qualifiers $\mathbb{Q}$. First, we describe dependent types, logical qualifiers, and liquid types, and then, through a series of examples, we show how our system infers dependent types.

Dependent Types. Following [2, 10], our system allows *base refinements* of the form $\{\nu : B \mid e\}$, where ν is a special *value variable* not appearing in the program, B is a *base type* and e is a boolean-valued expression constraining the value variable called the *refinement predicate*. Intuitively, the base refinement predicate specifies the set of values c of the base type B such that the predicate $[c/\nu]e$ evaluates to true. For example, $\{\nu : \texttt{int} \mid 0 < \nu\}$ specifies the set of positive integers, and $\{\nu : \texttt{int} \mid \nu \leq \texttt{n}\}$ specifies the set of integers whose value is less than or equal to the value of the variable n. Thus, B is an abbreviation for $\{\nu : B \mid \mathit{true}\}$. We use the base refinements to build up *dependent function types*, written $x : T_1 \rightarrow T_2$ (following [2, 10]). 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.

Logical Qualifiers and Liquid Types. A *logical qualifier* is a boolean-valued expression (*i.e.,* predicate) over the program variables, the special value variable ν which is distinct from the program variables, and the special placeholder variable $\star$ that can be instantiated with program variables. For the rest of this section, let us assume that $\mathbb{Q}$ is the set of logical qualifiers $\{0 \leq \nu,\ \star \leq \nu,\ \nu < \star,\ \nu < \texttt{len}\ \star\}$. In Section 5 we describe a simple set of qualifiers for array bounds checking. We say that a qualifier q *matches* the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $x \leq \nu$ matches the qualifier $\star \leq \nu$. We write $\mathbb{Q}^\star$ for the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. For example, when $\mathbb{Q}$ is as defined as above, $\mathbb{Q}^\star$ includes the qualifiers $\{0 \leq \nu,\ \texttt{x} \leq \nu,\ \texttt{y} \leq \nu,\ \texttt{k} \leq \nu,\ \nu < \texttt{n},\ \nu < \texttt{len a}\}$. A *liquid type over* $\mathbb{Q}$ is a dependent type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We write *liquid type* when $\mathbb{Q}$ is clear from the context. When checking or inferring dependent types over the logical qualifiers, our system ensures that the types are *well-formed*, *i.e.,* for each subexpression, the free variables appearing in the inferred type are bound in the environment for that subexpression.

Liquid Type Inference. Our liquid type inference algorithm proceeds in three steps. First, we perform Hindley-Milner (HM) type inference and use the results to generate *templates*, which are dependent types with unknown base refinements represented by *liquid type variables* κ. Second, we generate constraints on the templates that capture the subtyping relationships between the refinements. Third, we solve the constraints by using predicate abstraction to find, for each κ, the *strongest* conjunction of qualifiers from $\mathbb{Q}^\star$ that satisfies all the constraints. Note that for the third step, we need only consider the finite subset of $\mathbb{Q}^\star$ whose free variables belong to the program. Next, through a series of examples, we show how our type inference algorithm incorporates features essential for inferring precise dependent types — namely path-sensitivity, recursion, higher-order functions, and polymorphism — and thus can statically prove the safety of array accesses.

Notation: We write B as an abbreviation for $\{\nu : B \mid \mathit{true}\}$. Additionally, when the base type B is clear from the context, we abbreviate $\{\nu : B \mid \kappa\}$ as κ when κ is a *liquid type variable*, and $\{\nu : B \mid e\}$ as $\{e\}$ when e is a *refinement predicate*. For example, $\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \{\texttt{x} \leq \nu \wedge \texttt{y} \leq \nu\}$ denotes the type of a function that takes two (curried) integer arguments x, y and returns an integer no less than x and y.

Example 1: Path Sensitivity. Consider the max function shown in Figure 1 as an OCAML program. We will show how we infer that max returns a value no less than both arguments.

(Step 1) HM infers that max has the type $\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \texttt{int}$. Using this type, we create a *template* for the liquid type of max, $\texttt{x} : \kappa_\texttt{x} \rightarrow \texttt{y} : \kappa_\texttt{y} \rightarrow \kappa_1$, where $\kappa_\texttt{x}, \kappa_\texttt{y}, \kappa_1$ are *liquid type variables* representing the unknown refinements for the formals x, y and the body of max, respectively.

(Step 2) As the body is an if expression, our algorithm generates the following two constraints that stipulate that, under the appropriate branch condition, the then and else expressions, respectively x, y, have types that are *subtypes* of the entire body's type:

$$\texttt{x} : \kappa_\texttt{x}; \texttt{y} : \kappa_\texttt{y}; (\texttt{x} > \texttt{y}) \vdash \{\nu = \texttt{x}\} <: \kappa_1 \quad (1.1)$$

$$\texttt{x} : \kappa_\texttt{x}; \texttt{y} : \kappa_\texttt{y}; \neg(\texttt{x} > \texttt{y}) \vdash \{\nu = \texttt{y}\} <: \kappa_1 \quad (1.2)$$

Constraint (1.1) (resp. (1.2)) stipulates that when x and y have the types $\kappa_\texttt{x}$ and $\kappa_\texttt{y}$ respectively and $\texttt{x} > \texttt{y}$ (resp. $\neg(\texttt{x} > \texttt{y})$), the type of the expression x (resp. y), namely the set of all values equal to x (resp. y), must be a subtype of the body κ_1.

(Step 3) Since the program is "open", *i.e.,* there are no calls to max, we assign κ_x, κ_y *true*, meaning that *any* integer arguments can be passed, and use a theorem prover to find the strongest conjunction of qualifiers in $\mathbb{Q}^\star$ that satisfies the subtyping constraints. The theorem prover deduces that when $\mathtt{x} > \mathtt{y}$ (resp. $\neg(\mathtt{x} > \mathtt{y})$) if $\nu = \mathtt{x}$ (resp. $\nu = \mathtt{y}$) then $\mathtt{x} \leq \nu$ and $\mathtt{y} \leq \nu$. Hence, our algorithm infers that $\mathtt{x} \leq \nu \wedge \mathtt{y} \leq \nu$ is the strongest solution for κ_1 that satisfies the two constraints. By substituting the solution for κ_1 into the template for max, our algorithm infers

$$\mathtt{max} :: \mathtt{x}\!:\!\mathtt{int} \rightarrow \mathtt{y}\!:\!\mathtt{int} \rightarrow \{\nu\!:\!\mathtt{int} \mid (\mathtt{x} \leq \nu) \wedge (\mathtt{y} \leq \nu)\}$$

Example 2: Recursion. Next, we show how our algorithm infers that the recursive function sum from Figure 1 always returns a non-negative value greater than or equal to its argument k.

(Step 1) HM infers that sum has the type $\mathtt{k}\!:\!\mathtt{int} \rightarrow \mathtt{int}$. Using this type, we create a template for the liquid type of sum, $\mathtt{k}\!:\!\kappa_k \rightarrow \kappa_2$, where κ_k and κ_2 represent the unknown refinements for the formal k and body, respectively. Due to the let rec, we use the created template as the type of sum when generating constraints for the body of sum.

(Step 2) Again, as the body is an if expression, we generate constraints that stipulate that under the appropriate branch conditions, the "then" and "else" expressions have subtypes of the body κ_2. For the "then" branch, we get a constraint:

$$\mathtt{sum}\!:\!\ldots; \mathtt{k}\!:\!\kappa_k; \mathtt{k} < 0 \vdash \{\nu = 0\} <: \kappa_2 \tag{2.1}$$

The else branch is a let expression. First, considering the expression that is locally bound, we generate a constraint

$$\mathtt{sum}\!:\!\ldots; \mathtt{k}\!:\!\kappa_k; \neg(\mathtt{k} < 0) \vdash \{\nu = \mathtt{k} - 1\} <: \kappa_k \tag{2.2}$$

from the call to sum that forces the actual passed in at the callsite to be a subtype of the formal of sum. The locally bound variable s gets assigned the template corresponding to the output of the application, $[\mathtt{k}-1/\mathtt{k}]\kappa_2$, *i.e.,* the output template of sum with the formal replaced with the actual argument, and we get the next constraint that ensures the "else" expression is a subtype of the body κ_2.

$$\neg(\mathtt{k} < 0); \mathtt{s}\!:\![\mathtt{k}-1/\mathtt{k}]\kappa_2 \vdash \{\nu = \mathtt{s} + \mathtt{k}\} <: \kappa_2 \tag{2.3}$$

(Step 3) Here, as sum is called, we try to find the strongest conjunction of qualifiers for κ_k and κ_2 that satisfies the constraints. To satisfy (2.2), κ_k can only be assigned *true* (the empty conjunction), as when $\neg(\mathtt{k} < 0)$, the value of $\mathtt{k} - 1$ can be either negative, zero or positive. On the other hand, κ_2 is assigned $0 \leq \nu \wedge \mathtt{k} \leq \nu$, the strongest conjunction of qualifiers in $\mathbb{Q}^\star$ that satisfies (2.1) and (2.3). Constraint (2.1) is trivially satisfied as the theorem prover deduces that when $\mathtt{k} < 0$, if $\nu = 0$ then $0 \leq \nu$ and $\mathtt{k} \leq \nu$. When κ_2 is assigned the above conjunction, the binding for s in the environment for constraint (2.3) becomes $\mathtt{s}\!:\!\{0 \leq \nu \wedge \mathtt{k} - 1 \leq \nu\}$. Thus, constraint (2.3) is satisfied as the theorem prover deduces that when $\neg(\mathtt{k} < 0)$ and $[\mathtt{s}/\nu](0 \leq \nu \wedge \mathtt{k} - 1 \leq \nu)$, if $\nu = \mathtt{s} + \mathtt{k}$ then $0 \leq \nu$ and $\mathtt{k} \leq \nu$. The substitution simplifies to $0 \leq \mathtt{s} \wedge \mathtt{k} - 1 \leq \mathtt{s}$, which effectively asserts to the solver the knowledge about the type of s, and crucially allows the solver to use the fact that s is non-negative when determining the type of $\mathtt{s} + \mathtt{k}$, and hence, the output of sum. Thus, recursion enters the picture, as the solution for the output of the recursive call, which is bound to the type of s, is used in conjunction with the branch information to prove that the output expression is non-negative. Plugging the solutions for κ_k and κ_2 into the template, our system infers

$$\mathtt{sum} :: \mathtt{k}\!:\!\mathtt{int} \rightarrow \{\nu\!:\!\mathtt{int} \mid 0 \leq \nu \wedge \mathtt{k} \leq \nu\}$$

```
let max x y =
  if x > y then x else y

let rec sum k =
  if k < 0 then 0 else
    let s = sum (k-1) in
    s + k

let foldn n b f =
  let rec loop i c =
    if i < n then loop (i+1) (f i c) else c in
  loop 0 b

let arraymax a =
  let am l m = max (sub a l) m in
  foldn (len a) 0 am
```

Figure 1. Example OCAML Program

Example 3: Higher-Order Functions. Next, consider a program comprising only the higher-order accumulator foldn shown in Figure 1. We show how our algorithm infers that f is only called with arguments between 0 and n.

(Step 1) HM infers that foldn has the polymorphic type $\forall\alpha.\mathtt{n}\!:\!\mathtt{int} \rightarrow \mathtt{b}\!:\!\alpha \rightarrow \mathtt{f}\!:\!(\mathtt{int} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$. From this ML type, we create the new template $\forall\alpha.\mathtt{n}\!:\!\kappa_n \rightarrow \mathtt{b}\!:\!\alpha \rightarrow \mathtt{f}\!:\!(\kappa_3 \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$ for foldn, where κ_n and κ_3 represent the unknown refinements for the formal n and the first parameter for the accumulation function f passed into foldn. This is a *polymorphic* template, as the occurrences of α are preserved. This will allow us to *instantiate* α with an appropriate dependent type at places where foldn is called. HM infers that the type of loop is $\mathtt{i}\!:\!\mathtt{int} \rightarrow \mathtt{c}\!:\!\alpha \rightarrow \alpha$, from which we generate a template $\mathtt{i}\!:\!\kappa_i \rightarrow \mathtt{c}\!:\!\alpha \rightarrow \alpha$ for loop, which we will use when analyzing the body of loop.

(Step 2) First, we generate constraints inside the body of loop. As HM infers that the type of the body is α, we omit the trivial subtyping constraints on the "then" and "else" expressions. Instead, the two interesting constraints are:

$$\ldots; \mathtt{i}\!:\!\kappa_i; \mathtt{i} < \mathtt{n} \vdash \{\nu = \mathtt{i} + 1\} <: \kappa_i \tag{3.1}$$

which stipulates that the actual passed into the recursive call to loop is a subtype of the expected formal, and

$$\ldots; \mathtt{i}\!:\!\kappa_i; \mathtt{i} < \mathtt{n} \vdash \{\nu = \mathtt{i}\} <: \kappa_3 \tag{3.2}$$

which forces the actual i to be a subtype of the first parameter of the higher-order function f, in the environment containing the critical branch condition. Finally, the application loop 0 yields

$$\ldots \vdash \{\nu = 0\} <: \kappa_i \tag{3.3}$$

forcing the actual 0 to be a subtype of the formal i.

(Step 3) Here, as foldn is not called, we assign κ_n *true* and try to find the strongest conjunction of qualifiers in $\mathbb{Q}^\star$ for κ_i and κ_3. We can assign to κ_i the predicate $0 \leq \nu$, which trivially satisfies (3.3), and also satisfies (3.1) as when $[\mathtt{i}/\nu](0 \leq \nu)$, if $\nu = \mathtt{i} + 1$ then $0 \leq \nu$. That is, the theorem prover can deduce that if i is non-negative, then so is $\mathtt{i} + 1$. To κ_3 we can assign the conjunction $0 \leq \nu \wedge \nu < \mathtt{n}$ which satisfies (3.2) as when $[\mathtt{i}/\nu](0 \leq \nu)$ and $\mathtt{i} < \mathtt{n}$, if $\nu = \mathtt{i}$ then $0 \leq \nu$ and $\nu < \mathtt{n}$. By plugging the solutions for κ_3, κ_n into the template our algorithm infers

$$\mathtt{foldn} :: \forall\alpha.\mathtt{n}\!:\!\mathtt{int} \rightarrow \mathtt{b}\!:\!\alpha \rightarrow \mathtt{f}\!:\!(\{0 \leq \nu \wedge \nu < \mathtt{n}\} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$$

Example 4: Polymorphism and Array Bounds Checking. Consider the function arraymax that calls foldn with a helper that calls `max` to compute the max of the elements of an array and 0. Suppose there is a base type `intarray` representing arrays of integers. Arrays are accessed via a primitive function

$$\texttt{sub} :: \texttt{a}\!:\!\texttt{intarray}\!\rightarrow\!\texttt{j}\!:\!\{\nu\!:\!\texttt{int} \mid \texttt{0} \leq \nu \wedge \nu < \texttt{len a}\}\!\rightarrow\!\texttt{int}$$

where the primitive function `len` returns the number of elements in the array. The `sub` function takes an array and an index that is between `0` and the number of elements, and returns the integer at that index in the array. We show how our algorithm combines predicate abstraction, function subtyping, and polymorphism to prove that (a) the array `a` is safely accessed at indices between 0 and `len a`, and (b) `arraymax` returns a non-negative integer.

(Step 1) HM infers that (1) `arraymax` has the type $\texttt{a}\!:\!\texttt{intarray}\!\rightarrow\!\texttt{int}$, (2) `am` has the type $\texttt{l}\!:\!\texttt{int}\!\rightarrow\!\texttt{m}\!:\!\texttt{int}\!\rightarrow\!\texttt{int}$, and (3) `foldn` called in the body is a polymorphic instance where the type variable α has been instantiated with `int`. Consequently, our algorithm creates the following templates: (1) $\texttt{a}\!:\!\texttt{intarray}\!\rightarrow\!\kappa_4$ for `arraymax`, where κ_4 represents the unknown refinement for the output of `arraymax`, (2) $\texttt{l}\!:\!\kappa_1\!\rightarrow\!\texttt{m}\!:\!\kappa_{\texttt{m}}\!\rightarrow\!\kappa_5$ for `am`, where κ_1, $\kappa_{\texttt{m}}$ and κ_5 represent the unknown refinements for the parameters and output type of `am` respectively, and (3) κ_6 for the type that α is instantiated with, and so the template for the instance of `foldn` inside `arraymax` is the type computed in the previous example with κ_6 substituted for α, namely, $\texttt{n}\!:\!\texttt{int}\!\rightarrow\!\texttt{b}\!:\!\kappa_6\!\rightarrow\!\texttt{f}\!:\!(\{\texttt{0} \leq \nu \wedge \nu < \texttt{n}\}\!\rightarrow\!\kappa_6\!\rightarrow\!\kappa_6)\!\rightarrow\!\kappa_6$

(Step 2) First, for the application `sub a l`, our algorithm generates

$$\texttt{l}\!:\!\kappa_1; \texttt{m}\!:\!\kappa_{\texttt{m}} \vdash \{\nu = \texttt{l}\} <: \{\texttt{0} \leq \nu \wedge \nu < \texttt{len}\ a\} \qquad (4.1)$$

which states that the argument passed into `sub` must be within the array bounds. For the application `max (sub a l) m`, using the type inferred for `max` in Example 1, we get

$$\texttt{l}\!:\!\kappa_1; \texttt{m}\!:\!\kappa_{\texttt{m}} \vdash \{\texttt{sub a l} \leq \nu \wedge \texttt{m} \leq \nu\} <: \kappa_5 \qquad (4.2)$$

which constrains the output of `max` (with the actuals (`sub a l`) and `m` substituted for the parameters `x` and `y` respectively), to be a subtype of the output type κ_5 of `am`. The call `foldn (len a) 0` generates

$$\ldots \vdash \{\nu = \texttt{0}\} <: \kappa_6 \qquad (4.3)$$

which forces the actual passed in for `b` to be a subtype of κ_6 the type of the formal `b` in this polymorphic instance. Similarly, the call `foldn (len a) 0 am` generates a constraint (4.4)

$$\ldots \vdash \texttt{l}\!:\!\kappa_1\!\rightarrow\!\texttt{m}\!:\!\kappa_{\texttt{m}}\!\rightarrow\!\kappa_5 <: \{\texttt{0} \leq \nu \wedge \nu < \texttt{len a}\}\!\rightarrow\!\kappa_6\!\rightarrow\!\kappa_6$$

forcing the type of the actual `am` to be a subtype of the formal `f` inferred in Example 1, with the curried argument `len` a substituted for the formal `n` of `foldn`, and

$$\ldots \vdash \kappa_6 <: \kappa_4 \qquad (4.5)$$

forcing the output of the `foldn` application to be a subtype of the body of `arraymax`. Upon simplification using the standard rule for subtyping function types, constraint (4.4) reduces to

$$\ldots \vdash \{\texttt{0} \leq \nu \wedge \nu < \texttt{len a}\} <: \kappa_1 \qquad (4.6)$$

$$\ldots \vdash \kappa_6 <: \kappa_{\texttt{m}} \qquad (4.7)$$

$$\ldots \vdash \kappa_5 <: \kappa_6 \qquad (4.8)$$

(Step 3) The strongest conjunction of qualifiers from $\mathbb{Q}^\star$ that we can assign to: $\kappa_{\texttt{m}}$, κ_4, κ_5 and κ_6 is the predicate $\texttt{0} \leq \nu$. In essence the solution infers that we can "instantiate" the type variable α with the dependent type $\{\nu\!:\!\texttt{int} \mid \texttt{0} \leq \nu\}$. This is sound because the base value 0 passed in is non-negative (constraint (4.3) is satisfied), and the accumulation function passed in (`am`), is such that if its second argument (`m` of type $\kappa_{\texttt{m}}$) is non-negative then the output (of type κ_5) is non-negative (constraint (4.2) is satisfied). Plugging the solution into the template, our algorithm infers

$$\texttt{arraymax} :: \texttt{intarray}\!\rightarrow\!\{\nu\!:\!\texttt{int} \mid \texttt{0} \leq \nu\}$$

The strongest conjunction over $\mathbb{Q}^\star$ we can assign to κ_1 is $\texttt{0} \leq \nu \wedge \nu < \texttt{len a}$, which trivially satisfies constraint (4.6). Moreover, with this assignment, we have satisfied the "bounds check" constraint (4.1), *i.e.,* we have inferred an assignment of dependent types to all the program expressions that proves that all array accesses occur within bounds.

e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	`c`	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	if-then-else
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	let-binding
	$\mid$	$\texttt{let rec}\ f = \lambda x.e\ \texttt{in}\ e$	letrec-binding
	$\mid$	$[\Lambda\alpha]e$	type-abstraction
	$\mid$	$[\tau]e$	type-instantiation
Q	$::=$		*Liquid Refinements*
	$\mid$	$true$	true
	$\mid$	q	logical qualifier in $\mathbb{Q}^\star$
	$\mid$	$Q \wedge Q$	conjunction of qualifiers
B	$::=$		*Base Types:*
	$\mid$	`int`	base type of integers
	$\mid$	`bool`	base type of booleans
$\mathbb{T}(\mathbb{B})$	$::=$		*Type Skeletons:*
	$\mid$	$\{\nu\!:\!B \mid \mathbb{B}\}$	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 2. Syntax

3. Liquid Type Checking

We first present the syntax and static semantics of our core language λ_{L}, a variant of the λ-calculus with ML-style polymorphism extended with liquid types. We begin by describing the elements of λ_{L}, including expressions, types, and environments (Section 3.1). Next, we present the type judgments and derivation rules and state a soundness theorem which relates the static type system with the operational semantics (Section 3.2). We conclude this section by describing how the design of our type system enables automatic dependent type inference (Section 3.3).

3.1 Elements of λ_{L}

The syntax of expressions and types for λ_{L} is summarized in Figure 2. 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, λ_{L} includes as expressions the common constructs `if-then-else` and `let`, which the liquid type inference algorithm exploits to generate precise types, as well as `let rec` which is syntactic sugar for the standard `fix` operator.

Types and Schemas. We use B to denote base types such as `bool` or `int`. 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 write τ and σ for ML types and schemas, T and S for dependent types and schemas, and $\hat{T}$ and $\hat{S}$ for liquid types and schemas.

Environments and Well-formedness. A *type environment* Γ is a sequence of *type bindings* $x\!:\!S$ and *guard predicates* e. The former are standard; the latter capture constraints about the if-then-else branches under which an expression is evaluated, which is required to make the system "path-sensitive" (Section 3.3). A type is considered *well-formed* with respect to an environment if all the free variables appearing in the refinement predicates of the type are bound in the environment. An environment is considered *well-formed* if, in each type binding, the dependent type is well-formed with respect to the preceding (prefix) environment.

Shapes. The *shape* of the dependent type (schema) S, denoted by $\mathsf{Shape}(S)$, is the ML type (schema) obtained by replacing all refinement predicates with $true$. We lift the function Shape to type environments by applying it to each type binding and eliminating the guard predicates.

Constants. As in [22, 10], the basic units of computation are the constants $\mathtt{c}$ built into λ_{L}, each of which has a dependent type $ty(\mathtt{c})$ 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} includes:

$$\begin{array}{rcl}
\mathtt{true} & :: & \{\nu\!:\!\mathtt{bool} \mid \nu\} \\
\mathtt{false} & :: & \{\nu\!:\!\mathtt{bool} \mid \mathtt{not}\ \nu\} \\
\Leftrightarrow & :: & x\!:\!\mathtt{bool}\!\rightarrow\! y\!:\!\mathtt{bool}\!\rightarrow\!\{\nu\!:\!\mathtt{bool} \mid \nu \Leftrightarrow (x \Leftrightarrow y)\} \\
3 & :: & \{\nu\!:\!\mathtt{int} \mid \nu = 3\} \\
= & :: & x\!:\!\mathtt{int}\!\rightarrow\! y\!:\!\mathtt{int}\!\rightarrow\!\{\nu\!:\!\mathtt{bool} \mid \nu \Leftrightarrow (x = y)\} \\
+ & :: & x\!:\!\mathtt{int}\!\rightarrow\! y\!:\!\mathtt{int}\!\rightarrow\!\{\nu\!:\!\mathtt{int} \mid \nu = x + y\} \\
\mathtt{fix} & :: & \forall\alpha.(\alpha\!\rightarrow\!\alpha)\!\rightarrow\!\alpha \\
\mathtt{len} & :: & \mathtt{intarray}\!\rightarrow\!\{\nu\!:\!\mathtt{int} \mid 0 \leq \nu\} \\
\mathtt{sub} & :: & a\!:\!\mathtt{intarray}\!\rightarrow\! i\!:\!\{\nu\!:\!\mathtt{int} \mid 0 \leq \nu \wedge \nu < \mathtt{len}\ a\}\!\rightarrow\!\mathtt{int}
\end{array}$$

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 [10]), and the static semantics is defined via a sound overapproximation of the dynamic semantics [24]. For clarity, we will use infix notation for constants like $+$. To simplify the exposition, we assume 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.

3.2 Liquid Type Checking Rules

We now describe the key ingredients of the type system: the typing judgments and derivation rules summarized in Figure 3. Our system has three kinds of judgments relating environments, expressions, and types.

Well-formedness Judgment $\Gamma \vdash S$: 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 in the scope of the corresponding expression.

Subtype Judgment $\Gamma \vdash S_1 <: S_2$: states that dependent type schema S_1 *is a subtype of* schema S_2 in environment Γ.

Liquid Type Judgment $\Gamma \vdash_{\mathbb{Q}} e : S$: states that, using the logical qualifiers $\mathbb{Q}$, the expression e has the type schema S under the type environment Γ.

Soundness of Liquid Type Checking. Assume that variables are bound at most once in any type environment; in other words, assume that variables are α-renamed to ensure that substitutions (such as in [LT-APP]) avoid capture. Let $\hookrightarrow$ describe the single evaluation step relation for λ_{L} expressions and $\overset{*}{\hookrightarrow}$ describe the reflexive, transitive closure of $\hookrightarrow$.

As the conservative subtyping rule makes it hard to prove a substitution lemma, we prove soundness via two steps. First, we define an "exact" version of the type system, with a judgment $\Gamma \vdash e : S$, whose rules use an undecidable subtyping relation. We show the standard weakening, narrowing, and substitution lemmas for this system, and thereby obtain preservation and progress theorems. Second, we show that our decidable system is conservative: *i.e.,* if $\Gamma \vdash_{\mathbb{Q}} e : S$ then $\Gamma \vdash e : S$. Combining the results, we conclude that if an expression is well-typed in our decidable system then we are guaranteed that evaluation does not get "stuck", *i.e.,* at run-time, every primitive operation receives valid inputs.

THEOREM 1. **[Liquid Type Safety]**

1. *(Overapproximation) If* $\Gamma \vdash_{\mathbb{Q}} e : S$ *then* $\Gamma \vdash e : S$.
2. *(Preservation) If* $\Gamma \vdash e : S$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash e' : S$.
3. *(Progress) If* $\emptyset \vdash e : S$ *and* e *is not a value then there exists an* e', $e \hookrightarrow e'$.

We omit the details due to lack of space — the formalization and proofs can be found in [24]. Thus, if a program typechecks we are guaranteed that every call to `sub` gets an index that is within the array's bounds. Arbitrary safety properties (*e.g.,* divide-by-zero errors) can be expressed by using suitable types for the appropriate primitive constant (*e.g.,* requiring the second argument of $(/)$ to be non-zero).

3.3 Features of the Liquid Type System

Next, we describe some of the features unique to the design of the Liquid type system and how they contribute to automatic type inference and verification.

1. Path Sensitivity. Any analysis that aims to prove properties like the safety of array accesses needs to be sensitive to branch information; it must infer properties which hold for the entire `if` expression as well as for the individual `then` and `else` expressions. For example, without the branch information in the `sum` example from Section 2, the system would not be able to infer that the occurrence of `k` inside the `else` expression is non-negative, and hence that `sum` returned a non-negative value. For array bounds checking, programmers often compare the index to some other expression — either the array length, or some other variable that is known to be smaller than the array length (*e.g.,* in `arraymax` from Section 2), and only perform the array access under the appropriate guard. To capture this kind of information, the environment Γ also includes branch information, shown in rule [LT-IF] in Figure 3.

2. Decidable, Conservative Subtyping. As shown in Figure 3, checking that one type is a subtype of another reduces to a set of subtype checks on base refinement predicates, which further reduces to checking if the refinement predicate for the subtype *implies* the predicate for the supertype. As the refinement predicates contain arbitrary terms, exact implication checking is undecidable. To get around this problem, our system uses a conservative but decidable implication check, shown in rule [DEC-<:-BASE] of Figure 3. Let EUFA be the decidable logic of *equality*, *uninterpreted functions* and *linear arithmetic* [21]. We write $[\![e]\!]$ for the *embedding* of the expression e into terms of the logic EUFA by encoding expressions corresponding to integers, addition, multiplication and division by constant integers, equality, inequality, and disequality with corresponding terms in the EUFA logic, and encoding all other constructs, including λ-abstractions and applications, with uninterpreted function terms. We write:

$$[\![\Gamma]\!] \equiv \bigwedge \{e \mid e \in \Gamma\} \wedge \bigwedge \{[\![[x/\nu]e]\!] \mid x\!:\!\{\nu\!:\!B \mid e\} \in \Gamma\}$$

as the embedding for the environment. Notice that we use the guard predicates and *base type* bindings in the environment to *strengthen* the antecedent of the implication. However, we *substitute all occurrences of the value variable* ν in the refinements from Γ with the actual variable being refined, thereby asserting in the antecedent that the program variable satisfies the base refinement predicate. Thus, in the embedded formula, all occurrences of ν refer to the two types that are being checked for subtyping. The embedding is conservative, *i.e.,* the validity of the embedded implication implies the the standard, weaker, exact requirement for subtyping of refined types [10, 22]. For example, for the `then` expression in `max` from Section 2, the subtyping relation: $\mathtt{x}\!:\!\mathtt{int}; \mathtt{y}\!:\!\mathtt{int}; \mathtt{x} > \mathtt{y} \vdash \{\nu = \mathtt{x}\} <: \{\mathtt{x} \leq \nu \wedge \mathtt{y} \leq \nu\}$ holds as the following implication is valid in EUFA: $((\mathit{true} \wedge \mathit{true} \wedge \mathtt{x} > \mathtt{y}) \wedge (\nu = x)) \Rightarrow (\mathtt{x} \leq \nu \wedge \mathtt{y} \leq \nu)$

3. Recursion via Polymorphism. To handle polymorphism, our type system incorporates type generalization and instantiation annotations, which are over ML type variables α and monomorphic ML types τ, respectively, and thus can be reconstructed via a standard type inference algorithm. The rule [LT-INST] allows a type schema to be instantiated with an arbitrary liquid type $\hat{T}$ of the same shape as τ, the monomorphic ML type used for instantiation. We use polymorphism to encode recursion via the polymorphic type given to `fix`. That is, `let rec` bindings are syntactic sugar: `let rec f = e in e'` is internally converted to `let f = fix (fun f -> e) in e'`. The expression typechecks if there is an appropriate liquid type that can be instantiated for the α in the polymorphic type of `fix`; this liquid type corresponds to the type of the recursive function `f`.

4. The Liquid Type Restriction. The most critical difference between the rules for liquid type checking and other dependent systems is that our rules stipulate that certain kinds of expressions have liquid types. In essence, these expressions are the key points where appropriate dependent types must be inferred. By forcing the types to be liquid, we bound the space of possible solutions, thus making inference efficiently decidable.

[LT-INST] For polymorphic instantiation, also the mechanism for handling recursion, the liquid type restriction enables efficient inference by making the set of candidate dependent types finite.

[LT-FUN] For λ-abstractions, we impose the restriction that the input and output be liquid to ensure the types remain small, thereby making algorithmic checking and inference efficient. This is analogous to procedure "summarization" for first-order programs.

[LT-IF] For conditional expressions we impose the liquid restriction and implicitly force the `then` and `else` expressions to be subtypes of a fresh liquid type, instead of an explicit "join" operator as in dataflow analysis. We do so as the expression may have a function type and with a join operator, input type contravariance would introduce disjunctions into the dependent type which would have unpleasant algorithmic consequences.

[LT-LET] For let-in expressions we impose the liquid restriction as a means of *eliminating* the locally bound variable from the dependent type of the whole expression (as the local variable goes out of scope). The antecedent $\Gamma \vdash \hat{T}$ requires that the liquid type be well-formed in the outer environment, which, together with the condition, enforced via alpha renaming, that each variable is bound only once in the environment, is essential for ensuring the soundness of our system [24]. The alternative of existentially quantifying the local variable [18] makes algorithmic checking hard.

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S_1 \qquad \Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash S_2}{\Gamma \vdash_{\mathbb{Q}} e : S_2}\ [\text{LT-SUB}]$$

$$\frac{\Gamma(x) = \{\nu\!:\!B \mid e\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu\!:\!B \mid \nu = x\}}\ [\text{LT-VAR}] \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash_{\mathbb{Q}} x : \Gamma(x)}\ [\text{LT-VAR}]$$

$$\frac{}{\Gamma \vdash_{\mathbb{Q}} \mathtt{c} : \mathit{ty}(\mathtt{c})}\ [\text{LT-CONST}]$$

$$\frac{\Gamma; x\!:\!\hat{T}_x \vdash_{\mathbb{Q}} e : \hat{T} \qquad \Gamma \vdash x\!:\!\hat{T}_x \rightarrow \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \lambda x.e : (x\!:\!\hat{T}_x \rightarrow \hat{T})}\ [\text{LT-FUN}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : (x\!:\!T_x \rightarrow T) \qquad \Gamma \vdash_{\mathbb{Q}} e_2 : T_x}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 : [e_2/x]T}\ [\text{LT-APP}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : \mathtt{bool} \quad \Gamma; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{T} \quad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \hat{T}}\ [\text{LT-IF}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : S_1 \qquad \Gamma; x\!:\!S_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \qquad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \hat{T}}\ [\text{LT-LET}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e : \forall\alpha.S}\ [\text{LT-GEN}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : \forall\alpha.S \qquad \Gamma \vdash \hat{T} \qquad \mathsf{Shape}(\hat{T}) = \tau}{\Gamma \vdash_{\mathbb{Q}} [\tau]e : [\hat{T}/\alpha]S}\ [\text{LT-INST}]$$

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

$$\frac{\mathsf{Valid}([\![\Gamma]\!] \wedge [\![e_1]\!] \Rightarrow [\![e_2]\!])}{\Gamma \vdash \{\nu\!:\!B \mid e_1\} <: \{\nu\!:\!B \mid e_2\}}\ [\text{DEC-<:-BASE}]$$

$$\frac{\Gamma \vdash T'_x <: T_x \qquad \Gamma; x\!:\!T'_x \vdash T <: T'}{\Gamma \vdash x\!:\!T_x \rightarrow T <: x\!:\!T'_x \rightarrow T'}\ [\text{DEC-<:-FUN}]$$

$$\frac{}{\Gamma \vdash \alpha <: \alpha}\ [\text{<:-VAR}] \qquad \frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash \forall\alpha.S_1 <: \forall\alpha.S_2}\ [\text{<:-POLY}]$$

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

$$\frac{\Gamma; \nu\!:\!B \vdash e : \mathtt{bool}}{\Gamma \vdash \{\nu\!:\!B \mid e\}}\ [\text{WT-BASE}] \qquad \frac{}{\Gamma \vdash \alpha}\ [\text{WT-VAR}]$$

$$\frac{\Gamma; x\!:\!T_x \vdash T}{\Gamma \vdash x\!:\!T_x \rightarrow T}\ [\text{WT-FUN}] \qquad \frac{\Gamma \vdash S}{\Gamma \vdash \forall\alpha.S}\ [\text{WT-POLY}]$$

Figure 3. Rules for Liquid Type Checking

5. Placeholder Variables and α-Renaming. We use the placeholder variables $\star$ instead of "hard-coded" program variables to make our type system robust to α-renaming. If $\mathbb{Q}$ is $\{x < \nu\}$, then $\emptyset \vdash_{\mathbb{Q}} (\lambda x.x + 1) : x\!:\!\mathtt{int}\rightarrow\{x < \nu\}$ is a valid judgment, but $\emptyset \vdash_{\mathbb{Q}} (\lambda y.y + 1) : y\!:\!\mathtt{int}\rightarrow\{y < \nu\}$ is not, as $y < \nu$ is not in $\mathbb{Q}^\star$. If instead $\mathbb{Q}$ is $\{\star < \nu\}$, then $\mathbb{Q}^\star$ includes $\{x < \nu, y < \nu\}$ and so both of the above are valid judgments. In general, our type system is robust to renaming in the following sense: if $\Gamma \vdash_{\mathbb{Q}} e_1 : S_1$ and e_1 is α-equivalent to e_2 and the free variables of $\mathbb{Q}$ are bound[1] in Γ, then for some S_2 that is α-equivalent to S_1, we have $\Gamma \vdash_{\mathbb{Q}} e_2 : S_2$.

[1] Recall that variables are bound at most once in any environment

4. Liquid Type Inference

We now turn to the heart of our system: the algorithm Infer (shown in Figure 4) that takes as input a type environment Γ, an expression e, and a finite set of logical qualifiers $\mathbb{Q}$ and determines whether e is well-typed over $\mathbb{Q}$, *i.e.,* whether there exists some S such that $\Gamma \vdash_{\mathbb{Q}} e : S$. Our algorithm proceeds in three steps. First, we observe that the dependent type for any expression must be a refinement of its ML type, and so we invoke Hindley-Milner (HM) to infer the types of subexpressions, and use the ML types to generate *templates* representing the unknown dependent types for the subexpressions (Section 4.1). Second, we use the syntax-directed liquid typing rules from Figure 3 to build a system of constraints that capture the subtyping relationships between the templates that must hold for a liquid type derivation to exist (Section 4.2). Third, we use $\mathbb{Q}$ to solve the constraints using a technique inspired by predicate abstraction (Section 4.3).

4.1 ML Types and Templates

Our type inference algorithm is based on the observation that the liquid type derivations are refinements of the ML type derivations, and hence the dependent types for all subexpressions are *refinements* of their ML types.

ML Type Inference Oracle. Let HM be an ML type inference oracle, which takes an ML type environment Γ and an expression e and returns the ML type (schema) σ if and only if, using the classical ML type derivation rules [7], there exists a derivation $\Gamma \vdash e : \sigma$. The liquid type derivation rules are refinements of the ML type derivation rules. That is, if $\Gamma \vdash_{\mathbb{Q}} e : S$ then $\mathsf{HM}(\mathsf{Shape}(\Gamma), e) = \mathsf{Shape}(S)$. Moreover, we assume that the ML type derivation oracle has "inserted" suitable type generalization ($[\Lambda\alpha]e$) and instantiation ($[\tau]e$) annotations. Thus, the problem of dependent type inference reduces to inferring appropriate refinements of the ML types.

Templates. Let $\mathbb{K}$ be a set of *liquid type variables* used to represent unknown type refinement predicates. A *template* F is a dependent type schema described via the grammar shown below, where some of the refinement predicates are replaced with liquid type variables with *pending substitutions*. A *template environment* is a map Γ from variables to templates.

$$\theta \quad ::= \quad \epsilon \mid [e/x];\theta \qquad \text{(Pending Substitutions)}$$
$$F \quad ::= \quad \mathbb{S}(E \cup \theta \cdot \mathbb{K}) \qquad \text{(Templates)}$$

Variables with Pending Substitutions. A *sequence of pending substitutions* θ is defined using the grammar above. To understand the need for θ, consider rule [LT-APP] from Figure 3 which specifies that the dependent type of a function application is obtained by *substituting* 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 the type of e_1 is $x{:}B{\rightarrow}\{\nu{:}B \mid \kappa\}$, where κ is a liquid type variable. In this case, we will assign the application $e_1\ e_2$ the type $\{\nu{:}B \mid [e_2/x] \cdot \kappa\}$, where $[e_2/x] \cdot \kappa$ is a variable with a *pending* substitution [18]. Note that substitution can be "pushed inside" type constructors, *e.g.,* $\theta \cdot (\{\kappa_1\} \rightarrow \{\kappa_2\})$ is the same as $\{\theta \cdot \kappa_1\} \rightarrow \{\theta \cdot \kappa_2\}$ and so it suffices to apply the pending substitutions only to the root of the template.

4.2 Constraint Generation

We now describe how our algorithm generates constraints over templates by traversing the expression in the syntax-directed manner of a type checker, generating fresh templates for unknown types, constraints that capture the relationships between the types of various subexpressions, and well-formedness requirements. The generated constraints are such that they have a solution if and only if the expression has a valid liquid type derivation. Our inference algorithm uses two kinds of constraints over templates. **Well-formedness constraints** of the form $\Gamma \vdash F$, where Γ is template environment, and F is a template, ensure that the types inferred for each subexpression are over program variables that are in scope at that subexpression. **Subtyping constraints** of the form $\Gamma \vdash F_1 <: F_2$ where Γ is a template environment and F_1 and F_2 are two templates of the same shape, ensure that the types inferred for each subexpression can be combined using appropriate subsumption relationships to yield a valid type derivation.

Our constraint generation algorithm, Cons, shown in Figure 4, takes as input a template environment Γ and an expression e that we wish to infer the type of 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. Intuitively, Cons mirrors the type derivation rules and generates constraints C which capture exactly the relationships that must hold between the templates of the subexpressions in order for e to have a valid type derivation over $\mathbb{Q}$. To understand how Cons works, notice that the expressions of λ_{L} can be split into two classes: those whose types are constructable from the environment and the types of subexpressions, and those whose types are not.

1. Expressions with Constructable Types. The first class of expressions are variables, constants, function applications and polymorphic generalizations, whose types can be immediately constructed 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.

As an example, consider $\mathsf{Cons}(\Gamma, e_1\ e_2)$. 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 returned template 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 returned constraints are the union of the constraints for the subexpressions and a subtyping constraint ensuring that the type of the argument e_2 is a subtype of the input type of e_1.

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-bindings, and polymorphic instantiations (which includes recursive functions). We use two observations to infer the types of these expressions. First, the shape of the dependent type is the same as the ML type of the expression. Second, from the *liquid type restriction*, we know that the refinement predicates for these expressions are conjunctions of logical qualifiers from $\mathbb{Q}^\star$ (cf. rules [LT-LET], [LT-FUN], [LT-IF], [LT-INST] of Figure 3). Thus, to infer the types of these expressions, we invoke HM to determine the ML type of the expression and then use Fresh to generate a template with the same shape as the ML type but with fresh liquid type variables representing the unknown refinements.

As an example, consider $\mathsf{Cons}(\Gamma, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$. 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` subexpressions. Note that for the `then` (resp. `else`) subexpression, the environment is extended with e_1 (resp. $\neg e_1$) as in the type derivation rule ([LT-IF] from Figure 3). The fresh template is returned as the template for the whole expression. The constraints returned are the union of those for the subexpressions, a well-formedness constraint for the whole expression's template, and subtyping constraints forcing the templates for the `then` and `else` subexpressions to be subtypes of the whole expression's template

Example: Constraints. The well-formedness constraint $\emptyset \vdash \mathtt{x}\!:\!\kappa_{\mathtt{x}} \!\rightarrow\! \mathtt{y}\!:\!\kappa_{\mathtt{y}} \!\rightarrow\! \kappa_1$ is generated for the fresh template for `max` (from Figure 1). The constraint ensures that the inferred type for `max` only contains program variables that are in scope at the point where `max` is bound. The `if` expression that is the body of `max` is an expression with liquid type. For this expression, a fresh template $\kappa_{1'}$ is generated, and the subtyping constraints:

$$\mathtt{x}\!:\!\kappa_{\mathtt{x}}; \mathtt{y}\!:\!\kappa_{\mathtt{y}}; (\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{x}\} <: \kappa_{1'}$$
$$\mathtt{x}\!:\!\kappa_{\mathtt{x}}; \mathtt{y}\!:\!\kappa_{\mathtt{y}}; \neg(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{y}\} <: \kappa_{1'}$$
$$\mathtt{x}\!:\!\kappa_{\mathtt{x}}; \mathtt{y}\!:\!\kappa_{\mathtt{y}} \vdash \kappa_{1'} <: \kappa_1$$

are generated, capturing the relationships between the `then` and the `if` expression, the `else` and the `if` expression, and the `if` and the output expression, respectively. The constraints (1.1) and (1.2) are the above constraints simplified for exposition. The recursive application `sum (k-1)` from Figure 1 is an expression with a constructable type. For this expression the subtyping constraint (2.2) is generated, forcing the actual to be a subtype of the formal. The output of the application, *i.e.,* the output type κ_2 of `sum`, with the pending substitution of the formal `k` with the actual $(\mathtt{k} - 1)$ is shown bound to `s` in (2.3).

4.3 Constraint Solving

Next, we describe our two-step algorithm for solving the constraints, *i.e.,* assigning liquid types to all variables κ such that all constraints are satisfied. In the first step, we use the well-formedness and 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 weaken a trivial assignment, in which each liquid type variable is assigned the conjunction of all logical qualifiers, until we find the least fixpoint solution for all the simplified constraints or determine that the constraints have no solution. We first formalize the notion of a solution and then describe the two-step algorithm that computes solutions.

Satisfying Liquid Assignments. A *Liquid Assignment over* $\mathbb{Q}$ is a map A from liquid type variables to sets of qualifiers from $\mathbb{Q}^\star$. Assignments can be lifted to maps from templates F to dependent types $A(F)$ and template environments Γ to environments $A(\Gamma)$, by substituting each liquid type variable κ with $\bigwedge A(\kappa)$ and then applying the pending substitutions. A *satisfies* a constraint c if $A(c)$ is valid. That is, A satisfies a well-formedness constraint $\Gamma \vdash F$ if $A(\Gamma) \vdash A(F)$, and a subtyping constraint $\Gamma \vdash F_1 <: F_2$ if $A(\Gamma) \vdash A(F_1) <: A(F_2)$. A is a *solution* for a set of constraints C if it satisfies each constraint in C.

Step 1: Splitting into Simple Constraints. First, we call Split, which uses the rules for well-formedness and subtyping (Figure 3) to convert all the constraints over complex types (*i.e.,* function types) into simple constraints over base types. An assignment is a solution for C if and only if it is a solution for $\mathsf{Split}(C)$.

Example: Splitting. The well-formedness constraint $\emptyset \vdash \mathtt{x}\!:\!\kappa_{\mathtt{x}} \!\rightarrow\! \mathtt{y}\!:\!\kappa_{\mathtt{y}} \!\rightarrow\! \kappa_1$ splits into the three simple constraints: $\emptyset \vdash \kappa_{\mathtt{x}}$, $\mathtt{x}\!:\!\kappa_{\mathtt{x}} \vdash \kappa_{\mathtt{y}}$ and $\mathtt{x}\!:\!\kappa_{\mathtt{x}}; \mathtt{y}\!:\!\kappa_{\mathtt{y}} \vdash \kappa_1$, which ensure that: the parameter `x` must have a refinement over only constants and the value variable ν as the environment is empty; the parameter `y` must have a refinement over only `x` and ν; and the output type's refinement can refer to both parameters `x`, `y` and the value variable. The function subtyping constraint generated by the call `foldn (len a) 0 am` shown in (4.4) splits into the simple subtyping constraints (4.6),(4.7),(4.8). Notice how substitution and contravariance combine to cause the flow of the bounds information into input parameter κ_1 (4.6) thus allowing the system to statically check the array access.

Step 2: Iterative Weakening. Due to the well-formedness constraints, any solution over $\mathbb{Q}$ must map the liquid type variables to sets of qualifiers whose free variables are either the value variable ν or the variables in the input environment Γ (written $\mathsf{Var}(\Gamma)$), or the variables in the input expression e (written $\mathsf{Var}(e)$). That is, any solution maps the liquid variables to a set of qualifiers contained in $\mathsf{Inst}(\Gamma, e, \mathbb{Q})$ which is defined as

$$\{q \mid q \in \mathbb{Q}^\star \text{ and } \mathsf{FreeVar}(q) \subseteq \{\nu\} \cup \mathsf{Var}(\Gamma) \cup \mathsf{Var}(e)\}$$

where $\mathsf{Var}(\Gamma)$ and $\mathsf{Var}(e)$ are the set of variables in Γ and e respectively. Notice that if $\mathbb{Q}$ is finite, then $\mathsf{Inst}(\Gamma, e, \mathbb{Q})$ is also finite as the placeholder variables can only be instantiated with finitely many variables from Γ and e. Thus, to solve the constraints, we call the procedure Solve, shown in Figure 4, with the split constraints and a trivial initial assignment that maps each liquid type variable to $\mathsf{Inst}(\Gamma, e, \mathbb{Q})$.

Solve repeatedly picks a constraint that is not satisfied by the current assignment and calls Weaken to remove the qualifiers that prevent the constraint from being satisfied. For unsatisfied constraints of the form: (1) $\Gamma \vdash \{\nu\!:\!B \mid \theta \cdot \kappa\}$, Weaken removes 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 environment $\mathsf{Shape}(\Gamma); \nu\!:\!B$, (2) $\Gamma \vdash \{\nu\!:\!B \mid \rho\} <: \{\nu\!:\!B \mid \theta \cdot \kappa\}$, where ρ is either a refinement predicate or a liquid variable with pending substitutions, Weaken removes from the assignment for κ all the logical qualifiers q such that the implication $(\llbracket A(\Gamma) \rrbracket \wedge \llbracket A(\rho) \rrbracket) \Rightarrow \theta \cdot q$ is not valid in EUFA, (3) $\Gamma \vdash \{\nu\!:\!B \mid \rho\} <: \{\nu\!:\!B \mid e\}$, Weaken, and therefore Solve, returns **Failure**.

Correctness of Solve. For two assignments A and A', we say that $A \leq A'$ if for all κ, the set of logical qualifiers $A(\kappa)$ *contains* the set of logical qualifiers $A'(\kappa)$. We can prove that if a set of constraints has a solution over $\mathbb{Q}$ then it has a *unique minimum* solution w.r.t. $\leq$. Intuitively, we invoke Solve with the least possible assignment that maps each liquid variable to all the possible qualifiers. Solve then uses Weaken to iteratively weaken the assignment until the unique minimum solution is found. The correctness of Solve follows from the following invariant about the iterative weakening: if A^* is the minimum solution for the constraints, then in each iteration, the assignment $A \leq A^*$. Thus, if Solve returns a solution then it must be the minimum solution for C over $\mathbb{Q}$. If at some point a constraint $\Gamma \vdash \{\nu\!:\!B \mid \rho\} <: \{\nu\!:\!B \mid e\}$ is unsatisfied, subsequent weakening cannot make it satisfied. Thus, if Solve returns **Failure** then C has no solution over $\mathbb{Q}$.

By combining the steps of constraint generation, splitting and solving, we obtain our dependent type inference algorithm Infer shown in Figure 4. The algorithm takes as input an environment Γ, 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 in the environment Γ. The correctness properties of Infer are stated in the theorem below, whose proof is in [24]. From Theorems 1, 2, we conclude that if $\mathsf{Infer}(\emptyset, e, \mathbb{Q}) = S$ then every primitive operation invoked during the evaluation of e succeeds.

THEOREM 2. **[Liquid Type Inference]**

1. $\mathsf{Infer}(\Gamma, e, \mathbb{Q})$ *terminates,*
2. *If* $\mathsf{Infer}(\Gamma, e, \mathbb{Q}) = S$ *then* $\Gamma \vdash_{\mathbb{Q}} e : S$*, and,*
3. *If* $\mathsf{Infer}(\Gamma, e, \mathbb{Q}) =$ **Failure** *then there is no* S *s.t.* $\Gamma \vdash_{\mathbb{Q}} e : S$.

Running Time. Most of the time taken by Infer goes inside procedure Solve which asymptotically dominates the time taken to generate constraints. Solve returns the same output regardless of the order in which the constraints are processed. For efficiency, we implement Solve in two phases. First, Solve makes a (linear) pass that solves the well-formedness constraints, thus rapidly pruning away irrelevant qualifiers. Second, Solve uses a standard worklist-based algorithm that solves the subtyping constraints. The time taken in the first phase is asymptotically dominated by the time taken in the second. Let Q be the maximum number of qualifiers that any liquid variable is mapped to after the first well-formedness pass, V be the number of variables in the program e that have a base type, and D be the size of the ML type derivation for e in the environment Γ. A constraint is sent to Weaken only when the antecedent of its implication changes, *i.e.,* at most $V \times Q$ times. There are at most $O(D)$ constraints and so Weaken is called at most $O(D \times V \times Q)$ times. Each call to Weaken makes at most Q calls to the theorem prover. Thus, in all the running time of Infer is $O(D \times V \times Q^2)$ assuming each theorem prover call takes unit time. Of course, D can be exponential in the program size (but tends to be linear in practice), and the size of each (embedded) theorem prover query is $O(V \times Q)$. Though validity checking in EUFA is NP-Hard, several solvers for this theory exist which are very efficient for the simple queries that arise in our context [9].

4.4 Features of Liquid Type Inference

We now discuss some features of the inference algorithm.

1. Type Variables and Polymorphism. There are two kinds of type variables used during inference: ML type variables α obtained from the ML types returned by HM, and liquid type variables κ introduced during liquid constraint generation to stand for unknown liquid types. Our system is monomorphic in the liquid type variables. Polymorphism only enters via the ML type variables as fresh liquid type variables are created at each point where an ML type variable α is instantiated with a monomorphic ML type.

2. Whole Program Analysis and Non-General Types. Due to the above, the types we obtain for function inputs are the *strongest liquid supertype* of all the arguments passed into the function. This is in contrast with ML type inference which infers *the most general* type of the function independent of how the function is used. For example, consider the function `neg` defined as `fun x -> (-x)`, and suppose that $\mathbb{Q} = \{0 \leq \nu, 0 \geq \nu\}$. In a program comprising *only* the above function *i.e.,* where the function is never passed arguments, our algorithm infers `neg` :: $\{0 \leq \nu \wedge 0 \geq \nu\} \rightarrow \{0 \leq \nu \wedge 0 \geq \nu\}$ which is useless but sound. If `neg` is only called with (provably) non-negative (resp. non-positive) arguments, the algorithm infers `neg` :: $\{0 \leq \nu\} \rightarrow \{0 \geq \nu\}$ (resp. `neg` :: $\{0 \geq \nu\} \rightarrow \{0 \leq \nu\}$) If `neg` is called with arbitrary arguments, the algorithm infers `neg` :: `int`$\rightarrow$`int` and not a more general intersection of function types. We found this design choice greatly simplified the inference procedure by avoiding the expensive "case splits" on all possible inputs [14] while still allowing us to prove the safety of challenging benchmarks. Moreover, we can represent the intersection type in our system as: $\mathtt{x}\!:\!\mathtt{int} \rightarrow \{(0 \leq \mathtt{x} \Rightarrow 0 \geq \nu) \wedge (0 \geq \mathtt{x} \Rightarrow 0 \leq \nu)\}$, and so, if needed, we can recover the precision of intersection types by using qualifiers containing implications.

3. A-Normalization. Recall the `sum` example from Section 2. Our system as described would fail to infer that the output type of: `let rec sum k = if k < 0 then 0 else (s + sum (k-1))` was non-negative, as it cannot use the fact that `sum (k-1)` is non-negative when inferring the type of the `else` expression. This is solved by *A-Normalizing*[12] the program so that intermediate subexpressions are bound to temporary variables, thus allowing us to use information about types of intermediate expressions, as in the original `sum` implementation.

5. Experimental Results

We now describe our implementation of liquid type inference in the tool DSOLVE which does liquid type inference for OCAML. We

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

$\mathsf{Weaken}(c, A) =$
    match $c$ with
    | $\Gamma \vdash \{\nu\!:\!B \mid \theta \cdot \kappa\} \longrightarrow$
        $A[\kappa \mapsto \{q \mid q \in A(\kappa)$ and $\mathsf{Shape}(\Gamma); \nu\!:\!B \vdash \theta \cdot q : \mathtt{bool}\}]$
    | $\Gamma \vdash \{\nu\!:\!B \mid \rho\} <: \{\nu\!:\!B \mid \theta \cdot \kappa\} \longrightarrow$
        $A[\kappa \mapsto \{q \mid q \in A(\kappa)$ and $[\![A(\Gamma)]\!] \wedge [\![A(\rho)]\!] \Rightarrow [\![\theta \cdot q]\!]\}]$
    | $\_ \longrightarrow$ Failure

$\mathsf{Solve}(C, A) =$
    if exists $c \in C$ such that $A(c)$ is not valid
    then $\mathsf{Solve}(C, \mathsf{Weaken}(c, A))$ else $A$

$\mathsf{Infer}(\Gamma, e, \mathbb{Q}) =$
    let $(F, C) = \mathsf{Cons}(\Gamma, e)$ in
    let $A = \mathsf{Solve}(\mathsf{Split}(C), \lambda\kappa.\mathsf{Inst}(\Gamma, e, \mathbb{Q}))$ in
    $A(F)$
```

Figure 4. Liquid Type Inference Algorithm

describe experiments which demonstrate, over a set of benchmarks that were previously annotated in the DML project [27], that liquid types greatly reduce the burden of manual dependent type annotation required to prove the safety of array accesses.

DSOLVE takes as input a closed OCAML program and a set of qualifiers $\mathbb{Q}$, and outputs the dependent types inferred for the program expressions and the set of applications of primitive operations that *could not* statically be proven safe (ideally empty). DSOLVE is built on top of OCAML: DSOLVE uses the OCAML parser and type inference engine (to implement the oracle HM), and outputs the inferred dependent types in `.annot` files as documentation.

Array Bounds Checking Qualifiers. To automate array bounds checking, we observe that the safety of array accesses typically depends on the *relative ordering* of integer expressions. Thus, to statically prove the safety of array accesses, we use the mechanically-generated set of array bounds checking qualifiers $\mathbb{Q}_{BC}$ defined as

$$\{\nu \bowtie X \mid \bowtie \in \{<, \leq, =, \neq, >, \geq\} \text{ and } X \in \{0, \star, \texttt{len}\ \star\}\}$$

Next, we show experimental results demonstrating that liquid type inference over $\mathbb{Q}_{BC}$ suffices to prove the safety of most array accesses. Even when DSOLVE needs extra qualifiers, the types inferred using $\mathbb{Q}_{BC}$ help the programmer quickly identify the relevant extra qualifiers.

Benchmarks. We use benchmarks from the DML project [26] (ported to OCAML) that were previously annotated with dependent types with the goal of statically proving the safety of array accesses [27]. The benchmarks are listed in the first column of Table 1. The second column indicates the size of the benchmark (ignoring comments and whitespace). The benchmarks include OCAML implementations of: the Simplex algorithm for Linear Programming (`simplex`), the Fast Fourier Transform (`fft`), Gaussian Elimination (`gauss`), Matrix Multiplication (`matmult`), Binary Search in a sorted array (`bsearch`), computing the Dot Product of two vectors (`dotprod`), Insertion sort (`isort`), the n-Queens problem (`queen`), the Towers of Hanoi problem (`tower`), a fast byte copy routine (`bcopy`), and Heap sort (`heapsort`). The above include all DML array benchmarks except `quicksort`, whose invariants remain unclear to us. In addition, we ran DSOLVE on a simplified Quicksort routine from OCAML's Sort module (`qsort-o`), a version ported from the DML benchmark (`qsort-d`) where one optimization is removed, and BITV, an open source bit vector library (`bitv`).

Array Bounds Checking Results. As shown in column **DSOLVE** of Table 1, DSOLVE needs no manual annotations for most programs: that is, the qualifiers $\mathbb{Q}_{BC}$ suffice to automatically prove the safety of all array accesses. For some of the examples, *e.g.,* `tower`, we do need to provide extra qualifiers. However, even in this case, the annotation burden is typically just a few qualifiers. For example, in `tower`, we require a qualifier which is analogous to $\nu = \texttt{n} - \texttt{h1} - \texttt{h2}$, which describes the height of the "third" tower, capturing the invariant that the height is the total number of rings `n` minus the rings in the first two towers. Similarly, in `bitv`, one qualifier states the key invariant relating a bit vector's length to the length of its underlying data structure. The time for inference is robust to the number of qualifiers as most qualifiers are pruned away by the well-formedness constraints. In our prototype implementation, the time taken for inference is reasonable even for non-trivial benchmarks like `simplex`, `fft` and `gauss`.

Case Study: Bit Vectors. We applied DSOLVE to verify the open source BITV bit vector library (version `0.6`). A bit vector in BITV consists of a record with two fields: `length`, the number of bits stored, and `bits`, the actual data. If b is the number of bits stored per array element, `length` and `bits` are related by $(\texttt{len bits} - 1) \cdot b < \texttt{length} \leq (\texttt{len bits}) \cdot b$. The executed code, and hence, dependent types are different for 32- and 64-bit machines. Thus, to verify the code using our conjunctive types, we fixed the word size to 32. We were able to verify the array safety of 58 of BITV's 65 bit vector creation, manipulation, and iteration functions, which contain a total of 30 array access operations.

There are three kinds of manual annotation needed for verification: *extra qualifiers* (14 lines, 605 characters), *trusted assumptions* (8 lines, 143 characters), and *interface specifications* (43 lines, 2390 characters). The trusted assumptions (which are akin to dynamic checks) are needed due to current limitations of our system. These include the conservative way in which modular arithmetic is embedded into EUFA, the lack of refinements for type variables and recursive datatypes, and the conservative handling of control-flow. The interface specifications are needed because BITV is a library, *i.e.,* an open program. Thus, for verification, we need to specify that

	Size		**DML**		**DSOLVE**		
Program	**Line**	**Char**	**Line**	**Char**	**Line**	**Char**	**Time (s)**
`dotprod`	7	158	3 (30%)	110 (41%)	0 (0%)	0 (0%)	0.31
`bcopy`	8	199	3 (27%)	172 (46%)	0 (0%)	0 (0%)	0.15
`bsearch`	24	486	3 (11%)	157 (24%)	0 (0%)	0 (0%)	0.46
`queen`	30	886	7 (19%)	309 (26%)	0 (0%)	0 (0%)	0.70
`isort`	33	899	12 (27%)	702 (44%)	0 (0%)	0 (0%)	0.88
`tower`	36	927	8 (18%)	348 (27%)	1 (2%)	26 (2%)	3.33
`matmult`	43	797	10 (19%)	454 (36%)	0 (0%)	0 (0%)	1.79
`heapsort`	84	1414	11 (12%)	433 (23%)	0 (0%)	0 (0%)	0.53
`fft`	107	3279	13 (11%)	790 (19%)	1 (1%)	25 (1%)	9.13
`simplex`	118	2712	33 (22%)	1913 (41%)	0 (0%)	0 (0%)	7.73
`gauss`	142	2431	22 (13%)	999 (29%)	1 (1%)	67 (2%)	3.17
TOTAL	633	14188	125 (17%)	6387 (31%)	3(1%)	93(1%)	
`qsort-o`	62	1585			0 (0%)	0 (0%)	1.89
`qsort-d`	112	2735			5 (5%)	63 (2%)	18.28
`bitv`	426	10757			65 (15%)	3138 (29%)	63.11

Table 1. Experimental Results: Size is the amount of program text (without annotation) after removing whitespace and comments from the code. **DML** is the amount of manual annotation required in the DML versions of the benchmarks. **DSOLVE** is the amount of manual annotation required by DSOLVE, *i.e.,* qualifiers not in $\mathbb{Q}_{BC}$. **Time** is the time taken by DSOLVE to infer dependent types.

the API functions are called with valid input vectors that satisfy invariants like the one described above. The interface specifications, by far the largest category of annotations, are unavoidable. The extra qualifiers and expressiveness limitations are directions for future work.

DSOLVE was able to locate a serious bounds checking error in BITV. The error occurs in BITV's `blit` function, which copies `c` bits from `v1`, starting at bit `offset1`, to `v2`, starting at bit `offset2`. This function first *checks* that the arguments passed are safe, and then calls a fast but unsafe internal function `unsafe_blit`.

```
let blit v1 offset1 v2 offset2 c =
  if c < 0 || offset1 < 0 || offset1 + c > v1.length
             offset2 < 0 || offset2 + c > v2.length
  then invalid_arg "Bitv.blit";
  unsafe_blit v1.bits offset1 v2.bits offset2 c
```

`unsafe_blit` immediately accesses the bit at `offset1` in `v1`, regardless of the value of `c`. When the parameters are such that: $\texttt{offset1} = \texttt{v1.length}$ and $\texttt{v1.length} \bmod b = 0$ and $\texttt{c} = 0$, the `unsafe_blit` attempts to access the bit at index `v1.length`, which must be located in `v1.bits[v1.length / `b`]`; but this is `v1.bits[len v1.bits]`, which is out of bounds and can cause a crash, as we verified with a simple input. The problem is that `blit` does not verify that the starting offset is within the bounds of the bit vectors. This is fixed by adding the test `offset1 >= v1.length` (and `offset2 >= v2.length` for similar reasons). DSOLVE successfully typechecks the fixed version.

6. Related Work

The first component of our approach is predicate abstraction, [1, 15] which has its roots in early work on axiomatic semantics [8]. Predicate abstraction has proven effective for path-sensitive verification of control-dominated properties of first-order imperative programs [3, 16, 11, 4, 29, 17].

The second component of our approach is constraint-based program analysis, an example of which is the ML type inference algorithm. Unlike other investigations of type inference for HM with subtyping, *e.g.,* [19, 23, 25], our goal is algorithmic dependent type inference, which requires incorporating path-sensitivity and decision procedures for EUFA. We also draw inspiration from type qualifiers [13] that refine types with a lattice of built-in and programmer-specified annotations. Our Shape and Fresh functions are similar to *strip* and *embed* from [13]. Liquid types extend qualifiers by assigning them semantics via logical predicates, 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 [5], but with the intention of checking and inferring rules for qualifier derivations. Our approach is complementary in that the rules themselves are fixed, but allow for the use of guard and value binding information in type derivations, thereby yielding a more powerful analysis. For example, it is unclear whether the approach of [5] would be able to prove the safety of any of our benchmark programs, due to the inexpressivity of the qualifiers and inference rules. On the other hand, our technique is more computationally expensive as the decision procedure is integrated with type inference.

The notion of type refinements was introduced in [14] with refinements limited to restrictions on the structure of algebraic datatypes, for which inference is decidable. DML(C) [28] 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. 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 [18], 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 polymorphism and 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 impossible.

7. Conclusions and Future Work

In this paper, we have presented a dependent type system called liquid types, a tool DSOLVE that infers liquid types, and experiments showing that DSOLVE can significantly reduce the amount manual annotation required to statically prove the safety of array accesses. Even in very complex benchmarks like BITV, DSOLVE needs 22 lines of manual hints, which is only 5% of the entire code size. The other annotations, namely, types specifying correct usage of interface functions, are unavoidable. Thus, we believe that liquid types will prove useful even for modular verification. If the modules are designed well, their interfaces should have far fewer functions than their implementations and so the gains from not having to manually specify the types of *all* top-level bindings will be significant.

Several challenges need to be addressed in order to realize the full potential of liquid types. First, we would like to make the system more expressive, for example, by extending the system to allow refinements for type variables and recursive datatypes. This will allow us to apply liquid types to a larger class of programs and properties. Second, for the cases when typechecking fails, we require error reporting techniques that help the programmer determine whether there is either an error in her program, the set of qualifiers is insufficient, or, the conservativeness of the system is to blame. One approach would be to devise a notion of *type counterexample* and adapt proof-based methods to check if the counterexample is feasible (*i.e.,* there is an error) or if not, to lazily extract new qualifiers from the counterexample [6, 3, 16]. Third, we would like to extend the system to include reasoning about imperative features. With such an extension, liquid types could be profitably applied to verify C++, Java and C# programs which use generic datatypes.

Acknowledgments. We thank Adam Chlipala, Cormac Flanagan, Radha Jagadeesan, Suresh Jagannathan, Kenn Knowles, Sorin Lerner, Bill McLoskey, Todd Millstein, Corneliu Popea, Philip Wadler, Westley Weimer, and the anonymous referees for their helpful comments and suggestions for improving this paper.

References

[1] Tilak Agerwala and Jayadev Misra. Assertion graphs for verifying and synthesizing programs. Technical Report 83, University of Texas, Austin, 1978.

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

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

[4] S. Chaki, E. M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs. *FMSD*, 25(2-3):129–166, 2004.

[5] B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, pages 264–278, 2006.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, LNCS 1855, pages 154–169. Springer, 2000.

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

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

[9] B. Dutertre and L. De Moura. Yices SMT solver. http://yices.csl.sri.com/.

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

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

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

[13] J.S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, U.C. Berkeley, 2002.

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

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

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

[17] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking c programs using f-soft. In *ICCD*, pages 297–308, 2005.

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

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

[20] P. Martin-Lof. Constructive mathematics and computer programming. *Royal Society of London Philosophical Transactions Series A*, 312:501–518, October 1984.

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

[22] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

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

[24] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. Technical Report CSE Tech Report, UCSD, 2008.

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

[26] H. Xi. DML code examples. http://www.cs.bu.edu/fac/hwxi/DML/.

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

[28] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

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