Liquid Types

Patrick Rondon Ming Kawaguchi Ranjit Jhala

March 5, 2008

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 paying the heavy price of 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 an elementary method for automatically generating qualifiers from program text, DSOLVE reduces the amount of manual annotation required for proving safety from 31% of program text to under 1%.

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; we get the benefits of strong static typing for free.

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 the accessing of an array `a` of size 100 at the index `i`, or a division by `i`. Instead, the language can only provide a weaker dynamic safety guarantee at additional cost of high performance overhead.

In an exciting development, several authors [14, 23, 2, 24, 19, 9] have proposed the use of *dependent types* as a mechanism for enhancing the expressivity of type systems. 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 [23]. However, these benefits came at the price of automatic inference. In the benchmarks analyzed by Xi and Pfenning, about 31% of the code (or 17% by number of lines) was annotation that the programmer had to enter to enable the type checker to prove safety. We believe that this non-trivial annotation burden hampered the widespread adoption of dependent types despite their enormous 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 of manual annotation. The technical heart of our inference algorithm is a technique for synergistically blending Hindley-Milner type inference with *predicate abstraction*, a technique for synthesizing loop invariants for imperative programs that forms the algorithmic core of industrial-strength software model checkers like SLAM [3]. Our system takes as input a *closed* program and a set of *logical qualifiers* $\mathbb{Q}$, which are simple boolean predicates over the program variables and a special *value variable* ν. The system then uses the qualifiers to infer *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, which are deemed to hold iff 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 constraint-based algorithm for dependent type inference (Section 4).

Step 1: Hindley-Milner Type Inference: First, our algorithm invokes Hindley-Milner [6] 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.

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, 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 [19, 9].

To validate the utility of our technique, we have built DSOLVE, which does liquid type inference for OCAML. While liquid types can be used to statically prove a variety of properties [1], in this paper we focus on the canonical problem of proving the safety of array accesses, itself the foundation for other kinds of safety policies. Furthermore, by obviating run-time bounds checks, we achieve this safety without performance penalty.

We use a diverse set of challenging benchmarks taken from the DML project to demonstrate that, for many programs, DSOLVE, in conjunction with a simple *automatic qualifier generation* procedure, can prove safety *completely automatically* (Section 5). For the few programs where the automatic generator does not return sufficient qualifiers, the programmer typically needs to only specify one or two extra qualifiers. Even in these rare cases, the dependent types inferred by DSOLVE using *only* the generated qualifiers help the programmer rapidly identify the relevant extra qualifiers. We show that, over all the benchmarks, DSOLVE with automatic qualifier generation reduces the manual annotation required to prove safety from 31% of program text (or 17% by number of lines) to about 1%. Finally, we describe a case study where DSOLVE was able to pinpoint an error in an open-source OCAML `bitvector` library implementation, in a function that contained an *explicit* (but insufficient) safety check.

2 Overview

We begin with an overview of our technique 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, show how our system infers dependent types.

Dependent Types. Following [2, 9], 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 called the *refinement predicate* constraining the value variable. 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, $\{\nu : \texttt{int} \mid \texttt{0} < \nu\}$ 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, 9]). 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. The set of *logical qualifiers* $\mathbb{Q}$ is a finite set of boolean valued expression (or predicates) over the program variables and the special value variable ν distinct from the program variables. There are no restrictions on the expressions appearing in $\mathbb{Q}$. When synthesizing dependent types from the logical qualifiers, our system ensures that the types are well-formed, *i.e.,* for each expression, the inferred type is over variables in the innermost statically enclosing lexical scope. A *liquid type over* $\mathbb{Q}$ is a dependent type where the base refinement predicates are conjunctions of expressions from the logical qualifiers $\mathbb{Q}$.

Liquid Type Inference. For the rest of this section, assume that: $\mathbb{Q} \equiv \{0 \leq \nu,\ x \leq \nu,\ y \leq \nu,\ \nu < \texttt{n},\ \nu < \texttt{len}\ a\}$. Our algorithm proceeds in three steps: First, we perform Hindley-Milner (HM) type inference and use the results to generate *templates* which are complex 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 using predicate abstraction to infer dependent types over $\mathbb{Q}$. 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 $\kappa_{\texttt{x}}$, $\kappa_{\texttt{y}}$, κ_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 system 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:

$$\texttt{x} : \kappa_{\texttt{x}}; \texttt{y} : \kappa_{\texttt{y}}; (\texttt{x} > \texttt{y}) \vdash_{\mathbb{Q}} \{\nu = \texttt{x}\} <: \kappa_1 \tag{1.1}$$

$$\texttt{x} : \kappa_{\texttt{x}}; \texttt{y} : \kappa_{\texttt{y}}; \neg(\texttt{x} > \texttt{y}) \vdash_{\mathbb{Q}} \{\nu = \texttt{y}\} <: \kappa_1 \tag{1.2}$$

Constraint (1.1) (resp. (1.2)) stipulates that, under the condition that `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`), 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 $\kappa_{\texttt{x}}$, $\kappa_{\texttt{y}}$ to *true*, meaning that *any* integer arguments can be passed, and use a theorem prover to find the *strongest* conjunction of predicates in $\mathbb{Q}$ that satisfies the subtyping constraints. The theorem prover deduces that when $\texttt{x} > \texttt{y}$ (resp. $\neg(\texttt{x} > \texttt{y})$) if $\nu = \texttt{x}$ (resp. $\nu = \texttt{y}$) then $\texttt{x} \leq \nu$ and $\texttt{y} \leq \nu$. Hence, our technique infers that $\texttt{x} \leq \nu \wedge \texttt{y} \leq \nu$ is the strongest solution for κ_1 that satisfies the two constraints, and by substituting the solution for κ_1 into the template for `max`, infers

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

Example 2: Recursion. Next, we show how our system infers that the recursive function `sum` from Figure 1 always returns a non-negative value.

(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`, $\mathtt{k}{:}\kappa_{\mathtt{k}}{\rightarrow}\kappa_2$, where $\kappa_{\mathtt{k}}$ 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:

$$\mathtt{sum}{:}\ldots;\mathtt{k}{:}\kappa_{\mathtt{k}};\mathtt{k}<0 \vdash_{\mathbb{Q}} \{\nu = 0\} <: \kappa_2 \tag{2.1}$$

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

$$\mathtt{sum}{:}\ldots;\mathtt{k}{:}\kappa_{\mathtt{k}};\neg(\mathtt{k}<0) \vdash_{\mathbb{Q}} \{\nu = \mathtt{k}-1\} <: \kappa_{\mathtt{k}} \tag{2.2}$$

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.

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

(Step 3) Here, as `sum` is called, we try to find the strongest conjunction of qualifiers for $\kappa_{\mathtt{k}}$ and κ_2 that satisfies the constraints. To satisfy (2.2), $\kappa_{\mathtt{k}}$ 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 $0 \leq \nu$, the strongest conjunction of qualifiers that satisfies (2.1) and (2.3). Constraint (2.1) is trivially satisfied as the theorem prover deduces that if $\nu = 0$ then $0 \leq \nu$. Constraint (2.3) is satisfied as the theorem prover deduces that when $\neg(\mathtt{k}<0)$ and $[\mathtt{s}/\nu](0 \leq \nu)$, if $\nu = \mathtt{s}+\mathtt{k}$ then $0 \leq \nu$. The substitution $[\mathtt{s}/\nu](0 \leq \nu)$ 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 $\mathtt{s}+\mathtt{k}$. 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 $\kappa_{\mathtt{k}}$ and κ_2 into the template, our system infers

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

Example 3: Higher-Order Functions. Next, consider a program comprising only the higher-order accumulator `foldn` shown in Figure 1. We show how our system 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 $\forall\alpha.\mathtt{n}{:}\kappa_{\mathtt{n}}{\rightarrow}\mathtt{b}{:}\alpha{\rightarrow}\mathtt{f}{:}(\kappa_3{\rightarrow}\alpha{\rightarrow}\alpha){\rightarrow}\alpha$ for `foldn`, where $\kappa_{\mathtt{n}}$ 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 $\mathtt{i}{:}\kappa_{\mathtt{i}}{\rightarrow}\mathtt{c}{:}\alpha{\rightarrow}\alpha$, for `loop`, which, as for `sum`, 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 subtype constraints on the "then" and "else" expressions. Instead, the two interesting constraints are:

$$\ldots;\mathtt{i}{:}\kappa_{\mathtt{i}};\mathtt{i}<\mathtt{n} \vdash_{\mathbb{Q}} \{\nu = \mathtt{i}+1\} <: \kappa_{\mathtt{i}} \tag{3.1}$$

```
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

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

$$\ldots; \mathtt{i}{:}\kappa_{\mathtt{i}}; \mathtt{i} < \mathtt{n} \vdash_{\mathbb{Q}} \{\nu = \mathtt{i}\} <: \kappa_3 \tag{3.2}$$

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

$$\ldots \vdash_{\mathbb{Q}} \{\nu = 0\} <: \kappa_{\mathtt{i}} \tag{3.3}$$

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

(Step 3) Here, as `foldn` is not called, we assign $\kappa_{\mathtt{n}}$ to *true* and try to find the strongest conjunction of predicates in $\mathbb{Q}$ for $\kappa_{\mathtt{i}}$ and κ_3. We can assign to $\kappa_{\mathtt{i}}$ 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, $\kappa_{\mathtt{n}}$ into the template our system 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

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

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 system combines predicate abstraction, (1) function subtyping, and (2) polymorphism to prove (1) that array accesses are made with arguments between 0 and `len a` and hence safe, and (2) that `arraymax` returns a non-negative integer.

(Step 1) HM infers that (1) `arraymax` has the type $\mathtt{a}{:}\mathtt{intarray}{\rightarrow}\mathtt{int}$, (2) `am` has the type $\mathtt{l}{:}\mathtt{int}{\rightarrow}\mathtt{m}{:}\mathtt{int}{\rightarrow}\mathtt{int}$, and (3) `foldn` called in the body is a polymorphic instance where the type variable α has been instantiated with `int`. Consequently, our system creates the following templates: (1) $\mathtt{a}{:}\mathtt{intarray}{\rightarrow}\kappa_4$ for `arraymax`, where κ_4 represents the unknown refinement for the output of `arraymax`,

(2) $\mathtt{l}:\kappa_1 \rightarrow \mathtt{m}:\kappa_{\mathtt{m}} \rightarrow \kappa_5$ for `am`, where κ_1, $\kappa_{\mathtt{m}}$ 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 the previous example with κ_6 substitute for α namely, $\mathtt{n}:\mathtt{int} \rightarrow \mathtt{b}:\kappa_6 \rightarrow \mathtt{f}:(\{\mathtt{0} \leq \nu \land \nu < \mathtt{n}\} \rightarrow \kappa_6 \rightarrow \kappa_6) \rightarrow \kappa_6$

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

$$\mathtt{l}:\kappa_1; \mathtt{m}:\kappa_{\mathtt{m}} \vdash_{\mathbb{Q}} \{\nu = \mathtt{l}\} <: \{\mathtt{0} \leq \nu \land \nu < \mathtt{len}\ a\} \tag{4.1}$$

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

$$\mathtt{l}:\kappa_1; \mathtt{m}:\kappa_{\mathtt{m}} \vdash_{\mathbb{Q}} \{\mathtt{sub\ a\ l} \leq \nu \land \mathtt{m} \leq \nu\} <: \kappa_5 \tag{4.2}$$

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:

$$\ldots \vdash_{\mathbb{Q}} \{\nu = \mathtt{0}\} <: \kappa_6 \tag{4.3}$$

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)

$$\ldots \vdash_{\mathbb{Q}} \mathtt{l}:\kappa_1 \rightarrow \mathtt{m}:\kappa_{\mathtt{m}} \rightarrow \kappa_5 <: \{\mathtt{0} \leq \nu \land \nu < \mathtt{len\ a}\} \rightarrow \kappa_6 \rightarrow \kappa_6$$

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

$$\ldots \vdash_{\mathbb{Q}} \kappa_6 <: \kappa_4 \tag{4.5}$$

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

$$\ldots \vdash_{\mathbb{Q}} \{\mathtt{0} \leq \nu \land \nu < \mathtt{len\ a}\} <: \kappa_1 \tag{4.6}$$

$$\ldots \vdash_{\mathbb{Q}} \kappa_6 <: \kappa_{\mathtt{m}} \tag{4.7}$$

$$\ldots \vdash_{\mathbb{Q}} \kappa_5 <: \kappa_6 \tag{4.8}$$

(Step 3) The strongest conjunction of qualifiers that we can assign to: $\kappa_{\mathtt{m}}$, κ_4, κ_5 and κ_6 is the predicate $\mathtt{0} \leq \nu$. In essence the solution infers that we can "instantiate" the type variable α with the refinement $\{\nu:\mathtt{int} \mid \mathtt{0} \leq \nu\}$. 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 $\kappa_{\mathtt{m}}$) is non-negative then the output (of type κ_5) is non-negative (constraint (4.2) is satisfied). Plugging the solution into the template, the system infers

$$\mathtt{arraymax} :: \mathtt{intarray} \rightarrow \{0 \leq \nu\}$$

The strongest conjunction over $\mathbb{Q}$ we can assign to κ_1 is $\mathtt{0} \leq \nu \land \nu < \mathtt{len\ a}$, which trivially satisfies constraint (4.7). Moreover, with this assignment, we have satisfied the "bounds check" constraint (4.1), *i.e.,* we have found an assignment of dependent types to all the program expressions that suffices to prove that all array accesses occur within bounds.

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).

e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	$\mathtt{c}$	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$	if-then-else
	$\mid$	$\mathtt{let}\ x\ =\ e\ \mathtt{in}\ e$	let-binding
	$\mid$	$\mathtt{let\ rec}\ f\ =\ \lambda x.e\ \mathtt{in}\ e$	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:*
	$\mid$	$\mathtt{int}$	base type of integers
	$\mid$	$\mathtt{bool}$	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	$::=$	$\mathbb{T}(\mathit{true}), \mathbb{S}(\mathit{true})$	*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**

3.1 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 idioms `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). Figure 3 shows the rules describing well-formed environments. Notice that in each type binding, the dependent type must also be well-formed *i.e.,* the "free variables" appearing in the refinement predicates must be bound in the prefix of the environment.

Shapes. The function Shape maps arbitrary dependent types (schemas) to ML types (schemas), where 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 dropping the guards and observing $x{:}\mathsf{Shape}(S) \in \mathsf{Shape}(\Gamma)$ iff $x{:}S \in \Gamma$.

Constants. As in [19, 9], the basic units of computation are the constants $\mathtt{c}$ built into λ_{L}, each of which has a dependent type $\mathit{ty}(\mathtt{c})$ 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:

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

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 [9]), and the static semantics is defined via a sound overapproximation of the dynamic semantics [1]. 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

Next, we present the key ingredients of the type system, the typing judgements and derivation rules summarized in Figure 3. Our system has three kinds of judgements relating environments, expressions and types.

Well-formedness 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 in the scope of the corresponding expression.

Subtype Judgement $\Gamma \vdash_{\mathbb{Q}} S_1 <: S_2$: states that dependent type schema S_1 *is a subtype of* schema S_2 in environment Γ.

Liquid Type Judgement $\Gamma \vdash_{\mathbb{Q}} e : S$: 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, we 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 judgement $\Gamma \vdash e : S$, 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 [1] (Theorems 3, 4 and 5). 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, in the `sum` example from Section 2, without the branch information, 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. Let EUFA be the decidable logic [18] of *equality*, *uninterpreted functions* and *linear arithmetic*, shown in rule [DEC-<:-BASE] of Figure 3. 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. 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 [9, 19]. For example, for the `then` expression in `max` from Section 2, the subtyping relation: $\mathtt{x}\!:\!\mathtt{int};\mathtt{y}\!:\!\mathtt{int};\mathtt{x} > \mathtt{y} \vdash_{\mathbb{Q}} \{\nu = \mathtt{x}\} <: \{\mathtt{x} \leq \nu \wedge \mathtt{y} \leq \nu\}$ 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`. `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`; intuitively, 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 [19, 9] 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 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, 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 [1]. The alternative of existentially quantifying the local variable [16] makes algorithmic checking hard .

4 Liquid Type Inference

Next, we describe the heart of our technique: an algorithm that takes a set of logical qualifiers $\mathbb{Q}$ and an expression e as input and determines whether e is well-typed over $\mathbb{Q}$, *i.e.,* whether there exists some T such that $\emptyset \vdash_{\mathbb{Q}} e : T$. 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 the finite set of logical qualifiers $\mathbb{Q}$ to solve the constraints using a technique inspired by predicate abstraction, thereby inferring readable types for all subexpressions and determining whether the expression can be well-typed over $\mathbb{Q}$.

4.1 ML Types and Templates

Our type inference algorithm is based on the observation that the liquid type derivations and, hence, the dependent types for each subexpression 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 iff, using the classical ML type derivation rules [6], there exists a derivation $\Gamma \vdash e : \sigma$. Observe that as the liquid type derivation rules are "refinements" of the ML type derivation rules, 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.

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

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

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

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

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

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

$$\frac{\Gamma \vdash x : \hat{T}_x \rightarrow \hat{T}_1 \quad \Gamma[x \mapsto \hat{T}_x] \vdash_{\mathbb{Q}} e_1 : \hat{T}_1}{\Gamma \vdash_{\mathbb{Q}} (\lambda x. e_1) : (x : \hat{T}_x \rightarrow \hat{T}_1)} \text{[LT-Fun]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : (x : T_1 \rightarrow T) \qquad \Gamma \vdash_{\mathbb{Q}} e_2 : T_1}{\Gamma \vdash_{\mathbb{Q}} e_1 \; e_2 : [e_2/x]T} \text{[LT-App]}$$

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

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

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \qquad \alpha \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e : \forall\alpha.S} \text{[LT-Gen]}$$

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

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

$$\frac{\mathsf{Valid}([\![\Gamma]\!] \wedge [\![e_1]\!] \Rightarrow [\![e_2]\!])}{\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid e_1\} <: \{\nu : B \mid e_2\}} \text{[Dec-<:-Base]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} T'_2 <: T'_1 \qquad \Gamma[x \mapsto T'_2] \vdash_{\mathbb{Q}} T''_1 <: T''_2}{\Gamma \vdash_{\mathbb{Q}} x : T'_1 \rightarrow T''_1 <: x : T'_2 \rightarrow T''_2} \text{[Dec-<:-Fun]}$$

$$\frac{}{\Gamma \vdash_{\mathbb{Q}} \alpha <: \alpha} \text{[<:-Var]} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} S_1 <: S_2}{\Gamma \vdash_{\mathbb{Q}} \forall\alpha.S_1 <: \forall\alpha.S_2} \text{[<:-Poly]}$$

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

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

$$\frac{\Gamma; x : T_1 \vdash T_2}{\Gamma \vdash x : T_1 \rightarrow T_2} \text{[WT-Fun]} \qquad \frac{\Gamma \vdash S}{\Gamma \vdash \forall\alpha.S} \text{[WT-Poly]}$$

Figure 3: **Rules for Liquid Type Checking**

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 [16]. Note that substitution can be "pushed inside" type constructors, *i.e.,* $\theta \cdot (\{\kappa_1\} \rightarrow \{\kappa_2\})$ 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

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. 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, to ensure that the liquid types inferred for each expression are over program variables that are in the expression's scope. **Subtyping constraints** of the form $\Gamma \vdash_{\mathbb{Q}} F_1 <: F_2$ where Γ is a template environment and F_1 and F_2 are two templates of the same shape, ensure that the liquid types inferred for each sub-expression can be combined to yield a valid type derivation by ensuring that appropriate subsumption relationships hold.

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 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). To infer the types of these these expressions, we exploit two observations. 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 $\mathbb{Q}$ (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 *fresh template* which has the same "shape" as the ML type but which has *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` sub-expressions. Note that for the `then` (resp. `else`) sub-expression, 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 sub-expressions, together with subtyping constraints forcing the templates for the `then` and `else` sub-expressions 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: $\mathtt{x}\!:\!\kappa_{\mathtt{x}};\mathtt{y}\!:\!\kappa_{\mathtt{y}};(\mathtt{x} > \mathtt{y}) \vdash_{\mathbb{Q}} \{\nu = \mathtt{x}\} <: \kappa_{1'}$, $\mathtt{x}\!:\!\kappa_{\mathtt{x}};\mathtt{y}\!:\!\kappa_{\mathtt{y}};\neg(\mathtt{x} > \mathtt{y}) \vdash_{\mathbb{Q}} \{\nu = \mathtt{y}\} <: \kappa_{1'}$, and $\mathtt{x}\!:\!\kappa_{\mathtt{x}};\mathtt{y}\!:\!\kappa_{\mathtt{y}} \vdash_{\mathbb{Q}} \kappa_{1'} <: \kappa_1$ are generated, capturing the relationships between the then, else expressions and the body, and the body and the output type 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 the constraints are satisfied. 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 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 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 maps from templates (F) to dependent types (AF) and template environments (Γ) to environments $(A\Gamma)$, by substituting each liquid type variable κ with $A(\kappa)$ *and then* applying the pending substitutions. A *satisfies* a constraint c if Ac is valid. That is, A satisfies a well-formedness constraint $\Gamma \vdash F$ if $\mathsf{Shape}(\Gamma) \vdash AF$, and a subtyping constraint $\Gamma \vdash_{\mathbb{Q}} F_1 <: F_2$ if $A\Gamma \vdash_{\mathbb{Q}} AF_1 <: AF_2$. A satisfies 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 (*e.g.,* function types) into simple constraints over base types. An assignment satisfies C iff it satisfies $\mathsf{Split}(C)$.

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 as the environment has no bindings and the value variable ν; 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. Constraints (4.6),(4.7),(4.8) from Section 2, are the result of splitting 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 $\kappa_{\mathtt{k}}$ (4.6) thus allowing the system to statically check the array access.

Step 2: Iterative Weakening. Next, we call the procedure Solve, shown in Figure 5, to find a solution for the simple constraints. Solve takes a set of simple constraints and a finite set of logical qualifiers and returns either an assignment satisfying the constraints or fails indicating 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 $\mathbb{Q}$, and then repeatedly picks a constraint that is not satisfied by the current assignment and refines the assignment by removing qualifiers that prevent the constraint from being satisfied. For unsatisfied constraints of the form: (1) $\Gamma \vdash \{\nu\!:\!B \mid \theta \cdot \kappa\}$, 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 environment $\mathsf{Shape}(\Gamma);\nu\!:\!B$, (2) $\Gamma \vdash_{\mathbb{Q}} \{\nu\!:\!B \mid \rho\} <: \{\nu\!:\!B \mid \theta \cdot \kappa\}$, where ρ is either a refinement predicate or a liquid variable with pending substitutions, we remove 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) $\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid \rho\} <: \{\nu : B \mid e\}$, the weakening procedure, 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 whose conjunction is $A(\kappa)$ *contains* the set of logical qualifiers whose conjunction is $A'(\kappa)$. 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, the iterative weakening starts with the least possible solution where each liquid variable is assigned the conjunction of all qualifiers and then iteratively weakens the assignment until the *minimum* solution is found. The correctness of Solve follows from the following invariant about the iterative weakening: if A^* is the minimum assignment that satisfies all the constraints, then in each iteration, the assignment $A \leq A^*$. Thus, if $\mathsf{Solve}(C, \mathbb{Q})$ returns a solution then it is the minimum solution for C over $\mathbb{Q}$. If at some point a constraint $\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid \rho\} <: \{\nu : B \mid e\}$ is unsatisfied, subsequent weakening cannot make it become satisfied, and so if $\mathsf{Solve}(C, \mathbb{Q})$ 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 5. 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.

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* $\mathsf{Infer}(\Gamma, e, \mathbb{Q})$ *returns* ***Failure*** *then there is no* S, $\Gamma \vdash_{\mathbb{Q}} e : S$.

Due to space constraints the proof of the above theorem is omitted, but can found in [1]. Using standard worklist-based techniques, Infer can be implemented so that its running time is *linear* in the size of the ML type derivation times the number of qualifiers.

4.4 Features of Liquid Type Inference

Next, we 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 system 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 system 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 system 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: `x`:`int`$\rightarrow \{(0 \leq \mathtt{x} \Rightarrow 0 \geq \nu) \wedge (0 \geq x \Rightarrow 0 \leq \nu)\}$, and so, if needed, we can recover the precision of intersection types by using qualifiers containing implications. Of course, as our system is a whole program analysis, for open systems *e.g.,* libraries, manual annotations remain the only way to specify API usage.

3. Scoping and Well-formedness. The wellformedness constraints ensure that types are inferred soundly *regardless* of the free variables that appear in the qualifiers. If the set of constraints has a solution, system solves the constraints, then by Theorem 2 a valid type derivation exists and by Theorem 1 no run time errors occur.

4. 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.

```
$\mathsf{Cons}(\Gamma, e) =$
  match $e$ with
  | $x \longrightarrow$
      if $\mathsf{HM}(\Gamma, e) = B$ then $(\{\nu : B \mid \nu = x\}, \emptyset)$
      else $(\Gamma(x), \emptyset)$
  | $\mathsf{c} \longrightarrow$
      $(ty(\mathsf{c}), \emptyset)$
  | $e_1\ e_2 \longrightarrow$
      let $(x : F_f \to F_o, C_1) = \mathsf{Cons}(\Gamma, e_1)$ in
      let $(F_a, C_2) = \mathsf{Cons}(\Gamma, e_2)$ in
      $([e_2/x]F_o, C_1 \cup C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} F_a <: F_f\})$
  | $\lambda x.e_1 \longrightarrow$
      let $(x : F_f \to F_o) = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e))$ in
      let $(F_b, C_b) = \mathsf{Cons}(\Gamma; x : F_f, e_1)$ in
      $(x : F_f \to F_o, C_b \cup \{\Gamma \vdash x : F_f \to F_o\} \cup$
          $\{\Gamma; x : F_f \vdash_{\mathbb{Q}} F_b <: F_o\})$
  | if $e_1$ then $e_2$ else $e_3 \longrightarrow$
      let $F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e))$ in
      let $(\_, C_g) = \mathsf{Cons}(\Gamma, e_1)$ in
      let $(F_t, C_t) = \mathsf{Cons}(\Gamma; e_1, e_2)$ in
      let $(F_e, C_e) = \mathsf{Cons}(\Gamma; \neg e_1, e_3)$ in
      $(F, C_g \cup C_t \cup C_e \cup \{\Gamma \vdash F\} \cup$
          $\{\Gamma; e_1 \vdash_{\mathbb{Q}} F_t <: F\} \cup$
          $\{\Gamma; \neg e_1 \vdash_{\mathbb{Q}} F_e <: F\})$
  | let $x\ =\ e_1$ in $e_2 \longrightarrow$
      let $F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e))$ in
      let $(F_x, C_x) = \mathsf{Cons}(\Gamma, e_1)$ in
      let $(F_b, C_b) = \mathsf{Cons}(\Gamma; x : F_x, e_2)$ in
      $(F, C_x \cup C_b \cup \{\Gamma \vdash F\} \cup$
          $\{\Gamma; x : F_x \vdash_{\mathbb{Q}} F_b <: F\})$
  | $[\Lambda\alpha]e \longrightarrow$
      let $(F, C) = \mathsf{Cons}(\Gamma, e)$ in
      $(\forall\alpha.F, C)$
  | $[\tau]e \longrightarrow$
      let $F_i = \mathsf{Fresh}(\tau)$ in
      let $(\forall\alpha.F, C) = \mathsf{Cons}(\Gamma, e)$ in
      $([F_i/\alpha]F, C \cup \{\Gamma \vdash F_i\})$
```

Figure 4: **Liquid Constraint Generation**

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

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

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

Figure 5: **Liquid Constraint Solving**

5 Experimental Results

We now describe our implementation of liquid type inference in the tool Dsolve which does liquid type inference for Ocaml. We describe experiments which demonstrate, over a set of benchmarks that were pre-

	Size		DML		Dsolve		
Program	**Line**	**Char**	**Line**	**Char**	**Line**	**Char**	**Time (s)**
`dotprod`	7	158	3 (30%)	110 (41%)	0 (0%)	0 (0%)	0.43
`bcopy`	8	199	3 (27%)	172 (46%)	0 (0%)	0 (0%)	0.32
`bsearch`	24	486	3 (11%)	157 (24%)	0 (0%)	0 (0%)	0.96
`queen`	30	886	7 (19%)	309 (26%)	0 (0%)	0 (0%)	1.67
`isort`	33	899	12 (27%)	702 (44%)	0 (0%)	0 (0%)	1.97
`tower`	36	927	8 (18%)	348 (27%)	1 (2%)	28 (2%)	5.66
`matmult`	43	797	10 (19%)	454 (36%)	0 (0%)	0 (0%)	2.94
`heapsort`	85	1414	11 (12%)	433 (23%)	0 (0%)	0 (0%)	1.67
`fft`	107	3279	13 (11%)	790 (19%)	1 (1%)	27 (1%)	20.36
`simplex`	118	2712	33 (22%)	1913 (41%)	0 (0%)	0 (0%)	12.92
`gauss`	142	2431	22 (13%)	999 (29%)	1 (1%)	67 (2%)	5.01
TOTAL	633	14188	125 (17%)	6387 (31%)	3(1%)	122(1%)	
`qsort-o`	62	1585			0 (0%)	0 (0%)	4.25
`qsort-d`	112	2735			5 (5%)	172 (6%)	17.53
`bitv`	85	1982			3 (4%)	110 (6%)	4.59

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, but not automatically generated, by Dsolve. **Time** describes the time taken by Dsolve to infer dependent types for each of the examples.

viously annotated in the DML project [23], that liquid types greatly reduce (often eliminate) the significant 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, and uses the Ocaml parser, type inference engine (to implement the oracle HM), and also outputs the inferred dependent types in the `.annot` files produced during compilation so that they may be inspected easily.

Automatic Qualifier Generation. To automatically generate qualifiers, we observe that array bounds checking typically depends on the *relative ordering* of integer expressions. Thus, our qualifier generator traverses the program AST and creates qualifiers $\nu \bowtie x$ where $\bowtie \in \{<, \leq, =, >, \geq\}$ and x is an integer constant, an integer variable, or `len` a where a is an array. Next we show experimental results demonstrating that liquid type inference over the automatically generated qualifiers almost always suffices to prove the safety of all array accesses. Even in the rare cases where Dsolve needs extra qualifiers, the types inferred by Dsolve using only the generated qualifiers help the programmer quickly identify the relevant extra qualifiers.

Benchmarks. We use benchmarks from the DML project [22] (ported to Ocaml), that were previously annotated with dependent types with the goal of statically proving the safety of array accesses [23]. 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`), Heap sort (`heapsort`). The above include all DML array benchmarks except (`quicksort`), whose invariants remain unclear to us at time of writing. In addition, we ran Dsolve on a simplified Quicksort routine from Ocaml's Sort module (`qsort-o`), a version ported from the DML (`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, for the large majority of programs, unlike DML, Dsolve needs *no extra annotations*. The qualifiers generated by the simple syntactic method suffice to prove the safety of all array accesses in a fully automatic manner. 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 $\nu = \mathtt{n} - \mathtt{h1} - \mathtt{h2}$, 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. For `bitv`, one qualifier states the key invariant about the record-based implementation of the bitvector, and the others could be generated by extending the heuristic to include record fields. The time for inference is robust to the number of qualifiers, as most qualifiers are pruned away by the well-formedness constraints. This enables the use of a simple and greedy qualifier generation method. In our prototype implementation, the time taken for inference is reasonable even for non-trivial benchmarks like `simplex` and `fft`.

Case Study: Bit Vectors. A bit vector is an arbitrarily-sized array of bits which supports accessors, boolean operations, etc. We used DSOLVE to verify the safety of bit vector creation, initialization, and accessors in the BITV library[8] which represents bit vectors as records with two fields: `length`, the integral number of bits it contains, and `bits`, an integer array containing the data. If b is the number of bits stored per array element, `length` and `bits` are related by $(\texttt{len bits} - 1) \cdot b < \texttt{length} \leq (\texttt{len bits}) \cdot b$.

DSOLVE found a serious off-by-one error in BITV's `blit` function (in the latest `v0.6`), 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: `offset1` = `v1.length` and `v1.length` mod $b = 0$ and `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 an off-by-one error in the test performed in `blit`, that is fixed by replacing the `>` test with `>=`. DSOLVE successfully typechecks the fixed version.

6 Related Work

The first component of our approach is predicate abstraction, [15] which has its roots in early work on axiomatic semantics [7]. Predicate abstraction has proven extremely effective for the path-sensitive verification of imperative programs [11, 4, 25], and forms the algorithmic core of industrial-strength tools like SLAM [3]. While it is very effective for control-dominated software, it is less effective for automated reasoning about complex data and higher-order control flow.

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.,* [21, 20, 17, 10], 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 their derivation, thereby yielding a more powerful analysis. For example, it is unclear whether the approach of [5] would be able to prove the safety any of the above 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) [24] 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 [16], 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.

References

[1] Anonymous. Please contact PC chair for details.

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

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

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

[8] J.-C. Filliátre. Bitv: a bit vectors library. http://www.lri.fr/ filliatr/software.en.html.

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

[10] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM TOPLAS*, 21(2):370–416, 1999.

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

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

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

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

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

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

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

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

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

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

A 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:

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

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 ν 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:

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

where `size` is a special constant, similar to `len` for the `intarray` type. Figure 6 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 6 shows how we extend the language of types to incorporate lists. A *dependent list type* is of the form $\{\nu : T\ \texttt{list} \mid e\}$ 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 `list` as an abbreviation for $\{\nu : T\ \texttt{list} \mid \mathit{true}\}$. Thus, an *ML list type* is a dependent list type where all refinement predicates are *true*.

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

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

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.

Type Checking. Figure 6 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

Syntax

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

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

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

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

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

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

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

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

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

Figure 6: Rules for Lists

function application, rule $\hat{T}$-App from Figure 3. 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, →, 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 **??** and **??** 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 $\mathbb{Q} = \{0 \leq \nu; 0 < \texttt{size}\ \nu\}$, our system allows us to derive:

$$\emptyset, \mathit{true} \vdash \texttt{[]} : \{\nu : \{\nu : \texttt{int} \mid 0 \leq \nu\}\ \texttt{list} \mid \texttt{size}\ \nu = 0\}$$

and from this, derive:

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

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

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

has the type $\{\nu : \texttt{int} \mid 0 \leq \nu\}$. 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: $\Gamma \triangleq [x \mapsto \{\nu : \{\nu : \texttt{int} \mid 0 \leq \nu\}\ \texttt{list} \mid 0 < \texttt{size}\ \nu\}]$ and guard environment $G \triangleq \texttt{size}\ x = 0$ obtained by substituting x for ν in $g_{\texttt{[]}}$. 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 $\{\nu : \texttt{int} \mid 0 \leq \nu\}$. 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*[12], where all the intermediate subexpressions get bound to temporary variables, thereby allowing us to infer the strongest possible liquid types.

B Other Applications of Liquid Types

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 7. Recall that / is a constant of the type $x\!:\!\texttt{int} \rightarrow y\!:\!\{\nu : \texttt{int} \mid \nu \neq 0\} \rightarrow \texttt{int}$. Using the logical qualifiers $\mathbb{Q} = \{0 \leq \nu\}$, our system is able to infer that *abs* from Section 1 has the liquid type $\texttt{int} \rightarrow \{\nu\!:\!\texttt{int} \mid 0 \leq \nu\}$. Thus, both i_a and n_a have the type $\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\}$. 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 $\{\nu : \texttt{int} \mid 0 < \nu\}$, 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 [23], shown in the "Binary Search" box of Figure 7. Our system is able to infer that all array accesses in the are safe using just the logical qualifiers $\mathbb{Q} = \{0 \leq \nu;\ \nu < \texttt{len}\ a\}$. The generated constraints have the minimal assignment which yields the liquid type: $l\!:\!\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\} \rightarrow h\!:\!\{\nu\!:\!\texttt{int} \mid \nu < (\texttt{len}\ a)\} \rightarrow \texttt{int}$ 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 7 (adapted from [23]) are safe, using the logical qualifiers: $\mathbb{Q} = \{0 \leq \nu;\ \nu \leq \texttt{len}\ u;\ \nu \leq \texttt{len}\ v\}$. The system infers the liquid type: $i\!:\!\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\} \rightarrow n\!:\!\{\nu\!:\!\texttt{int} \mid \nu \leq (\texttt{len}\ u) \wedge \nu \leq (\texttt{len}\ v)\} \rightarrow \texttt{int}$ 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: $\{\nu\!:\!\mathtt{int} \mid \nu \leq (\mathtt{len}\ u) \wedge \nu \leq (\mathtt{len}\ v)\}$ 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 **??**.

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 7 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 $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathtt{int} \rightarrow \alpha\ \mathtt{list}$. Using the qualifier set $\mathbb{Q} = \{0 < \nu\}$, we are able to determine that the function *double* has the type $k\!:\!\{\nu : \mathtt{int} \mid 0 < \nu\} \rightarrow \{\nu : \mathtt{int} \mid 0 < \nu\}$, *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 $\{\nu\!:\!\mathtt{int} \mid 0 < \nu\}\ \ \mathtt{list}$.

This is effected by automatically instantiating the polymorphic type variable α with a fresh liquid type variable κ. and generating the constraints: $\cdot \vdash \kappa_1 \rightarrow \kappa_2 <: \kappa \rightarrow \kappa$, $\cdot \vdash \{\nu\!:\!\mathtt{int} \mid \nu = 1\} <: \kappa$, for the curried application to *generate*, where $\kappa_1 \rightarrow \kappa_2$ is the template for *double*, whose body generates the constraint: $[k \mapsto \kappa_1], \mathit{true} \vdash \{\nu\!:\!\mathtt{int} \mid \nu = k + k\} <: \kappa_2$. As the minimal satisfying assignment to these constraints maps $\kappa, \kappa_1, \kappa_2$ to $0 < \nu$, 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 7. λ_{L} can encode the polymorphic option type in a manner akin to lists. *mapfilter* has the type: $(\alpha \rightarrow \beta\ \mathtt{option}) \rightarrow \alpha\ \ \mathtt{list} \rightarrow \beta\ \ \mathtt{list}$. Using the logical qualifier *prime* ν, 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 $\{\nu : \mathtt{int} \mid \mathit{prime}\ \nu\}\ \mathtt{list}$, and so x_1 has the liquid type $\{\nu : \mathtt{int} \mid \mathit{prime}\ \nu\}$, 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 7 shows the *append* function on two lists. Using only the logical qualifier: $\mathbb{Q} = \{\mathtt{size}\ \nu = \mathtt{size}\ l + \mathtt{size}\ m\}$, our system infers that *append* has the liquid type $l\!:\!\alpha\ \mathtt{list} \rightarrow m\!:\!\alpha\ \mathtt{list} \rightarrow L(l, m)$ where $L(l, m)$ is an abbreviation for $\{\nu\!:\!\alpha\ \mathtt{list} \mid \mathtt{size}\ \nu = \mathtt{size}\ l + \mathtt{size}\ m\}$.

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 $[l/\nu]g_{\mathtt{[]}}$. In the $x_1\mathtt{::}x_2$ case, notice that the guard environment contains contains the predicate $[l/\nu]g_{\mathtt{::}}$, which is $\mathtt{size}\ l = \mathtt{size}\ x_2 + 1$. 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 $\{\nu\!:\!\alpha\ \mathtt{list} \mid \mathtt{size}\ \nu = \mathtt{size}\ x_2 + \mathtt{size}\ m\}$. This, coupled with the guard predicate $g_{\mathtt{::}}$, allows the system to infer that the type of $x_1\mathtt{::}(\mathit{append}\ x_2\ m)$ is $\{\nu\!:\!\alpha\ \mathtt{list} \mid \mathtt{size}\ \nu = \mathtt{size}\ x_2 + \mathtt{size}\ m + 1\}$. As this occurs in the case where l matches with $x_1\mathtt{::}x_2$, the predicate $[l/\nu]g_{\mathtt{::}}$ in the guard environment allows the system to infer in that environment, $\{\mathtt{size}\ \nu = \mathtt{size}\ x_2 + \mathtt{size}\ m + 1\}$ is a subtype of $\{\mathtt{size}\ \nu = \mathtt{size}\ l + \mathtt{size}\ m\}$, thereby showing that the inferred type is indeed a fixpoint.

Using similar reasoning, our system infers, using only the logical qualifers $\mathbb{Q} = \{\mathtt{size}\ \nu \leq \mathtt{size}\ l\}$, that *mapfilter* from box "Map Filter" in Figure 7, has the liquid type:

$$(\alpha \rightarrow \beta\ \mathtt{option}) \rightarrow \alpha\ \mathtt{list} \rightarrow \{\nu\!:\!\beta\ \ \mathtt{list} \mid \mathtt{size}\ \nu \leq \mathtt{size}\ l\}$$

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 [23].

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 $\mathbb{Q} = \{0 < \mathtt{size}\ \nu\}$ our system

```
let trunc = λn. λi.                                        Truncation
   let i_a = abs i in
   let n_a = abs n in
   if i_a ≤ n_a then i else n_a ∗ (i/i_a)

let bsearch = λk.λa.                                    Binary Search
   let rec look = λl.λh.
      if l ≤ 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 = λu.λv.                                      Dot Product
   let rec loop = λi.λn.λs.
      if n ≤ 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

let rec generate = λf.λb.λn.                                 Generate
   if n = 0 then b::[]
   else let h = f b in h::(generate (n − 1) f h) in
let double = λk.k + k in
   generate double 1 10

let rec mapfilter = λf.λl.                                  MapFilter
   match l with [] → []
   | (x1::x2) →
      match f h with None → mapfilter f x2
      | Some x → x::(mapfilter f x2) in
...
let prime = λx.(∗ tricky primality test∗) in
...
let xs′ =
   mapfilter (λx. if prime x then Some x else None) xs
in ...
match xs′ with [] → ...
| x1::x2 → if prime x1 then ...else error 0

let rec append = λl.λm.                                       Append
   match l with [] → m
   | x1::x2 → (x1::(append x2 m))

let pow2 = λn.                                               Power 2
   let x = generate double 1 n
   (match x with []→ error 0 | x1::x2→ x1)
```

Figure 7: Liquid Type Examples

infers that the function *generate* in the box "Generate", has the liquid type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \texttt{int} \rightarrow \{\nu{:}\alpha\ \texttt{list} \mid 0 < \texttt{size}\ \nu\}$. Consider the function *pow2* in the box "Power 2" in Figure 7. Our system uses the liquid type inferred for *generate*, to infer that x has the liquid type $\{\nu{:}\texttt{int list} \mid 0 < \texttt{size}\ \nu\}$, *i.e.,* is not an empty list. Using reasoning similar to the example from Section A, our system is able to typecheck *pow2* by showing that `error` is never called.

C Formal Semantics of λ_{L}

Syntax. The syntax for λ_{L} expressions, types and schemas is shown in Fig 2.

Dynamic Semantics. The (call-by-value) dynamic semantics of λ_{L}, are formalized using the usual, small-step (contextual) operational semantics, whose rules are shown in Figure 8.

Static Semantics. The *exact* dependent type derivation rules of λ_{L}, that yield judgements of the form $\Gamma \vdash e : S$, are shown in Figures 9 and 10. The rules for liquid type derivation, from Figure 3 are a conservative approximation of the exact rules (Theorem 5).

Contexts			$\boxed{C}$
v	$::=$		*Values:*
		$\mid \mathtt{c}$	constants
		$\mid \lambda x.e$	λ-terms
C	$::=$		*Contexts:*
		$\mid \bullet$	hole
		$\mid C\ e$	application left
		$\mid v\ C$	application right
		$\mid \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e$	if-then-else
		$\mid \texttt{let}\ x\ =\ C\ \texttt{in}\ e$	let-binding

Evaluation			$\boxed{e \hookrightarrow e'}$
$\mathtt{c}\ v$	$\hookrightarrow$	$[\![\mathtt{c}]\!](v)$	[E-Prim]
$(\lambda x.e)\ v$	$\hookrightarrow$	$[v/x]e$	[E-β]
$\texttt{if true then}\ e\ \texttt{else}\ e'$	$\hookrightarrow$	e	[E-If-True]
$\texttt{if false then}\ e\ \texttt{else}\ e'$	$\hookrightarrow$	e'	[E-If-False]
$\texttt{let}\ x\ =\ v\ \texttt{in}\ e$	$\hookrightarrow$	$[v/x]e$	[E-Let]
$C[e]$	$\hookrightarrow$	$C[e']$ if $e \hookrightarrow e'$	[E-Compat]

Figure 8: **Small-Step Operational Semantics**

D Correctness of Type Checking

Definition 1. *(Constants) Each constant* $\mathtt{c}$ *has a type* $ty(\mathtt{c})$ *such that:*

1. $\emptyset \vdash ty(\mathtt{c})$,
2. *If* $ty(\mathtt{c})$ *is* $x{:}T_1 \rightarrow T_2$ *then for all values* v *such that* $\emptyset \vdash v : T_1$, $[\![\mathtt{c}]\!](v)$ *is defined and,* $\emptyset \vdash [\![\mathtt{c}]\!](v) : [v/x]T_2$, *and,*
3. *If* $ty(\mathtt{c})$ *is* $\{\nu{:}B \mid e\}$ *then* $e \equiv \nu = \mathtt{c}$.

Definition 2. *(Embedding) We define* $[\![]\!]$ *to be a map from expressions and environments to formulas in a decidable logic such that for all* Γ, e_1, e_2, *if* $\Gamma \vdash e_1 : \texttt{bool}$, $\Gamma \vdash e_2 : \texttt{bool}$, $\mathsf{Valid}([\![\Gamma]\!] \wedge [\![e_1]\!] \Rightarrow [\![e_2]\!])$, *then* $\Gamma \vdash e_1 \Rightarrow e_2$.

Lemma 1. *(Free Variables)*

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

$$\frac{\Gamma; \nu{:}B \vdash e : \texttt{bool}}{\Gamma \vdash \{\nu{:}B \mid e\}}\ [\text{WT-Base}] \qquad \frac{}{\Gamma \vdash \alpha}\ [\text{WT-Var}]$$

$$\frac{\Gamma; x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \to T_2}\ [\text{WT-Fun}] \qquad \frac{\Gamma \vdash S \quad \alpha \notin \Gamma}{\Gamma \vdash \forall \alpha.S}\ [\text{WT-Poly}]$$

Well-Formed Environments $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ [\text{WE-Empty}] \quad \frac{\vdash \Gamma \quad \Gamma \vdash S}{\vdash \Gamma; x{:}S}\ [\text{WE-Ext}] \quad \frac{\vdash \Gamma \quad \Gamma \vdash e : \texttt{bool}}{\vdash \Gamma; e}\ [\text{WE-Gxt}]$$

Well-Formed Substitutions $\boxed{\Gamma \models \rho}$

$$\frac{}{\emptyset \models \emptyset}\ [\text{WS-Empty}] \quad \frac{\Gamma \models \rho \quad \emptyset \vdash v : \rho S}{\Gamma; x{:}S \models \rho; [x \mapsto v]}\ [\text{WS-Ext}] \quad \frac{\Gamma \models \rho \quad \rho e \overset{*}{\hookrightarrow} \texttt{true}}{\Gamma; e \models \rho}\ [\text{WS-Gxt}]$$

Figure 9: **Rules for Well-formed Dependent Types, Environments, Substitutions**

1. *If* $\Gamma \vdash S$ *then* $\mathsf{FreeVars}(S) \subseteq \mathsf{Dom}(\Gamma)$.
2. *If* $\Gamma \vdash e : S$ *then* $\mathsf{FreeVars}(e) \cup \mathsf{FreeVars}(S) \subseteq \mathsf{Dom}(\Gamma)$.
3. *If* $\Gamma \models \rho$ *then* $\mathsf{Dom}(\rho) = \mathsf{Dom}(\Gamma)$ *and* $\mathsf{FreeVars}(\mathsf{Rng}(\rho)) = \emptyset$.

Proof. Induction on the derivation of $\Gamma \vdash S$, $\Gamma \vdash e : S$, $\Gamma \models \rho$ respectively. □

Lemma 2. *(Well-Formedness) If* $\Gamma \vdash e : S$ *then* $\Gamma \vdash S$.

Proof. Induction on the derivation of $\Gamma \vdash e : S$. □

Lemma 3. *(Substitution Permutation) If* $\Gamma \models \rho_1; \rho_2$ *then:*

1. $\mathsf{Dom}(\rho_1) \cap \mathsf{Dom}(\rho_2) = \emptyset$,
2. *for all* e, $(\rho_1; \rho_2)e = (\rho_2; \rho_1)e$,
3. *for all* S, $(\rho_1; \rho_2)S = (\rho_2; \rho_1)S$.

Proof. (1) Induction on the derivation of $\Gamma \models \rho_1; \rho_2$ and the fact that a variable is bound at most once in Γ. (2) As $\Gamma \models \rho_1; \rho_2$, from Lemma 1 we have: $\mathsf{FreeVars}(\mathsf{Rng}(\rho_1)) = \mathsf{FreeVars}(\mathsf{Rng}(\rho_2)) = \emptyset$. The proof follows by induction on the structure of e using (1). (3) Induction on the structure of S using (2). □

Lemma 4. *(Well-formed Substitution)*

1. *If* $\Gamma \models \rho_1; \rho_2$ *then there are* Γ_1, Γ_2 *such that* $\Gamma \equiv \Gamma_1; \Gamma_2$, $\mathsf{Dom}(\rho_1) = \mathsf{Dom}(\Gamma_1)$, $\mathsf{Dom}(\rho_2) = \mathsf{Dom}(\Gamma_2)$.
2. *If* $\Gamma_1; \Gamma_2 \models \rho$ *then there are* ρ_1, ρ_2 *such that* $\rho \equiv \rho_1; \rho_2$, $\mathsf{Dom}(\rho_1) = \mathsf{Dom}(\Gamma_1)$, $\mathsf{Dom}(\rho_2) = \mathsf{Dom}(\Gamma_2)$.
3. $\Gamma_1; \Gamma_2 \models \rho_1; \rho_2$, $\mathsf{Dom}(\rho_1) = \mathsf{Dom}(\Gamma_1)$, $\mathsf{Dom}(\rho_2) = \mathsf{Dom}(\Gamma_2)$ *iff* $\Gamma_1 \models \rho_1$, $\rho_1\Gamma_2 \models \rho_2$.

Proof. We consider each case in turn.

1. By induction on Γ.

Dependent Type Checking $\boxed{\Gamma \vdash e : S}$

$$\frac{\Gamma \vdash e : S_1 \qquad \Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash S_2}{\Gamma \vdash e : S_2}\ [\textsc{T-Sub}]$$

$$\frac{\Gamma(x) = \{\nu{:}B \mid e\}}{\Gamma \vdash x : \{\nu{:}B \mid \nu = x\}}\ [\textsc{T-Var-Base}] \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash x : \Gamma(x)}\ [\textsc{T-Var}]$$

$$\frac{}{\Gamma \vdash \mathtt{c} : ty(\mathtt{c})}\ [\textsc{T-Const}]$$

$$\frac{\Gamma \vdash T \quad \Gamma; x{:}T \vdash e_1 : T_1}{\Gamma \vdash (\lambda x.e_1) : (x{:}T \to T_1)}\ [\textsc{T-Fun}]$$

$$\frac{\Gamma \vdash e_1 : (x{:}T_2 \to T) \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : [e_2/x]T}\ [\textsc{T-App}]$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \qquad \Gamma; e_1 \vdash e_2 : S \qquad \Gamma; \neg e_1 \vdash e_3 : S}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : S}\ [\textsc{T-If}]$$

$$\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma; x{:}S_1 \vdash e_2 : S_2 \quad \Gamma \vdash S_2}{\Gamma \vdash \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : S_2}\ [\textsc{T-Let}]$$

$$\frac{\Gamma \vdash e : S \qquad \alpha \notin \Gamma}{\Gamma \vdash [\Lambda\alpha]e : \forall\alpha.S}\ [\textsc{T-Gen}] \quad \frac{\Gamma \vdash e : \forall\alpha.S \qquad \Gamma \vdash T \quad \mathsf{Shape}(T) = \tau}{\Gamma \vdash [\tau]e : [T/\alpha]S}\ [\textsc{T-Inst}]$$

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

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : \mathtt{bool} \quad \forall\rho.(\Gamma \models \rho \text{ and } \rho e_1 \overset{*}{\hookrightarrow} \mathtt{true} \text{ implies } \rho e_2 \overset{*}{\hookrightarrow} \mathtt{true}}{\Gamma \vdash e_1 \Rightarrow e_2}\ [\textsc{Imp}]$$

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

$$\frac{\Gamma; \nu{:}B \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{\nu{:}B \mid e_1\} <: \{\nu{:}B \mid e_2\}}\ [\textsc{<:-Base}]$$

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

$$\frac{}{\Gamma \vdash \alpha <: \alpha}\ [\textsc{<:-Var}] \quad \frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash \forall\alpha.S_1 <: \forall\alpha.S_2}\ [\textsc{<:-Poly}]$$

Figure 10: **Rules for Dependent Type Checking**

2. By induction on ρ_2.

3. By induction on Γ_2.

4. By induction on Γ_2.

   - case $\Gamma_2 \equiv \emptyset$: Trivial, as $\mathsf{Dom}(\rho_2) \subseteq \emptyset$, *i.e.*,, $\rho_2 \equiv \emptyset$.
   - case $\Gamma_2 \equiv \Gamma_2'; e$:

$$\begin{aligned}
\Gamma_1; \Gamma_2'; e \models \rho_1; \rho_2; \iff & \Gamma_1; \Gamma_2'; \models \rho_1; \rho_2,\ (\rho_1; \rho_2)e \overset{*}{\hookrightarrow} \texttt{true} \\
& \text{(by rule [WS-G}\textsc{xt}\text{])} \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2' \models \rho_2,\ (\rho_2; \rho_1)e \overset{*}{\hookrightarrow} \texttt{true} \\
& \text{(by IH, Lemma 3)} \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2'; \rho_1 e \models \rho_2 \\
& \text{(by rule [WS-G}\textsc{xt}\text{])} \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2 \models \rho_2
\end{aligned}$$

   - case $\Gamma_2 \equiv \Gamma_2'; x\!:\!S$:

$$\begin{aligned}
\Gamma_1; \Gamma_2'; x\!:\!S \models \rho_1; \rho_2; \iff & \Gamma_1; \Gamma_2'; \models \rho_1; \rho_2',\ \emptyset \models v : (\rho_1; \rho_2')S,\ \rho_2 \equiv \rho_2'; [x \mapsto v] \\
& \text{(by rule [WS-E}\textsc{xt}\text{])} \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2' \models \rho_2',\ \emptyset \models v : (\rho_2'; \rho_1)S,\ \rho_2 \equiv \rho_2'; [x \mapsto v] \\
& \text{(by IH, Lemma 3, rule [WS-E}\textsc{xt}\text{])} \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2'; x\!:\!\rho_1 S \models \rho_2 \\
\iff & \Gamma_1 \models \rho_1,\ \rho_1 \Gamma_2 \models \rho_2
\end{aligned}$$

□

Corollary 1. *(Well-formed Substitution)*

1. $\Gamma_1; x\!:\!S_x; \Gamma_2 \models \rho_1; [x \mapsto v_x]; \rho_2 \iff \Gamma_1 \models \rho_1,\ \emptyset \vdash v_x : \rho_1 S_x,\ (\rho_1; [x \mapsto v_x])\Gamma_2 \models \rho_2,$
2. *If* $\Gamma \models \rho$ *and* $x\!:\!S \in \Gamma$ *then* $\emptyset \models \rho(x) : \rho S$.

Proof. Corollary of Lemma 4. □

Lemma 5. *(Weakening) If*

$$\begin{aligned}
\Gamma &= \Gamma_1; \Gamma_2 \\
\Gamma' &= \Gamma_1; x\!:\!S_x; \Gamma_2 \\
x &\notin \mathsf{FreeVars}(\Gamma_2)
\end{aligned}$$

then:

1. *if* $\Gamma' \models \rho_1; [x \mapsto v]; \rho_2$ *then* $\Gamma \models \rho_1; \rho_2$,
2. *if* $\Gamma \vdash e_1 \Rightarrow e_2$ *then* $\Gamma' \vdash e_1 \Rightarrow e_2$,
3. *if* $\Gamma \vdash S_1 <: S_2$ *then* $\Gamma' \vdash S_1 <: S_2$,
4. *if* $\Gamma \vdash S$ *then* $\Gamma' \vdash S$,
5. *if* $\Gamma \vdash e : S$ *then* $\Gamma' \vdash e : S$.

Proof. By simultaneous induction on the derivations. Without loss of generality, assume that all type variables that are generalized under an environment with prefix Γ are suitably α-renamed to variables different from the *free* type variables in S_x.

1. Assume

$$\Gamma' \models \rho_1; [x \mapsto v]; \rho_2$$

Using the definition of Γ' and Corollary 1, we have

$$\Gamma_1 \models \rho_1,\ \emptyset \vdash v : \rho_1 S_x,\ (\rho_1; [x \mapsto v])\Gamma_2 \models \rho_2$$

As $x \notin \mathsf{FreeVars}(\Gamma_2)$ we have $(\rho_1; [x \mapsto v])\Gamma_2 \equiv \Gamma_2$, thus

$$\Gamma_1 \models \rho_1, \emptyset \vdash v : \rho_1 S_x,\ \rho_1\Gamma_2 \models \rho_2$$

By Corollary 1,

$$\Gamma_1; \Gamma_2 \models \rho_1; \rho_2$$

Finally, by the definition of Γ

$$\Gamma \models \rho_1; \rho_2$$

2. Assume

$$\Gamma \vdash e_1 \Rightarrow e_2 \tag{a}$$

From the rule [Imp], we have

$$\Gamma \vdash e_1 : \texttt{bool},\ \Gamma \vdash e_2 : \texttt{bool}, \tag{b}$$
$$\forall \rho. \Gamma \models \rho \text{ and } \rho e_1 \overset{*}{\hookrightarrow} \texttt{true} \text{ implies } \rho e_2 \overset{*}{\hookrightarrow} \texttt{true} \tag{c}$$

By induction (4.) we conclude that

$$\Gamma' \vdash e_1 : \texttt{bool},\ \Gamma' \vdash e_2 : \texttt{bool} \tag{d}$$

Next, consider any ρ' such that $\Gamma' \models \rho'$ and $\rho' e_1 \overset{*}{\hookrightarrow} \texttt{true}$. We shall prove that $\rho' e_2 \overset{*}{\hookrightarrow} \texttt{true}$. By Lemma 4, $\rho' \equiv \rho_1; [x \mapsto v_x]; \rho_2$ where $\mathsf{Dom}(\rho_1) = \mathsf{Dom}(\Gamma_1)$ and $\mathsf{Dom}(\rho_2) = \mathsf{Dom}(\Gamma_2)$. Consider $\rho \equiv \rho_1; \rho_2$. From (b) and Lemma 1, $x \notin \mathsf{FreeVars}(e_1) \cup \mathsf{FreeVars}(e_2)$, and so

$$\rho e_1 = \rho' e_1,\ \rho e_2 = \rho' e_2 \tag{e}$$

applying (c) completes the proof.

3. By induction on the derivation of $\Gamma \vdash S_1 <: S_2$ and (2.).

   - case $S_1, S_2 \equiv \alpha, \alpha$: Trivial.
   - case $S_1, S_2 \equiv \{\nu : B \mid e_1\}, \{\nu : B \mid e_2\}$: Assume

$$\Gamma \vdash S_1 <: S_2$$

   By rule [<:-Base]

$$\Gamma; \nu : B \vdash e_1 \Rightarrow e_2$$

By induction (2.)

$$\Gamma'; \nu\!:\!B \vdash e_1 \Rightarrow e_2$$

By rule [<:-Base]

$$\Gamma' \vdash S_1 <: S_2$$

- case $S_1, S_2 \equiv y\!:\!T_1 \to T_1', y\!:\!T_2 \to T_2'$: Assume

  $$\Gamma \vdash S_1 <: S_2$$

  By rule [<:-Fun]

  $$\Gamma \vdash y\!:\!T_2 <: y\!:\!T_1,\ \Gamma; y\!:\!T_1 \vdash T_1' <: T_2'$$

  By induction

  $$\Gamma' \vdash y\!:\!T_2 <: y\!:\!T_1,\ \Gamma'; y\!:\!T_1 \vdash T_1' <: T_2'$$

  By rule [<:-Fun]

  $$\Gamma' \vdash S_1 <: S_2$$

- case $S_1, S_2 \equiv \forall\alpha.S_1', \forall\alpha.S_2'$: Assume

  $$\Gamma \vdash S_1 <: S_2$$

  By rule [<:-Poly]

  $$\Gamma \vdash S_1' <: S_2'$$

  By induction

  $$\Gamma' \vdash S_1' <: S_2'$$

  By rule [<:-Poly]

  $$\Gamma' \vdash S_1 <: S_2$$

4. By induction on the derivation $\Gamma \vdash S$. We split cases on the structure of S. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

   - case $S \equiv \{\nu\!:\!B \mid e\}$: Assume,

     $$\Gamma \vdash S$$

     By inversion (rule [WT-Base]

     $$\Gamma; \nu\!:\!B \vdash e : \texttt{bool}$$

     By IH (5.)

     $$\Gamma'; \nu\!:\!B \vdash e : \texttt{bool}$$

     By rule [WT-Base]

     $$\Gamma' \vdash S$$

- case $S \equiv \alpha$: Trivial.
- case $S \equiv y\!:\!T \rightarrow T'$: Assume,

  $$\Gamma \vdash S$$

  By inversion (rule [WT-Fun])

  $$\Gamma \vdash T,\ \Gamma; y\!:\!T \vdash T'$$

  By IH

  $$\Gamma' \vdash T,\ \Gamma'; y\!:\!T \vdash T'$$

  By rule [WT-Fun]

  $$\Gamma' \vdash S$$

- case $S \equiv \forall\alpha.T'$: Where due to renaming, α not free in S_x. Assume,

  $$\Gamma \vdash S$$

  By inversion (rule [WT-Poly])

  $$\Gamma \vdash S',\ \alpha \notin \Gamma$$

  By IH

  $$\Gamma' \vdash S',\ \alpha \notin \Gamma'$$

  By rule [WT-Poly]

  $$\Gamma' \vdash S$$

5. By induction on the derivation $\Gamma \vdash e : S$. We split cases on the rule used at the root of the derivation. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

   - case [T-Sub]: Assume,

     $$\Gamma \vdash e : S$$

     By inversion

     $$\Gamma \vdash e : S',\ \Gamma \vdash S' <: S,\ \Gamma \vdash S$$

     For some S'. By IH, (3.) and (4.)

     $$\Gamma' \vdash e : S',\ \Gamma' \vdash S' <: S,\ \Gamma' \vdash S$$

     By rule [T-Sub]

     $$\Gamma' \vdash e : S$$

- case [T-Var-Base]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv y$ for some variable y and $S \equiv \{\nu : B \mid \nu = y\}$. By inversion

$$\Gamma(y) = \{\nu : B \mid e_y\}$$

By Lemma 1, $y \in \mathsf{Dom}(\Gamma)$, *i.e.,* y is different from x. Thus, we have $\Gamma'(y) = \Gamma(y)$, and hence

$$\Gamma'(y) = \{\nu : B \mid e_y\}$$

By rule [T-Var-Base]

$$\Gamma' \vdash e : S$$

- case [T-Var]: Similar to case [T-Var-Base].
- case [T-Const]: Trivial.
- case [T-Fun]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv \lambda y.e_1$, and $S \equiv y : T \to T_1$. By inversion

$$\Gamma \vdash T,\ \Gamma; y : T \vdash e_1 : T_1$$

By IH, (4.)

$$\Gamma' \vdash T,\ \Gamma'; y : T \vdash e_1 : T_1$$

By rule [T-Fun]

$$\Gamma' \vdash e : S$$

- case [T-App]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv e_1 e_2$, and $S \equiv [e_2/y]T$ for some T. By inversion

$$\Gamma \vdash e_1 : y : T_2 \to T,\ \Gamma \vdash e_2 : T_2$$

By IH

$$\Gamma' \vdash e_1 : y : T_2 \to T,\ \Gamma' \vdash e_2 : T_2$$

By rule [T-App]

$$\Gamma' \vdash e : S$$

- case [T-IF]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. By inversion

$$\Gamma \vdash e_1 : \texttt{bool},\ \Gamma; e_1 \vdash e_2 : S,\ \Gamma; \neg e_1 \vdash e_3 : S$$

By IH

$$\Gamma' \vdash e_1 : \texttt{bool},\ \Gamma'; e_1 \vdash e_2 : S,\ \Gamma'; \neg e_1 \vdash e_3 : S$$

By rule [T-IF]

$$\Gamma' \vdash e : S$$

- case [T-LET]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv \texttt{let}\ y\ =\ e_1\ \texttt{in}\ e_2$. By inversion

$$\Gamma \vdash e_1 : S_y,\ \Gamma; y{:}S_y \vdash e_2 : S,\ \Gamma \vdash S$$

By IH, (4.)

$$\Gamma' \vdash e_1 : S_y,\ \Gamma'; y{:}S_y \vdash e_2 : S$$
$$\Gamma' \vdash S$$

By rule [T-LET]

$$\Gamma' \vdash e : S$$

- case [T-GEN]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv [\Lambda\alpha]e_1$ and $S \equiv \forall\alpha.S_1$. By inversion

$$\Gamma \vdash e_1 : S_1,\ \alpha \notin \Gamma$$

By IH and α not in free variables of S_x,

$$\Gamma' \vdash e_1 : S_1,\ \alpha \notin \Gamma'$$

By rule [T-GEN]

$$\Gamma' \vdash e : S$$

- case [T-INST]: Assume,

$$\Gamma \vdash e : S$$

Where $e \equiv [\tau]e_1$, $S \equiv [T/\alpha]S_1$ and $\mathsf{Shape}(T) = \tau$. By inversion

$$\Gamma \vdash e_1 : \forall\alpha.S_1,\ \Gamma \vdash T$$

By IH, (4.)

$$\Gamma' \vdash e_1 : \forall\alpha.S_1,\ \Gamma' \vdash T$$

By rule [T-Inst]

$$\Gamma' \vdash e : S$$

□

Lemma 6. *(Guard Weakening) If*

$$\Gamma = \Gamma_1; \Gamma_2$$
$$\Gamma' = \Gamma_1; e; \Gamma_2$$

then,

1. *if* $\Gamma' \models \rho$ *then* $\Gamma \models \rho$,
2. *if* $\Gamma \vdash e_1 \Rightarrow e_2$ *then* $\Gamma' \vdash e_1 \Rightarrow e_2$,
3. *if* $\Gamma \vdash S_1 <: S_2$ *then* $\Gamma' \vdash S_1 <: S_2$,
4. *if* $\Gamma \vdash S$ *then* $\Gamma' \vdash S$,
5. *if* $\Gamma \vdash e : S$ *then* $\Gamma' \vdash e : S$.

Proof. By simultaneous induction on the typing derivations, similar to proof of Lemma 5. □

Lemma 7. *(True Guard) If*

$$\Gamma = \Gamma_1; \texttt{true}; \Gamma_2$$
$$\Gamma' = \Gamma_1; \Gamma_2$$

then,

1. $\Gamma \models \rho$ *iff* $\Gamma' \models \rho$,
2. $\Gamma \vdash e_1 \Rightarrow e_2$ *iff* $\Gamma' \vdash e_1 \Rightarrow e_2$,
3. $\Gamma \vdash S_1 <: S_2$ *iff* $\Gamma' \vdash S_1 <: S_2$,
4. $\Gamma \vdash S$ *iff* $\Gamma' \vdash S$,
5. $\Gamma \vdash e : S$ *iff* $\Gamma' \vdash e : S$.

Proof. The proof follows by using (a) the fact that as `true` is a constant, for any ρ, we have $\rho\ \texttt{true} \overset{*}{\hookrightarrow} \texttt{true}$ and (b) simultaneous induction on the typing derivations, similar to Lemma 5. □

Lemma 8. *(Narrowing) If*

$$\Gamma_1 \vdash S'_x$$
$$\Gamma_1 \vdash S_x <: S'_x$$
$$\Gamma = \Gamma_1; x{:}S_x; \Gamma_2$$
$$\Gamma' = \Gamma_1; x{:}S'_x; \Gamma_2$$

then:

1. if $\Gamma \models \rho$ *then* $\Gamma' \models \rho$*,*

2. if $\Gamma' \vdash e_1 \Rightarrow e_2$ *then* $\Gamma \vdash e_1 \Rightarrow e_2$*,*

3. if $\Gamma' \vdash S_1 <: S_2$ *then* $\Gamma \vdash S_1 <: S_2$*,*

4. if $\Gamma' \vdash S$ *then* $\Gamma \vdash S$*,*

5. if $\Gamma' \vdash e : S$ *then* $\Gamma \vdash e : S$*.*

Proof. By simultaneous induction on the derivations. Without loss of generality, assume that all type variables that are generalized under an environment with prefix Γ or Γ' are suitably α-renamed to variables different from the *free* type variables in S_x and S'_x.

1. Assume

$$\Gamma \models \rho$$

That is,

$$\Gamma_1; x\!:\!S_x; \Gamma_2 \models \rho$$

By Lemma 4, $\rho \equiv \rho_1; [x \mapsto v_x]; \rho_2$ where $\mathsf{Dom}(\rho_1) = \mathsf{Dom}(\Gamma_1)$ and $\mathsf{Dom}(\rho_2) = \mathsf{Dom}(\Gamma_2)$ and by Cor. 1

$$\Gamma_1 \models \rho_1,\ \emptyset \vdash v_x : \rho_1 S_x,\ (\rho_1; [x \mapsto v_x])\Gamma_2 \models \rho_2 \tag{a}$$

As $\Gamma_1 \vdash S_x <: S'_x$, Lemma 10 yields

$$\emptyset \vdash \rho_1 S_x <: \rho_1 S'_x \tag{b}$$

Combining (a), (b) with rule [T-Sub] yields

$$\Gamma_1 \models \rho_1,\ \emptyset \vdash v_x : \rho_1 S'_x,\ (\rho_1; [x \mapsto v_x])\Gamma_2 \models \rho_2$$

Finally, using Cor. 1

$$\Gamma' \models \rho$$

2. Assume

$$\Gamma' \vdash e_1 \Rightarrow e_2$$

By rule [Imp]

$$\Gamma' \vdash e_1 : \mathtt{bool},\ \Gamma' \vdash e_2 : \mathtt{bool},\ \forall \rho \text{s.t.} \Gamma' \models \rho,\ \text{if } \rho e_1 \overset{*}{\hookrightarrow} \mathtt{true} \text{ then } \rho e_2 \overset{*}{\hookrightarrow} \mathtt{true} \tag{a}$$

By IH (5.),

$$\Gamma \vdash e_1 : \mathtt{bool},\ \Gamma \vdash e_2 : \mathtt{bool} \tag{b}$$

Consider an arbitrary ρ s.t. $\Gamma \models \rho$. By IH (1.), $\Gamma' \models \rho$ and so by (a) if $\rho e_1 \overset{*}{\hookrightarrow} \mathtt{true}$ then $\rho e_2 \overset{*}{\hookrightarrow} \mathtt{true}$. Hence

$$\forall \rho \text{s.t.} \Gamma \models \rho,\ \text{if } \rho e_1 \overset{*}{\hookrightarrow} \mathtt{true} \text{ then } \rho e_2 \overset{*}{\hookrightarrow} \mathtt{true} \tag{c}$$

Using (b),(c) and rule [Imp]

$$\Gamma' \vdash e_1 \Rightarrow e_2$$

3. We split cases on the structure of S_1, S_2.
   - case $S_1, S_2 \equiv \alpha, \alpha$: Trivial.
   - case $S_1, S_2 \equiv \{\nu{:}B \mid e_1\}, \{\nu{:}B \mid e_2\}$: Assume
$$\Gamma' \vdash S_1 <: S_2$$
By inversion
$$\Gamma'; \nu{:}B \vdash e_1 \Rightarrow e_2$$
By IH (2.)
$$\Gamma; \nu{:}B \vdash e_1 \Rightarrow e_2$$
By rule [<:-Base]
$$\Gamma \vdash S_1 <: S_2$$
   - case $S_1, S_2 \equiv y{:}T_1 \to T_1', y{:}T_2 \to T_2'$: Assume
$$\Gamma' \vdash S_1 <: S_2$$
By inversion
$$\Gamma' \vdash T_2 <: T_1,\ \Gamma'; y{:}T_2 \vdash T_1' <: T_2'$$
By IH
$$\Gamma \vdash T_2 <: T_1,\ \Gamma; y{:}T_2 \vdash T_1' <: T_2'$$
By rule [<:-Fun]
$$\Gamma \vdash S_1 <: S_2$$
   - case $S_1, S_2 \equiv \forall\alpha.S_1', \forall\alpha.S_2'$: Assume
$$\Gamma' \vdash S_1 <: S_2$$
By inversion
$$\Gamma' \vdash S_1' <: S_2'$$
By IH
$$\Gamma \vdash S_1' <: S_2'$$
By rule [<:-Poly]
$$\Gamma \vdash S_1 <: S_2$$

4. By induction on the derivation $\Gamma' \vdash S$. We split cases on the structure of S. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

- case $S \equiv \{\nu\!:\!B \mid e\}$: Assume,

  $$\Gamma' \vdash S$$

  By inversion (rule [WT-Base]

  $$\Gamma'; \nu\!:\!B \vdash e : \texttt{bool}$$

  By IH (5.)

  $$\Gamma; \nu\!:\!B \vdash e : \texttt{bool}$$

  By rule [WT-Base]

  $$\Gamma \vdash S$$

- case $S \equiv \alpha$: Trivial.
- case $S \equiv y\!:\!T \to T'$: Assume,

  $$\Gamma' \vdash S$$

  By inversion (rule [WT-Fun])

  $$\Gamma' \vdash T,\ \Gamma'; y\!:\!T \vdash T'$$

  By IH

  $$\Gamma \vdash T,\ \Gamma; y\!:\!T \vdash T'$$

  By rule [WT-Fun]

  $$\Gamma \vdash S$$

- case $S \equiv \forall\alpha.T'$: Assume,

  $$\Gamma' \vdash S$$

  By inversion (rule [WT-Poly])

  $$\Gamma' \vdash S',\ \alpha \notin \Gamma'$$

  By IH and as due to renaming, α not free in S_x or S'_x

  $$\Gamma \vdash S',\ \alpha \notin \Gamma$$

  By rule [WT-Poly]

  $$\Gamma \vdash S$$

5. By induction on the derivation $\Gamma' \vdash e : S$. We split cases on the rule used at the root of the derivation. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

- case [T-Sub]: Assume,

$$\Gamma' \vdash e : S$$

By inversion

$$\Gamma' \vdash e : S',\ \Gamma' \vdash S' <: S,\ \Gamma' \vdash S$$

For some S'. By IH, (3.) and (4.)

$$\Gamma \vdash e : S',\ \Gamma \vdash S' <: S,\ \Gamma \vdash S$$

By rule [T-Sub]

$$\Gamma \vdash e : S$$

- case [T-Var-Base]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv y$ for some variable y and $S \equiv \{\nu : B \mid \nu = y\}$. By inversion

$$\Gamma'(y) = \{\nu : B \mid e'_y\}$$

As $\Gamma_1 \vdash S_x <: S'_x$, we have $\Gamma(y) = \{\nu : B \mid e_y\}$, and hence by rule [T-Var-Base]

$$\Gamma \vdash e : S$$

- case [T-Var]: Similar to case [T-Var-Base].
- case [T-Const]: Trivial.
- case [T-Fun]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv \lambda y.e_1$, and $S \equiv y : T \to T_1$. By inversion

$$\Gamma' \vdash T,\ \Gamma'; y : T \vdash e_1 : T_1$$

By IH, (4.)

$$\Gamma \vdash T,\ \Gamma; y : T \vdash e_1 : T_1$$

By rule [T-Fun]

$$\Gamma \vdash e : S$$

- case [T-App]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv e_1 e_2$, and $S \equiv [e_2/y]T$ for some T. By inversion

$$\Gamma' \vdash e_1 : y : T_2 \to T,\ \Gamma' \vdash e_2 : T_2$$

By IH

$$\Gamma \vdash e_1 : y : T_2 \to T,\ \Gamma \vdash e_2 : T_2$$

By rule [T-App]

$$\Gamma \vdash e : S$$

- case [T-If]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. By inversion

$$\Gamma' \vdash e_1 : \texttt{bool},\ \Gamma'; e_1 \vdash e_2 : S,\ \Gamma'; \neg e_1 \vdash e_3 : S$$

By IH

$$\Gamma \vdash e_1 : \texttt{bool},\ \Gamma; e_1 \vdash e_2 : S,\ \Gamma; \neg e_1 \vdash e_3 : S$$

By rule [T-If]

$$\Gamma \vdash e : S$$

- case [T-Let]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv \texttt{let}\ y\ =\ e_1\ \texttt{in}\ e_2$. By inversion

$$\Gamma' \vdash e_1 : S_y,\ \Gamma'; y\!:\!S_y \vdash e_2 : S,\ \Gamma' \vdash S$$

By IH, (4.)

$$\Gamma \vdash e_1 : S_y,\ \Gamma; y\!:\!S_y \vdash e_2 : S,\ \Gamma \vdash S$$

By rule [T-Let]

$$\Gamma \vdash e : S$$

- case [T-Gen]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv [\Lambda\alpha]e_1$ and $S \equiv \forall\alpha.S_1$. By inversion

$$\Gamma' \vdash e_1 : S_1,\ \alpha \notin \Gamma'$$

By IH and as α not in free variables of S_x or S'_x,

$$\Gamma \vdash e_1 : S_1,\ \alpha \notin \Gamma$$

By rule [T-Gen]

$$\Gamma \vdash e : S$$

- case [T-Inst]: Assume,

$$\Gamma' \vdash e : S$$

Where $e \equiv [\tau]e_1$, $S \equiv [T/\alpha]S_1$ and $\mathsf{Shape}(T) = \tau$. By inversion

$$\Gamma' \vdash e_1 : \forall\alpha.S_1,\ \Gamma' \vdash T$$

By IH, (4.)

$$\Gamma \vdash e_1 : \forall\alpha.S_1,\ \Gamma \vdash T$$

By rule [T-Inst]

$$\Gamma \vdash e : S$$

□

Lemma 9. *(Subtyping Reflexive Transitive)*

1. *if* $\Gamma \vdash S$ *then* $\Gamma \vdash S <: S$*,*
2. *if* $\Gamma \vdash e_1 \Rightarrow e_2$ *and* $\Gamma \vdash e_2 \Rightarrow e_3$ *then* $\Gamma \vdash e_1 \Rightarrow e_3$*,*
3. *if* $\Gamma \vdash S_1 <: S_2$ *and* $\Gamma \vdash S_2 <: S_3$ *then* $\Gamma \vdash S_1 <: S_3$*.*

Proof. By induction on the typing derivations, using Lemmas 5 and 8.

1. By induction on the derivation $\Gamma \vdash S$.
2. Transitivity of implication.
3. By lexicographic induction on the structure of S_1, S_2, S_3 and the derivations $\Gamma \vdash S_1 <: S_2$ and $\Gamma \vdash S_2 <: S_3$.
   - case $S_1, S_2, S_3 \equiv \alpha, \alpha, \alpha$: Trivial.
   - case $S_1, S_2, S_3 \equiv \{\nu : B \mid e_1\}, \{\nu : B \mid e_2\}, \{\nu : B \mid e_3\}$: Assume

     $$\Gamma \vdash S_1 <: S_2,\ \Gamma \vdash S_2 <: S_3$$

     By inversion (rule [<:-Base])

     $$\Gamma; \nu : B \vdash e_1 \Rightarrow e_2,\ \Gamma; \nu : B \vdash e_2 \Rightarrow e_3$$

     By (2.)

     $$\Gamma; \nu : B \vdash e_1 \Rightarrow e_3$$

     By rule [<:-Base]

   - case $S_1, S_2, S_3 \equiv x : T_1 \to T_1', x : T_2 \to T_2', x : T_3 \to T_3'$: Assume

     $$\Gamma \vdash S_1 <: S_2,\ \Gamma \vdash S_2 <: S_3$$

     By inversion (rule [<:-Fun])

     $$\Gamma \vdash T_2 <: T_1,\ \Gamma \vdash T_3 <: T_2 \quad \text{(a)}$$
     $$\Gamma; x : T_2 \vdash T_1' <: T_2' \quad \text{(b)}$$
     $$\Gamma; x : T_3 \vdash T_2' <: T_3' \quad \text{(c)}$$

     By IH on (a)

     $$\Gamma \vdash T_3 <: T_1 \quad \text{(d)}$$

By (a),(b) and Lemma 8

$$\Gamma; x\!:\!T_3 \vdash T_1' <: T_2' \tag{e}$$

By IH on (e), (c)

$$\Gamma; x\!:\!T_3 \vdash T_1' <: T_3' \tag{f}$$

Finally, using rule [<:-Fun] on (d),(f)

$$\Gamma \vdash S_1 <: S_3$$

- case $S_1, S_2, S_3 \equiv \forall\alpha.S_1', \forall\alpha.S_2', \forall\alpha.S_3'$: Assume

  $$\Gamma \vdash S_1 <: S_2,\ \Gamma \vdash S_2 <: S_3$$

  By inversion (rule [<:-Poly])

  $$\Gamma \vdash S_1' <: S_2',\ \Gamma \vdash S_2' <: S_3'$$

  By IH

  $$\Gamma \vdash S_1' <: S_3'$$

  By rule [<:-Poly]

  $$\Gamma \vdash S_1 <: S_3$$

□

Lemma 10. *(Value Substitution) If* $\Gamma \models \rho$ *then*

1. *If* $\Gamma; \Gamma' \models \rho; \rho'$ *then* $\rho\Gamma' \models \rho'$,
2. *If* $\Gamma; \Gamma' \vdash e_1 \Rightarrow e_2$ *then* $\rho\Gamma' \vdash \rho e_1 \Rightarrow \rho e_2$,
3. *If* $\Gamma; \Gamma' \vdash S_1 <: S_2$ *then* $\rho\Gamma' \vdash \rho S_1 <: \rho S_2$,
4. *If* $\Gamma; \Gamma' \vdash S$ *then* $\rho\Gamma' \vdash \rho S$,
5. *If* $\Gamma; \Gamma' \vdash e : S$ *then* $\rho\Gamma' \vdash \rho e : \rho S$,

Proof. By simultaneous induction on the derivations.

1. Assume

   $$\Gamma \models \rho,\ \Gamma; \Gamma' \models \rho; \rho'$$

   By Lemma 1, $\mathsf{Dom}(\Gamma) = \mathsf{Dom}(\rho)$ and $\mathsf{Dom}(\Gamma') = \mathsf{Dom}(\rho')$. Hence, by Lemma 4

   $$\rho\Gamma \models \rho'$$

2. Assume

$$\Gamma \models \rho,\ \Gamma; \Gamma' \models \rho; \rho'$$

By inversion (rule [Imp])

$$\Gamma; \Gamma' \vdash e_1 : \texttt{bool},\ \Gamma; \Gamma' \vdash e_2 : \texttt{bool} \tag{a}$$

$$\forall \rho, \rho'.\ \text{if } \Gamma; \Gamma' \models \rho; \rho' \text{ and } (\rho; \rho')e_1 \overset{*}{\hookrightarrow} \texttt{true} \text{ then } (\rho; \rho')e_2 \overset{*}{\hookrightarrow} \texttt{true} \tag{b}$$

By IH (5.), (a) yields

$$\rho\Gamma' \vdash \rho e_1 : \texttt{bool},\ \rho\Gamma' \vdash \rho e_2 : \texttt{bool} \tag{c}$$

By Lemma 3, (b) yields

$$\forall \rho'.\ \text{if } \Gamma; \Gamma' \models \rho; \rho' \text{ and } \rho'(\rho e_1) \overset{*}{\hookrightarrow} \texttt{true} \text{ then } \rho'(\rho e_2) \overset{*}{\hookrightarrow} \texttt{true} \tag{d}$$

By (1.), (d) becomes

$$\forall \rho'.\ \text{if } \rho\Gamma' \models \rho' \text{ and } \rho'(\rho e_1) \overset{*}{\hookrightarrow} \texttt{true} \text{ then } \rho'(\rho e_2) \overset{*}{\hookrightarrow} \texttt{true} \tag{e}$$

Which combined with (c) and rule [Imp] yields

$$\rho\Gamma' \vdash \rho e_1 \Rightarrow \rho e_2$$

3. We split cases on the structure of S_1, S_2.

   - case $S_1, S_2 \equiv \alpha, \alpha$: Trivial.
   - case $S_1, S_2 \equiv \{\nu : B \mid e_1\}, \{\nu : B \mid e_2\}$: Assume

     $$\Gamma; \Gamma' \vdash S_1 <: S_2$$

     By inversion (rule [<:-Base])

     $$\Gamma; \Gamma'; \nu : B \vdash e_1 \Rightarrow e_2$$

     By IH (2.), and as $\rho B \equiv B$

     $$\rho\Gamma'; \nu : B \vdash \rho e_1 \Rightarrow \rho e_2$$

     By rule [<:-Base]

     $$\rho\Gamma' \vdash \rho S_1 <: \rho S_2$$

   - case $S_1, S_2 \equiv y : T_1 \to T_1', y : T_2 \to T_2'$: Assume

     $$\Gamma; \Gamma' \vdash S_1 <: S_2$$

     By inversion (rule [<:-Fun])

     $$\Gamma; \Gamma' \vdash T_2 <: T_1,\ \Gamma; \Gamma'; y : T_2 \vdash T_1' <: T_2'$$

     By IH

     $$\rho\Gamma' \vdash \rho T_2 <: \rho T_1,\ \rho\Gamma'; y : \rho T_2 \vdash \rho T_1' <: \rho T_2'$$

     By rule [<:-Fun]

     $$\rho\Gamma' \vdash \rho S_1 <: \rho S_2$$

- case $S_1, S_2 \equiv \forall\alpha.S'_1, \forall\alpha.S'_2$: Assume

  $$\Gamma; \Gamma' \vdash S_1 <: S_2$$

  By inversion (rule [<:-Poly])

  $$\Gamma; \Gamma' \vdash S'_1 <: S'_2$$

  By IH

  $$\rho\Gamma' \vdash \rho S'_1 <: \rho S'_2$$

  By rule [<:-Poly]

  $$\rho\Gamma' \vdash S_1 <: S_2$$

4. We split cases on the structure of S. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

   - case $S \equiv \{\nu : B \mid e\}$: Assume,

     $$\Gamma; \Gamma' \vdash S$$

     By inversion (rule [WT-Base]

     $$\Gamma; \Gamma'; \nu : B \vdash e : \texttt{bool}$$

     By IH (5.)

     $$\rho(\Gamma'; \nu : B) \vdash \rho e : \texttt{bool}$$

     As $\rho(B) = B$

     $$\rho\Gamma'; \nu : B \vdash \rho e : \texttt{bool}$$

     By rule [WT-Base]

     $$\rho\Gamma' \vdash \rho S$$

   - case $S \equiv \alpha$: Trivial.
   - case $S \equiv y : T \to T'$: Assume,

     $$\Gamma; \Gamma' \vdash S$$

     By inversion (rule [WT-Fun])

     $$\Gamma; \Gamma' \vdash T,\ \ \Gamma; \Gamma'; y : T \vdash T'$$

     By IH

     $$\rho\Gamma' \vdash \rho T,\ \ \rho(\Gamma'; y : T) \vdash \rho T'$$

     Pushing the substitution inside

     $$\rho\Gamma' \vdash \rho T,\ \ \rho\Gamma'; y : \rho T) \vdash \rho T'$$

     By rule [WT-Fun]

     $$\rho\Gamma' \vdash \rho S$$

- case $S \equiv \forall\alpha.T'$: Assume,

  $$\Gamma;\Gamma' \vdash S$$

  By inversion (rule [WT-Poly])

  $$\Gamma;\Gamma' \vdash S',\ \alpha \notin \Gamma;\Gamma'$$

  By IH and as ρ is constant substitution, α not in $\mathsf{Rng}(\rho)$

  $$\rho\Gamma' \vdash \rho S',\ \alpha \notin \rho\Gamma'$$

  By rule [WT-Poly]

  $$\rho\Gamma' \vdash \rho S$$

5. We split cases on the rule used at the root of the derivation. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

   - case [T-Sub]: Assume,

     $$\Gamma;\Gamma' \vdash e : S$$

     By inversion

     $$\Gamma;\Gamma' \vdash e : S',\ \Gamma;\Gamma' \vdash S' <: S,\ \Gamma;\Gamma' \vdash S$$

     For some S'. By IH, (3.) and (4.)

     $$\rho\Gamma' \vdash \rho e : \rho S',\ \rho\Gamma' \vdash \rho S' <: \rho S,\ \rho\Gamma' \vdash \rho S$$

     By rule [T-Sub]

     $$\rho\Gamma' \vdash \rho e : \rho S$$

   - case [T-Var-Base]: Assume,

     $$\Gamma;\Gamma' \vdash e : S$$

     Where $e \equiv y$ for some variable y and $S \equiv \{\nu : B \mid \nu = y\}$. By inversion

     $$\Gamma;\Gamma'(y) = \{\nu : B \mid e_y\}$$

     Thus, by Lemmas 1, 4 either

     $$\Gamma(y) = \{\nu : B \mid e_y\},\ y \in \mathsf{Dom}(\rho) \tag{a}$$

     or

     $$\Gamma'(y) = \{\nu : B \mid e_y\},\ y \notin \mathsf{Dom}(\rho) \tag{b}$$

     Assume (a). Then by Cor. 1

     $$\emptyset \models \rho(y) : \{\nu : B \mid \rho e_y\}$$

As $\rho(y)$ is a value, $\rho(y)$ is a constant of base type B. Hence, by rule [T-CONST] and Definition 1

$$\rho\Gamma' \vdash \rho(y) : \{\nu{:}B \mid \nu = \rho(y)\}$$

That is

$$\rho\Gamma' \vdash \rho e : \rho S$$

Assume (b). Then

$$\rho y \equiv y \tag{c}$$
$$\rho\Gamma'(y) \equiv \{\nu{:}B \mid \rho e_y\} \tag{d}$$

From (d) and rule [T-VAR-BASE]

$$\rho\Gamma' \vdash y : \{\nu{:}B \mid \nu = y\}$$

Finally, from (c)

$$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-VAR]: Assume,

$$\Gamma; \Gamma' \vdash e : S$$

Where $e \equiv y$ for some variable y and S is not a base type. By inversion

$$\Gamma; \Gamma'(y) = S$$

Thus, by Lemmas 1, 4 either

$$\Gamma(y) = S,\ y \in \mathsf{Dom}(\rho) \tag{a}$$

or

$$\Gamma'(y) = S,\ y \notin \mathsf{Dom}(\rho) \tag{b}$$

Assume (a). Then by Cor. 1

$$\emptyset \models \rho(y) : \rho S$$

By Lemma 5

$$\rho\Gamma' \models \rho(y) : \rho S$$

That is

$$\rho\Gamma' \vdash \rho e : \rho S$$

Assume (b). Then

$$\rho y \equiv y \tag{c}$$
$$(\rho\Gamma')(y) \equiv \rho(\Gamma'(y)) \equiv \rho S \tag{d}$$

From (d) and rule [T-Var]

$$\rho\Gamma' \vdash y : \rho S$$

Finally, from (c)

$$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-Const]: Trivial, as by Def. 1 and Lemma 1, $ty(c)$ has no free variables.
- case [T-Fun]: Assume,

  $$\Gamma; \Gamma' \vdash e : S$$

  Where $e \equiv \lambda y.e_1$, and $S \equiv y{:}T \rightarrow T_1$. By inversion

  $$\Gamma; \Gamma' \vdash T,\ \Gamma; \Gamma'; y{:}T \vdash e_1 : T_1 \tag{a}$$

  And as y is bound at most once in $\Gamma; \Gamma'; y{:}T$

  $$y \notin \mathsf{Dom}(\Gamma) \tag{b}$$

  By IH, (4.) (a) yields

  $$\rho\Gamma' \vdash \rho T,\ \rho\Gamma'; y{:}\rho T \vdash \rho e_1 : \rho T_1$$

  By rule [T-Fun] and (b)

  $$\rho\Gamma' \vdash \lambda y.\rho e_1 : y{:}\rho T \rightarrow \rho T_1$$

  Finally, by (a)

  $$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-App]: Assume,

  $$\Gamma; \Gamma' \vdash e : S$$

  Where $e \equiv e_1 e_2$, and $S \equiv [e_2/y]T$ for some T, where $y \notin \mathsf{Dom}(\Gamma; \Gamma')$ *i.e.*, by Lemma 1, $y \notin \mathsf{Dom}(\rho)$. By inversion

  $$\Gamma; \Gamma' \vdash e_1 : y{:}T_2 \rightarrow T,\ \Gamma; \Gamma' \vdash e_2 : T_2$$

  By IH

  $$\rho\Gamma' \vdash \rho e_1 : \rho(y{:}T_2 \rightarrow T),\ \rho\Gamma' \vdash \rho e_2 : \rho T_2$$

  As $y \notin \mathsf{Dom}(\rho)$

  $$\rho\Gamma' \vdash \rho e_1 : y{:}\rho T_2 \rightarrow \rho T$$

  By rule [T-App]

  $$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-IF]: Assume,

$$\Gamma; \Gamma' \vdash e : S$$

Where $e \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. By inversion

$$\Gamma; \Gamma' \vdash e_1 : \texttt{bool},\ \Gamma; \Gamma'; e_1 \vdash e_2 : S,\ \Gamma; \Gamma'; \neg e_1 \vdash e_3 : S$$

By IH

$$\rho\Gamma' \vdash \rho e_1 : \texttt{bool},\ \rho\Gamma'; \rho e_1 \vdash \rho e_2 : \rho S,\ \rho\Gamma'; \neg\rho e_1 \vdash \rho e_3 : \rho S$$

By rule [T-IF]

$$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-LET]: Assume,

$$\Gamma; \Gamma' \vdash e : S$$

Where $e \equiv \texttt{let}\ y\ =\ e_1\ \texttt{in}\ e_2$, where $y \notin \mathsf{Dom}(\Gamma; \Gamma')$ *i.e.,* by Lemma 1, $y \notin \mathsf{Dom}(\rho)$. By inversion

$$\Gamma; \Gamma' \vdash e_1 : S_y,\ \Gamma; \Gamma'; y{:}S_y \vdash e_2 : S,\ \Gamma; \Gamma' \vdash S$$

By IH, (4.)

$$\rho\Gamma' \vdash \rho e_1 : \rho S_y,\ \rho\Gamma'; y{:}\rho S_y \vdash \rho e_2 : \rho S$$
$$\rho\Gamma' \vdash \rho S$$

By rule [T-LET]

$$\rho\Gamma' \vdash \texttt{let}\ y\ =\ \rho e_1\ \texttt{in}\ \rho e_2 : \rho S$$

As $y \notin \mathsf{Dom}(\rho)$

$$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-GEN]: Assume,

$$\Gamma; \Gamma' \vdash e : S$$

Where $e \equiv [\Lambda\alpha]e_1$ and $S \equiv \forall\alpha.S_1$. By inversion

$$\Gamma; \Gamma' \vdash e_1 : S_1,\ \alpha \notin \Gamma; \Gamma'$$

As ρ is only applied to refinement predicates, the free type variables of $\rho\Gamma'$ are the same as those of Γ'. Hence

$$\rho\Gamma' \vdash \rho e_1 : \rho S_1,\ \alpha \notin \rho\Gamma'$$

By rule [T-GEN]

$$\rho\Gamma' \vdash \rho e : \rho S$$

- case [T-Inst]: Assume,

$$\Gamma; \Gamma' \vdash e : S$$

Where $e \equiv [\tau]e_1$, $S \equiv [T/\alpha]S_1$ and $\mathsf{Shape}(T) = \tau$. By inversion

$$\Gamma; \Gamma' \vdash e_1 : \forall\alpha.S_1,\ \ \Gamma; \Gamma' \vdash T$$

By IH, (4.)

$$\rho\Gamma' \vdash \rho e_1 : \forall\alpha.\rho S_1,\ \ \rho\Gamma' \vdash \rho T$$

As $\mathsf{Shape}(\rho T) = \mathsf{Shape}(T) = \tau$, by rule [T-Inst]

$$\rho\Gamma' \vdash [\tau]\rho e_1 : [\rho T/\alpha]\rho S_1$$

As $\alpha \notin \mathsf{Dom}(\rho) \cup \mathsf{Rng}(\rho)$

$$\rho\Gamma' \vdash \rho e : \rho S$$

□

Notice that we use a non-standard substitution lemma, where the substitutions only involve *values*. The value substitution lemma is easier to prove than the usual substitution lemma that allows arbitrary (well-typed) expressions in the substitutions, as in the latter case, to handle substitutions of a variable x of base type B with another (well-typed) expression e_x of base type, we would have to prove $\Gamma \vdash e_x : \{\nu : B \mid \nu = e_x\}$. As our dynamic semantics (Figure 8) require that only values are substituted during a single step, the weaker Lemma 10 suffices to the following preservation theorem.

Lemma 11. *(Guard Evaluation) If*

$$\begin{aligned} &e \hookrightarrow e' \\ &\emptyset \vdash e : \texttt{bool} \\ &\emptyset \vdash e' : \texttt{bool} \\ &\Gamma = \Gamma_1; e; \Gamma_2 \\ &\Gamma = \Gamma_1; e'; \Gamma_2 \end{aligned}$$

then:

1. $\Gamma \models \rho$ *iff* $\Gamma' \models \rho$,
2. $\Gamma \vdash e_1 \Rightarrow e_2$ *iff* $\Gamma' \vdash e_1 \Rightarrow e_2$,
3. $\Gamma \vdash S_1 <: S_2$ *iff* $\Gamma \vdash S_1 <: S_2$,
4. $\Gamma \vdash S$ *iff* $\Gamma' \vdash S$,
5. $\Gamma \vdash e_1 : S$ *iff* $\Gamma' \vdash e_1 : S$.

Proof. By simultaneous induction on the derivations, similar to the proof of Lemma 8. The key difference is the proof of (1.).

1. By Lemma 1, $\emptyset \vdash e : \texttt{bool}$ and $\emptyset e' : \texttt{bool}$ imply:

$$\text{For all } \rho \text{ we have } \rho e \equiv e,\ \ \rho e' \equiv e' \tag{a}$$

By Lemma 12

$$e \overset{*}{\hookrightarrow} \texttt{true} \iff e' \overset{*}{\hookrightarrow} \texttt{true} \tag{b}$$

Next, observe that

$$\Gamma \models \rho \iff \Gamma_1; e; \Gamma_2 \models \rho$$

By Lemma 4

$$\iff \rho \equiv \rho_1; \rho_2,\ \Gamma_1 \models \rho_1,\ \rho_1 e \overset{*}{\hookrightarrow} \texttt{true},\ \rho_1 \Gamma_2 \models \rho_2$$

By (a),(b)

$$\iff \rho \equiv \rho_1; \rho_2,\ \Gamma_1 \models \rho_1,\ \rho_1 e' \overset{*}{\hookrightarrow} \texttt{true},\ \rho_1 \Gamma_2 \models \rho_2$$

By Lemma 4

$$\iff \Gamma \models \rho$$

2. Follows from Rule [Imp] and (1.).
3. We split cases on the structure of S_1, S_2, and in each case, use inversion on the appropriate rule, the IH (or (2.)) and the rule (as in Lemma 8).
4. By induction on the derivation $\Gamma' \vdash S$. We split cases on the structure of S. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.
5. By induction on the derivation $\Gamma' \vdash e : S$. We split cases on the rule used at the root of the derivation. In each case, the proof proceeds by inversion, applying the IH and finally, applying the rule.

□

Lemma 12. *(Confluence) If $e \hookrightarrow e'$ then $e \overset{*}{\hookrightarrow} v$ iff $e' \overset{*}{\hookrightarrow} v$.*

Proof. First, observe that the relation $\hookrightarrow$ is deterministic – for each e there is at most one e' such that $e \hookrightarrow e'$. The $\Rightarrow$ (resp. $\Leftarrow$) proof follows from induction on the derivations $e \overset{*}{\hookrightarrow} v$ (resp. $e' \overset{*}{\hookrightarrow} v$). □

Theorem 3. (Preservation) *If $\emptyset \vdash e : S$ and $e \hookrightarrow e'$ then $\emptyset \vdash e' : S$.*

Proof. By induction on the typing derivation $\emptyset \vdash e : S$. We split cases on the rule used at the root of the derivation.

- case [T-Sub]: Assume,

$$\emptyset \vdash e : S$$

By inversion

$$\emptyset \vdash e : S' \tag{a}$$
$$\emptyset \vdash S' <: S,\ \emptyset \vdash S \tag{b}$$

For some S'. By IH and (a)

$$\emptyset \vdash e' : S'$$

Which, with (b) and rule [T-Sub] yields

$$\emptyset \vdash e' : S$$

- case [T-Var-Base], [T-Var], [T-Const]: Trivial as there is no e' such that $e \hookrightarrow e'$.
- case [T-Fun]: We split cases on the structure of e.
  - case $e \equiv (\lambda x.e_1)\ v$: Here $e' \equiv [v/x]e_1$. By pushing applications of rule [T-Sub] down, we can ensure the rule [T-Fun] is used at the root of the derivation of the type for $\lambda x.e_1$. By inversion

$$\emptyset \vdash \lambda x.e_1 : x{:}T \to T' \tag{a}$$
$$\emptyset \vdash v : T \tag{b}$$
$$S \equiv [v/x]T' \tag{c}$$

From (b) and rule [WS-Ext]

$$x{:}T \models [x \mapsto v] \tag{d}$$

By inversion (rule [T-Fun]) (a) implies

$$x{:}T \models e_1 : T'$$

Which, by (d) and Lemma 10 yields

$$\emptyset \vdash [v/x]e_1 : [v/x]T'$$

That is, from (c)

$$\emptyset \vdash e' : S$$

  - case $e \equiv \mathtt{c}\ v$: By pushing applications of rule [T-Sub] down, we can ensure the rule [T-Const] is used at the root of the derivation of the type for $\mathtt{c}$. Here $e' \equiv [\![\mathtt{c}]\!](v)$. By inversion

$$\emptyset \vdash \mathtt{c} : x{:}T \to T'$$
$$\emptyset \vdash v : T$$
$$S \equiv ty(\mathtt{c}) \equiv [v/x]T'$$

By Definition 1, the above imply

$$\emptyset \vdash [\![\mathtt{c}]\!](v) : [v/x]T'$$

That is

$$\emptyset \vdash e' : S'$$

  - case $e \equiv e_1\ e_2$ where e_1 is not a value: Here $e' \equiv e_1'\ e_2$ where $e_1 \hookrightarrow e_1'$. Assume

$$\emptyset \vdash e_1\ e_2 : S$$

By inversion

$$\emptyset \vdash e_1 : x{:}T \to T' \tag{a}$$
$$\emptyset e_2 : T,\ S \equiv [e_2/x]T' \tag{b}$$

By IH, (a) implies

$$\emptyset \vdash e_1' : x{:}T \to T'$$

Which with (b) and rule [T-App] yields

$$\emptyset \vdash e_1' \ e_2 : S$$

That is

$$\emptyset \vdash e' : S$$

– case $e \equiv e_1 e_2$ where e_2 is not a value: Here $e' \equiv e_1 e_2'$ where $e_2 \hookrightarrow e_2'$. Assume

$$\emptyset \vdash e_1 \ e_2 : S$$

By inversion

$$\emptyset \vdash e_1 : x{:}T \rightarrow T' \quad \text{(a)}$$
$$\emptyset e_2 : T, \ S \equiv [e_2/x]T' \quad \text{(b)}$$

By IH, (a) implies

$$\emptyset \vdash e_2' : T$$

Which with (b) and rule [T-App] yields

$$\emptyset \vdash e_1 \ e_2' : S$$

That is

$$\emptyset \vdash e' : S$$

- case [T-If]: We split cases on the structure of e.

– case $e \equiv$ `if` v_1 `then` e_2 `else` e_3 where v_1 is a value: Assume

$$\emptyset \vdash e \equiv S$$

By inversion

$$\emptyset \vdash v_1 : \texttt{bool} \quad \text{(a)}$$
$$v_1 \vdash e_2 : S \quad \text{(b)}$$
$$\neg v_1 \vdash e_3 : S \quad \text{(c)}$$

As `true` and `false` are the only values of type `bool`, (a) implies

$$v_1 \equiv \texttt{true} \quad \text{(d)}$$

or

$$v_1 \equiv \texttt{true} \quad \text{(e)}$$

Assume (d). By (b) and Lemma 7

$$\emptyset \vdash e_2 : S$$

By (d) and rule [E-If-True], $e' \equiv e_2$. Hence

$$\emptyset \vdash e' : S$$

Assume (e). As $\neg$`false` $\equiv$ `true`, by (c) and Lemma 7

$$\emptyset \vdash e_3 : S$$

By (e) and rule [E-If-False], $e' \equiv e_3$. Hence

$$\emptyset \vdash e' : S$$

– case $e \equiv$ `if` e_1 `then` e_2 `else` e_3 where e_1 is not a value: Here, $e' \equiv$ `if` e_1' `then` e_2 `else` e_3 where $e_1 \hookrightarrow e_1'$ and as e_1 is not a value, $\neg e_1 \hookrightarrow \neg e_1'$. Assume

$$\emptyset \vdash e \equiv S$$

By inversion

$$\emptyset \vdash e_1 : \texttt{bool} \tag{a}$$
$$e_1 \vdash e_2 : S \tag{b}$$
$$\neg e_1 \vdash e_3 : S \tag{c}$$

By IH

$$\emptyset \vdash e_1' : \texttt{bool} \tag{d}$$

and hence

$$\emptyset \vdash \neg e_1' : \texttt{bool} \tag{e}$$

Thus, by Lemma 11 (b), (c) imply

$$e_1' \vdash e_2 : S$$
$$\neg e_1' \vdash e_3 : S$$

Which, with (a) and rule [T-If] yield

$$\emptyset \vdash e' : S$$

- case [T-Let]: We split cases on the structure of e.
  – case $e \equiv$ `let` $x = v_1$ `in` e_2 where v_1 is a value: Here $e' \equiv [v_1/x]e_2$. By inversion,

$$\emptyset \vdash v_1 : S_1 \tag{a}$$
$$x{:}S_1 \vdash e_2 : S \tag{b}$$
$$\emptyset \vdash S \tag{c}$$

By (c), $\mathsf{FreeVars}(S) \equiv \emptyset$ hence

$$[v_1/x]S \equiv S \tag{d}$$

By Lemma 10, (a),(b) imply

$$\emptyset \vdash [v_1/x]e_2 : [v_1/x]S \tag{d}$$

Which, with (d) yields

$$\emptyset \vdash e' : S$$

- case $e \equiv \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$ where e_1 is not a value: Here $e' \equiv \texttt{let}\ x\ =\ e_1'\ \texttt{in}\ e_2$ where $e_1 \hookrightarrow e_1'$. By inversion,

$$\emptyset \vdash e_1 : S_1 \tag{a}$$
$$x{:}S_1 \vdash e_2 : S \tag{b}$$
$$\emptyset \vdash S \tag{c}$$

  By IH (a) implies

$$\emptyset \vdash e_1' : S_1$$

  Which, with (b),(c) and rule [T-Let] yields

$$\emptyset \vdash e' : S$$

- case [T-Gen],[T-Inst]: By inversion, applying the induction hypothesis and then the rule.

□

Theorem 4. (Progress) *If $\emptyset \vdash e : S$ and e is not a value then there exists an $e \hookrightarrow e'$.*

Proof. By induction on the typing derivation. We split cases on the rule used at the root of the derivation (as in the proof of Theorem 3. The important case is when the rule is [T-App] and $e \equiv \texttt{c}\ v$, *i.e.,* the application of a value v to a primitive constant $\texttt{c}$. By inversion, $ty(\texttt{c})$ must be of the form $x{:}T \rightarrow T'$, and as $\emptyset \vdash v : T$. Hence, by Definition 1, $[\![\texttt{c}]\!](v) \equiv e'$ is well defined. □

Theorem 5. (Soundness of Decidable Checking) *If $\Gamma \vdash_{\mathbb{Q}} e : S$ then $\Gamma \vdash e : S$.*

Proof. By induction on the typing derivation. The key observations are that each liquid type (schema) is also a dependent type schema, each liquid type deriviation rule [LT-*] has a matching dependent type derivation rule, and the soundness of the embedding Definition 2. □

E Correctness of Type Inference

Lemma 13. *(Fresh) For each type schema σ and assignment A over $\mathbb{Q}$, $A(\mathsf{Fresh}(\sigma))$ is a liquid type over $\mathbb{Q}$.*

Proof. By induction on the structure of σ. □

Lemma 14. *(Shape) For every liquid type assignment A:*

1. $\mathsf{Shape}(F) = \mathsf{Shape}(AF)$,
2. $\mathsf{Shape}(\Gamma) = \mathsf{Shape}(A\Gamma)$.

Proof. (1) follows by induction on the structure of F. (2) follows from (1). □

Lemma 15. *(Derivation Projection) If $\Gamma \vdash_{\mathbb{Q}} e : S$ then $\mathsf{Shape}(\Gamma) \vdash_{ML} e : \mathsf{Shape}(S)$.*

Proof. Induction on the derivation of $\Gamma \vdash_{\mathbb{Q}} e : S$, and observing that each derivation rule for $\vdash_{\mathbb{Q}}$ is a *refinement* of a matching rule for $\vdash_{ML}$. □

Lemma 16. *(Constraint Substitution) For every template environment Γ, expression e, and liquid type assignment A, if $\mathsf{Cons}(\Gamma, e) = (F, C)$ then $\mathsf{Cons}(A\Gamma, e) = (AF, AC)$.*

Proof. By induction on the structure of e. □

Lemma 17. *(Update) For any assignment A, template F fresh with respect to A (if a liquid type variable κ appears in F then it appears only once and it is not in $\mathsf{Dom}(A)$) and liquid type $\hat{T}$ such that $\mathsf{Shape}(\hat{T}) = \mathsf{Shape}(F)$:*

1. $\mathsf{SolUpd}(A, F, \hat{T})(F) = \hat{T}$
2. *if* $\mathsf{LiquidVars}(F') \subseteq \mathsf{Dom}(A)$ *then* $\mathsf{SolUpd}(A, F, \hat{T})F' = AF'$.

Proof. By induction on the structure of F and $\hat{T}$. □

Theorem 6. (Constraint Generation) *For every type environment Γ and expression e such that $\mathsf{Cons}(\Gamma, e) = (F, C)$, $\Gamma \vdash_{\mathbb{Q}} e : S$ iff there exists an assignment A over $\mathbb{Q}$ such that $AF = S$ and AC is valid.*

Proof. Only if ($\Rightarrow$): By induction on the derivation $\Gamma \vdash_{\mathbb{Q}} e : S$.

- case $e \equiv \mathsf{c}$ or $e \equiv x$: Here $\mathsf{LiquidVars}(F) = \emptyset$ *i.e.,* F has no liquid type variables, and $C = \emptyset$ so *any* solution A suffices.
- case $e \equiv \lambda x.e_1$: Here,

$$\begin{aligned}
F &= x\!:\!F_x \to F_1 \\
C &= C_1 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x\!:\!F_x \vdash_{\mathbb{Q}} F_1' <: F_1\} \\
S &= x\!:\!\hat{T}_x \to \hat{T}_1 \\
x\!:\!F_x \to F_1 &= \mathsf{Fresh}(\mathsf{Shape}(x\!:\!\hat{T}_x \to \hat{T}_1)) \\
(F_1', C_1) &= \mathsf{Cons}(\Gamma; x\!:\!F_x)
\end{aligned}$$

Let $A_0 = \mathsf{SolUpd}(\emptyset, F, S)$. By Lemma 16,

$$\begin{aligned}
(A_0 C_1, A_0 F_1') &= \mathsf{Cons}(\Gamma; x\!:\!A_0 F_x, e_1) \\
&= \mathsf{Cons}(\Gamma; x\!:\!\hat{T}_x, e_1)
\end{aligned}$$

By inversion, $\Gamma \vdash \hat{T}_x$ and $\Gamma; x\!:\!\hat{T}_x \vdash_{\mathbb{Q}} e_1 : \hat{T}_1$. Thus, by IH, there exists A_1 such that:

$$A_1(A_0C_1), \text{is valid} \qquad (a)$$
$$A_1(A_0F_1') = \hat{T}_1 \qquad (b)$$

Thus, $A = A_1; A_0$ is such that:

$$\begin{aligned} AF &= A_1(A_0F) \\ &= A_1(S) \\ &= S \quad \text{as } \mathsf{LiquidVars}(S) = \emptyset \end{aligned}$$

Moreover,

$$A; A_1C = A_1; A_0C_1 \cup \{\Gamma \vdash_{\mathbb{Q}} A_1; A_0F\} \cup \{\Gamma; x\!:\!A_1; A_0F_x \vdash_{\mathbb{Q}} A_1; A_0F_1' <: F_1\}$$

as $\mathsf{Dom}(A_0)$ and $\mathsf{Dom}(A_1)$ are disjoint, and (b)

$$= A_1; A_0C_1 \cup \{\Gamma \vdash_{\mathbb{Q}} T\} \cup \{\Gamma; x\!:\!\hat{T}_x \vdash_{\mathbb{Q}} \hat{T}_1 <: \hat{T}_1\}$$

which, by (a), Lemma 2 and Lemma 9 respectively, is valid.

- case $e \equiv e_1e_2$: Here,

$$\begin{aligned} F &= [e_2/x]F' \\ C &= C_1 \cup C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} F_2' <: F_2''\} \\ (c\!:\!F_2'' \to F', C_1) &= \mathsf{Cons}(\Gamma, e_1) \\ (F_2', C_2) &= \mathsf{Cons}(\Gamma, e_2) \end{aligned}$$

  By inversion there exist T_2 and T such that:

$$\Gamma \vdash_{\mathbb{Q}} e_1 : x\!:\!T_2 \to T \qquad (a)$$
$$\Gamma \vdash_{\mathbb{Q}} e_2 : T_2 \qquad (b)$$
$$S = [e_2/x]T \qquad (c)$$

  By IH and (a), there exist A_1 such that:

$$A_1F_2'' = T_2 \quad \text{and} \quad A_1F' = T \qquad (d)$$
$$A_1C_1 \text{ is valid} \qquad (e)$$

  By IH and (b), there exist A_2 such that:

$$A_2F_2' = T_2 \qquad (f)$$
$$A_2C_2 \text{ is valid} \qquad (g)$$

  Moreover,

$$\begin{aligned} \mathsf{Dom}(A_1) &= \mathsf{LiquidVars}(x\!:\!F_2'' \to F') \cup \mathsf{LiquidVars}(C_1) \\ \mathsf{Dom}(A_2) &= \mathsf{LiquidVars}(F_2') \cup \mathsf{LiquidVars}(C_2) \\ &\text{are disjoint} \qquad (h) \end{aligned}$$

as they result from generating constraints on different subexpressions and $\mathsf{LiquidVars}(\Gamma) = \emptyset$. Consider $A = A_1; A_2$.

$$AF = A_1; A_2[e_2/x]F'$$

which, due to delayed substitutions

$$= [e_2/x]A_1; A_2F'$$

which, because of disjoint domains (g)

$$= [e_2/x]A_1F'$$

which, due to (d)

$$\begin{aligned} &= [e_2/x]T \\ &= S \end{aligned}$$

Moreover,

$$AC = A_1; A_2C_1 \cup A_1; A_2C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} A_1; A_2F_2' <: A_1; A_2F_2''\}$$

which, due to disjoint domains (g)

$$= A_1C_1 \cup A_2C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} A_2F_2' <: A_1F_2''\}$$

which, due to (f) and (d)

$$= A_1C_1 \cup A_2C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} T_2 <: T_2\}$$

which, by (e),(g) and Lemma 9 is valid.

- case $e \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$: Here,

$$\begin{aligned} F &= \mathsf{Fresh}(\mathsf{Shape}(S)) \quad \text{(by Lemma 15)} \\ C &= C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \\ &\quad \cup \{\Gamma; e_1 \vdash_{\mathbb{Q}} F_2' <: F\} \\ &\quad \cup \{\Gamma; \neg e_1 \vdash_{\mathbb{Q}} F_3' <: F\} \\ (\cdot, C_1) &= \mathsf{Cons}(\Gamma, e_1) \\ (F_2', C_2) &= \mathsf{Cons}(\Gamma; e_1, e_2) \\ (F_3', C_3) &= \mathsf{Cons}(\Gamma; \neg e_1, e_3) \end{aligned}$$

where $\mathsf{LiquidVars}(C_1)$,$\mathsf{LiquidVars}(C_2)$,$\mathsf{LiquidVars}(C_3)$ are disjoint. By inversion, and applying the IH, there exist solutions A_1, A_2, A_3 such that:

$$\begin{aligned} A_1C_1,\ A_2C_2,\ A_3C_3 &\text{ are valid} && \text{(a)} \\ A_2F_2' &= A_3F_3' = S && \text{(b)} \\ \Gamma &\vdash S && \text{(c)} \end{aligned}$$

Consider $A = \mathsf{SolUpd}(A_1; A_2; A_3, F, S)$, By Lemma 17,

$$\begin{aligned}AF =& S\\ AC =& A_1C_1 \cup A_2C_2 \cup A_3C_3 \cup \{\Gamma \vdash S\}\\ &\cup \{\Gamma; e_1 \vdash_{\mathbb{Q}} A_2F_2' <: S\}\\ &\cup \{\Gamma; \neg e_1 \vdash_{\mathbb{Q}} A_3F_3' <: S\}\end{aligned}$$

by (b) and (c)

$$\begin{aligned}=& A_1C_1 \cup A_2C_2 \cup A_3C_3 \cup \{\Gamma \vdash S\}\\ &\cup \{\Gamma; e_1 \vdash_{\mathbb{Q}} S <: S\}\\ &\cup \{\Gamma; \neg e_1 \vdash_{\mathbb{Q}} S <: S\}\end{aligned}$$

which, by (a), inversion and Lemma 9 is valid.

- case $e \equiv \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$: Here,

$$\begin{aligned}F &= \mathsf{Fresh}(\mathsf{Shape}(S)) \quad \text{(by Lemma 15)}\\ C &= C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x\!:\!F_1'F_2' <: F\} && \text{(a0)}\\ (F_1', C_1) &= \mathsf{Cons}(\Gamma, e_1)\\ (F_2', C_2) &= \mathsf{Cons}(\Gamma; x\!:\!F_1', e_2) && \text{(a1)}\end{aligned}$$

By inversion there exists S_1 such that:

$$\begin{aligned}\Gamma &\vdash e_1 : S_1 && \text{(a)}\\ \Gamma; x\!:\!S_1 &\vdash e_2 : S && \text{(b)}\\ \Gamma &\vdash S && \text{(c)}\end{aligned}$$

By (a) and IH there exists A_1 such that:

$$\begin{aligned}&A_1C_1 \text{ is valid} && \text{(d)}\\ &A_1F_1' = S_1 && \text{(e)}\end{aligned}$$

By Lemma 16 and (a1),

$$\begin{aligned}(A_1F_2', A_1C_2) &= \mathsf{Cons}(\Gamma; x\!:\!A_1F_1', e_2)\\ &= \mathsf{Cons}(\Gamma; x\!:\!S_1', e_2) \quad \text{by (e)}\end{aligned}$$

By (b) and IH, there exists A_2 such that:

$$\begin{aligned}\mathsf{Dom}(A_2) &= \mathsf{LiquidVars}(C_2) \text{ which is disjoint from } \mathsf{Dom}(A_1) && \text{(f)}\\ A_2(A_1C_2) &\text{ is valid} && \text{(g)}\\ A_2(A_1F_2') &= S && \text{(h)}\end{aligned}$$

Consider $A = \mathsf{SolUpd}(A_2; A_1, F, S)$. By Lemma 17,

$$AF = S$$

By (a0), (f), and (i):

$$AC = A_1C_1 \cup A_2(A_1C_2) \cup \{\Gamma \vdash AF\} \cup \{\Gamma; x\!:\!A_1F_1'A_2(A_1F_2') <: AF\}$$

by (i), (e), (h)

$$= A_1C_1 \cup A_2(A_1C_2) \cup \{\Gamma \vdash S\} \cup \{\Gamma; x\!:\!S_1 \vdash_{\mathbb{Q}} S <: S\}$$

which, by (d), (g), (c), and Lemma 9 is valid.

- case $e \equiv [\Lambda\alpha]e_1$: Here,

$$\begin{aligned}(F, C) &= (\forall\alpha.F_1', C_1)\\ (F_1', C_1) &= \mathsf{Cons}(\Gamma, e_1)\end{aligned}$$

By inversion, exists S_1 such that $\Gamma \vdash_{\mathbb{Q}} e_1 : S_1$. Thus, by IH, exists a A_1 such that:

$$A_1C_1 \text{ is valid} \tag{a}$$
$$A_1F_1' = S_1 \tag{b}$$

Consider $A = A_1$.

$$\begin{aligned}AF =& A_1F\\ =& A_1(\forall\alpha.F_1')\\ =& \forall\alpha.A_1F_1'\end{aligned}$$

which, by (b),

$$\begin{aligned}=& \forall\alpha.S_1\\ =& S\end{aligned}$$

Finally,

$$AC = A_1C_1 \text{ which, by (a), is valid.}$$

- case $e \equiv [\tau]e_1$: Here,

$$\begin{aligned}F =& [F_\alpha/\alpha]F_1'\\ C =& C_1 \cup \{\Gamma \vdash F_\alpha\})\\ (F_1', C_1) =& \mathsf{Cons}(\Gamma, e_1)\\ F_\alpha =& \mathsf{Fresh}(\tau)\end{aligned}$$

By inversion, there exists $\hat{T}$, S_1 such that:

$$\Gamma \vdash \hat{T} \tag{a}$$
$$\mathsf{Shape}(\hat{T}) = \tau \tag{b}$$
$$\gamma \vdash_{\mathbb{Q}} e_1 : \forall\alpha.S_1 \tag{c}$$

By IH, there exists A_1 such that:

$$A_1C_1 \text{ is valid} \tag{d}$$
$$A_1F_1' = \forall\alpha.S_1 \tag{e}$$

Consider $A = \mathsf{SolUpd}(A_1, F_\alpha, \hat{T})$:

$$\begin{aligned} AF =& A([F_\alpha/\alpha]F_1') \\ =& [AF_\alpha/\alpha]AF_1' \end{aligned}$$

by Lemma 17

$$\begin{aligned} =& [\hat{T}/\alpha]A_1F_1' \\ =& [\hat{T}/\alpha]\forall\alpha.S_1 \\ =& S \end{aligned}$$

If (⇐): By induction on the structure of e.

- case $e \equiv \mathsf{c}$ or $e \equiv x$ Trivial as $C = \emptyset$ and F such that $\mathsf{LiquidVars}(F) = \emptyset$ and $\Gamma \vdash_{\mathbb{Q}} e : F$.
- case $e \equiv \lambda x.e_1$: Here

$$\begin{aligned} F =& x{:}F_x \to F_1 \\ C =& C_1 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x{:}F_x \vdash_{\mathbb{Q}} F_1' <: F_1\} \\ S =& x{:}\hat{T}_x \to \hat{T}_1 \\ x{:}F_x \to F_1 =& \mathsf{Fresh}(\mathsf{Shape}(x{:}\hat{T}_x \to \hat{T}_1)) \\ (F_1', C_1) =& \mathsf{Cons}(\Gamma; x{:}F_x) \end{aligned}$$

As AC is valid,

$$AC_1 \text{ is valid} \tag{a}$$

$$\Gamma \vdash x{:}AF_x \to AF_1 \tag{b}$$

$$\Gamma; x{:}AF_x \vdash_{\mathbb{Q}} AF_1' <: AF_1 \tag{c}$$

By Lemma 16,

$$(AF_1', AC_1) = \mathsf{Cons}(\Gamma; x{:}AF_x, e_1)$$

By (a) and the IH,

$$\Gamma; x{:}AF_x \vdash_{\mathbb{Q}} e_1 : AF_1' \tag{d}$$

By (d),(b),(c) and rule [LT-Sub],

$$\Gamma; x{:}AF_x \vdash_{\mathbb{Q}} e_1 : AF_1 \tag{e}$$

By Lemma , and rule [LT-Fun],

$$\begin{aligned} &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : x{:}AF_x \to AF_1 \\ \text{implies } &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : A(x{:}F_x \to F_1) \\ \text{implies } &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : AF \end{aligned}$$

- case $e \equiv e_1 e_2$: Here,

$$\begin{aligned}
F =& [e_2/x]F' \\
C =& C_1 \cup C_2 \cup \{\Gamma \vdash_{\mathbb{Q}} F'_2 <: F''_2\} \\
(c\!:\!F''_2 \to F', C_1) =& \mathsf{Cons}(\Gamma, e_1) \\
(F'_2, C_2) =& \mathsf{Cons}(\Gamma, e_2) \\
AC &\text{ is valid} \\
AC_1 \cup AC_2 &\text{ is valid} \\
\Gamma \vdash_{\mathbb{Q}} & AF'_2 <: AF''_2 && \text{(a)}
\end{aligned}$$

Hence, by the IH,

$$\begin{aligned}
\Gamma \vdash_{\mathbb{Q}} & e_1 : x\!:\!AF''_2 \to AF' && \text{(b)} \\
\Gamma \vdash_{\mathbb{Q}} & e_2 : AF'_2 && \text{(c)}
\end{aligned}$$

From (b), and Lemma 2,

$$\Gamma \vdash AF''_2 \qquad \text{(d)}$$

Thus, from (a),(c), (d) and rule [LT-Sub],

$$\Gamma \vdash_{\mathbb{Q}} e_2 : AF''_2$$

From (b) and rule [LT-App],

$$\Gamma \vdash_{\mathbb{Q}} e_1 e_2 : [e_2/x]AF' \qquad \text{(e)}$$

As substitutions are delayed,

$$[e_2/x]AF' = A[e_2/x]F' = AF$$

and so, from (e),

$$\Gamma \vdash_{\mathbb{Q}} e_1 e_2 : AF$$

- case $e \equiv$ `if` e_1 `then` e_2 `else` e_3: Here,

$$\begin{aligned}
F =& \mathsf{Fresh}(\mathsf{Shape}(S)) \quad \text{(by Lemma 15)} \\
C =& C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup \{\Gamma; e_1 \vdash_{\mathbb{Q}} F'_2 <: F\}\{\Gamma; \neg e_1 \vdash_{\mathbb{Q}} F'_3 <: F\} \\
(\cdot, C_1) =& \mathsf{Cons}(\Gamma, e_1) \\
(F'_2, C_2) =& \mathsf{Cons}(\Gamma; e_1, e_2) \\
(F'_3, C_3) =& \mathsf{Cons}(\Gamma; \neg e_1, e_3)
\end{aligned}$$

As AC is valid,

$$\begin{aligned}
AC_1, AC_2, AC_3 &\text{ are valid} && \text{(a)} \\
\Gamma \vdash & AF && \text{(b)} \\
\Gamma; e_1 \vdash_{\mathbb{Q}} & AF'_2 <: AF && \text{(ct)} \\
\Gamma; \neg e_1 \vdash_{\mathbb{Q}} & AF'_3 <: AF && \text{(cf)}
\end{aligned}$$

By (a) and IH,

$$\begin{aligned}\Gamma; e_1 \vdash_{\mathbb{Q}} e_2 &: AF'_2\\ \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 &: AF'_3\end{aligned}$$

By (b), (ct), (cf), Lemma 6,

$$\begin{aligned}\Gamma; e_1 \vdash_{\mathbb{Q}} e_2 &: AF\\ \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 &: AF\end{aligned}$$

By (a), Lemma and rule [LT-If],

$$\Gamma \vdash_{\mathbb{Q}} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : AF$$

- case $e \equiv \texttt{let } x \ = \ e_1 \texttt{ in } e_2$: Here,

$$\begin{aligned}F =& \mathsf{Fresh}(\mathsf{Shape}(S)) \quad \text{(by Lemma 15)}\\ C =& C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma; x\!:\!F'_1 F'_2 <: F\}\\ (F'_1, C_1) =& \mathsf{Cons}(\Gamma, e_1)\\ (F'_2, C_2) =& \mathsf{Cons}(\Gamma; x\!:\!F'_1, e_2)\end{aligned}$$

As AC is valid

$$AC_1, AC_2 \text{ are valid} \tag{a}$$
$$\Gamma \vdash AF \tag{b}$$
$$\Gamma; x\!:\!AF'_1 \vdash_{\mathbb{Q}} AF'_2 <: F \tag{c}$$

By (a) and IH,

$$\Gamma \vdash_{\mathbb{Q}} e_1 : AF'_1 \tag{d1}$$
$$\Gamma; x\!:\!AF'_1 \vdash_{\mathbb{Q}} e_2 : AF'_2 \tag{d2}$$

By (b), (c), (d1), Lemma 5, and rule [LT-Sub],

$$\Gamma; x\!:\!AF'_1 \vdash_{\mathbb{Q}} e_2 : AF \tag{e}$$

Thus, by (b), (c), (d1), (e) and rule [LT-Let],

$$\Gamma \vdash_{\mathbb{Q}} \texttt{let } x \ = \ e_1 \texttt{ in } e_2 : AF$$

- case $e \equiv [\Lambda\alpha]e_1$: Here,

$$\begin{aligned}(F, C) =& (\forall\alpha.F'_1, C_1)\\ (F'_1, C_1) =& \mathsf{Cons}(\Gamma, e_1)\end{aligned}$$

As AC is valid, AC_1 is valid, and so,

$$\Gamma \vdash_{\mathbb{Q}} e : AF'_1 \tag{a}$$

As $\mathsf{Shape}(\Gamma) \vdash_{ML} e : \sigma$,

$$\alpha \notin \Gamma \tag{b}$$

Thus, by rule [LT-Gen],

$$\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e_1 : \forall\alpha.AF_1'$$

as $\alpha \notin A$

$$\begin{aligned}\text{implies } \Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e_1 : A\forall\alpha.F_1'\\ \text{implies } \Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e_1 : AF\end{aligned}$$

- case $e \equiv [\tau]e_1$: Here,

$$\begin{aligned}F =&[F_\alpha/\alpha]F_1'\\ C =&C_1 \cup \{\Gamma \vdash F_\alpha\})\\ (\forall\alpha.F_1', C_1)&\mathsf{Cons}(\Gamma, e_1)\\ F_\alpha =& \mathsf{Fresh}(\tau)\end{aligned}$$

As AC is valid,

$$AC_1 \text{ is valid} \tag{a}$$
$$\Gamma \vdash AF_\alpha \tag{b}$$

By (a) and IH,

$$\Gamma \vdash_{\mathbb{Q}} A\forall\alpha.F_1'$$

As $\alpha \notin A$, $A\forall\alpha.F_1' = \forall\alpha.AF_1'$. Thus, by (a) and IH,

$$\Gamma \vdash_{\mathbb{Q}} A\forall\alpha.F_1' \tag{c}$$

Thus, by (b), (c), Lemma 14 and rule [T-Inst],

$$\Gamma \vdash_{\mathbb{Q}} [\tau]e_1 : [AF_\alpha/\alpha]AF_1'$$

as $\alpha \notin A$, $F_\alpha \notin \mathsf{Rng}(A)$

$$\begin{aligned}&\Rightarrow \Gamma \vdash_{\mathbb{Q}} [\tau]e_1 : A[F_\alpha/\alpha]F_1'\\ &\Rightarrow \Gamma \vdash_{\mathbb{Q}} [\tau]e_1 : AF\end{aligned}$$

□

Definition 3. *(Simple Constraints) A* simple constraint *is of the form:*

- $\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid b\}$
- $\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid b\} <: \{\nu : \tau \mid b'\}$

where b, b' are either expressions or liquid type variables with pending substitutions.

Lemma 18. *(Constraint Splitting) For every set of constraints C,*

1. $\mathsf{Split}(C)$ *is a set of simple constraints,*

2. *For every assignment A, AC is valid iff $A(\mathsf{Split}(C))$ is valid.*

Proof. For both (1),(2), we prove the lemma when C is a singleton set and then lift to arbitrary sets. For singleton sets $\{c\}$, both (1), (2) follow by induction on the structure of c. □

Definition 4. *(Minimum Solution) For two assignments A and A' over $\mathbb{Q}$, we say $A \leq A'$ if for all κ, $A(\kappa) \supseteq A'(\kappa)$. For C, a set of constraints, A^* is the* minimum solution *over $\mathbb{Q}$ if*

1. *A^*C is valid and,*

2. *For each A over $\mathbb{Q}$, if AC is valid then $A^* \leq A$.*

Lemma 19. *(Embedding) If $A \leq A'$ are two assignments over $\mathbb{Q}$ then:*

1. $[\![A\kappa]\!] \Rightarrow [\![A\kappa]\!]$,

2. $[\![A(\theta \cdot \kappa)]\!] \Rightarrow [\![A(\theta \cdot \kappa)]\!]$,

3. $[\![A\Gamma]\!] \Rightarrow [\![A'\Gamma]\!]$.

Proof. Immediate from definition of solution ordering ($\leq$).

□

Theorem 7. (Minimum Solution) *If C has a solution over $\mathbb{Q}$ then C has a minimum solution over $\mathbb{Q}$.*

Proof. By Lemma 18 it suffices to prove the theorem for sets of simple constraints C. As the number of liquid type variables and logical qualifiers $\mathbb{Q}$ is finite, any C can only have a finite number of solutions over $\mathbb{Q}$. Suppose that $A_1, \ldots, A_n$ are the solutions for C, *i.e.,* for each $1 \leq i \leq n$, we have A_iC is valid. Then we shall show that:

$$A^* = \lambda\kappa. \cap_i A_i(\kappa)$$

is a minimum solution for C over $\mathbb{Q}$. Trivially, for each i, we have $A^* \leq A_i$. Next, we shall prove for each simple constraint $c \in C$, that as each A_ic is valid, A^*c is also valid.

- case $c \equiv \Gamma \vdash \{\nu : B \mid b\}$: where b is either an expression e or a variable with pending substitutions $\theta \cdot \kappa$. In the first case $b \equiv e$,

$$\begin{aligned} & A_i(\Gamma \vdash \{\nu : B \mid e\}) \\ \textit{i.e.,} \quad & \mathsf{Shape}(A_i\Gamma); \nu : B \vdash e : \texttt{bool} \end{aligned}$$

by Lemma 14, $\mathsf{Shape}(A^*\Gamma) = \mathsf{Shape}(A_i\Gamma) = \mathsf{Shape}(\Gamma)$, thus,

$$\begin{aligned} & \mathsf{Shape}(A^*\Gamma); \nu : B \vdash e : \texttt{bool} \\ \textit{i.e.,} \quad & A^*(\Gamma \vdash \{\nu : B \mid e\}) \end{aligned}$$

In the second case $b \equiv \theta \cdot \kappa$,

$$\begin{aligned} & A_i(\Gamma \vdash \{\nu : B \mid \theta \cdot \kappa\}) \\ \textit{i.e.,} \quad & \mathsf{Shape}(A_i\Gamma); \nu : B \vdash \theta \cdot A_i(\kappa) : \texttt{bool} \end{aligned}$$

i.e., for each $e \in A_i(\kappa)$,

$$\begin{aligned} & A_i\Gamma \vdash \{\nu : B \mid \theta e\}) \\ \textit{i.e.,}\quad & \mathsf{Shape}(A_i\Gamma); \nu : B \vdash \theta e : \texttt{bool} \end{aligned}$$

as $A^*(\kappa) \subseteq A_i(\kappa)$ and by Lemma 14, $\mathsf{Shape}(A^*\Gamma) = \mathsf{Shape}(A_i\Gamma) = \mathsf{Shape}(\Gamma)$, for each $e \in A^*(\kappa)$,

$$\begin{aligned} & \mathsf{Shape}(A^*\Gamma); \nu : B \vdash \theta e : \texttt{bool} \\ \textit{i.e.,}\quad & A^*(\Gamma \vdash \{\nu : B \mid \theta \cdot \kappa\}) \end{aligned}$$

- case $c \equiv \Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid b\} <: \{\nu : B \mid b'\}$: where each of b, b' is either an expression or a variable with pending substitution.

$$\begin{aligned} \text{For each } i,\quad & A_i(\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid b\} <: \{\nu : B \mid b'\}) \\ \textit{i.e.,}\quad & A_i\Gamma \vdash_{\mathbb{Q}} A_i b \Rightarrow A_i b' \\ \textit{i.e.,}\quad & [\![A_i\Gamma]\!] \wedge [\![A_i b]\!] \Rightarrow [\![A_i b']\!] \end{aligned}$$

  by the properties of implication,

$$[\![\wedge_i A_i\Gamma]\!] \wedge [\![\wedge_i A_i b]\!] \Rightarrow [\![\wedge_i A_i b']\!]$$

  as $[\![A^*\Gamma]\!] = [\![\wedge_i A_i\Gamma]\!]$ and $[\![A^*b]\!] = [\![\wedge_i A_i b]\!]$ and $[\![A^*b']\!] = [\![\wedge_i A_i b']\!]$, we have:

$$\begin{aligned} & [\![A^*\Gamma]\!] \wedge [\![A^*b]\!] \Rightarrow [\![A^*b']\!] \\ \textit{i.e.,}\quad & A^*(\Gamma \vdash_{\mathbb{Q}} \{\nu : B \mid b\} <: \{\nu : B \mid b'\}) \end{aligned}$$

□

Lemma 20. *(Refinement) If* $A' = \mathsf{Refine}(A, c)$ *then:*

1. $A \leq A'$,
2. *if* Ac *is not valid, then* $A \neq A'$,
3. *if* $A''c$ *is valid and* $A \leq A''$ *then* $A' \leq A''$.

Proof. 1. From the definition of Refine, we have:

$$A' \equiv A[\kappa_c \mapsto A(\kappa) \cap Q']$$

for some κ_c and Q'. We shall show that for any κ, we have $A'(\kappa) \subseteq A(\kappa)$.

Consider $\kappa \neq \kappa_c$. Here $A'(\kappa) = A(\kappa) \subseteq A(\kappa)$.

Consider $\kappa = \kappa_c$. Here $A'(\kappa) = A(\kappa) \cap Q' \subseteq A(\kappa)$.

2. We split cases on the type of constraint used for refinement, and in each case, show that if $A' = A$ then Ac must be valid.

   - case $c \equiv \Gamma \vdash \{\nu : B \mid b\}$: where b is an expression or a liquid type variable with pending substitutions. From the definition of Refine, we have:

$$\mathsf{Shape}(A\Gamma) \vdash A'b : \texttt{bool}$$

If $A = A'$ then,

$$\mathsf{Shape}(A\Gamma) \vdash Ab : \texttt{bool}$$

i.e., Ac is valid.

- case $c \equiv \Gamma \vdash b \Rightarrow \theta \cdot \kappa_c$: From the definition of Refine, we have:

  $$A\Gamma \vdash Ab \Rightarrow \theta \cdot A'(\kappa_c)$$

  If $A = A'$ then,

  $$A\Gamma \vdash Ab \Rightarrow \theta \cdot A(\kappa_c)$$

  i.e., Ac is valid.

3. We split cases on the type of constraint used for refinement.

- case $c \equiv \Gamma \vdash \{\nu : B \mid b\}$: where b is an expression or a liquid type variable with pending substitutions. We have,

  $$\begin{aligned} &A''c \text{ is valid} && \text{(a)} \\ &\forall \kappa. A''(\kappa) \subseteq A(\kappa) && \text{(b)} \end{aligned}$$

  as $A \leq A''$. From the definition of Refine, we have:

  $$\begin{aligned} A' \equiv& A[\kappa_c \mapsto A(\kappa) \cap Q'] && \text{(c)} \\ Q' \equiv& \{e \mid e \in \mathbb{Q}, \mathsf{Shape}(\Gamma); \nu : B \vdash e : \texttt{bool}\} && \text{(d)} \end{aligned}$$

  for some κ_c. We shall show that for any κ, we have $A''(\kappa) \subseteq A'(\kappa)$.

  Consider $\kappa \neq \kappa_c$. From (b)

  $$A''(\kappa) \subseteq A(\kappa) = A'(\kappa)$$

  Consider $\kappa = \kappa_c$. Now (a) implies that

  $$\begin{aligned} \text{for each} \quad & e \in A''(\kappa_c),\ e \in \mathbb{Q}, \text{and} \mathsf{Shape}(\Gamma); \nu : B \vdash e : \texttt{bool} \\ \textit{i.e.,}\text{for each} \quad & e \in A''(\kappa_c),\ e \in Q' \end{aligned}$$

  From (b),

  $$A''(\kappa_c) \subseteq A(\kappa_c) \cap Q' = A'(\kappa_c)$$

- case $c \equiv \Gamma \vdash b \Rightarrow \theta \cdot \kappa_c$: We have,

  $$\begin{aligned} &A''c \text{ is valid} && \text{(a)} \\ &\forall \kappa. A''(\kappa) \subseteq A(\kappa) && \text{(b)} \end{aligned}$$

as $A \leq A''$. From the definition of Refine, we have:

$$\begin{aligned} A' \equiv& A[\kappa_c \mapsto A(\kappa) \cap Q'] \\ Q' \equiv& \{e \mid e \in \mathbb{Q} \text{ and } [\![A\Gamma]\!] \wedge [\![Ab]\!] \Rightarrow \theta e\}d \end{aligned} \qquad \text{(c)}$$

We shall show that for any κ, we have $A''(\kappa) \subseteq A'(\kappa)$.

Consider $\kappa \neq \kappa_c$. From (b)

$$A''(\kappa) \subseteq A(\kappa) = A'(\kappa)$$

Consider $\kappa = \kappa_c$. Now (a) implies that

$$\text{for each} \quad e \in A''(\kappa_c),\ e \in \mathbb{Q}, \text{and} [\![A''\Gamma)]\!] \wedge [\![A''b]\!] \Rightarrow \theta e$$

As $A \leq A''$, from Lemma 19 we have

$$\begin{aligned} [\![A\Gamma]\!] \Rightarrow& [\![A''\Gamma]\!] \\ [\![Ab]\!] \Rightarrow& [\![A''b]\!] \\ \textit{i.e.,} \quad [\![A\Gamma]\!] \wedge [\![Ab]\!] \Rightarrow& [\![A''\Gamma]\!] \wedge [\![A''b]\!] \Rightarrow \end{aligned}$$

and so,

$$\begin{aligned} \text{for each} \quad & e \in A''(\kappa_c),\ e \in \mathbb{Q}, \text{and} [\![A''\Gamma)]\!] \wedge [\![A''b]\!] \Rightarrow \theta e \\ \textit{i.e.,}\text{for each} \quad & e \in A''(\kappa_c),\ e \in Q' \\ \textit{i.e.,} A''(\kappa_c) \subseteq& Q' \end{aligned}$$

From (b),

$$A''(\kappa_c) \subseteq A(\kappa_c) \cap Q' = A'(\kappa_c)$$

□

Theorem 8. (Constraint Solving) *For every set of constraints C and qualifiers $\mathbb{Q}$,*

1. $\mathsf{Solve}(C, \lambda\kappa.\mathbb{Q})$ *terminates,*
2. *if* $\mathsf{Solve}(C, \lambda\kappa.\mathbb{Q})$ *returns A then A is the minimum solution for C over $\mathbb{Q}$,*
3. *if* $\mathsf{Solve}(C, \lambda\kappa.\mathbb{Q})$ *returns* ***Failure*** *then C has no solution over $\mathbb{Q}$.*

Proof. 1. To prove that Solve terminates, we associate a well-founded measure with solutions and show that in each iteration of the refinement loop, the potential of the solution A strictly decreases. Let,

$$\mu A \equiv \sum_{\kappa} \|A(\kappa)\|$$

where $\|A(\kappa)\|$ is the cardinality of $A(\kappa)$. Consider any loop iteration. From the definition of Solve, we know that the constraint c chosen for refinement is such that Ac is not valid. Let A' be the *refined* solution returned by calling $\mathsf{Solve}(A, c)$. By Lemma 20,

$$\begin{aligned} A \leq& A' \\ A \neq& A' \end{aligned}$$

Thus, by the definition of $\leq$, we have $\mu A' < \mu A$, *i.e.,* the potential of the solution strictly decreases in the iteration. As the potential is non-negative, Refine must terminate.

2. Assume that Solve returns a solution A. Then AC is valid (as otherwise the loop would finish), and so by Theorem 7, C has a minimum solution A^* over $\mathbb{Q}$. To prove that the returned solution is the same as A^*, we show by induction over n that after n iterations of the loop in Solve, the solution $A \leq A^*$. In the base case, A has the initial assignment mapping each liquid type variable to $\mathbb{Q}$ and thus A is less than *every* solution over $\mathbb{Q}$ including A^*. Let us assume the induction hypothesis, that after n iterations, $A \leq A^*$. The value of A after $n+1$ iterations is $\mathsf{Refine}(A, c)$ where A is the solution after n iterations. As A^*c is valid (A^* is the minimum solution for C over $\mathbb{Q}$, and $A \leq A^*$ (by the induction hypothesis), from Lemma 20, we deduce that $\mathsf{Refine}(A, c) \leq A^*$, and so after $n+1$ iterations, $A \leq A^*$. Thus, if A is the solution returned by Solve, then $A \leq A^*$. As A^* is the minimum solution over $\mathbb{Q}$, and A is a valid solution, $A = A^*$.

3. Suppose that Solve fails,but that there is a valid solution for C over $\mathbb{Q}$. Then, there exists a minimum solution A^* over $\mathbb{Q}$, and by the reasoning above, at each iteration, $A \leq A^*$. By the definition of Refine, the outcome **Failure** only happens when A and the constraint c are such that $c \equiv \Gamma \vdash_{\mathbb{Q}} b \Rightarrow e$, where b is either an expression or a liquid variable with pending substitutions, and:

$$[\![A\Gamma]\!] \wedge [\![Ab]\!] \not\Rightarrow [\![e]\!]$$

   Now, as $A \leq A^*$, by Lemma 19, we have

$$[\![A\Gamma]\!] \Rightarrow [\![A^*\Gamma]\!]$$
$$[\![Ab]\!] \Rightarrow [\![A^*b]\!]$$

   and therefore,

$$[\![A^*\Gamma]\!] \wedge [\![A^*b]\!] \not\Rightarrow [\![e]\!]$$

   i.e., A^*c is not valid, which is a contradiction, and so there is no valid solution for C over $\mathbb{Q}$.

□

Proof. (of Theorem 2 Immediate corollary of Theorems 6 and 8.

□