Counterexample-Guided Refinement for Higher-Order Programs

Ranjit Jhala[1] Rupak Majumdar[2]

[1]UC San Diego [2]UC Los Angeles

Abstract. Counterexample-guided Abstraction Refinement (CEGAR) is a popular technique to automatically find an abstract domain tailored to proving a safety property for a particular program. While CEGAR has been well-studied and applied to software model checking for first-order imperative programs, its potential for the type-based analysis of higher-order functional programs has not been studied before. We extend CEGAR to dependent type inference for higher-order functional programs, by formalizing the notion of counterexample in this setting, and presenting an algorithm that analyzes a counterexample to determine if it is genuine, *i.e.,* shows that the program is unsafe, or spurious, arising due to an imprecise choice of abstract domain for dependent types. In the latter case, we show how Craig Interpolation can be used to refine the abstract domain in order to refute the counterexample, thereby allowing the use of CEGAR to iteratively compute an abstract domain that suffices to verify the safety of functional programs.

1 Introduction

Counterexample-guided abstraction refinement (CEGAR) [13,3] is a popular paradigm for automatically refining the precision of a program analysis. In this paradigm one uses a coarse abstract domain to build a *finite-state* (or *push-down*) automaton that overapproximates the behaviors of the infinite-state program, and uses (push-down) model checking to verify the given property on the automaton. The overapproximation is iteratively refined using abstract *counterexamples* produced by the model checker, until either an abstract domain is found whose resulting automaton is safe, or an abstract counterexample is found that corresponds to a genuine property violation.

Unfortunately, the success of the CEGAR paradigm has been limited to the setting of *first-order*, low-level programs (e.g., device drivers), whose semantics can be modeled using (push-down) automata [1,9,2,10,14]. High-level software written in modern languages make heavy use of *higher-order functions* to modularize code. Examples include functional languages where higher-order functions are explicit (e.g. Lisp, ML, and Haskell), imperative dynamic languages with support for closures (e.g. JavaScript, Python and Ruby), imperative static languages with support for classes and dynamic dispatch (e.g. Java, C#, C++), and finally, even large systems written in C, such as the Linux kernel and the Apache Web server, make heavy use of higher-order programming via function pointers, to implement event handlers, asynchronous callbacks and lightweight objects.

In this paper, we show how the CEGAR paradigm can be freed from reliance upon push-down modelling and thus be applied to the setting of higher-order programs. In particular, we show how CEGAR can be applied to infer *dependent types*, a generalization of invariants,

pre- and post- conditions, for a core higher-order functional language. To achieve this goal, we work in the framework of liquid types [15], a dependent type system in which ML types are refined by Boolean predicates. While [15] shows a type inference algorithm which, given a program and a set of logical qualifiers, automatically infers most precise refinement types over the given set of logical qualifiers for each expression, the question of automatically inferring logical qualifiers precise enough to ensure type safety was left open. In this paper, we show how to apply CEGAR to automatically find logical qualifiers that suffice to prove a given set of assertions safe. To this end, we show how to extend counterexample analysis to the setting of higher-order functional programs, by showing how additional logical qualifiers can be automatically inferred by considering failed runs of the liquid type inference algorithm when using fewer logical qualifiers.

In the higher-order setting, the CEGAR loop proceeds as follows. To verify that a closed program e is safe, we start with a set of qualifiers $\mathbb{Q}$ (which could initially be empty) and an *unroll depth* i (which could initially be 0). In each iteration of the loop, we apply the abstract liquid type inference algorithm over the qualifiers $\mathbb{Q}$. This algorithm proves safety by computing the strongest refinements that are conjunctions of qualifiers from $\mathbb{Q}$ which overapproximate the set of values for each *sub-expression*. If these refinements are strong enough to prove safety, then we exit the loop and return SAFE. If instead the program cannot be proven safe using $\mathbb{Q}$, then either (a) the program is indeed unsafe, or, (b) the qualifiers are insufficient. To determine whether (a) or (b) holds, we unroll all recursive functions to a depth of i, to obtain a *counterexample*, a bounded recursion-free program e_i. If the counterexample analysis determines that (a) e_i is unsafe, then the counterexample is *genuine* and e is also unsafe and hence we exit the loop return UNSAFE, (b) e_i is safe, then the counterexample is *spurious*, and our analysis returns a set of qualifiers $\mathbb{Q}_i$ that suffice to refute the counterexample, *i.e.,* prove e_i safe. In this case, these qualifers are added to $\mathbb{Q}$, the unrolling depth i is incremented, and the loop is repeated.

The key technical contribution of this paper is an algorithm that generalizes the counterexample analysis for imperative programs [8] to the setting of higher-order functional programs. Our main insight is that subtyping constraints in the functional setting play a role analogous to data flow in the imperative setting. Our counterexample analysis takes as input a recursion free counterexample program, and computes, in the following four steps, a set of qualifiers using which appropriate type refinements can be found that refute the counterexample.

Step 1. Renaming. In the first step, we generate a system of constraints over type variables κ which represent the unknown refinements, for each sub-expression of the counterexample. The constraints and program variables are cloned and renamed so that each distinct value held by a variable, *i.e.,* each distinct binding (analogous to each distinct assignment in the imperative case) is represented by a distinct renamed version of the original variable.

Step 2. Trace Formula Generation. In the second step, renamed variables are used to construct a logical formula called the *trace formula*, which is satisfiable iff the counterexample is genuine, *i.e.,* the recursion-free expression is unsafe.

Step 3. Formula Partitioning. In the third step, for each type variable κ we partition the trace formula into two formulas. The first captures the relationships between the values that are used to *define* the sub-expression described by the liquid variable κ. The second specifies how the sub-expression is *used* by the remainder of the computation.

```
1: let rec sum n =
2:   if n <= 0 then 0 else
3:     let t = sum (n-1) in
4:     t + n
5:
6: let a = read_int ()
7: let b = sum a
8: assert (b >= a)
```

Fig. 1. Example

```
let sum0 n = diverge () in
let sum1 n =
  if n <= 0 then 0 else
    let t = sum0 (n-1) in
    t + n in
let a = read_int () in
let b = sum1 a in
assert (b >= a)
```

Fig. 2. Counterexample at depth 1

Step 4. Interpolation. In the fourth step, we use the type variable κ's formula partition, to compute a *Craig Interpolant* [4], a formula that overapproximates the values that define κ in a manner precise enough to prove safe the uses of κ. With a few syntactic transformations, the set of interpolants can be converted to a set of qualifers that suffice to refute the counterexample, *i.e.,* which suffice to prove the safety of the bounded recursion-free program.

Together, our four-step algorithm cleanly combines the constraint-based formulation of dependent type inference with interpolation thereby enabling, for the first time, the application of the CEGAR paradigm to higher-order programming languages. In the rest of the paper, we formalize the syntax and semantics of the core of ML within which we work (Section 2), briefly recall the liquid type inference algorithm from [15] (Section 3), and formalize and explain the counterexample analysis algorithm AnalyzeCEX.

2 Overview

We start with an informal description of safety verification using CEGAR-based dependent type inference. Consider the program in Figure 1 which shows a function `sum` that takes an argument `n` and returns a sum of the numbers $1, \ldots, n$. Suppose that we would like to verify that when invoked on an arbitrary integer `a` read from the input, the result of `sum a`, which is bound to the variable `b` is greater than `a`. This safety property is captured by the `assert` on line 8. Using just the ML type system, we can only infer: `sum` :: `int`→`int` which is too weak to statically verify the assertion.

Dependent Types. One way to prove the assertion is to use a stronger, *dependent* type system which enriches the ML types using refinement predicates. In such a system, the dependent type for `assert` is

$$\texttt{b:}\{\nu\texttt{:bool} \mid \nu = \mathit{true}\}\texttt{→unit}$$

stating the requirement that `assert` takes as input a parameter `b`, which must be a boolean value that is equal to `true`, and returns as output the unit value. In this system, the verification of safety properties like the assert, reduces to finding appropriate dependent types for all the program sub-expressions, using which each to call to assert, and hence, the entire program type checks.

Liquid Types. We work in the framework of *liquid types* [15], where the ML types of (sub-)expressions are refined using boolean combinations over a given set of predicates (called *logical qualifiers*) over program variables and the special value variable ν that represents the type being defined. For example, given the logical qualifier $\{\mathtt{n} \leq \nu\}$, the liquid type inference algorithm infers that:

$$\mathtt{sum} :: \mathtt{n}{:}\mathtt{int} \rightarrow \{\nu{:}\mathtt{int} \mid \mathtt{n} \leq \nu\}$$

and hence that, $\mathtt{b}:: \{\nu{:}\mathtt{int} \mid \mathtt{a} \leq \nu\}$, thereby typechecking the call to `assert`, *i.e.,* proving that the assert is never violated.

CEGAR Framework. While dependent type systems offer precise program reasoning, in practice, it is often tedious to figure out exactly what qualifiers are required to verify a given program. A similar problem arises in the safety verification of imperative programs, where the programmer has to specify an abstract domain over which the analysis operates. CEGAR has proven to be crucial in that setting, by enabling verifiers to automatically *infer* new predicates from failed runs of the model checker. Algorithm CEGTypeInfer shown in Figure 5 describes the overall CEGAR-based dependent type inference framework. The input to the algorithm is an environment Γ, and an expression e. The CEGAR loop starts by initializing the set of qualifiers $\mathbb{Q}$ with the empty set, and the unroll depth i to 1. In each iteration of the loop, the algorithm first type checks e by running liquid type inference using the qualifiers $\mathbb{Q}$ (procedure Infer). If this succeeds, the algorithm returns SAFE. Otherwise, a counterexample is computed by unrolling i times, each recursive definition in e (procedure Unroll). Next, counterexample analysis is performed on the resulting, non-recursive program (procedure AnalyzeCEX). Counterexample analysis either returns no qualifiers, indicating that the unrolled program, and hence the original program e is not safe (*i.e.,* no valid dependent type derivation exists), or returns a new set of logical qualifiers. In the former case, the algorith returns UNSAFE, and in the latter case, the CEGAR loop continues, after adding the new qualifiers to $\mathbb{Q}$ and incrementing the unroll depth.

Suppose we invoked Algorithm CEGTypeInfer on the empty environment and the program from Figure 1. In the first iteration, the type inference over the empty set of qualifiers would fail. By unrolling the recursion to a depth of 1 we get the recursion-free counterexample program shown in Figure 2. When the counterexample analysis procedure AnalyzeCEX is invoked with this counterexample, it returns the set of qualifiers $\{n \leq \nu\}$ which suffice to refute the counterexample, *i.e.,* prove the recursion-free counterexample safe. These qualifiers are added to $\mathbb{Q}$, the unroll depth is incremented and the loop is repeated. In the next iteration, the algorithm Infer is able to find a valid type derivation for the entire program using $\mathbb{Q}$ and thus, CEGTypeInfer returns SAFE indicating that the `assert` in line 8 never fails.

3 Syntax, Semantics, Types

We present our refinement algorithm on a core language λ_{L}, which is a simply-typed λ-calculus with `let`- and `let rec`- bindings.

Syntax. Figure 3 shows the syntax of expressions and types for λ_{L}. An *expression* is either a variable, a special constant (including arithmetic constants and arithmetic operators, and primitive operators described below), λ-abstractions, applications (labeled with label ℓ), the special constructs *conditionals* `if` $\cdot$ `then` $\cdot$ `else` $\cdot$ and *let-bindings* `let` and `let rec`.

e	::=		*Expressions:*
		x	variable
		$\texttt{c}$	constant
		$\lambda x.e$	abstraction
		$x^{\ell}x$	application
		$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	if-then-else
		$\texttt{let}\ x\ =\ e\ \texttt{in}\ e$	let-binding
		$\texttt{let rec}\ f\ =\ \lambda x.e\ \texttt{in}\ e$	letrec-binding
Q	::=		*Liquid Refinements*
		true	true
		q	qualifier in $\mathbb{Q}$
		$Q \wedge Q$	qualifier conjunction
B	::=		*Base Types:*
		$\texttt{int}$	base type, integers
		$\texttt{bool}$	base type, booleans
$\mathbb{T}(\mathbb{B})$	::=		*Type Skeletons:*
		$\{\nu : B \mid \mathbb{B}\}$	base
		$x{:}\mathbb{T}(\mathbb{B}){\rightarrow}\mathbb{T}(\mathbb{B})$	function
		α	type variable
$\mathbb{S}(\mathbb{B})$	::=		*Type Schema Skeletons:*
		$\mathbb{T}(\mathbb{B})$	monotype
		$\forall\alpha.\mathbb{S}(\mathbb{B})$	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*

Fig. 3. Syntax of expressions and types

In what follows, we assume expressions to be in A-normal form [6], where each intermediate subexpression is let-bound to a variable, and each application is between two variables. Further, we assume that each application in the program is labelled with a unique label ℓ, written ${x_1}^{\ell}x_2$. We denote by $\mathbb{L}$ the set of labels annotating the applications.

Semantics. Let $\hookrightarrow$ denote the single evaluation step relation for λ_{L} expressions defined in the standard way (see [16] for details). We write $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$.

For an expression e, we say e is *stuck* if (a) e is not a value (a constant or a λ-term), but (b) there is no e' such that $e \hookrightarrow e'$. An expression e can *get stuck* if $e \hookrightarrow^* e'$ and e' is stuck.

Types. We use B to denote base types such as `unit`, `bool`, and `int`. λ_{L} has a system of *refined* base types, dependent function types.

- A *base refinement* is an expression of the form $\{\nu{:}B \mid e\}$ where ν is a special *value* variable, B is a base type, and e is a boolean-valued expression called a *refinement predicate*, which constrains the values populating the type. Intuitively, the base refinement $\{\nu{:}B \mid e\}$ denotes the subset of values c of type B such that $[c/\nu]e$ holds. With this understanding, B is an abbreviation for $\{\nu{:}B \mid \mathit{true}\}$.
- A *dependent function type* $x : T_1 \rightarrow T_2$ consists of a domain type T_1, a range type T_2, and the variable x can occur in the refinement predicates in T_2.

– A *logical qualifier* is a Boolean-valued expression over the program variables, the special value variable ν which does not appear in the program, and the placeholder variable $\star$ that can be instantiated with program variables. A logical qualifier q *matches* a logical qualifier q' if replacing a subset of occurrences of $\star$ in q' with program variables produces q. For a set of logical qualifiers $\mathbb{Q}$, let $\mathbb{Q}^\star$ be the set of all logical qualifiers not containing $\star$ that matches some logical qualifier in $\mathbb{Q}$. A *liquid type* over $\mathbb{Q}$ is a dependent type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We omit $\mathbb{Q}$ when it is clear from the context.
– An *environment* Γ is a sequence of *type bindings* $x{:}S$ for variable x and (dependent) type schema S, and *guard predicates* e (of boolean type).

Notation. When the base type B is clear from the context, we abbreviate $\{\nu{:}B \mid \kappa\}$ as κ, where κ is a liquid type variable, and $\{\nu{:}B \mid e\}$ as $\{e\}$ when e is a refinement predicate. In the following, we use τ and σ for ML types and schemas, T and S for dependent types and schemas, and $\hat{T}$, $\hat{S}$ for liquid types and schemas.

Well-formedness. A type (schema) S is *well-formed* with respect to an environment Γ, written $\Gamma \vdash S$, if all free variables appearing in the refinement predicates of the type are bound in Γ. An environment Γ is *well-formed*, written $\vdash \Gamma$, if whenever $\Gamma \equiv \Gamma_0;\ x : S;\ \Gamma'$, the type S is well-formed under Γ_0, *i.e.,* the free variables appearing in the dependent type schema S are bound in the prefix Γ_0. The rules describing well-formed types and environments are standard and given in Figure 7 in the Appendix.

Constants. We assume that λ_{L} contains certain constants c built-in, each constant with a type $ty(c)$ that precisely captures its semantics. In what follows, we shall assume the following constants (including basic constants corresponding to integers and Boolean values and primitive operations on the basic constants):

$$
\begin{aligned}
\mathit{true} &:: \{\nu{:}\mathtt{bool} \mid \nu\} \\
\Leftrightarrow &:: x{:}\mathtt{bool}{\to}y{:}\mathtt{bool}{\to}\{\nu{:}\mathtt{bool} \mid \nu \Leftrightarrow (x \Leftrightarrow y)\} \\
c \in \mathtt{int} &:: \{\nu{:}\mathtt{int} \mid \nu = c\} \\
= &:: x{:}\mathtt{int}{\to}y{:}\mathtt{int}{\to}\{\nu{:}\mathtt{bool} \mid \nu \Leftrightarrow (x = y)\} \\
+ &:: x{:}\mathtt{int}{\to}y{:}\mathtt{int}{\to}\{\nu{:}\mathtt{bool} \mid \nu = (x + y)\} \\
\mathtt{assert} &:: \{\nu{:}\mathtt{bool} \mid \nu\}{\to}\mathtt{unit} \\
\mathtt{diverge}_\tau &:: \mathtt{unit}{\to}\tau
\end{aligned}
$$

With abuse of notation, in certain cases, we have used the same symbol for the operation (e.g., +), and its semantics (the integer addition operation). For clarity, we use infix notation for constants like + and $\Leftrightarrow$.

Dependent Type System. We can design an exact dependent type system for λ_{L} programs that types a program iff it does not get stuck. The type derivation rules are shown in Figure 8 in the Appendix. Notice that this system is "exact" in the sense that the base-level subsumption checks require an "exact" form of implication, which is defined using the operational semantics of the language. The following preservation and progress theorem [16] relates the static and dynamic semantics. Informally, it states that if e is a (closed) expression that is well-typed in the empty environment, then no primitive operation gets stuck during the course of evaluating e.

Theorem 1. *(Soundness) Let Γ be an environment and e an expression. If $\Gamma \vdash e : S$ for some S, then (a) if $e \hookrightarrow e'$ then $\Gamma \vdash e' : S$, and (b) if e is not a value, then there exists e' such that $e \hookrightarrow e'$.*

As the exact type system is undecidable due to the exact implication check, we look for conservative overapproximations of the type system.

4 Abstract Type Inference

The *liquid type inference algorithm* [15] performs an abstract type inference that ensures that programs typable using the algorithm are guaranteed to be typable in the exact type system. It takes as input a type environment, an expression, and a set of qualifiers $\mathbb{Q}$, and generates type constraints from e such that if the constraints are satisfiable, then $\Gamma \vdash e : S$ for some S which can be obtained from the constraint solving algorithm. We recall the algorithm from [15].

4.1 ML Types and Templates

In the liquid type inference algorithm, the inferred dependent types for all subexpressions are *refinements* of their ML types.

ML Type Inference Oracle. Let HM be an ML type inference oracle, which takes an ML type environment Γ and an expression e and returns the ML type (schema) σ if and only if, using the classical ML type derivation rules [5], there exists a derivation $\Gamma \vdash e : \sigma$. The *shape* of a dependent type schema S, denoted $\mathsf{Shape}(S)$, is the ML type schema obtained by replacing all refinement predicates with *true*. We extend Shape to environments Γ by applying Shape to each type binding and eliminating each guard in Γ. Notice that the dependent type derivation rules are refinements of the ML type derivation rules. That is, if $\Gamma \vdash e : S$ then $\mathsf{HM}(\mathsf{Shape}(\Gamma), e) = \mathsf{Shape}(S)$.

Templates. Let $\mathbb{K}$ be a set of *liquid type variables* used to represent unknown type refinement predicates. Let $\mathbb{L}$ be a set of labels. A *tagged variable* $\langle x, l\rangle$ is a pair consisting of a program variable x and a string $l \in \mathbb{L}^*$. Similarly, a *tagged liquid type variable* $\langle \kappa, l\rangle$ is a pair consisting of a liquid type variable $\kappa \in \mathbb{K}$ and a string $l \in \mathbb{L}^*$. A variable x (resp. liquid type variable κ) is an abbreviation for $\langle x, \epsilon\rangle$ (resp. $\langle \kappa, \epsilon\rangle$). For two strings $l_1, l_2 \in \mathbb{L}^*$, we write $l_1 \preceq l_2$ if l_1 is a prefix of l_2. A *template* F is a dependent type schema described as:

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

That is, a template is a dependent type where some of the refinement predicates are replaced with tagged liquid type variables. with *pending substitutions*. A *template environment* is a map Γ from variables to templates.

4.2 Constraint-Based Algorithm

The liquid type inference algorithm is based on constraint generation and solving (the algorithm is shown in Figure 9 of the appendix. The algorithm generates two kinds of constraints.

Well-formedness Constraints. A *well-formedness constraint* is of the form $\Gamma \vdash F$, where Γ is a template environment and F is a template. Well-formedness constraints ensure that

```
let dinc y = y + 1 in let inc x = dinc^0 x in
let a = inc^1 2 in let b = inc^2 3 in let c = a + b in
assert (c > 0)
```

Fig. 4. Example program

the types inferred for each subexpression are over program variables that are in scope at the subexpression.

Subtyping Constraints. A *subtyping constraint* is of the form $\Gamma \vdash F_1 <: F_2$ where Γ is a template environment, and F_1 and F_2 are two templates of the same shape. Subtyping constraints ensure that types inferred for each subexpression can be combined using the subsumption rule to yield a valid type derivation.

Context Tagging. For an expression e and $l \in \mathbb{L}^*$, define the function $\mathsf{Tag}(e, l)$ which replaces every tagged variable $\langle x, l' \rangle$ (other than ν) in e with the tagged variable $\langle x, l' \cdot l \rangle$, where $l' \cdot l$ is the concatenation of l' and l. Similarly, for a template F, we define $\mathsf{Tag}(F, l)$ as the function which replaces every tagged type variable $\langle \kappa, l' \rangle$ with $\langle \kappa, l' \cdot l \rangle$. We lift Tag to constraints in the natural way. Dually, for an expression e, define the function $\mathsf{Untag}(e)$ which replaces every tagged variable $\langle x, l \rangle$ in e with x. Similarly, for a template F, we define $\mathsf{Untag}(F)$ as the function which replaces every tagged type variable $\langle \kappa, l \rangle$ with κ. Finally, we lift Untag to constraints in the natural way.

The constraint generation procedure Cons (Figure 9 in the Appendix) differs from the constraint generation procedure in liquid type inference [15] in that each constraint generated by an application is tagged with the unique label corresponding to the application. These tags are ignored during the abstract type inference phase, but are used for qualifer discovery.

Example 1. Consider the example program shown in Figure 4. Assuming $\texttt{dinc} : y : \kappa_3 \to \kappa_4$ and $\texttt{inc} : x : \kappa_1 \to \kappa_2$. For this program, the following subtyping constraints are generated.

$$y{:}\kappa_3 \vdash \{\nu = y + 1\} <: \kappa_4 \quad (1)$$
$$x{:}\kappa_1 \vdash \langle \kappa_4, 0 \rangle <: \kappa_2 \quad (2)$$
$$x{:}\kappa_1 \vdash \{\nu = x\} <: \langle \kappa_3, 0 \rangle \quad (3)$$
$$\emptyset \vdash \{\nu = 2\} <: \langle \kappa_1, 1 \rangle \quad (4)$$
$$\emptyset \vdash \{\nu = 3\} <: \langle \kappa_1, 2 \rangle \quad (5)$$
$$a{:}[2/x]\langle \kappa_2, 1 \rangle;\ b{:}[3/x]\langle \kappa_2, 2 \rangle;\ c{:}\{\nu = a + b\} \vdash \{\nu = c > 0\} <: \{\nu = \mathit{true}\} \quad (6)$$

Constraint Solving The constraints are solved using a two-step algorithm to assign liquid types to all variables κ such that all constraints are satisfied.

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

Algorithm Infer shown in Figure 9 in the Appendix describes the two-step algorithm that finds an assignment over $\mathbb{Q}$ that satisfies the constraints. In the first step, we use the well-formedness and subtyping rules to split the complex constraints, which may contain function

```
CEGTypeInfer(Γ, e) =
  ℚ ← ∅
  i ← 0
  while true do
    if Infer(Γ, e, ℚ) = T then
      return SAFE
    e′ ← Unroll(e, i)
    ℚ′ ← AnalyzeCEX(Γ, e′)
    if ℚ′ is empty then return UNSAFE
    ℚ ← ℚ ∪ ℚ′
    i ← i + 1
```

```
AnalyzeCEX(Γ, e) =
  C ← Cons(Γ, e)
  foreach assertion constraint c in C do
    if TF(c, C) is satisfiable then return ∅
    C′ ← Clone(c, C)
    ℚ ← ∅
    foreach κ in C′ do
      (φ⁻κ, φ⁺κ) ← Partition(κ, c, C′)
      ℚ ← ℚ ∪ renamed clauses of ITP(φ⁻κ, φ⁺κ)
  return ℚ
```

Fig. 5. (a) CEGAR-based Dependent Type Inference and (b) Algorithm AnalyzeCEX

types, into simple constraints over variables with pending substitutions. In the second step, we iteratively weaken a trivial assignment, in which each liquid type variable is assigned the conjunction of all logical qualifiers, until we find the least fixpoint solution for all the simplified constraints or determine that the constraints have no solution.

Theorem 2. *[Liquid Type Inference] For every environment Γ, expression e, and logical qualifiers $\mathbb{Q}$,* $\mathsf{Infer}(\Gamma, e, \mathbb{Q})$ *terminates, and if* $\mathsf{Infer}(\Gamma, e, \mathbb{Q}) = S$ *then* $\Gamma \vdash e : S$.

In case $\mathsf{Infer}(\Gamma, e, \mathbb{Q})$ returns **Failure**, there are two possibilities: first, there is an actual error in the program, or second, the set of qualifiers $\mathbb{Q}$ is not strong enough to show well-typedness, but there may be a $\mathbb{Q}' \supseteq \mathbb{Q}$ such that $\mathsf{Infer}(\Gamma, e, \mathbb{Q}')$ returns some S.

5 Counterexample Refinement

We now describe the technical core of the counterexample-guided type inference algorithm – the algorithm AnalyzeCEX which takes a version of the (possibly recursive) program unrolled to a finite depth i, and determines whether this bounded unrolling is safe, and if so, finds a set of qualifiers that suffice to prove the unrolled program safe.

Notation. A *recursion-free* expression is a λ_{L}-expression which does not have a `let rec` subexpression. In this section we assume a fixed recursion-free expression e. An *assertion constraint* is a constraint of the form $\Gamma \vdash F <: \{e\}$, where F is a liquid type variable or a refinement. For a constraint $c \doteq \Gamma \vdash F$ (resp., $c \doteq \Gamma \vdash F_1 <: F_2$), we define $\mathsf{lhs}(c) = \Gamma;\ F_1$ and $\mathsf{rhs}(c) = F$ (resp., $\mathsf{rhs}(c) = F_2$). For a template environment Γ, we define $vars(\Gamma)$, the set of type variables in Γ, inductively as $vars(\emptyset) = \emptyset$, $vars(\Gamma;\ e) = vars(\Gamma)$, $vars(\Gamma;\ x : \{\nu \mid e\}) = vars(\Gamma)$, and $vars(\Gamma;\ x : \kappa) = \{\kappa\} \cup vars(\Gamma)$. Finally, we write $vars(\{\nu \mid e\}) = \emptyset$ and $vars(\kappa) = \{\kappa\}$. We extend $vars$ to sets in the natural way. For a constraint $c \doteq \Gamma \vdash t$, we write $vars(c)$ for $vars(\mathsf{lhs}(c)) \cup vars(\mathsf{rhs}(c))$.

Algorithm AnalyzeCEX shown in Figure 5(b) finds the qualifiers needed to prove a recursion-free program safe, using the four steps: renaming, trace formula generation, formula partitioning, and interpolation. We now describe each step in detail.

Step 1. Renaming. In the first step, we clone and rename each type and program variable in order to get a unique variable for each binding, that is, for each calling context. In the

$\mathsf{Clone}(c, C) =$
  $C' \leftarrow \{c\}$; $C_{old} \leftarrow \emptyset$
  while $C' \neq C_{old}$
    $C_{old} \leftarrow C'$
    pick $c \in C, c' \in C'$ *s.t.*
      exists $\langle \kappa, l' \rangle \in vars(\mathsf{lhs}(c'))$,
        $\langle \kappa, l \rangle \in vars(\mathsf{rhs}(c)) : l \preceq l'$
    $C' \leftarrow C' \cup \{\mathsf{Tag}(c, l' - l)\}$
  return C'

$\mathsf{Partition}(\kappa, c, C) =$
  let $\Gamma \vdash \kappa$ be the well-formedness constraint for κ
  $C_\kappa \leftarrow \emptyset \quad W \leftarrow \{\kappa\}$
  while $W \neq \emptyset$
    pick and **remove** $\kappa' \in W$
    foreach $c \in C$ *s.t.* $\kappa' \in vars(\mathsf{rhs}(c))$
      $C_\kappa \leftarrow C_\kappa \cup \{c\}$
      $W \leftarrow W \cup \{\kappa'' \mid \kappa'' \in vars(c)\} \setminus vars(\Gamma)$
  if $c \in C_\kappa$ **then**
    $(\psi^-, \psi^+) = (\wedge\{e \mid e \in \mathsf{lhs}(c)\}, true)$
  else $(\psi^-, \psi^+) = (true, \wedge\{e \mid e \in \mathsf{lhs}(c)\})$
  $\phi_\kappa^- \leftarrow [\![C_\kappa]\!] \wedge \psi^-$ and $\phi_\kappa^+ \leftarrow [\![C \setminus C_\kappa]\!] \wedge \psi^+$
  return $(\phi_\kappa^-, \phi_\kappa^+)$

Fig. 6. Algorithms Clone and Partition

imperative setting, this step is analogous to the SSA conversion or conversion to *passive form* [7,8]. This step is carried out using Algorithm Clone shown in Figure 6. This algorithm takes as input an assertion constraint c and a set of constraints C, and constructs, by saturation, a set of renamed versions of constraints C', which contains a distinct version of each constraint of C for each of the calling contexts under which the variables of c get bound.

We can view an occurrence of a liquid variable on the RHS of a constraint as a *definition* of the variable, and an occurrence on the LHS of a constraint as a *use* of a variable. For each use of a liquid variable κ in a calling context l' (*i.e.,* for each occurrence of $\langle \kappa, l' \rangle$ in the LHS of a constraint c' in C') that has a definition in a calling context l that is a prefix of l', (*i.e.,* for each occurrence of $\langle \kappa, l \rangle$ in the RHS of an original constraint c in C), we generate a cloned version of the defining constraint c, obtained by renaming c with the remainder of the context-string $l' - l$.

Example 2. Recall the constraints C generated on the program from Figure 4 shown in Example 1. Algorithm Clone returns the following renamed constraints C' when invoked with C and the assertion constraint (6). Next to each renamed constraint, we write the pair i, j where i is the index of the defining constraint $c \in C$, and j is the index of the using constraint $c' \in C'$ to whose context the constraint c is renamed.

$$
\begin{array}{lll}
[c = 2, c' = 6] & \langle x, 1 \rangle : \langle \kappa_1, 1 \rangle \vdash [\langle x, 1 \rangle / \langle y, 01 \rangle] \langle \kappa_4, 01 \rangle <: \langle \kappa_2, 1 \rangle & (7) \\
[c = 2, c' = 6] & \langle x, 2 \rangle : \langle \kappa_1, 2 \rangle \vdash [\langle x, 2 \rangle / \langle y, 02 \rangle] \langle \kappa_4, 02 \rangle <: \langle \kappa_2, 2 \rangle & (8) \\
[c = 4, c' = 7] & \emptyset \vdash \{\nu = 2\} <: \langle \kappa_1, 1 \rangle & (9) \\
[c = 5, c' = 8] & \emptyset \vdash \{\nu = 3\} <: \langle \kappa_1, 2 \rangle & (10) \\
[c = 3, c' = 9] & \langle x, 1 \rangle : \langle \kappa_1, 1 \rangle \vdash \{\nu = \langle x, 1 \rangle\} <: \langle \kappa_3, 01 \rangle & (11) \\
[c = 3, c' = 10] & \langle x, 2 \rangle : \langle \kappa_1, 2 \rangle \vdash \{\nu = \langle x, 2 \rangle\} <: \langle \kappa_3, 02 \rangle & (12) \\
[c = 1, c' = 11] & \langle y, 01 \rangle : \langle \kappa_3, 01 \rangle \vdash \{\nu = \langle y, 01 \rangle + 1\} <: \langle \kappa_4, 01 \rangle & (13) \\
[c = 1, c' = 12] & \langle y, 02 \rangle : \langle \kappa_3, 02 \rangle \vdash \{\nu = \langle y, 02 \rangle + 1\} <: \langle \kappa_4, 02 \rangle & (14)
\end{array}
$$

Note that in the cloned constraints, each type (resp. program) variable is defined and used in exactly one calling context. For example, the type variable κ_3 (resp. `y`), representing the input type (resp. parameter) for the function `dinc` has two clones $\langle \kappa_3, 01 \rangle$ and $\langle \kappa_3, 02 \rangle$ (resp. $\langle y, 01 \rangle$ and $\langle y, 02 \rangle$), one for each of the calling contexts through which `dinc` can be invoked.

Suppose $C = \mathsf{Cons}(\emptyset, e)$ for a recursion-free closed expression e, and $c \in C$. Intuitively, Algorithm Clone terminates on the input (c, C) as there are no definition-use cycles in the generated constraints. The renamed constraints have fresh clones for each binding of a type (resp. program variable). Thus, implicitly, the renamed constraints encode the exact semantics of all the (finitely many) executions of the bounded, recursion free program.

Step 2. Trace Formula Generation. In the second step, we explicate the relationship between the cloned constraints and the program executions by constructing a logical formula called the trace formula that is satisfiable iff there is an execution of the bounded program that gets stuck at the assertion from which the constraints were cloned. The trace formula is simply the conjunction of separate constraint formulas which are generated as follows, from the individual cloned constraints.

First, with each a (tagged) liquid type variable κ, we associate a value variable ν_κ. Next, with each constraints c we associate a *constraint formula* $[\![c]\!]$:

$$
\begin{aligned}
[\![\Gamma \vdash F]\!] &\doteq \mathit{true} && \text{(Well-Formedness)}\\
[\![x{:}\{e\};\ \Gamma \vdash F_1 <: F_2]\!] &\doteq [x/\nu]e \wedge [\![\Gamma \vdash F_1 <: F_2]\!] &&\\
[\![e;\ \Gamma \vdash F_1 <: F_2]\!] &\doteq e \Rightarrow [\![\Gamma \vdash F_1 <: F_2]\!] &&\\
[\![\emptyset \vdash \theta_1 \cdot \kappa_1 <: \theta_2 \cdot \kappa_2]\!] &\doteq (\nu_{\kappa_1} = \nu_{\kappa_2}) && \text{(Subtype)}\\
[\![\emptyset \vdash \{e\} <: \theta \cdot \kappa]\!] &\doteq [\nu_\kappa/\nu]e &&\\
[\![\emptyset \vdash \theta \cdot \kappa <: \{e\}]\!] &\doteq \neg([\nu_\kappa/\nu]e) &&\\
[\![\emptyset \vdash \{e_1\} <: \{e_2\}]\!] &\doteq e_1 \wedge \neg e_2 &&
\end{aligned}
$$

We can ignore the pending substitutions in the logical encoding as the cloned constraints have a unique version for possible binding due in different calling contexts. Finally, for a set C of constraints and and assertion constraint $c \doteq \Gamma \vdash F <: \{e\}$ in C, we associate the *trace formula*:

$$
\mathsf{TF}(c, C) \doteq \bigwedge_{c' \in \mathsf{Clone}(c,C)} [\![c']\!] \wedge [\![\Gamma \vdash F <: \{e\}]\!] \wedge \bigwedge \{e \mid e \in \Gamma\}
$$

Example 3. For the cloned constraints from Example 2 and the assertion constraint (6) from Example 1, we get the trace formula:

$$
\begin{aligned}
& a = \nu_{\langle\kappa_2,1\rangle} \wedge b = \nu_{\langle\kappa_2,2\rangle} \wedge c = a + b \wedge \nu_{\langle\kappa_2,1\rangle} + \nu_{\langle\kappa_2,2\rangle} \leq 0\\
\wedge\ & \langle x,1\rangle = \nu_{\langle\kappa_1,1\rangle} \wedge \nu_{\langle\kappa_4,01\rangle} = \nu_{\langle\kappa_2,1\rangle}\\
\wedge\ & \langle x,2\rangle = \nu_{\langle\kappa_1,2\rangle} \wedge \nu_{\langle\kappa_4,02\rangle} = \nu_{\langle\kappa_2,2\rangle}\\
\wedge\ & \nu_{\langle\kappa_1,1\rangle} = 2\\
\wedge\ & \nu_{\langle\kappa_1,2\rangle} = 3\\
\wedge\ & \langle x,1\rangle = \nu_{\langle\kappa_1,1\rangle} \wedge \nu_{\langle\kappa_3,01\rangle} = \langle x,1\rangle\\
\wedge\ & \langle x,2\rangle = \nu_{\langle\kappa_1,2\rangle} \wedge \nu_{\langle\kappa_3,02\rangle} = \langle x,2\rangle\\
\wedge\ & \langle y,01\rangle = \nu_{\langle\kappa_3,01\rangle} \wedge \nu_{\langle\kappa_4,01\rangle} = \langle y,01\rangle + 1\\
\wedge\ & \langle y,02\rangle = \nu_{\langle\kappa_3,02\rangle} \wedge \nu_{\langle\kappa_4,02\rangle} = \langle y,02\rangle + 1
\end{aligned}
$$

The first clause is from the assertion constraint, and the remaining clauses are from the remaining cloned constraints.

The connection between the operational semantics of the bounded program and the trace formula is captured by the following theorem.

Theorem 3. *Let e be a recursion-free closed expression, and $C = \mathsf{Cons}(\emptyset, e)$. Let $C_a \subseteq C$ be the set of assertion constraints in C. Then $\bigvee \{\mathsf{TF}(c, C) \mid c \in C_a\}$ is satisfiable iff e can get stuck.*

Step 3. Formula Partitioning. In the third step, for each (cloned) liquid variable κ, we partition the conjuncts of the trace formula into a pair of formulas ϕ^-_κ, and ϕ^+_κ. Intuitively, ϕ^-_κ constrains the values that *define* (*i.e.,* flow into) the liquid type represented by κ, and ϕ^+_κ constrains the values that *use* (*i.e.,* flow from) the liquid type represented by κ.

Let e be a recursion-free closed expression, $C \doteq \mathsf{Cons}(\emptyset, e)$, and $c \in C$ be an assertion constraint. Figure 6 shows the Algorithm Partition which takes as input the liquid type variable κ called the *pivot*, the current assertion constraint c, and the set of cloned constraints $C' \doteq \mathsf{Clone}(c, C)$, and returns as output a pair of formulas $(\phi^-_\kappa, \phi^+_\kappa)$ such that (1) the conjunction of the two formulas is equivalant to the trace formula $\mathsf{TF}(c, C)$, (2) the variables in ϕ^-_κ and ϕ^+_κ correspond directly to the variables in scope for the expression whose unknown liquid type is represented by the pivot κ.

Intuitively, the partition algorithm splits the set of cloned constraints into C_κ, the constraints that transitively define the pivot κ and the remaining constraints which use κ. The former set is used to generate ϕ^-_κ and the latter to generate ϕ^+_κ. To compute the set C_κ, the algorithm starts with the (unique, due to cloning) constraint that defines the pivot κ, and transitively adds those constraints that define the liquid variables that are used to define κ. In other words, the algorithm iteratively adds constraints on variables that "flow into" the variable κ, while not transitively expanding any variable that appears in the environment Γ of the well-formedness constraint on κ.

The variable W contains the liquid variables that (transitively) define κ, and is initialized with the pivot κ. In each iteration, the algorithm selects a variable κ' from W and adds to C_κ the constraint where κ' is defined. Further, it adds all the LHS variables of c', except those that appear in the environment Γ that represents the scope for κ. Finally, when all the possible transitive constraints have been added to C_κ, the procedure returns the pair of formulas ϕ^-_κ and ϕ^+_κ generated from the constraint formulas for C_κ and $C' \setminus C_\kappa$ respectively.

Example 4. [Partitioning] For the cloned constraints from Example 2 and the assertion constraint (6) from Example 1, and the pivot variable $\langle \kappa_4, 01 \rangle$, the algorithm Partition splits the constraints into constraint (14) and all the other constraints, and returns the pair of formulas $\phi^-_{\langle \kappa_4, 01 \rangle}$ which is

$$(\langle y, 01 \rangle = \nu_{\langle \kappa_3, 01 \rangle} \wedge \nu_{\langle \kappa_4, 01 \rangle} = \langle y, 01 \rangle + 1)$$

and $\phi^+_{\langle \kappa_4, 01 \rangle}$ which is the conjunction of all the remaining clauses from Example 3.

Theorem 4. *Let e be a closed recursion-free expression, $C = \mathsf{Cons}(\emptyset, e)$, and, $C' = \mathsf{Clone}(c, C)$. For any pivot κ in C', if $\Gamma \vdash \kappa$ is the well-formedness constraint for κ in C' and $(\phi^-_\kappa, \phi^+_\kappa) = \mathsf{Partition}(\kappa, c, \mathsf{Clone}(c, C))$, then:*

1. *$\phi^-_\kappa \wedge \phi^+_\kappa$ is equivalent to $\mathsf{TF}(c, C)$,*
2. *if x appears in ϕ^-_κ and ϕ^+_κ then x is bound in Γ, and,*
3. *if $\nu_{\kappa'}$ appears in ϕ^-_κ and ϕ^+_κ then κ' is either κ or is bound to some variable in Γ.*

Step 4. Interpolation. In the fourth step, we use Craig Interpolation [8] on each partition to find appropriate qualifiers for the type variable corresponding to the partition.

Given predicates (ϕ_1, ϕ_2) such that $\phi_1 \wedge \phi_2$ is unsatisfiable, an *interpolant* $\mathsf{ITP}(\phi_1, \phi_2)$ is a predicate ψ such that: (I1) the variables in ψ are the variables common to ϕ_1 and ϕ_2 (I2) $\phi_1 \Rightarrow \psi$, and (I3) $\psi \wedge \phi_2$ is unsatisfiable [4].

In the fourth step, our qualifier discovery algorithm uses interpolation to compute for each pivot κ, a set of logical qualifiers, such that when all the qualifiers are combined, they suffice to prove the (bounded) program safe. To do so, for each pivot variable κ in the cloned constraints we compute the predicate $\mathsf{ITP}(\phi^-_\kappa, \phi^+_\kappa)$ where ϕ^-_κ and ϕ^+_κ are obtained by partitioning the trace formula using κ as described in Step 3.

Note that by the properties of the partition described in Theorem 4, if the trace formula is unsatisfiable, *i.e.,* the bounded program does not get stuck, then the interpolant is guaranteed to exist. Further, property (I1) ensures that the interpolant contains variables that are in scope for κ. We replace each value variable $\nu_{\kappa'}$ in the interpolant with the unique program variable x' to which $\nu_{\kappa'}$ is bound in the well-formedness environment for the pivot κ, and replace the pivot's value variable ν_κ with ν, and break the resulting formula into its conjuncts to obtain the qualifiers Q_κ. Let $\mathbb{Q} \doteq \cup_\kappa Q_\kappa$. Property (I1) ensures that the assignment of the conjunction of Q_κ to κ satisfies the well-formedness constraint of κ. Properties (I2) and (I3) ensure that the assignment satisfies the subtyping constraints, and thus using $\mathbb{Q}$ we can infer liquid types that prove that the (bounded) does not get stuck at the assertion.

Example 5. [Interpolation] For the pair of formulas from Example 4 we have:

$$\mathsf{ITP}(\phi^-_{\langle\kappa_4,01\rangle}, \phi^+_{\langle\kappa_4,01\rangle}) \doteq (\nu_{\langle\kappa_4,01\rangle} > \nu_{\langle\kappa_3,01\rangle})$$

Note that the only variables that appear in the interpolant are the value variable corresponding to: (a) the pivot variable $\langle\kappa_4, 01\rangle$, and, (b) the (renamed) type variable which is bound to the (renamed) program variable $\langle y, 01\rangle$ in the well-formedness constraint for the pivot. Finally, by replacing the value variable of (a) the pivot with ν, and, (b) $\langle y, 01\rangle$ with y (*i.e.,* dropping the context-tags), we infer for the pivot variable κ_4, the qualifier $\nu > y$. By interpolating across the partitions induced by all the pivots, we discover the qualifers $\{\nu > x, \nu > y\}$ which suffice to prove the program safe.

The four steps are combined in Algorithm AnalyzeCEX shown in Figure 5(b). The properties of the algorithm are stated by the following theorem.

Theorem 5. *For every Γ and recursion-free expression e, let $\mathbb{Q} \doteq \mathsf{AnalyzeCEX}(\Gamma, e)$. Then*

1. *if $\mathbb{Q} = \emptyset$ then e has an execution that gets stuck,*
2. *if $\mathbb{Q} \neq \emptyset$ then there exists an S s.t. $\mathsf{Infer}(\Gamma, e, \mathbb{Q}) = S$, and hence $\Gamma \vdash e : S$.*

As in the first-order, imperative setting, liquid type checking with counterexample refinement is not guaranteed to terminate. As in the imperative setting, we can enforce completeness of the algorithm by limiting the predicate language over which interpolants are discovered [11].

6 Conclusions and Future Work

In this paper, we have shown how CEGAR can be applied to verify safety properties of programs that use modern programming abstractions like higher-order functions. It is straightforward to implement the presented algorithm for functional programs. Such an implementation would have the same benefits of the first order case – namely, it would succeed on control-dominated programs with simple linear invariants. However, most interesting verification questions on high-level software require sophisticated invariants that capture universally quantified facts over data structures. Recent work [12] shows how these universal invariants can be captured by a rich abstract domain that combines simple predicates with type skeletons. We believe our CEGAR approach can be extended to this domain and in future work we would like to develop and evaluate our algorithm in that context, thereby obtaining a CEGAR-based technique for automatically verifying high-level software.

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*. ACM, 2002.
2. 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.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00:*, LNCS 1855, pages 154–169. Springer, 2000.
4. W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.
5. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.
6. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.
7. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL 00: Principles of Programming Languages*, pages 193–205. ACM, 2000.
8. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*. ACM, 2004.
9. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
10. H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, 2005.
11. R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV 05*, LNCS, pages 39–51. Springer, 2005.
12. M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI 09: Programming Languages Design and Implementation (to appear)*. ACM, 2009.
13. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
14. K. L. McMillan. Lazy abstraction with interpolants. In *CAV 2006*, LNCS, pages 123–136. Springer-Verlag, 2006.
15. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 08: Programming Languages Design and Implementation*. ACM, 2008.
16. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. Technical report, UC San Diego, 2008.

A Typing Rules and Constraint-Based Type Inference

In this appendix, we give the typing rules for well-formed types, an "exact" dependent type system, as well as a constraint-based algorithm for liquid type inference. The first two sets of rules are standard. The constraint-based algorithm is a slight modification of [15] in which type variables are tagged with application labels.

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

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

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

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 $\boxed{\Gamma \models \rho}$

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

Fig. 7. Well-formed Dependent Types, Environments, Substitutions

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

$$\frac{\Gamma \vdash e : S_1 \qquad \Gamma \vdash S_1 <: S_2 \qquad \Gamma \vdash S_2}{\Gamma \vdash e : S_2}\ [\text{T-Sub}] \quad \frac{}{\Gamma \vdash \mathtt{c} : ty(\mathtt{c})}\ [\text{T-Const}]$$

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

$$\frac{\Gamma \vdash T \quad \Gamma;\ x{:}T \vdash e_1 : T_1}{\Gamma \vdash (\lambda x.e_1) : (x{:}T \to T_1)}\ [\text{T-Fun}] \quad \frac{\Gamma \vdash e_1 : (x{:}T_2 \to T) \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : [e_2/x]T}\ [\text{T-App}]$$

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

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

$$\frac{\Gamma, y : S_0, x : S_0 \to S_1 \vdash e_1 : S_1 \quad \Gamma;\ x{:}S_0 \to S_1 \vdash e_2 : S_2 \quad \Gamma \vdash S_2}{\Gamma \vdash \mathtt{let\ rec}\ x\ =\ \lambda y.e_1\ \mathtt{in}\ e_2 : S_2}\ [\text{T-Letrec}]$$

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

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

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

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

$$\frac{\Gamma;\ \nu{:}B \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{\nu{:}B \mid e_1\} <: \{\nu{:}B \mid e_2\}}\ [<:\text{-Base}] \quad \frac{\Gamma \vdash T_2' <: T_1' \qquad \Gamma[x \mapsto T_2'] \vdash T_1'' <: T_2''}{\Gamma \vdash x{:}T_1' \to T_1'' <: x{:}T_2' \to T_2''}\ [<:\text{-Fun}]$$

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

Fig. 8. Rules for Dependent Type Checking

$\mathsf{Cons}(\Gamma, x) =$
  if $\mathsf{HM}(\Gamma, e) = B$
  then $(\{\nu\!:\!B \mid \nu = x\}, \emptyset)$
  else $(\Gamma(x), \emptyset)$

$\mathsf{Cons}(\Gamma, \mathsf{c}) =$
  $(ty(\mathsf{c}), \emptyset)$

$\mathsf{Cons}(\Gamma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) =$
  let $F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e))$ **in**
  let $(\_, C_1) = \mathsf{Cons}(\Gamma, e_1)$ **in**
  let $(F_2, C_2) = \mathsf{Cons}(\Gamma;\ e_1, e_2)$ **in**
  let $(F_3, C_3) = \mathsf{Cons}(\Gamma;\ \neg e_1, e_3)$ **in**
  $(F, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup$
    $\{\Gamma;\ e_1 \vdash F_2 <: F\} \cup \{\Gamma;\ \neg e_1 \vdash F_3 <: F\})$

$\mathsf{Cons}(\Gamma, x_1\ ^{\ell} x_2) =$
  let $(x\!:\!F_x \to F, C_1) = \mathsf{Cons}(\Gamma, e_1)$ **in**
  let $(F'_x, C_2) = \mathsf{Cons}(\Gamma, e_2)$ **in**
  $(\langle [x_2/x]F, \ell\rangle, C_1 \cup C_2 \cup \{\Gamma \vdash F'_x <: \langle F_x, \ell\rangle\})$

$\mathsf{Cons}(\Gamma, \lambda x.e \textbf{ as } e') =$
  let $(x\!:\!F_x \to F) = \mathsf{Fresh}(\mathsf{HM}(\Gamma, e'))$ **in**
  let $(F', C) = \mathsf{Cons}(\Gamma;\ x\!:\!F_x, e)$ **in**
  $(x\!:\!F_x \to F, C \cup \{\Gamma \vdash x\!:\!F_x \to F\} \cup \{\Gamma;\ x\!:\!F_x \vdash F' <: F\})$

$\mathsf{Cons}(\Gamma, \texttt{let } x \ = \ e_1 \texttt{ in } e_2 \textbf{ as } e') =$
  let $F = \mathsf{Fresh}(\mathsf{HM}(\Gamma, e'))$ **in**
  let $(F_1, C_1) = \mathsf{Cons}(\Gamma, e_1)$ **in**
  let $(F_2, C_2) = \mathsf{Cons}(\Gamma;\ x\!:\!F_1, e_2)$ **in**
  $(F, C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma;\ x\!:\!F_1 \vdash F_2 <: F\})$

$\mathsf{Cons}(\Gamma, \texttt{let rec } x \ = \ \lambda y.e_1 \texttt{ in } e_2 \textbf{ as } e') =$
  let $F = \mathsf{Fresh}(\mathsf{HM}(\Gamma, e'))$ **in**
  let $(F_1, C_1) = \mathsf{Cons}(\Gamma, e_1)$ **in**
  let $(F_2, C_2) = \mathsf{Cons}(\Gamma;\ x\!:\!F_1, e_2)$ **in**
  $(F, C_1 \cup C_2 \cup \{\Gamma \vdash F\} \cup \{\Gamma;\ x\!:\!F_1 \vdash F_2 <: F\})$

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

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

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

Fig. 9. Abstract Type Inference Algorithm