Counterexample Refinement for Functional Programs

Ranjit Jhala[1] Rupak Majumdar[2]

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

Abstract. Counterexample refinement is a popular technique to automatically refine the abstract domain for static analysis relative to a property. While counterexample-guided abstraction refinement has been well-studied in the imperative context, and applied to software model checking for first-order imperative programs, its potential in dependent type inference of higher-order programs has not been studied before. We extend the notion of counterexample refinement to higher-order functional programs. We formalize the notion of a *type counterexample* and show how type counterexamples can be used to infer logical qualifiers in a dependent type system based on liquid types. In contrast to first-order imperative programs where counterexamples are sequences of instructions, counterexamples are directed graphs constraining type variables. We generalize the counterexample analysis procedure for imperative programs —SSA-renaming, trace formula generation, partitioning, and interpolation— to this setting.

1 Introduction

Counterexample-guided abstraction refinement (CEGAR) [3,1,9] is a popular technique for automatically refining the precision of a program analysis. The techniques starts by analyzing the program with a coarse abstract domain, and iteratively refines the domain (e.g., by adding more elements to precisely track more sets of concrete states) based on abstract counterexamples produced while analyzing the program using the current abstraction. So far, the application of CEGAR has been limited to first order, imperative programs [1,9,2,**?**]. In this paper, we extend the idea in the setting of higher-order functional programs and dependent type inference.

We work in the framework of liquid types [**?**], a dependent type system in which ML types are refined by Boolean predicates. While [**?**] 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. We complete the CEGAR loop in this paper, by extending counterexample analysis to the setting of higher order functional programs and dependent type inference, and 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.

```
let rec bs len l u =
  if l<= u then
    let m = (l+u)/2 in
    let _ = assert (m <= len) in
    if * then bs len (m+1) u
    else if * then bs len l (m-1)
         else m
  else -1

let search len =
  if (len>=0) then bs len 0 len
  else -1
```

Fig. 1. Simple binary search example

Counterexample analysis for safety model checking of imperative programs proceeds in the following way [8]. First, an abstract path in the program control flow graph is extracted from the failure of the model checker to prove the property of interest using the current abstraction. Second, the operations in the path are SSA-renamed to ensure that any variable is assigned at most once. Third, from the SSA-renamed path, a logical formula, called the *trace formula*, is extracted such that the formula is satisfiable iff the path is concretely feasible. Fourth, at each program point along the path, the trace formula is partitioned into a "before" and an "after" formula such that the only variables common to the two parts are those in scope and live at the interface. Finally, an interpolation-based procedure is used to infer new predicates such that an abstract analysis that tracks these predicates is guaranteed to not find the current counterexample path.

Our algorithm generalizes counterexample refinement for imperative programs in the following way. Imagine that the original program is recursion-free (we deal with recursion by unrolling, as in the imperative case). First, instead of a path in the control flow graph, a counterexample is now a set of constraints such that the constraint solver in the type inference algorithm does not find a valid typing. Second, we show an algorithm *clone* which takes a set of type constraints and renames type variables in the constraints, one for each constraint, in a way that each type variable is cloned for each context. ♣ explain better?♣ This generalizes the SSA-renaming step in the imperative setting. Third, we construct a trace formula from a cloned set of constraints such that the formula is satisfiable iff there is some input to the program that causes the program to "get stuck", and so no liquid typing exists. Fourth, for each type variable κ, we construct a partition of the trace formula into two parts such that the only common variables between them are those in scope for the subexpression whose type is given by (a solution to) κ. Finally, we apply interpolation-based predicate discovery to discover new logical qualifiers which ensure the removal of the current cause of inconsistency in the type inference engine.

Example We start with an informal description of counterexample-guided dependent type inference. Consider the program in Figure 1 which shows the simplified skeleton of a binary search procedure. The actual array access is replaced by a (partial) check that the access is within bounds. We use "$*$" to denote non-deterministic choice in branches where the branch condition is not important for the example. We also assume that integers are unbounded (i.e., do not wrap-around). The ML type of the function `bs` is

$$\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$$

which is too weak to prove statically that the assertion on line 5 always holds. One way to prove the assertion is to use a stronger, *dependent* type system which refines the ML type system by *qualifying* each ML datatype with logical predicates. For example, the dependent type for `assert` is

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

stating the requirement that the argument to `assert` must be a predicate evaluating to true.

We work in the framework of *liquid types* [**?**], where the ML types of (sub-)expressions are refined using Boolean combinations over a given set of predicates (called *logical qualifiers*) over program variables. For example, given the logical qualifiers ♣ CHECK♣

$$\nu \leq \texttt{len}$$

the liquid type inference algorithm from [**?**] is able to conclude that the assertion is never violated. The liquid type inferred for `bs` is:

$$\texttt{len} : \texttt{int} \rightarrow l : \texttt{int} \rightarrow \{\nu : \nu \leq \texttt{len}\}\texttt{int}$$

and of the expression `m` is $\{\nu : \nu \leq \texttt{len}\}$, which proves that the argument to the `assert` is always true. Similar reasoning can be performed in closely related dependent type systems such as DML [11,12].

While dependent type systems offer precise program reasoning, in practice, it is often tedious to figure out exactly what facts are required to get a program to type check. A similar problem arises the safety verification of imperative programs, where the programmer has to specify an abstract domain over which the analysis operates. In that setting, counterexample-guided abstraction refinement is used to *infer* new predicates based on failed runs of the model checker. In this paper, our goal is to import this idea to the setting of dependent type inference. Once this is performed, ideally, one can run the liquid type inference algorithm in a loop, and every failed attempt to type check the program can be mapped either to a concrete violation in the program, or new logical qualifiers that rule out all causes of type failure so far. (Since the problem is undecidable in general, the process may not terminate.)

Suppose we run the liquid type inference algorithm *without* specifying any predicates (but using the dependent type for `assert`). Then, liquid type inference reduces to ML type inference, and the program does not type check: it cannot be proved that the argument to assert is always true. Technically, the constraints generated by the liquid type inference algorithm are unsatisfiable. We show how this unsatisfiable set of constraints can be mined to automatically derive logical qualifiers, in this case, the qualifier $\nu \leq \texttt{len}$. ♣ MORE♣

$$
\begin{array}{l}
\mathsf{cegTypeInfer}(e, \Pi, i) = \\
\quad \textbf{if } (\mathsf{LiquidTypeInfer}(e, \Pi) = \mathsf{ok})\textbf{then} \\
\quad\quad \textbf{return } \text{“safe”} \\
\quad \textbf{else} \\
\quad\quad \texttt{let } e' = \mathsf{Unroll}(e, i) \texttt{ in} \\
\quad\quad \textbf{match } \mathsf{CtrexAnalyze}(e') \textbf{ with} \\
\quad\quad \mid \mathsf{Error} \rightarrow \textbf{return } \text{“errror”} \\
\quad\quad \mid \mathsf{Preds}(\Pi') \rightarrow \mathsf{cegTypeInfer}(e, \Pi \cup \Pi', i+1)
\end{array}
$$

Fig. 2. Counterexample-guided liquid type inference

Overall Algorithm Figure 2 shows our overall algorithm. The input to the algorithm is a program e, a set of logical qualifiers Π, and an “unfolding depth” i. The algorithm first type checks e by running liquid type inference using the qualifiers Π (algorithm LiquidTypeInfer). If type checking succeeds, the algorithm returns “safe”. Otherwise, every recursive definition in e is unrolled i times (using the procedure Unroll), and counterexample analyusis is performed on the resulting, non-recursive program (procedure CtrexAnalyze). Counterexample analysis either shows a concrete execution which causes the program to get stuck, or returns a new set of logical qualifiers. In the latter case, the algorithm recursively calls itself with the combined set $\Pi \cup \Pi'$ of qualifiers and an incremented unfolding depth $i + 1$. The procedure Unroll simply unrolls every recursive definition i times, thus constructing a non-recursive program. For example, $\mathsf{Unroll}(\texttt{let rec} f x = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1) \texttt{ in } f\ 2, 2)$ gives

$$
\begin{array}{l}
\texttt{let} f_1\ x\ = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } \texttt{diverge}() \texttt{ in} \\
\texttt{let} f_2\ x\ = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f_1\ (x-1) \texttt{ in } f_2\ 2
\end{array}
$$

The rest of the paper formalizes and explains CtrexAnalyze.

2 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 · then · else ·` and *let-bindings* `let` and `let rec`.

$$
\begin{array}{lll}
e & ::= & \textit{Expressions:} \\
& \mid x & \text{variable} \\
& \mid \mathtt{c} & \text{constant} \\
& \mid \lambda x.e & \text{abstraction} \\
& \mid x^{\ell} x & \text{application} \\
& \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & \text{if-then-else} \\
& \mid \mathtt{let}\ x\ =\ e\ \mathtt{in}\ e & \text{let-binding} \\
& \mid \mathtt{let\ rec}\ f\ =\ \lambda x.e\ \mathtt{in}\ e & \text{letrec-binding} \\
Q & ::= & \textit{Liquid Refinements} \\
& \mid \mathit{true} & \text{true} \\
& \mid q & \text{qualifier in } \mathbb{Q} \\
& \mid Q \wedge Q & \text{qualifier conjunction}
\end{array}
$$

$$
\begin{array}{lll}
B & ::= & \textit{Base Types:} \\
& \mid \mathtt{int} & \text{base type, integers} \\
& \mid \mathtt{bool} & \text{base type, booleans} \\
\mathbb{T}(\mathbb{B}) & ::= & \textit{Type Skeletons:} \\
& \mid \{\nu : B \mid \mathbb{B}\} & \text{base} \\
& \mid x : \mathbb{T}(\mathbb{B}) \rightarrow \mathbb{T}(\mathbb{B}) & \text{function} \\
& \mid \alpha & \text{type variable} \\
\mathbb{S}(\mathbb{B}) & ::= & \textit{Type Schema Skeletons:} \\
& \mid \mathbb{T}(\mathbb{B}) & \text{monotype} \\
& \mid \forall \alpha.\mathbb{S}(\mathbb{B}) & \text{polytype} \\
\tau, \sigma & ::= \mathbb{T}(\mathit{true}), \mathbb{S}(\mathit{true}) & \textit{Types, Schemas} \\
T, S & ::= \mathbb{T}(E), \mathbb{S}(E) & \textit{Dep. Types, Schemas} \\
\hat{T}, \hat{S} & ::= \mathbb{T}(Q), \mathbb{S}(Q) & \textit{Liquid Types, Schemas}
\end{array}
$$

Fig. 3. Syntax

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 [**?**] 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* $x : T_1 \to T_2$ 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):

$$\begin{aligned}
\mathit{true} &:: \{\nu : \mathtt{bool} \mid \nu\} \qquad \mathit{false} :: \{\nu : \mathtt{bool} \mid \mathtt{not}\ \nu\}\\
\Leftrightarrow &:: x : \mathtt{bool} \to y : \mathtt{bool} \to \{\nu : \mathtt{bool} \mid \nu \Leftrightarrow (x \Leftrightarrow y)\}\\
c \in \mathtt{int} &:: \{\nu : \mathtt{int} \mid \nu = c\}\\
= &:: x : \mathtt{int} \to y : \mathtt{int} \to \{\nu : \mathtt{bool} \mid \nu \Leftrightarrow (x = y)\}\\
+ &:: x : \mathtt{int} \to y : \mathtt{int} \to \{\nu : \mathtt{bool} \mid \nu = (x + y)\}\\
\mathtt{assert} &:: \{\nu : \mathtt{bool} \mid \nu\} \to \mathtt{unit}\\
\mathtt{diverge}_\tau &:: \mathtt{unit} \to \tau
\end{aligned}$$

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. Figure 8 in the Appendix summarizes the type derivation rules for our dependent type system. 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 [**?**] 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 $e \hookrightarrow e'$.*

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

3 Abstract Type Inference

The *liquid type inference algorithm* [**?**] 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 produces dependent types for all subexpressions of e such that if the constraint-based algorithm finds a satisfying assignment to the constraints, then $\Gamma \vdash e : S$ for some S. We briefly recall the algorithm from [**?**].

3.1 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 via the grammar shown below,

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

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.

3.2 Constraint Generation

Figure 4 shows the constraint generation algorithm for type inference. 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 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 appropriate subsumption rules 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 in Figure 4 differs from the constraint generation procedure in liquid type inference [**?**] in that each constraint generated by an application is tagged with the unique label corresponding to the application. These tags are ignored during the type inference phase, *i.e.,* during constraint solving, but are used for qualifer discovery.

Example 1. Consider the example program shown in Figure 5. Assuming `dinc` : $y : \kappa_3 \rightarrow \kappa_4$ and `inc` : $x : \kappa_1 \rightarrow \kappa_2$. ♣ give the well-formedness constraints♣ For this program, the following constraints are generated.

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

3.3 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. In the first step, we use the well-formedness and subtyping rules to split the complex constraints, which may contain function types, into simple constraints over variables with pending substitutions. In the second step, we iteratively weaken a trivial assignment, in which each liquid type variable is assigned the conjunction of all logical qualifiers, until we find the least fixpoint solution for all the simplified constraints or determine that the constraints have no solution. We first formalize the notion of a solution and then describe the two-step algorithm that computes solutions.

Satisfying Liquid Assignments. A *liquid assignment over* $\mathbb{Q}$ is a map A from liquid type variables to sets of qualifiers from $\mathbb{Q}^\star$. Assignments can be lifted to maps from templates F to dependent types $A(F)$ and template environments Γ to environments $A(\Gamma)$, by substituting each liquid type variable κ with $\bigwedge A(\kappa)$ and then applying the pending substitutions. A 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.

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.

4 Counterexample Refinement

♣ the notion of bounding by rec-free-ness needs to be said before, perhaps with an overall "loop" pseudo-code somewhere ?♣

We introduce some 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 $c \equiv \Gamma \vdash F$ (resp., $c \equiv \Gamma \vdash F_1 <: F_2$), we define $\mathsf{lhs}(c) = \Gamma$ and $\mathsf{rhs}(c) = F$ (resp., $\mathsf{rhs}(c) = F_1 <: F_2$).

Counterexample refinement is carried out using the following steps.

Step 1. Renaming. In the first step, program variables are cloned and renamed so that each distinct value held by a variable, *i.e.,* each distinct assignment in the imperative case, and each distinct binding in the functional 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 that captures *all* the (finitely many) possible executions of the recursion-free program.

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

Step 4. Interpolation In the fourth step, we compute for each cut, a Craig Interpolant, from which after dropping the tags and renaming, we can find an appropriate set of qualifiers for the liquid variable corresponding to the cut. Thus, the qualifiers resulting from all the interpolants form a valid dependent type assignment that proves the safety of the bounded recursion-free program.

We now describe how each step is carried out in detail.

For a template environment Γ, we define $\mathit{vars}(\Gamma)$, the set of type variables in Γ, inductively as $\mathit{vars}(\emptyset) = \emptyset$, $\mathit{vars}(\Gamma_0; e) = \mathit{vars}(\Gamma_0)$, $\mathit{vars}(\Gamma_0; x : \{\nu \mid e\}) = \mathit{vars}(\Gamma_0)$, and $\mathit{vars}(\Gamma_0; x : \kappa) = \{\kappa\} \cup \mathit{vars}(\Gamma_0)$,

Similarly, with abuse of notation we define $\mathit{vars}(\{\nu \mid e\}) = \emptyset$ and $\mathit{vars}(\kappa) = \{\kappa\}$. We extend *vars* to sets in the natural way. For a constraint $c \equiv \Gamma \vdash t$, we write $\mathit{vars}(c)$ for $\mathit{vars}(\mathsf{lhs}(c)) \cup \mathit{vars}(\mathsf{rhs}(c))$.

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

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 $l' - l$.

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.

Example 2. [Cloning] Recall the constraints C generated on the program from Figure 5 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.

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

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.

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. ♣ Some Proposition that says what clone does ?♣ ♣ Fix Pending Substs♣

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* $[\![c]\!]$:

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

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. [Trace Formula] 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 connection between the operational semantics of the bounded program and the trace formula is captured by the following theorem.

Theorem 3. *[Trace Formulas] Let e be a recursion-free closed expression, and $C = \mathsf{Cons}(\emptyset, e)$. For any assertion constraint $c \in C$, we have that $\mathsf{TF}(()c, C)$ is satisfiable iff the program e gets stuck* ♣ at that assertion...♣

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, $C = \mathsf{Cons}(\emptyset, e)$, and $c \in C$ be an assertion constraint. Figure 6 shows the Algorithm Partition which takes as input (1) the liquid type variable κ called the *pivot* and (2) the set of cloned constraints $C' = \mathsf{Clone}(c, C)$, and returns as output a pair of formulas $(\phi_\kappa^-, \phi_\kappa^+)$ such that:

1. $\phi_\kappa^- \wedge \phi_\kappa^+$ is equivalent to $\mathsf{TF}(c, C)$,
2. $vars(\phi_\kappa^-) \cap vars(\phi_\kappa^+)$ is contained in $\{\nu_\kappa\} \cup vars(\Gamma)$.

That is, the conjunction of the definition and use formulas is the entire trace formula, and the only variables common to the definition and use formulas are the variables that are in scope for the expression whose unknown liquid is represented by 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 $(\langle y, 01 \rangle = \nu_{\langle \kappa_3, 01 \rangle} \wedge \nu_{\langle \kappa_4, 01 \rangle} = \langle y, 01 \rangle + 1)$ $\phi^+_{\langle \kappa_4, 01 \rangle}$ 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) the only variables common to $vars(\phi_\kappa^-)$ and $vars(\phi_\kappa^+)$ are program variables bound in $vars(\Gamma)$, and value variables $\nu_{\kappa'}$ where κ' is either κ or bound 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.

♣ TBD♣

```
1. fix pseudo code to reflect text
3. fix other mynotes
```

Theorem 5. *Let e be a closed recursion-free expression and let* $C' = \mathsf{Clone}(c, \mathsf{Cons}(\Gamma, e))$. *Suppose* $[\![C']\!]$ *is unsatisfiable and* $\mathbb{Q} = \mathsf{ITP}(C')$. *Then there exists S such that* $\mathsf{Infer}(\Gamma, e, \mathbb{Q}) = S$.

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 [10].

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. R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV 05*, LNCS, pages 39–51. Springer, 2005.
11. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.
12. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

$$
\begin{array}{l}
\mathsf{Cons}(\Gamma, x) = \\
\quad \textbf{if}\ \ \mathsf{HM}(\Gamma, e) = B \\
\quad \textbf{then}\ \ (\{\nu\!:\!B \mid \nu = x\}, \emptyset) \\
\quad \textbf{else}\ \ (\Gamma(x), \emptyset)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Cons}(\Gamma, \mathsf{c}) = \\
\quad (ty(\mathsf{c}), \emptyset)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Cons}(\Gamma, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) = \\
\quad \textbf{let}\ F = \mathsf{Fresh}(\mathsf{HM}(\mathsf{Shape}(\Gamma), e))\ \textbf{in} \\
\quad \textbf{let}\ (\_, C_1) = \mathsf{Cons}(\Gamma, e_1)\ \textbf{in} \\
\quad \textbf{let}\ (F_2, C_2) = \mathsf{Cons}(\Gamma; e_1, e_2)\ \textbf{in} \\
\quad \textbf{let}\ (F_3, C_3) = \mathsf{Cons}(\Gamma; \neg e_1, e_3)\ \textbf{in} \\
\quad (F, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash F\} \cup \\
\qquad \{\Gamma; e_1 \vdash F_2 <: F\} \cup \{\Gamma; \neg e_1 \vdash F_3 <: F\})
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{Cons}(\Gamma, \lambda x.e\ \textbf{as}\ e') = \\
\quad \textbf{let}\ (x\!:\!F_x \rightarrow F) = \mathsf{Fresh}(\mathsf{HM}(\Gamma, e'))\ \textbf{in} \\
\quad \textbf{let}\ (F', C) = \mathsf{Cons}(\Gamma; x\!:\!F_x, e)\ \textbf{in} \\
\quad (x\!:\!F_x \rightarrow F, C \cup \{\Gamma \vdash x\!:\!F_x \rightarrow F\} \cup \{\Gamma; x\!:\!F_x \vdash F' <: F\})
\end{array}
$$

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

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

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

$$
\begin{array}{l}
\mathsf{Solve}(C, A) = \\
\quad \textbf{if}\ \text{exists}\ c \in C\ \text{such that}\ A(c)\ \text{is not valid} \\
\quad \textbf{then}\ \mathsf{Solve}(C, \mathsf{Weaken}(c, A))\ \textbf{else}\ A
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Infer}(\Gamma, e, \mathbb{Q}) = \\
\quad \textbf{let}\ (F, C) = \mathsf{Cons}(\Gamma, e)\ \textbf{in} \\
\quad \textbf{let}\ A = \mathsf{Solve}(\mathsf{Split}(C), \lambda\kappa.\mathsf{Inst}(\Gamma, e, \mathbb{Q}))\ \textbf{in} \\
\quad A(F)
\end{array}
$$

Fig. 4. Abstract Type Inference Algorithm

```
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. 5. Example program and constraints

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

$\mathsf{Partition}(\kappa, C) =$
$\quad$ let $\Gamma \vdash \kappa$ be the well-formedness constraint for κ
$\quad X = \textit{vars}(\Gamma)$
$\quad C' = \emptyset;\ W = \{\kappa\}$
$\quad \textbf{while } W \neq \emptyset$
$\quad\quad \textbf{pick} \text{ and } \textbf{remove } \kappa' \in W$
$\quad\quad \textbf{foreach } c \in C \textit{ s.t. } \kappa' \in \textit{vars}(\mathsf{rhs}(c))$
$\quad\quad\quad C' = C' \cup \{c\}$
$\quad\quad\quad W = W \cup \{\kappa'' \mid \kappa'' \in \textit{vars}(c), \kappa'' < \kappa'\} \setminus X$
$\quad \textbf{return } (C', C \setminus C')$

Fig. 6. Algorithms Clone and $\mathsf{Partition}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Rules for Dependent Type Checking