Gradual Liquid Type Inference

ANONYMOUS AUTHOR(S)

Refinement types allow for lightweight program verification by enriching types types with logical predicates. Liquid typing provides a decidable refinement inference mechanism that is convenient but subject to two major issues: (1) inference is global and requires top-level annotations, making it unsuitable for inference of modular code components and prohibiting its applicability to library code, and (2) inference failure results in obscure error messages. These difficulties seriously hamper the migration of existing code to use refinements.

This paper shows that *gradual liquid type inference*–a novel combination of liquid inference and gradual refinement types–addresses both issues. Gradual refinement types, which support imprecise predicates that are optimistically interpreted, can be used in argument positions to constrain liquid inference so that the global inference process effectively infers modular specifications usable for library components. Dually, when gradual refinements appear as the result of inference, they signal an inconsistency in the use of static refinements. Because liquid refinements are drawn from a finite set of predicates, in gradual liquid type inference we can enumerate the *safe concretizations* of each imprecise refinement, *i.e.* the static refinements that justify why a program is gradually well-typed. This enumeration is useful for static liquid type error explanation, since the safe concretizations exhibit all the potential inconsistencies that lead to static type errors.

We develop the theory of gradual liquid type inference and explore its pragmatics in the setting of Liquid Haskell. To demonstrate the utility of our approach, we develop an interactive tool, `GuiLT`, for gradual liquid type inference in Liquid Haskell that both infers modular types and explores safe concretizations of gradual refinements. We report on the use of `GuiLT` for error reporting and discuss a case study on the migration of three commonly-used Haskell list manipulation libraries into Liquid Haskell.

1 INTRODUCTION

Refinement types [Freeman and Pfenning 1991] allow for lightweight program verification by decorating existing program types with logical predicates. For instance, the type `{x:Int | 0 < x}` denotes strictly positive integer values and can be used to validate *at compile time* the absence of division-by-zero errors by refining the type of division:

```
(/) :: Int → {x:Int | 0 < x} → Int
```

A major challenge with refinement types is to support decidable automatic checking and *inference*. To this end, *liquid types* restrict refinement predicates to decidable theories [Rondon et al. 2008]. The type `{x:Int | k}` describes integer values refined with some predicate `k`, that is automatically solved based on unifying the constraints generated at each use of `x`, resulting in a concrete refinement drawn from a *finite* domain of template refinements. The attractiveness of liquid types for programmers is usability. Verification only requires specification of top-level functions, while all intermediate types can be automatically inferred and checked.

It has long been recognized that automatic type inference makes error reporting very challenging [Wand 1986], resulting in severe usability problems. In particular, at an ill-typed function application, the system should decide whether to blame the function definition or the client; the particular choice will unfortunately and inevitably expose some of the internals of the inference procedure on to the user. The situation is even worse with liquid type inference, because refinement verification is more involved than standard Hindley-Milner style unification. The difficulty of understanding error messages from liquid inference in turn makes it really hard to progressively migrate non-refined code to refined code, *e.g.* from Haskell to Liquid Haskell [Vazou et al. 2014b].

For instance, consider the following function `divIf` that either inverts its argument `x` if it is positive, or inverts `1-x` otherwise:

```
divIf x = if isPos x then 1/x else 1/(1-x)
```

The function relies on an imported function `isPos` of type `Int → Bool`. If we want to give `divIf` a refined type, liquid inference starts from the signature:

```
divIf :: x:{ Int | k_x } → {o:Int | k_o }
```

and solves the predicate variables k_x and k_o based on the use sites. However, without any uses of `divIf` in the considered source code, and for reasons we will clarify in due course, liquid inference infers the useless refinements:

```
divIf :: x:{ Int | false } → {o:Int | false }
```

The false refinement on the argument means that `divIf` is dead code, since liquid type inference relies on a *closed world assumption*; and `divIf` has, at inference time, no clients.

Conversely, if the code base at inference time includes a "positive" client call `divIf 1`, the inferred precondition would be `0 < x`, triggering a type error in the *definition* of `divIf` due to the lack of information about `isPos`. This hard-to-predict and moving blame is not unique to liquid type inference; it frequently appears in type inference engines, yielding hard-to-debug error messages.

A key contribution of this paper is to recognize that, in such situations, treating the inferred refinement as an *unknown* refinement — in the sense of gradual typing [Siek and Taha 2006] — allows us to eliminate the closed world assumption, as well as globally explain liquid type errors. Specifically, we adapt the gradual refinement types of Lehmann and Tanter [2017] to the setting of liquid type inference, yielding gradual liquid type inference. Programmers can introduce gradual refinements such as `{x: Int | ?}` and inference exhaustively searches for *safe concretizations* (SCs for short), *i.e.* the concrete refinements that can replace each occurrence of a gradually-refined variable to make the program well-typed. These SCs can then be used to understand liquid type errors and assist in migrating programs to adopt liquid types.

Contributions. This paper makes the following contributions:

- We give a semantics and inference algorithm for gradual liquid types (§ 4), by exploiting the abstract interpretation approach to gradual language design [Garcia et al. 2016]. We prove that inference is correct and satisfies the static criteria for gradual languages [Siek et al. 2015]. The latter result is, to the best of our knowledge, novel for a gradual inference system.
- We implement gradual liquid type inference in `GuiLT`, as an extension of Liquid Haskell (§ 5). The implementation integrates a number of optimizations and heuristics in order to be applicable to existing Haskell libraries.
- `GuiLT` takes as input a Haskell program annotated with gradual refinements and generates an interactive program that presents all valid static choices to interpret each separate occurrence of an unknown refinement, if any. The user can explore suggested predicates and decide which ones to use so that their program is well-typed (§ 6).
- We use `GuiLT` for user-guided migration of three existing Haskell libraries (1260 LoC) to Liquid Haskell (§ 7), demonstrating that gradual liquid type inference can be effectively supported and used interactively for program migration.

The essence of the novelty of our approach—that gradual inference based on abstract interpretation can be fruitfully used for error reporting and program migration—is not restricted to refinement types. We conjecture that the principles of gradual types can be used for type error explanation in further typing systems.

This paper proceeds as follows. We first provide an informal overview of liquid type inference, gradual refinement types, and gradual liquid type inference (§ 2). Next, we formally establish the necessary background on liquid types and gradual refinements (§ 3), before presenting the main results (§ 4–7). Finally, we discuss related work (§ 8) and conclude (§ 9). Auxiliary definitions as well as proofs of theorems can be found in Appendix A. The github location of `GuiLT` is submitted as supplementary material with this submission.

2 BACKGROUND AND OVERVIEW

We start by recalling the basic notions of refinement types (§ 2.1), their decidable fragment known as liquid types (§ 2.2), and recent work on gradual refinements (§ 2.3). Then (§ 2.4) we combine gradual and liquid types to build a usable interactive inference procedure that can be used for both error explanation and program migration.

2.1 Refinement Types

To explain refinement type checking, assume the below type signatures for the example of § 1:

```
isPos :: x:Int → {b:Bool | b ⇔ 0 < x}
divIf :: Int → Int
```

With these specifications, `divIf` is well-typed because the positive restriction in the precondition of `(/)` is provably satisfied. Refinement type checking proceeds in three steps, described below.

Step 1: Constraint Generation. Based on the code and the specifications, refinement subtyping constraints are generated; for our example, two subtyping constraints are generated, one for each call to `(/)`. They stipulate that, in the environment with the argument `x` and the boolean branching guard `b`, the second argument to `(/)` (*i.e.* `v = x` and `v = 1-x`, *resp.*) is *safe*, *i.e.* it respects the precondition `0 < v`.

```
b:{b ⇔ 0 < x ∧ b}   ⊢ {v | v = x}   ≤ {v | 0 < v}
b:{b ⇔ 0 < x ∧ ¬b} ⊢ {v | v = 1-x} ≤ {v | 0 < v}
```

For space, we write `{v | p}` to denote `{v:t | p}` when the type `t` is clear; we omit the refinement variables from the environment, simplifying `x:{x | p}` to `x:{p}`, and we skip uninformative refinements such as `x:{true}`.

In both constraints the branching guard `b` is refined with the result refinement of `isPos`: `b ⇔ 0 < x`. Also, the environment is strengthened with the value of the condition in each branch: `b` in the **`then`** branch, `¬ b` in the **`else`** branch.

Step 2: Verification Conditions. Each subtyping constraint is reduced to a logical verification condition (*VC*), that validates if, assuming all the refinements in the environment, the refinement on the left-hand side implies the one on the right-hand side. For instance, the two constraints above reduce to the following VCs.

```
b ⇔ 0 < x ∧  b ⇒ v = x   ⇒ 0 < v
b ⇔ 0 < x ∧ ¬b ⇒ v = 1-x ⇒ 0 < v
```

Step 3: Implication Checking. Finally, an SMT solver is used to check the validity of the generated VCs, and thus determine if the program is well-typed. Here, the SMT solver determines that both VCs are valid, thus `divIf` is well-typed.

2.2 Liquid Types

When not all refinement types are spelled out explicitly, one can use *inference*. For instance, the well-typedness of `divIf` crucially relies on the guard predicate, as propagated by the refinement type of `isPos`. We now explain how liquid typing [Rondon et al. 2008] infers a type for `divIf` in case `isPos` is an imported, *unrefined* function. Liquid types introduce refinement type variables, known as *liquid variables*, for unspecified refinements. As in § 1, the type of `divIf` is assigned the liquid variables k_x and k_o for the input and the output refinements, *resp.*:

```
divIf :: x:{ Int | kₓ } → {o:Int | kₒ }
```

After generating subtyping constraints as in § 2.1, the inference procedure attempts to find a solution for the liquid variables such that all the constraints are satisfied. If no solution can be found, the program is deemed ill-typed.

Step 1: Constraint Generation. After introduction of the liquid variables, the following subtyping constraints are generated for `divIf`.

```
x:{kₓ}, b:{b}  ⊢ {v | v = x  } ⪯ {v | 0 < v}
x:{kₓ}, b:{¬b} ⊢ {v | v = 1-x} ⪯ {v | 0 < v}
```

Step 2: Constraint Solving. Liquid inference then solves the liquid variables k so that the subtyping constraints are satisfied. The solving procedure takes as input a *finite* set of refinement *templates* $\mathbb{Q}^\star$ abstracted over program variables. For example, the template set $\mathbb{Q}^\star$ below describes ordering predicates, with $\star$ ranging over program variables.

$$\mathbb{Q}^\star = \{0 < \star,\ 0 \le \star,\ \star < 0,\ \star \le 0,\ \star < \star,\ \star \le \star\}$$

Next, for each liquid variable, the set $\mathbb{Q}^\star$ is instantiated with all program variables in scope, to generate well-sorted predicates. Instantiation of $\mathbb{Q}^\star$ for the liquid variables k_x and k_o leads to the following concrete predicate candidates.

$$\mathbb{Q}^x = \{\ 0<x,\ 0\le x,\ x<0,\ x\le 0\ \}$$
$$\mathbb{Q}^o = \{\ 0<o,\ 0\le o,\ o<0,\ o\le 0,\ o<x,\ x<o,\ \ldots\ \}$$

Finally, inference iteratively computes the strongest solution for each liquid variable that satisfies the constraints. It starts from an initial solution that maps each variable to the logical conjunction of all the instantiated templates $A = \{\mathsf{k}_x \mapsto \bigwedge \mathbb{Q}^x,\ \mathsf{k}_o \mapsto \bigwedge \mathbb{Q}^o\}$. It then repeatedly filters out predicates of the solution until all constraints are satisfied.

In our example, the initial solution—which includes the contradictory predicates $0 < x$ and $x < 0$— solves both liquid variables to false. As discussed in § 1, this inferred as false, solution is based on a closed world assumption and is practically useless. With client code that imposes additional constraints, the inferred solution can be more useful, though the reported errors can be hard to interpret.

2.3 Gradual Refinement Types

We observe that we can exploit gradual typing in order to assist inference and provide better support for error explanation and program migration. Instead of interpreting an unspecified refinement as a liquid variable, let us use the unknown gradual refinement `{ Int | ? }` for the argument type of `divIf` [Lehmann and Tanter 2017].

```
divIf :: x:{ Int | ? } → Int
```

This gradual precondition specifies that for each *usage occurrence* of the argument x, there must *exist* a concrete refinement (which we call a *safe concretization*, SC) for which the (non-gradual) program type checks. Key to this definition is that the refinement that exists need not be unique to all occurrences of the identifier. Gradual refinement type checking proceeds as follows.

Step 1: Constraint Generation. First, we generate the subtyping constraints derived from the definition of `divIf` that now contain gradual refinements.

```
x:{?}, b:{b}  ⊢ {v | v = x  } ⪯ {v | 0 < v}
x:{?}, b:{¬b} ⊢ {v | v = 1-x} ⪯ {v | 0 < v}
```

Step 2: Gradual Verification Conditions. Each subtyping reduces to a VC, where each gradual refinement such as `x:{?}` translates intuitively to an existential refinement (`∃ p. p x`). The solution of these existentials are the safe concretizations (SCs) of the program. Here, we informally use $\exists^?$ `p` to denote such existentials over predicates, and call the corresponding verification conditions *gradual VCs* (GVCs). For example, the GVCs for `divIf` are the following.

$$(\exists^? \; p_{then}. \; p_{then} \; x) \land b \Rightarrow v{=}x \Rightarrow 0 < v$$
$$(\exists^? \; p_{else}. \; p_{else} \; x) \land \lnot b \Rightarrow v{=}1{-}x \Rightarrow 0 < v$$

Step 3: Gradual Implication Checking. Checking the validity of the generated GVCs is an open problem. In the `divIf` example, we can, by observation, find the SCs that render the GVCs valid: $p_{then}\ x \mapsto 0 < x$ and $p_{else}\ x \mapsto x \leq 0$.

More importantly, we can present these solutions to the user as the conditions under which `divIf` is well-typed. We use the SCs to explain to the user that for `divIf` to type check under a static type, the gradual precondition `?` should be replaced with a refinement that implies `0 < x` in the **`then`** branch and `x ≤ 0` in the **`else`** branch. Since such a refinement cannot exist, we use SCs to explain to the user that the `?` cannot be replaced in the type of `divIf`, unless the user modifies the code (here, strengthen the postcondition of `isPos`, as in § 2.1).

Our goal is to find an algorithmic procedure that solves GVCs. Lehmann and Tanter [2017] describe how GVCs over linear arithmetic can be checked, while Courcelle and Engelfriet [2012] describe a more general logical fragment (monadic second order logic) with an algorithmic decision procedure. Yet, in both cases we lose the justifications, *i.e.* the SCs, and thus the opportunity to use such SCs for error explanation and migration assistance.

2.4 Gradual Liquid Type Inference

To algorithmically solve GVCs we can exhaustively search for SCs in the *finite* predicate domain of liquid types. In between constraint generation and constraint solving, gradual liquid type inference concretizes the constraints by instantiating gradual refinements with each possible liquid template.

Step 1: Constraint Generation. Constraint generation is performed exactly like gradual refinement types, leading to the constraints of § 2.3 for the `divIf` example.

Step 2: Constraint Concretization. We exhaustively generate all the possible concretizations of the constraints. For example, `x:{?}` can be concretized to any predicate from the $\mathbb{Q}^x$ set of § 2.2, yielding $|\mathbb{Q}^x|^2 = 16$ concrete constraint sets, among which the following two:

(1) **Concretization for** $p_{then}\ x \mapsto 0 < x$, $p_{else}\ x \mapsto 0 < x$:

```
x:{0<x}, b:{b}  ⊢ {v | v = x  } ⪯ {v | 0 < v}
x:{0<x}, b:{¬b} ⊢ {v | v = 1-x} ⪯ {v | 0 < v}
```

(2) **Concretization for** $p_{then}\ x \mapsto 0 < x$, $p_{else}\ x \mapsto x \leq 0$:

```
x:{0<x},  b:{b}  ⊢ {v | v = x  } ⪯ {v | 0 < v}
x:{x≤0}, b:{¬b} ⊢ {v | v = 1-x} ⪯ {v | 0 < v}
```

Step 3: Constraint Solving. After concretization, liquid constraint solving finds out the valid ones. In our example, the constraint 1 above is invalid while 2 is valid. Out of the 16 concrete constraints, only two are valid, with $\texttt{p}_{\texttt{then}}$ `x ↦ 0 < x` and $\texttt{p}_{\texttt{else}}$ `x ↦` $x \leq 0$ or $\texttt{p}_{\texttt{else}}$ `x ↦` $x < 0$. Thus, `divIf` type checks and also the inference provides to the user the SCs of $\texttt{p}_{\texttt{then}}$ and $\texttt{p}_{\texttt{else}}$ as an explanation of type checking.

Application to Error Explanation. The contradictory solutions of the gradual refinement of the argument indicates to the user that the code cannot statically type check. The user needs to edit either the code or the refinement types. For example, the code can be fixed by providing a precise type for `isPos` (following § 2.1) or by restricting `divIf`'s domain to positive numbers (rendering the **`else`** branch as dead code).

Unlike current liquid type inference, our algorithm does not assume a closed world, since the SCs it provides do not depend on call sites of the functions. Thus, the output of the algorithm is modular: it explains the code contradictions that lead to type errors but these contradictions are generated in the function definition and do not rely on the arguments of function calls.

Importantly, gradual liquid types are used for error explanation using the language of contradictions in refinement predicates that the user understands. In this example, the output of our algorithm informs the user that the program cannot type check because the same refinement should be solved to `0 < x` in the **`then`** branch and to `x < 0` in the **`else`** branch, which is impossible. This explanation is much more informative than the current liquid type algorithm, which would, generate a type error either in the definition of `divIf` or at its call sites, following the closed world assumption, as discussed in § 1.

In § 6 we use gradual liquid types to explain to the user the infamous off-by-one bug. Since the algorithm is exhaustively searching all potential solutions, it is exponentially slow on the number of potential solutions. Yet, in § 7 we show that we can apply our technique on real Haskell code due to the three following reasons:

- Our algorithm is a modular, per-function analysis. Thus inference time depends on the size of the analyzed function, and not on the size of the whole codebase to be type checked.
- Our implementation is user interactive, thus the exponential complexity is not a problem in practice, since our algorithm runs in the background while expecting the user input. Once a SC is found, the system presents it to the user, while looking for the next SCs in the background.
- Finally, in § 5 we discuss further technical optimizations that make our theoretically-exponential algorithm tractable and usable in practice.

In § 4 we formalize the inference steps and prove the correctness and the gradual criteria of our algorithm. Various implementation considerations necessary for the algorithm to scale are described in § 5. We report on its use for error explanation and program migration in § 6 and § 7.

3 LIQUID TYPES AND GRADUAL REFINEMENTS

We briefly provide the technical background required to describe gradual liquid types. We start with the semantics and rules of a generic refinement type system (§ 3.1) which we then adjust to describe both liquid types (§ 3.2) and gradual refinement types (§ 3.3).

3.1 Refinement Types

Syntax. Figure 1 presents the syntax of a standard functional language with refinement types, λ_R^e. The expressions of the language include constants, lambda terms, variables, function applications, conditionals, and let bindings. Note that the argument of a function application needs to be syntactically a variable, as must the condition of a conditional; this normalization is standard in

$$
\begin{array}{rrcl}
\textbf{\textit{Constants}} & c & ::= & \land \mid \lnot \mid = \mid \ldots \\
 & & \mid & \texttt{true} \mid \texttt{false} \\
 & & \mid & 0, 1, -1, \ldots \\
\textbf{\textit{Values}} & v & ::= & c \mid \lambda x.e \\
\textbf{\textit{Expressions}} & e & ::= & v \mid x \mid e\ x \\
 & & \mid & \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid & \texttt{let}\ x{:}t = e\ \texttt{in}\ e \\
\textbf{\textit{Predicates}} & p & ::= & e
\end{array}
\qquad
\begin{array}{rrcl}
\textbf{\textit{Basic Types}} & b & ::= & \texttt{int} \mid \texttt{bool} \\
\textbf{\textit{Types}} & t & ::= & \{x{:}b \mid p\} \mid x{:}t \rightarrow t \\
\textbf{\textit{Environment}} & \Gamma & ::= & \cdot \mid \Gamma, x{:}t \\
\textbf{\textit{Substitution}} & \sigma & ::= & \cdot \mid \sigma, (x, e) \\
\textbf{\textit{Constraint}} & C & ::= & \Gamma \vdash \{v{:}b \mid p\} \\
 & & \mid & \Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid p\}
\end{array}
$$

Fig. 1. Syntax of λ_R^e.

Typing $\boxed{\Gamma \vdash e{:}t}$

$$
\frac{\Gamma(x) = \{v{:}b \mid \_\}}{\Gamma \vdash x{:}\{v{:}b \mid v = x\}}\ \textsc{T-Var-Base}
\qquad
\frac{\Gamma(x) \text{ is a function type}}{\Gamma \vdash x{:}\Gamma(x)}\ \textsc{T-Var}
\qquad
\frac{}{\Gamma \vdash c{:}ty(c)}\ \textsc{T-Const}
$$

$$
\frac{\Gamma \vdash e{:}t_e \qquad \Gamma \vdash t \qquad \Gamma \vdash t_e \preceq t}{\Gamma \vdash e{:}t}\ \textsc{T-Sub}
\qquad
\frac{\Gamma, x{:}t_x \vdash e{:}t \qquad \Gamma \vdash x{:}t_x \rightarrow t}{\Gamma \vdash \lambda x.e{:}(x{:}t_x \rightarrow t)}\ \textsc{T-Fun}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash y{:}t_x \\ \Gamma \vdash e{:}(x{:}t_x \rightarrow t)\end{array}}{\Gamma \vdash e\ y{:}t[y/x]}\ \textsc{T-App}
\qquad
\frac{\begin{array}{c}\text{fresh } x' \qquad \Gamma_1 \doteq \Gamma, x'{:}\{v{:}\texttt{bool} \mid x\} \qquad \Gamma_2 \doteq \Gamma, x'{:}\{v{:}\texttt{bool} \mid \lnot x\} \\ \Gamma \vdash x{:}\{v{:}\texttt{bool} \mid \_\} \qquad \Gamma_1 \vdash e_1{:}t \qquad \Gamma_2 \vdash e_2{:}t \qquad \Gamma \vdash t\end{array}}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2{:}t}\ \textsc{T-If}
$$

$$
\frac{\Gamma \vdash e_x{:}t_x \qquad \Gamma, x{:}t_x \vdash e{:}t \qquad \Gamma \vdash t}{\Gamma \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e{:}t}\ \textsc{T-Let}
\qquad
\frac{\Gamma \vdash e_x{:}t_x \qquad \Gamma, x{:}t_x \vdash e{:}t \qquad \Gamma \vdash t \qquad \Gamma \vdash t_x}{\Gamma \vdash \texttt{let}\ x{:}t_x = e_x\ \texttt{in}\ e{:}t}\ \textsc{T-Spec}
$$

Sub-Typing $\boxed{\Gamma \vdash t \preceq t}$

$$
\frac{\text{isValid}(\Gamma \vdash \{v{:}b \mid p_1\} \preceq \{v{:}b \mid p_2\})}{\Gamma \vdash \{v{:}b \mid p_1\} \preceq \{v{:}b \mid p_2\}}\ \textsc{S-Base}
\qquad
\frac{\Gamma \vdash t_{x2} \preceq t_{x1} \qquad \Gamma, x{:}t_{x2} \vdash t_1 \preceq t_2}{\Gamma \vdash x{:}t_{x1} \rightarrow t_1 \preceq x{:}t_{x2} \rightarrow t_2}\ \textsc{S-Fun}
$$

Well-Formedness $\boxed{\Gamma \vdash t}$

$$
\frac{\text{isValid}(\Gamma \vdash \{v{:}b \mid p\})}{\Gamma \vdash \{v{:}b \mid p\}}\ \textsc{W-Base}
\qquad
\frac{\Gamma \vdash t_x \qquad \Gamma, x{:}t_x \vdash t}{\Gamma \vdash x{:}t_x \rightarrow t}\ \textsc{W-Fun}
$$

Fig. 2. Static Semantics of λ_R^e. (Types colored in blue need to be inferred.)

refinement types as it simplifies the formalization [Rondon et al. 2008]. There are two let binding forms, one where the type of the bound variable is inferred and one where it is explicitly declared.

λ_R^e types include *base refinements* $\{x{:}b \mid p\}$ where b is a base type (`int` or `bool`) refined with the logical predicate p. A predicate can be any expression e, which can refer to x. Types also include dependent function types $x{:}t_x \rightarrow t$, where x is bound to the function argument and can appear in the result type t. As usual, we write b as a shortcut for $\{x{:}b \mid \texttt{true}\}$ and $t_x \rightarrow t$ as a shortcut for $x{:}t_x \rightarrow t$ when x does not appear in t.

Denotations. Following Knowles and Flanagan [2010], each type of λ_R^e denotes a set of expressions. The denotation of a base refinement includes all expressions that either diverge or evaluate to base values that satisfy the associated predicate. We write $\models e$ to represent that e is (operationally) valid and $e \Downarrow$ to represent that e terminates:

$$\models e \ \doteq\ e \hookrightarrow^{\star} \texttt{true} \qquad e \Downarrow\ \doteq\ \exists v.e \hookrightarrow^{\star} v$$

where $\cdot \hookrightarrow^{\star} \cdot$ is the reflexive, transitive closure of the small-step reduction relation. Denotations are naturally extended to function types and environments (as sets of substitutions).

$$\begin{aligned} [\![\{x{:}b \mid p\}]\!] &\doteq \{e \mid\ \vdash e : b, \text{if } e \Downarrow \text{ then } \models p[e/x]\} \\ [\![x{:}t_x \to t]\!] &\doteq \{e \mid \forall e_x \in [\![t_x]\!].e\ e_x \in [\![t[e_x/x]]\!]\} \\ [\![\Gamma]\!] &\doteq \{\sigma \mid \forall x{:}t \in \Gamma.(x,e) \in \sigma \land e \in [\![\sigma \cdot t]\!]\} \end{aligned}$$

Static semantics. Figure 2 summarizes the standard typing rules that characterize whether an expression belongs to the denotation of a type [Knowles and Flanagan 2010; Rondon et al. 2008]. Namely, $e \in [\![t]\!]$ *iff* $\vdash e{:}t$. We define three kinds of relations 1. typing, 2. subtyping, and 3. well-formedness.

(1) *Typing:* $\Gamma \vdash e{:}t$ *iff* $\forall \sigma \in [\![\Gamma]\!].\sigma \cdot e \in [\![\sigma \cdot t]\!]$.
Rule T-Var-Base refines the type of a variable with its exact value. Rule T-Const types a constant c using the function $ty(c)$ that is assumed to be sound, *i.e.* we assume that for each constant c, $c \in [\![ty(c)]\!]$. Rule T-Sub allows to weaken the type of a given expression by subtyping, discussed below. Rule T-If achieves path sensitivity by typing each branch under an environment strengthened with the value of the condition. Finally, the two let binding rules T-Let and T-Spec only differ in whether the type of the bound variable is inferred or taken from the syntax. Note that the last premise, a well-formedness condition, ensures that the bound variable does not escape (at the type level) the scope of the let form.

(2) *Subtyping:* $\Gamma \vdash t_1 \preceq t_2$ *iff* $\forall \sigma \in [\![\Gamma]\!], e \in [\![\sigma \cdot t_1]\!].\ e \in [\![\sigma \cdot t_2]\!]$.
Rule S-Base uses the relation isValid($\cdot$) to check subtyping on basic types; we leave this relation abstract for now since we will refine it in the course of this section. Knowles and Flanagan [2010] define subtyping between base refinements as:

$$\text{isValid}(\Gamma \vdash \{x{:}b \mid p_1\} \preceq \{x{:}b \mid p_2\}) \qquad \textit{iff} \qquad \forall \sigma \in [\![\Gamma, x{:}b]\!].\text{if } \models \sigma \cdot p_1 \text{ then } \models \sigma \cdot p_2$$

This definition makes checking undecidable, as it quantifies over all substitutions. We come back to decidability below.

(3) *Well-Formedness:* Rule W-Base overloads isValid($\cdot$) to refer to well-formedness on base refinements. A base refinement $\{x{:}b \mid p\}$ is well-formed only when p is typed as a boolean:

$$\text{isValid}(\Gamma \vdash \{x{:}b \mid p\}) \qquad \textit{iff} \qquad \Gamma, x{:}b \vdash p{:}\texttt{bool}$$

Inference. In addition to being undecidable, the typing rules in Figure 2 are not syntax directed: several types do not come from the syntax of the program, but have to be guessed—they are colored in blue in Figure 2. These are: the argument type of a function (Rule T-Fun), the common (least upper bound) type of the branches of a conditional (Rule T-If), and the resulting type of let expressions (Rules T-Let and T-Spec), which needs to be weakened to not refer to variable x in order to be well-formed. Thus, to turn the typing relation into a type checking algorithm, one needs to address both decidability of subtyping judgments and inference of the aforementioned types.

3.2 Liquid Types

Liquid types [Rondon et al. 2008] provide a decidable and efficient inference algorithm for the typing relation of Figure 2. For decidability, the key idea of liquid types is to restrict refinement predicates to be drawn from a *finite* set of *predefined*, SMT-decidable predicates $q \in \mathbb{Q}$.

```
Infer :: Env → Expr → Quals → Maybe Type
Infer Γ̂ ê ℚ = A <*> t̂
  where
    A = Solve C A₀
    (t̂, C) = Cons Γ̂ ê

Cons  :: Env → Expr → (Maybe Type, [Cons])
Solve :: [Cons] → Sol → Maybe Sol
```

Fig. 3. **Liquid Inference Algorithm** (`Cons` and `Solve` are defined in [Material 2018]).

Syntax. The syntax of *liquid predicates*, written $\hat{p}$, is:

$$\begin{array}{rcll} \hat{p} & ::= & \texttt{true} & \text{True} \\ & | & q & \text{Predicate, with } q \in \mathbb{Q} \\ & | & \hat{p} \wedge \hat{p} & \text{Conjunction} \\ & | & \kappa & \text{Liquid Variable} \\ A & ::= & \cdot \mid A, \kappa \mapsto \overline{q} & \textit{Solution} \end{array}$$

A liquid predicate can be true (`true`), an element from the predefined set of predicates (q), a conjunction of predicates ($\hat{p} \wedge \hat{p}$), or a predicate variable (κ), called a *liquid variable*. A *solution* A is a mapping from liquid variables to a set of elements of $\mathbb{Q}$. The set $\overline{q}$ represents a variable-free liquid predicate using `true` for the empty set and conjunction to combine the elements otherwise.

Checking. When all the predicates in $\mathbb{Q}$ belong to SMT-decidable theories, validity checking of λ^e_R: isValid($\Gamma \vdash \{x{:}b \mid p_1\} \preceq \{x{:}b \mid p_2\}$) which quantifies over all embeddings of the typing environment, can be SMT automated in a sound and complete way. Concretely, a subtyping judgment $\Gamma \vdash \{x{:}b \mid \hat{p}_1\} \preceq \{x{:}b \mid \hat{p}_2\}$ is valid *iff* under all the assumptions of Γ, the predicate $\hat{p}_1$ implies the predicate $\hat{p}_2$.

$$\begin{gathered} \text{isValid}(\Gamma \vdash \{x{:}b \mid \hat{p}_1\} \preceq \{x{:}b \mid \hat{p}_2\}) \\ \textit{iff} \\ \text{isSMTValid}(\bigwedge\{\hat{p} \mid x{:}\{x{:}b \mid \hat{p}\} \in \Gamma\} \Rightarrow \hat{p}_1 \Rightarrow \hat{p}_2) \end{gathered}$$

Inference. The liquid inference algorithm, defined in Figure 3, first applies the rules of Figure 2 using liquid variables as the refinements of the types that need to be inferred and then uses an iterative algorithm to solve the liquid variable as a subset of $\mathbb{Q}$ [Rondon et al. 2008] (steps 1 and 2 of § 2.2).

More precisely, given a typing environment $\hat{\Gamma}$, an expression $\hat{e}$, and the fixed set of predicates $\mathbb{Q}$, the function `Infer` $\hat{\Gamma}$ $\hat{e}$ $\mathbb{Q}$ returns the type of the expression $\hat{e}$ under the environment $\hat{\Gamma}$, if it exists, or nothing otherwise. It first generates a template type $\hat{t}$ and a set of constraints `C` that contain liquid variables in the types to be inferred. Then it generates a solution A that satisfies all the constraints in `C`. Finally, it returns the type $\hat{t}$ in which all the liquid variables have been substituted by concrete refinements from the mapping in A.

The function `Cons` $\hat{\Gamma}$ $\hat{e}$ uses the typing rules in Figure 2 to generate the template type `Just` $\hat{t}$ of the expression $\hat{e}$, *i.e.* a type that potentially contains liquid variables, and the basic constraints that appear in the leaves of the derivation tree of the judgment $\hat{\Gamma} \vdash \hat{e}{:}\hat{t}$. If the derivation rules fail, then `Cons` $\hat{\Gamma}$ $\hat{e}$ returns `Nothing` and an empty constraint list. The function `Solve` C A uses the decidable validity checking to iteratively pick a constraint in $c \in C$ that is not satisfied, while such a constraint exists, and weakens the solution A so that c is satisfied. The function A `<*>` $\hat{t}$ applies the solution A to the type $\hat{t}$, if both contain `Just` values, otherwise returns `Nothing`. Here and in the following, we pose: $A_0 \doteq \lambda\kappa.\mathbb{Q}$.

The algorithm `Infer` $\hat{\Gamma}$ $\hat{e}$ $\mathbb{Q}$ is terminating and sound and complete with respect to the typing relation $\hat{\Gamma} \vdash \hat{e}{:}\hat{t}$ as long as all the predicates are conjunctions of predicates drawn from $\mathbb{Q}$.

3.3 Gradual Refinement Types

Gradual refinement types [Lehmann and Tanter 2017] extend the refinements of λ_R^e to include imprecise refinements like $x > 0 \wedge ?$. While they describe the static and dynamic semantics of gradual refinements, inference is left as an open challenge. Our work extends liquid inference to gradual refinements, therefore we hereby summarize their basics.

Syntax. The syntax of gradual predicates in $\lambda_G^{\tilde{p}}$ is

$$\begin{array}{rcll} \tilde{p} & ::= & p & \text{Precise Predicate} \\ & | & p \wedge ? & \text{Imprecise Predicates, where } p \text{ is local} \end{array}$$

A predicate is either *precise* or *imprecise*. The syntax of an imprecise predicate $p \wedge ?$ allows for a *static part* p. Intuitively, with the predicate $x > 0 \wedge ?$, x is statically (and definitely) positive, but the type system can optimistically assume stronger, non-contradictory requirements about x. To make this intuition precise and derive the complete static and dynamic semantics of gradual refinements, Lehmann and Tanter [2017] follow the Abstracting Gradual Typing methodology (AGT) [Garcia et al. 2016]. Following AGT, a gradual refinement type (resp. predicate) is given meaning by *concretization* to the set of static types (resp. predicates) it represents. Defining this concretization requires introducing two important notions.

Specificity. First, we say that p_1 is *more specific* than p_2, written $p_1 \preceq p_2$, *iff* p_2 is true when p_1 is:

$$p_1 \preceq p_2 \doteq \forall \sigma.\ \text{if } \models \sigma \cdot p_1 \text{ then } \models \sigma \cdot p_2$$

Locality. Additionally, in order to prevent imprecise formulas from introducing contradictions—which would defeat the purpose of refinement checking—Lehmann and Tanter [2017] identify the need for the static part of an imprecise refinement to be *local*. Using an explicit syntax $p(x)$ to explicitly declare the variable x refined by the predicate p, a refinement is local if there exists a value v for which $p[v/x]$ is true; and this, for any (well-typed) substitution that closes the predicate:

$$\text{isLocal}(p(x)) \doteq \forall \sigma, \exists v.\ \models \sigma \cdot p[v/x]$$

Concretization. Using specificity and locality, the concretization function $\gamma(\cdot)$ maps gradual predicates to the set of the static predicates they represent.

$$\begin{array}{rcl} \gamma(p(x)) & \doteq & \{p\} \\ \gamma((p \wedge ?)(x)) & \doteq & \{p' \mid p' \preceq p, \text{isLocal}(p'(x))\} \end{array}$$

A precise predicate concretizes to itself (singleton), while an imprecise predicate denotes all the local predicates more specific than its static part. This definition extends naturally to types and environments.

$$\begin{array}{rcl} \gamma(\{x{:}b \mid \tilde{p}\}) & \doteq & \{\{x{:}b \mid p\} \mid p \in \gamma(\tilde{p}(x))\} \\ \gamma(x{:}\tilde{t_x} \rightarrow \tilde{t}) & \doteq & \{x{:}t_x \rightarrow t \mid t_x \in \gamma(\tilde{t_x}), t \in \gamma(\tilde{t})\} \\ \gamma(\tilde{\Gamma}) & \doteq & \{\Gamma \mid x{:}t \in \Gamma \textit{ iff } x{:}\tilde{t} \in \tilde{\Gamma}, t \in \gamma(\tilde{t})\} \end{array}$$

The denotations of gradual refinement types are similar to those from § 3.1. The denotation of a base *imprecise* gradual refinement $\{x{:}b \mid p \wedge ?\}$ includes all (gradually-typed) expressions that satisfy at least p.

Type Checking. Figure 2 is used "as is" to type gradual expressions $\tilde{\Gamma} \vdash \tilde{e}{:}\tilde{t}$, save for the fact that the validity predicate must be lifted to operate on gradual types. $\widetilde{\text{isValid}}(\cdot)$ holds if there exists a justification, by concretization, that the static judgment holds. Precisely:

$$\begin{aligned}\widetilde{\text{isValid}}(\tilde{\Gamma} \vdash \tilde{t}_1 \preceq \tilde{t}_2) &\doteq \exists \Gamma \in \gamma(\tilde{\Gamma}), t_1 \in \gamma(\tilde{t}_1), t_2 \in \gamma(\tilde{t}_2).\text{isValid}(\Gamma \vdash t_1 \preceq t_2)\\ \widetilde{\text{isValid}}(\tilde{\Gamma} \vdash \tilde{t}) &\doteq \exists \Gamma \in \gamma(\tilde{\Gamma}), t \in \gamma(\tilde{t}).\text{isValid}(\Gamma \vdash t)\end{aligned}$$

4 GRADUAL LIQUID TYPES

We now formalize the combination of liquid type inference and gradual refinements to later use gradual liquid types for both error explanation and program migration. We extend the work of Lehmann and Tanter [2017] by adapting the liquid type inference algorithm to the gradual setting. To do so, we apply the abstract interpretation approach of AGT [Garcia et al. 2016] to lift the `Infer` function (defined in § 3.2) so that it operates on gradual liquid types.

Below is the syntax of predicates in $\lambda^{\check{p}}_{GL}$, a gradual liquid core language whose predicates are gradual predicates where the static part of an imprecise predicate is a liquid predicate, with the additional requirement that it is *local* (def. in § 3.3).

$$\begin{aligned}\check{p} ::= &\ \hat{p} && \text{Precise Liquid Predicate}\\ \mid &\ \hat{p} \wedge ? && \text{Imprecise Liquid Predicate, where } \hat{p} \text{ is local}\end{aligned}$$

The elements of $\lambda^{\check{p}}_{GL}$ are both gradual and liquid; *i.e.* expressions $\check{e}$ could also be written as $\tilde{\hat{e}}$. Also, we write ? as a shortcut for the imprecise predicate `true ∧ ?`.

Our goal is to define $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ so that it returns a type $\check{t}$ such that $\check{\Gamma} \vdash \check{e}{:}\check{t}$. After deriving $\check{\texttt{Infer}}$ using AGT (§ 4.1), we provide an algorithmic characterization of $\check{\texttt{Infer}}$ (§ 4.3), which serves as the basis for our implementation. We present the properties that $\check{\texttt{Infer}}$ satisfies in § 4.4.

4.1 Lifting Liquid Inference

We define the function $\check{\texttt{Infer}}$ using the abstracting gradual typing methodology [Garcia et al. 2016]. In general, AGT defines the consistent lifting of a function `f` as: $\tilde{\texttt{f}}\ \tilde{t} = \alpha(\{\texttt{f}\ t \mid t \in \gamma(\tilde{t})\})$, where α is the sound and optimal abstraction function that, together with γ, forms a Galois connection.

The question is how to apply this general approach to the liquid type inference algorithm. We answer this question via trial-and-error.

Try 1. Lifting `Infer`. Assume we lift `Infer` in a similar manner, *i.e.* we pose

$$\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q} = \alpha(\{\texttt{Infer}\ \Gamma\ e\ \mathbb{Q} \mid \Gamma \in \gamma(\check{\Gamma}), e \in \gamma(\check{e})\})$$

This definition of $\check{\texttt{Infer}}$ is too strict: it rejects expressions that should be accepted. Consider for instance the following expression $\check{e}$ that defines a function `f` with an imprecisely-refined argument:

```
// onlyPos :: {v:Int | 0 < v} → Int
// check :: Int → Bool
let f :: x:{Int | ?} → Int
    f x = if check x then onlyPos x else onlyPos (-x)
in f 42
```

There is no single static expression $e \in \gamma(\check{e})$ such that the definition of `f` above type checks. For any $\mathbb{Q}$ we will get `Infer` $\{\}\ e\ \mathbb{Q}$ = `Nothing` which denotes a type inference failure. This behavior occurs because by the definition of $\check{\texttt{Infer}}$ the gradual argument of `f` needs to be concretized before calling `Infer`, which breaks the flexibility programmers expect from gradual refinements (this example is based on the motivation example of Lehmann and Tanter [2017]). One expects that

$\check{\texttt{Infer}}$ {} $\check{e}$ $\mathbb{Q}$ should simply return `Int`. A similar scenario appears in Garcia et al. [2016], where lifting the typing relation as a whole would be too imprecise and instead the lifted typing relation is defined by lifting the type functions and predicates used to define typing. Here, as described in § 3.2, `Infer` calls the functions `Cons` and `Solve`, which in turn calls the function `isValid`. Which of these functions should we lift? Since `Cons` is merely an algorithmic definition of the typing rules, it is not affected by the gradualization of the system, thus does not require lifting. On the contrary, `Solve` is calling `isValid` that operates on gradual refinements. To get a precise inference system we first attempted to lift `isValid`.

Try 2. Lifting `isValid`. To our surprise, using gradual validity checking (the lifting of `isValid`, § 3.3) leads to an unsound inference algorithm! This is because, soundness of static inference implicitly relies on the property of validity checking that if two refinements p_1 and p_2 are right-hand-side valid, then so is their conjunction, *i.e.*:

$$\text{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid p_1\}) \quad \text{and} \quad \text{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid p_2\})$$
$$\Rightarrow$$
$$\text{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq p_1 \wedge p_2)$$

But this property does not hold for gradual validity checking, because for any logical predicate q, it is true that $(q \Rightarrow p_1 \wedge q \Rightarrow p_1)$ implies $(q \Rightarrow p_1 \wedge p_2)$, but $(\exists q.(q \Rightarrow p_1)) \wedge (\exists q.(q \Rightarrow p_1))$ does not imply that $\exists q.(q \Rightarrow p_1 \wedge p_2)$.

Try 3. Lifting `Solve`. Let us try to lift `Solve`:

$\check{\texttt{Solve}}\ A\ \check{C} = \{\texttt{Solve}\ A\ C \mid C \in \gamma(\check{C})\}$

where $\check{C}$ denotes a gradual constraint (from Figure 1). This approach is successful and leads to a provably sound and complete inference algorithm (§ 4.4).

Note that in the definition of $\check{\texttt{Solve}}$, we do not appeal to abstraction. This is because we can directly define $\check{\texttt{Infer}}$ to consider all produced solutions instead.

```
Infěr :: Ěnv → Expř → Quals → Set Type
Infěr Γ̌ ě ℚ = {ť′ | Just ť′ <- A <*> ť, A ∈ Solvě A₀ Č}
  where (ť, Č) = Cons Γ̌ ě
```

First, function `Cons` derives the typing constraints $\check{C}$, and if successful, the template type $\check{t}$ (step 1 of § 2.4).[1] Next, we use the lifted $\check{\texttt{Solve}}$ to concretize and solve all the derived constraints (steps 2 and 3 of § 2.4, *resp.*). By keeping track of the concretizations that return non-`Nothing` solutions, we derive the safe concretizations of § 2. Finally, we apply each solution to the template gradual type, yielding a set of inferred types. We do not explicitly abstract the set of inferred types back to a single gradual type. Adding abstraction by exploiting the abstraction function defined by Lehmann and Tanter [2017] is left for future work.

The core of the gradual inference algorithm is a combination of the liquid and gradual refinement type systems. Yet, as we exposed by the failing attempt to lift `isValid`, this combination was not trivial, since blind application of the AGT [Garcia et al. 2016] methodology could lead to unsound inference. In § 4.4, we prove that our algorithm is sound and in § 5 we discuss an optimized implementation and its applications. The applications discussed in § 6 and § 7 make explicit use of the inferred set in order to assist users in understanding errors and migrating programs.[2]

[1]Note that `Cons` is unchanged from the static system, because it only depends on the structure of the types, and not on the refinements themselves.
[2]In standard gradual typing, the set of static types denoted by a gradual type can be infinite, hence abstraction is definitely required. In contrast, here the structure of types is fixed, and the set of possible liquid refinements, even if potentially large, is finite. We can therefore do without abstraction. We discuss implementation considerations in § 6.

4.2 When Does Gradual Liquid Inference Fail?

Given the flexibility induced by gradual refinements, at this point the reader might wonder when an actual type inference failure can occur. First, as we will formally prove later in this section, gradual liquid inference is a conservative extension of liquid inference, and therefore, in the absence of imprecise refinements, gradual liquid inference fails exactly when standard liquid inference fails.

More crucially, in presence of imprecise refinements, gradual inference only succeeds when there exist possible justifications for each individual occurrences of gradually-refined variables. Said otherwise, gradual liquid inference fails when there exists at least one occurrence of a gradually-refined variable for which there does not exist any valid concretization.

Consider the example below:

```
// onlyPos :: {v:Int | 0 < v} → Int
f :: x:{Int | x <= 0 ∧ ?} → Int
f x = onlyPos x
```

Each valid concretization of the imprecise refinement `x <= 0 ∧ ?` should imply the precondition of `onlyPos`, namely `0 < v`. This however contradicts the static part of the imprecise refinement, `x <= 0`. Therefore, gradual liquid inference fails for this program.

4.3 Algorithmic Concretization

To make $\check{\texttt{Infer}}$ algorithmic, we need to define an algorithmic concretization function of a set of constraints, $\gamma(\check{C})$. We do so, by using the the finite domain of predicates $\mathbb{Q}$.

Recall the concretization of gradual predicates: $\gamma((p \wedge ?)(x)) \doteq \{p' \mid p' \preceq p, \text{isLocal}(p'(x))\}$. In general, this function cannot be algorithmically computed, since it ranges over the infinite domain of predicates. In gradual liquid refinements, the domain of predicates is restricted to the powerset of the *finite* domain $\mathbb{Q}$. We define the algorithmic concretization function $\gamma_{\mathbb{Q}}(\check{p}(x))$ as the intersection of the powerset of the finite domain $\mathbb{Q}$ with the gradual concretization function.

$$\gamma_{\mathbb{Q}}(\check{p}(x)) \doteq 2^{\mathbb{Q}} \cap \gamma(\check{p}(x))$$

Concretization of gradual predicates reduces to (decidable) locality and specificity checking on the elements of $\mathbb{Q}$.

$$\gamma_{\mathbb{Q}}((\hat{p} \wedge ?)(x)) \doteq \{\hat{p}' \mid \hat{p}' \in 2^{\mathbb{Q}}, \hat{p}' \preceq \hat{p}, \text{isLocal}(\hat{p}'(x))\}$$

We naturally extend the algorithmic concretization function to typing environments, constraints, and list of constraints.

$$\begin{aligned}
\gamma_{\mathbb{Q}}(\check{\Gamma}) &\doteq \{\hat{\Gamma} \mid x{:}\hat{t} \in \hat{\Gamma} \textit{ iff } x{:}\check{t} \in \check{\Gamma}, \hat{t} \in \gamma_{\mathbb{Q}}(\check{t})\} \\
\gamma_{\mathbb{Q}}(\check{\Gamma} \vdash \check{t}_1 \preceq \check{t}_2) &\doteq \{\hat{\Gamma} \vdash \hat{t}_1 \preceq \hat{t}_2 \mid \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t}_i \in \gamma_{\mathbb{Q}}(\check{t}_i)\} \\
\gamma_{\mathbb{Q}}(\check{\Gamma} \vdash \check{t}) &\doteq \{\hat{\Gamma} \vdash \hat{t} \mid \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t} \in \gamma_{\mathbb{Q}}(\check{t}), \} \\
\gamma_{\mathbb{Q}}(\check{C}) &\doteq \{\hat{C} \mid c \in \hat{C} \textit{ iff } \check{c} \in \check{C}, \hat{c} \in \gamma_{\mathbb{Q}}(\check{c})\}
\end{aligned}$$

We use $\gamma_{\mathbb{Q}}(\cdot)$ to define an algorithmic version of $\check{\texttt{Solve}}$:

$$\check{\texttt{Solve}}\ A\ \check{C} = \{\texttt{Solve}\ A\ \hat{C} \mid \hat{C} \in \gamma_{\mathbb{Q}}(\check{C})\}$$

which in turn yields an algorithmic version of $\check{\texttt{Infer}}$.

4.4 Properties of Gradual Liquid Inference

We prove that the inference algorithm $\check{\texttt{Infer}}$ satisfies the the correctness criteria of Rondon et al. [2008], as well as the static criteria for gradually-typed languages [Siek et al. 2015].[3] The corresponding proofs can be found in supplementary material [Material 2018].

4.4.1 Correctness of Inference. The algorithm $\check{\texttt{Infer}}$ is sound, complete, and terminates.

Theorem 4.1 (Correctness). *Let $\mathbb{Q}$ be a finite set of predicates from an SMT-decidable logic, $\check{\Gamma}$ a gradual liquid environment, and $\check{e}$ a gradual liquid expression. Then*

- ***Termination*** $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ *terminates.*
- ***Soundness*** *If $\check{t} \in \check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$, then $\check{\Gamma} \vdash \check{e}{:}\check{t}$.*
- ***Completeness*** *If* $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q} = \emptyset$, *then $\nexists \check{t}.\ \check{\Gamma} \vdash \check{e}{:}\check{t}$.*

The proof of termination is straightforward from the careful definition of the $\check{\texttt{Infer}}$ algorithm. Soundness and completeness rely on the property that a constraint is gradually valid *iff* there exists a concrete solution that renders it valid (*i.e.* $\widetilde{\text{isValid}}(A \cdot \check{c})$ *iff* $\exists \hat{c} \in \gamma_{\mathbb{Q}}(\check{c}).\text{isValid}(A \cdot \check{c})$). We use this property to prove soundness by the definition of the algorithm and completeness by contradiction. We note that, unlike the $\texttt{Infer}$ algorithm that provably returns the strongest possible solution, it is not clear how to relate the set of solutions returned by $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ with the rest of the types that satisfy $\check{\Gamma} \vdash \check{e}{:}\check{t}$.

4.4.2 Gradual Typing Criteria. Siek et al. [2015] list three criteria for the static semantics of a gradual language, which the $\check{\texttt{Infer}}$ algorithm satisfies. These criteria require the gradual type system *(i)* is a conservative extension of the static type system, *(ii)* is flexible enough to accommodate the dynamic end of the typing spectrum (in our case, unrefined types), and *(iii)* supports a smooth connection between both ends of the spectrum.

(i) Conservative Extension. The gradual inference algorithm $\check{\texttt{Infer}}$ coincides with the static algorithm $\texttt{Infer}$ on terms that only rely on precise predicates. More specifically, if $\texttt{Infer}$ infers a static type for a term, then $\check{\texttt{Infer}}$ returns only that type, for the same term. Conversely, if a term is not typeable with $\texttt{Infer}$, it is also not typeable with $\check{\texttt{Infer}}$.

Theorem 4.2 (Conservative Extension). *If* $\texttt{Infer}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \texttt{Just}\ \hat{t}$, *then* $\check{\texttt{Infer}}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \{\hat{t}\}$. *Otherwise,* $\check{\texttt{Infer}}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \emptyset$.

The proof follows by the definition of $\check{\texttt{Infer}}$ a n d the concretization function.

(ii) Embedding of imprecise terms. We then prove that given a well-typed unrefined term (*i.e.* simply-typed), refining all base types with the unknown predicate ? yields a well-typed gradual term. This property captures the fact that it is possible to "import" a simply-typed term into the gradual liquid setting. (In contrast, this is not possible without gradual refinements: just putting `true` refinements to all base types does not yield a well-typed program.)

To state this theorem, we use t_s to denote simple types (b and $t_s \rightarrow t_s$) and similarly e_s and Γ_s for terms and environments. The simply-typed judgment is the standard one.

The $\lceil\cdot\rceil$ function turns simple types into gradual liquid types by introducing the unknown predicate on every base type (and naturally extended to environments and terms):

$$\lceil b \rceil = \{v{:}b \mid ?\} \qquad\qquad \lceil t_1 \rightarrow t_2 \rceil = x{:}\lceil t_1 \rceil \rightarrow \lceil t_2 \rceil$$

Theorem 4.3 (Embedding of Unrefined Terms). *If* $\Gamma_s \vdash e_s{:}t_s$, *then* $\check{\texttt{Infer}}\ \lceil\Gamma_s\rceil\ \lceil e_s \rceil\ \mathbb{Q} \neq \emptyset$.

[3]Because this work focuses on the static semantics, *i.e.* the inference algorithm, we do not discuss the dynamic part of the gradual guarantee, which has been proven for gradual refinement types [Lehmann and Tanter 2017].

Let t_s denote the unrefined version of the gradually refined type $\check{t}$. We first prove that for any gradual type $\check{t}$ with only local refinement both $\Gamma \vdash \check{t} \preceq \lceil t_s \rceil$ and $\Gamma \vdash \lceil t_s \rceil \preceq \check{t}$ hold. Then, we prove that if $\Gamma_s \vdash e_s{:}t_s$, then $\lceil \Gamma_s \rceil \vdash \lceil e_s \rceil{:}\lceil t_s \rceil$. Finally, we use completeness of the inference algorithm to prove theorem 4.3.

For the expression $\lceil e_s \rceil$, the inference algorithm will generate a set of refinement types, all of which have the structure of t_s. The liquid algorithm infers types with strongest postconditions and the gradual inference algorithm is not generating any fresh ?, so, the inference returns all valid types with shape t_s with the dynamic refinements instantiated to all valid local predicates and for each instantiation, the unknown refinements are solved following the principle of strongest postconditions.

(iii) Static Gradual Guarantee. Finally, the solid foundations for gradualization based on abstract interpretation allow us to effectively satisfy the *gradual guarantee*, which we believe was never proved for any gradual inference work. The gradual guarantee stipulates that typeability is monotonic in the precision of type information. In other words, making type annotations less precise cannot introduce new type errors.

We first define the notion of precision in terms of algorithmic concretization:

Definition 4.4 (Precision of Gradual Types). $\check{t}_1$ *is less precise than* $\check{t}_2$*, written as* $\check{t}_1 \sqsubseteq \check{t}_2$*, iff* $\gamma_{\mathbb{Q}}(\check{t}_1) \subseteq \gamma_{\mathbb{Q}}(\check{t}_2)$.

Precision naturally extends to type environments and terms.

Theorem 4.5 (Static Gradual Guarantee). *If* $\check{\Gamma}_1 \sqsubseteq \check{\Gamma}_2$ *and* $\check{e}_1 \sqsubseteq \check{e}_2$*, then for every* $\check{t}_{1i} \in \mathtt{In\check{f}er}\ \check{\Gamma}_1\ \check{e}_1\ \mathbb{Q}$ *there exists* $\check{t}_{1i} \sqsubseteq \check{t}_{2i}$ *so that* $\check{t}_{2i} \in \mathtt{In\check{f}er}\ \check{\Gamma}_2\ \check{t}_2\ \mathbb{Q}$.

The intuition of the proof is that since $\check{t}_{1i} \in \mathtt{In\check{f}er}\ \check{\Gamma}_1\ \check{e}_1\ \mathbb{Q}$, then the algorithm inferred a solution of the unknown refinements. We prove that this solution is also inferred for $\check{e}_1$.

For a given term and type environment, the theorem ensures that, for every inferred type, the algorithm infers a less precise type when run on a less precise term and environment.

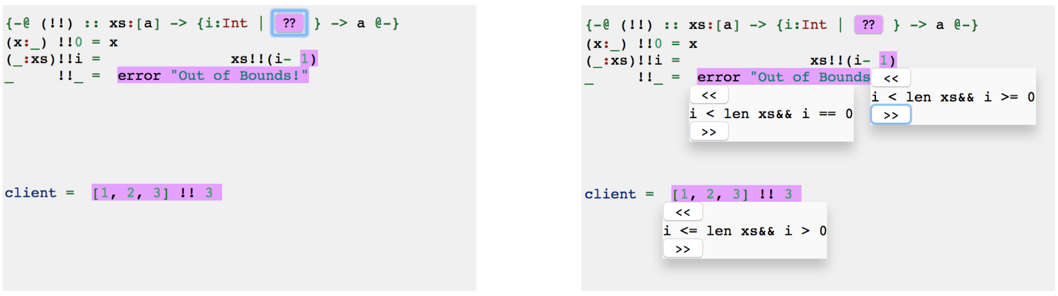
5 IMPLEMENTATION

We implemented `Infer` as `GuiLT`, an extension to Liquid Haskell [Vazou et al. 2014a] that takes a Haskell program annotated with gradual refinement specifications and returns an `.html` interactive file that lets the user explore all safe concretizations.

Concretely, `GuiLT` uses the existing API of Liquid Haskell to implement the three steps of gradual liquid type checking steps described in § 2.4 (and formalized in § 4): *1)* First, `GuiLT` calls the Liquid Haskell API to generate subtyping constraints that contain both liquid variables and imprecise predicates. *2)* Next, it calls the liquid API to collect all the refinement templates. The templates are used to map each occurrence of imprecise predicates in the constraints to a set of concretizations. These concretizations are combined to generate the possible concretizations of the constraints. *3)* Finally, using Liquid Haskell's constraint solving it decides the validity of each concretized constraint, while all the safe concretizations SCs are interactively presented to the user.

The implementation of `GuiLT` closely follows the theory of § 4, apart from a syntactic detail (we use ?? instead of ? to denote the unknown part of a refinement), and several practical adjustments that we discuss here and are crucial for real-world applicability.

Templates. To generate the refinement templates we use Liquid Haskell's existing API. The generated templates consist of a predefined set of predicates for linear arithmetic ($v\ [<\ |\ \leq\ |\ >\ |\ \geq\ |\ =\ |\ \neq]\ x$), comparison with zero ($v\ [<\ |\ \leq\ |\ >\ |\ \geq\ |\ =\ |\ \neq]\ 0$) and length operations ($\text{len}\ v[\geq\ |\ >]0$, $\text{len}\ v = \text{len}\ x$, $v = \text{len}\ x$, $v = \text{len}\ x + 1$), where `v` and `x` respectively range over the refinement



```
{-@ (!!) :: xs:[a] -> {i:Int | ?? } -> a @-}
(x:_) !!0 = x
(_:xs)!!i =                    xs!!(i- 1)
_     !!_ =  error "Out of Bounds!"


client =  [1, 2, 3] !! 3
```

```
{-@ (!!) :: xs:[a] -> {i:Int | ?? } -> a @-}
(x:_) !!0 = x
(_:xs)!!i =                    xs!!(i- 1)
_     !!_ =  error "Out of Bounds
                                   i < len xs&& i >= 0
             i < len xs&& i == 0

client =  [1, 2, 3] !! 3
          i <= len xs&& i > 0
```

Fig. 4. GuiLT GUI for error explanation of incompatible indexing.

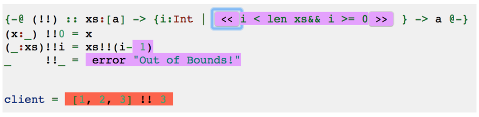
variable and any program variable. Application-specific templates are automatically abstracted from user-provided specifications and the user can explicitly define custom templates. For instance, assume a user defined data type `Tree a` and an uninterpreted function `size :: Tree a → Int`. If the user writes a specification `x: Tree a → {v:Int | v < size x}`, then the template `v < size x` is generated where `v` and `x` respectively range over the refinement variable and any program variable.

Depth. For completeness in the theory, we check the validity of any solution, including all possible *conjunctions* of elements of $\mathbb{Q}$, which is not tractable in practice. The implementation uses an instantiation depth parameter that is 1 by default, meaning each ?? ranges over single templates. At depth 2, ?? ranges over conjunctions of (single) templates, *etc.*

Sensibility Checking. Each ?? can be instantiated with any templates that are local and specific. The implementation uses the SMT solver to check both. As an *optimization*, we perform a syntactic locality check to reject templates that are "non-sensible", for instance, syntactic contradictions of the form `x < v && v < x`. As a *heuristic*, we further filter out as non-sensible type-directed instantiations of the templates that, based on our experience, a user would not write, such as arithmetic operations on lists and booleans (*e.g.* `x < False`)—although potentially correct in Haskell through overloading.

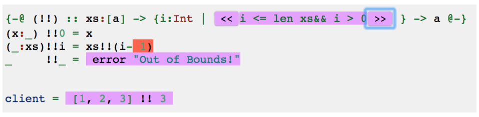
Locality Checking. To encode Haskell functions (*e.g.* `len`) in the refinements, Liquid Haskell is using uninterpreted SMT functions [Vazou et al. 2018]. As Lehmann and Tanter [2017] note, locality checking breaks under the presence of uninterpreted functions. For instance, the predicate `0 < len i` is not local on `i`, because `∃i. 0 < len i` is not SMT valid due to a model in which `len` is always negative. To check locality under uninterpreted functions we define a fresh variable (*e.g.* `leni`) for each function application (*e.g.* `len i`). For instance, `0 < len i` is local on `i` because under the new encoding `∃leni. 0 < leni` is SMT valid.

Partitions. A critical optimization for efficiency is that after generation and before solving, the set of constraints is partitioned based on the constraint dependencies so that each partition is solved independently. Two constraints depend on each other when they contain the same liquid variable or when they contain different variables (*e.g.* k_1 and k_2) that depend on each other (*e.g.* $\texttt{x:\{}k_1\texttt{\}} \vdash \texttt{\{v|true\}} \preceq \texttt{\{v|}k_2\texttt{\}}$). That way, we reduce the number of ?? that appear in each set of independent constraints and thus the number of concretization combinations that need to be checked (which increases exponentially with the number of ?? accounting for all combinations of all concretizations).



```
{-@ (!!) :: xs:[a] -> {i:Int | << i < len xs&& i >= 0 >> } -> a @-}
(x:_) !!0 = x
(_:xs)!!i = xs!!(i- 1)
_     !!_ = error "Out of Bounds!"


client = [1, 2, 3] !! 3
```



```
{-@ (!!) :: xs:[a] -> {i:Int | << i <= len xs&& i > 0 >> } -> a @-}
(x:_) !!0 = x
(_:xs)!!i = xs!!(i- 1)
_     !!_ = error "Out of Bounds!"


client = [1, 2, 3] !! 3
```

Fig. 5. GuiLT GUI for liquid types exploration.

6 APPLICATION I: ERROR EXPLANATION

We illustrate how GuiLT can be used interactively for error explanation. Consider the list indexing function (!!) in Haskell:

```
(x:_)  !! 0 = x
(_:xs) !! i = xs!!(i - 1)
_      !! _ = error "Out of Bounds!"
```

Indexing is 0-based and signals a runtime error for `length xs <= i`. Now consider a client indexing a list by its length:

```
client = [1, 2, 3] !! 3
```

Refinement types can be used to impose a false precondition on the error function and detect the out-of-bound crash:

```
error :: {i:String | false} → a
```

But when the two above incompatible definitions coexist is the error due to the *definition* of indexing, or to the *client* of indexing?

This question of who is to blame has no definitive answer in the general case. Gradual liquid type inference is helpful to explore possible resolutions of the error and decide for a suitable solution. To do so, we transform the statically ill-typed program into a gradually well-typed program by giving an imprecise refinement as a precondition to the indexing function:

```
(!!) :: xs:[a] → {i:Int | ?? } → a
```

Using GuiLT, we can explore all potential predicates that can be substituted for ??. Among the candidates are both `0 <= i < len xs` and `0 < i <= len xs`. While the former induces a type error in the client, the latter induces a type error in the definition of (!!).

Figure 4 was generated after running GuiLT on the above specification and code. The result of GuiLT is an .html file where each user specified ?? is turned into a colored button. Pressing a ?? button once highlights all of its usage occurrence, using different colors for each ?? (Figure 4(left)). Pressing it again presents all safe concretizations to scroll through (Figure 4(right)). In the indexing example, three gradual constraints are generated: *1)* for the recursive call of indexing, *2)* for the unreachable (due to the false precondition) error call, and *3)* for the client. Unsurprisingly, the SCs for the recursive call and the client include the incompatible `0 <= i < len xs` and `0 < i <= len xs`, respectively. Interestingly, the unreachable constraint enjoys many SCs including `0 <= i <= len xs` and one given in the right of Figure 4, *i.e.* `i < len xs && i == 0`, since the case of indexing 0 from a non-empty list is covered in the first case of the indexing function.

The user explores all SCs with the << and >> buttons. In the example, the three SCs are independent, but in many cases SCs can depend on each other (*e.g.* dependencies in function preconditions). In such cases, pressing a navigation button changes the values of all the dependent occurrences.

When the goal is to replace the ?? with a concrete refinement, going through all SCs can be overwhelming. In the example, there are 22, 10, and 13 SCs for the unreachable, client, and recursive calls, respectively. To accommodate the user, GuiLT generates an alternative interactive .out.html

Depth	# ?	Occs	Cands	Sens	Local	Spec	Parts	# γ	SCs	Sols	Time (s)
1	1	[3]	[12*]	[11*]	[11*]	[11*]	3/12	11*	[8,0,6]	0	0.47
2	1	[3]	[68*]	[38*]	[34*]	[34*]	3/12	34*	[22,10,13]	0	4.91

Table 1. Quantitative Evaluation of the Indexing Example

file by which the user can replace the `??` with any SC and observe the generated refinement errors. In Figure 5-left the `??` is replaced with the concrete refinement `0 <= i < len xs`, generating an error at the client site. Pressing >>, the user explores the next concrete refinement, where `0 < i <= len xs` generates an error in the recursive case of indexing (Figure 5-right). This exposes all the concretizations of `??` that render at least one constraint safe (here 22).

Gradual types for Error Explanation. This interactive replacement of `??` allows the user to try all the possible concretizations inferred by the algorithm. For each selection the associated type errors are generated and highlighted. This process is effective for two major reasons:

(1) *Provide all error locations:* The inference algorithm does not arbitrarily blame one program location based on one inferred predicate, instead the user navigates via *all* the potential error locations. Thus, during navigation the user can observe all potential sources of program errors.
(2) *Predicate Synthesis:* Moreover, the user does not have to come up with the refinements, since the inference procedure synthesizes them. It is only up to the user to pick the adequate refinement that she chooses as the desired specification—which could be fully static, or imprecise.[4]

Quantitative Evaluation. Table 1 summarizes a quantitative evaluation of the indexing example. We run `GuiLT` with instantiation depth (Depth) 1 and 2, *i.e.* the size of the conjunctions of the templates we consider (§ 5). The # ? column gives the number of imprecise refinements (as added by the user) and Occs gives the number each `??` appearing in the generated constraints. Column Cands denotes the candidate (*i.e.* well-typed) templates for each occurrence (§ 5). In the example, for depth 2, 68 templates are generated for each occurrence (*i.e.* [68, 68, 68] simplified as 68* for space). Then, `GuiLT` decides how many of the templates are sensible (Sens), local (Local), and specific (Spec); here, 38, 34, and 34, respectively. We note that all the local templates are specific, when the static part of the gradual refinement is true (*i.e.* `true && ??`, simplified as `??`). Moreover, note that most non-local solutions were filtered out by the sensibility check. The constraints were split in 12 partitions (Parts), out of which 3 contained gradual refinements and had to be concretized. Each partition had 34 concretizations (# γ) out of which 22, 10, and 13 were safe concretizations (SCs), respectively. None of these concretizations was common for all the occurrences of the `??`, thus the `GuiLT` reports 0 static solutions (Sols), as expected. Finally, the running time of `GuiLT` was 4.91 sec.

In short, we observe how the implementation optimizations, as discussed in § 5 , allow for practical error reporting using gradual liquid types. With depth of 2 and 3 occurrences of the ? each with 68 candidates, the exponential inference algorithm would need to run 68^3 times. In practice, we filter the candidates to 34 and since each occurrence appears in a different partition, we only run the algorithm $34 * 3$ times. Next, we use these metrics to evaluate `GuiLT` as a practical tool for migrating three standard Haskell list libraries to Liquid Haskell.

7 APPLICATION II: MIGRATION ASSISTANCE

As a second application, we use `GuiLT`'s error reporting GUI from § 6 to assist the migration of commonly used Haskell libraries to Liquid Haskell. Inter-language migration to strengthen type

[4]For sound imprecise refinements, runtime support for gradual refinements is needed, as described by Lehmann and Tanter [2017]. Implementing such support in Liquid Haskell is an interesting perspective, outside the scope of this work.

safety guarantees is one of the main motivations for gradual type systems [Tobin-Hochstadt and Felleisen 2006]. Our study confirms that gradual liquid type inference provides an effective bridge to migrate programs written in a traditional polymorphic functional language like Haskell to a stronger discipline with refinement types as provided by Liquid Haskell.

Benchmarks. We used `GuiLT` to migrate three interdependent Haskell list libraries:

- `GHC.List`: provides commonly used list functions available at Haskell's `Prelude`,
- `Data.List`: defines more sophisticated list functions, *e.g.* list transposing, and
- `Data.List.NonEmpty`: lifts list functions to a non-empty list data type.

Migration Process. A library consists of a set of function imports and definitions. Each function comes with its Haskell type and may be assigned to a gradual refinement type during migration. Migration is complete when, if possible, all functions are given fully-static refinement types, and type check under Liquid Haskell. The process proceeds in four steps:

Step 1: run Liquid Haskell to generate a set of type errors,
Step 2: fix the errors by *manually* inserting gradual refinements (`??`),
Step 3: use `GuiLT` to replace `??` with an *automatically* generated SC, and
Step 4: go back to **Step 1** until no type errors are reported.

This process is iterative and interactive since refinement errors propagate between imported and client libraries and it is up to the user to decide how to resolve these errors.

Step 1: At the beginning of the migration process the source files given to Liquid Haskell have no refinement type specifications. Still, there are two sources[5] of refinement type errors: *1)* failure to satisfy imported functions preconditions, *e.g.* in § 6 the `error` function assumes the `false` precondition and *2)* incomplete patterns, *i.e.* a pattern-match that might fail at runtime, *e.g.* `scanr`'s result is matched to a non-empty list.

```
scanr _ q []     =  [q]
scanr f q (x:xs) =  f x q : qs where qs@(q:_) = scanr f q xs
```

Each time **Step 1** is reiterated, new type errors can occur as a consequence of new function preconditions added in the following steps.

Step 2: The insertion of gradual refinements (`??`) is left to the user; they can add `??` either in the preconditions of defined functions or postconditions of imported functions, thereby resolving the type errors of **Step 1**. If the user places redundant `??`, then they will be solved to the `True` static refinement. If the user misses some `??`, type errors will remain.

Step 3: Once the program gradually typechecks, the user explores the generated SCs, and chooses which one to replace `??` with. In our benchmarks, often, the decision is trivial since only one SC coincides for all occurrences of the gradually-refined variable.

Step 4: Depending on the refinement inserted at **Step 3**, new errors might appear in the current or imported libraries; there are three possible scenarios:

(a) If the user refined the precondition of a function (*e.g.* `head` requires non-empty lists), then a type error might be generated at clients of the function both inside and outside the library;
(b) If the user refined the post-condition of a function (*e.g.* `scanr` always returns non-empty lists), then no error can be generated;
(c) If the user refined the postcondition of an imported function upon which the function to be verified relies, then the imported function's specification may not be satisfied by its

[5]Termination checking, by default activate in Liquid Haskell, is a third source of type errors in programs without any refinement type specifications. To simplify our case study we deactivated termination checking.

implementation. In this case the user can either *assume* the imported type (thus relying on gradual checking), or update and re-check the imported library.

Rnd	Function	# ?	Occs	Cands	Spec	Parts	# γ	SCs	Sols	Time (s)
`GHC.List` (56 functions defined and verified)										
1st	`errorEmp`	1	[1]	[[5]]	[[4]]	1/4	[4]	[0]	0	1.00
	`scanr`	1	[4]	[6*]	[5*]	1/5	[625]	[125]	1	4.6K
	`scanr1`	2	[4,6]	[12*,5*]	[5*,4*]	1/5	[2.5M]	N/A	N/A	N/A
2nd	`head`	1	[1]	[[5]]	[[4]]	1/3	[4]	[1]	1	0.70
	`tail`	1	[1]	[[5]]	[[4]]	1/5	[4]	[1]	1	0.77
	`last`	1	[2]	[5*]	[4*]	2/4	4*	[4,1]	1	1.04
	`init`	1	[3]	[5*]	[4*]	2/8	[16,4]	[16,1]	1	3.12
	`foldl`	1	[3]	[5*]	[4*]	2/5	[4,16]	[1,16]	1	2.41
	`foldr1`	1	[1]	[[5]]	[[4]]	1/2	[4]	[1]	1	1.08
	`(!!)`	2	[4,4]	[5*,10*]	[4*,9*]	4/9	36*	[12,24,36,36]	6	7.81
	`cycle`	1	[2]	[5*]	[4*]	2/6	4*	[4,1]	1	1.37
3rd	`maximum`	1	[3]	[5*]	[4*]	2/4	[4,16]	[1,16]	1	3.38
	`minimum`	1	[3]	[5*]	[4*]	2/4	[4,16]	[1,16]	1	2.80
`Data.List` (115 functions defined and verified)										
1st	`maximumBy`	1	[3]	[5*]	[4*]	2/5	[16,4]	[16,1]	1	2.24
	`minimumBy`	1	[3]	[5*]	[4*]	2/5	[16,4]	[16,1]	1	2.40
	`transpose`	1	[3]	[12*]	[5*]	2/11	[25,5]	[0,4]	1	51.02
	`genIndex`	2	[6,6]	[2*,5*]	[1*,4*]	6/12	4*	[3,4,4,1,1,4]	0	1.83
`Data.List.NonEmpty` (57 functions defined and verified)										
1st	`fromList`	1	[2]	[5*]	[4*]	2/11	4*	[4,1]	1	1.41
	`cycle`	1	[1]	[[2]]	[[1]]	1/3	[1]	[0]	0	1.27
	`- toList`	2	[1,2]	[[2],5*]	[[1],4*]	2/3	4*	[1,3]	1	3.11
	`(!!)`	1	[4]	[10*]	[9*]	4/22	9*	[9,4,4,9]	1	5.96
2nd	`cycle`	2	[1,1]	[[12],[5]]	[[5],[1]]	1/2	[5]	[2]	2	3.13
	`lift`	1	[1]	[[6]]	[[5]]	1/2	[5]	[2]	2	2.90
	`inits`	1	[1]	[[6]]	[[5]]	1/5	[5]	[2]	2	2.95
	`tails`	2	[1,1]	[[5],[6]]	[[4],[5]]	1/5	[20]	[6]	6	10.22
	`scanl`	1	[1]	[[6]]	[[5]]	1/5	[5]	[2]	2	2.23
	`scanl1`	1	[1]	[[6]]	[[5]]	1/2	[5]	[1]	1	1.13
	`insert`	1	[1]	[[12]]	[[5]]	1/5	[5]	[2]	2	2.25
	`transpose`	2	[1,1]	[[7],[12]]	[[6],[5]]	1/7	[30]	[6]	6	37.48
3rd	`reverse`	2	[1,1]	[[5],[12]]	[[4],[5]]	2/3	[5,4]	[2,3]	5	0.96
	`sort`	2	[1,1]	[[12],[5]]	[[5],[4]]	2/3	[5,4]	[2,3]	5	1.03
	`sortBy`	2	[1,1]	[[12],[5]]	[[5],[4]]	2/3	[5,4]	[2,3]	5	0.97

Table 2. Evaluation of Migrations Assistance. Rnd: number of iterations to verify the function. Function: name of the function. # ? : number of ? inserted. Occs: times each ? is used. For each occurrence, we give the number of template candidates (Cands) and how many are specific (Spec). Parts: the number of partitions. For each partition, we give the number of concretizations (# γ) and safe concretizations (SCs). Sols: number of static solutions found. Time: time in sec.

Evaluation. Table 2 summarizes the migration case study; there are three subtables, one for each library: `GHC.List`, `Data.List`, and `Data.List.NonEmpty`. Within each of these tables, there is a row for every function which requires a refinement type to type check. Rows are collected into groupings of rounds (*i.e.* **Steps 1 - 4**), where round i lead to refinement type preconditions that trigger type errors in the functions of round $i + 1$. The columns of the table have the same meaning as that of Table 1; all results are for depth 1.

`GHC.List`*:* Liquid Haskell reported three static errors on the original version of `GHC.List`, *i.e.* with no user refinement type specifications. The function `errorEmp` is rejected as it is merely a wrapper around the `error` function; `scanr` and `scanr1` each have incomplete patterns assuming a non-empty list postcondition. `GuiLT` performed bad at all these three initial cases. It was unable to generate any SC for `errorEmp` , since the required false precondition is non local. It required more than one hour to generate SCs of `scanr` and it timed-out in the case of `scanr1`. The reason for this is that ?? in post-conditions generated dependent set of constraints that made the partitioning optimization (§ 5) useless, despite its importance in the other scenarios. Yet, `GuiLT` was very efficient in the next two rounds. The specification of `errorEmp` introduced errors in eight functions that were fixed using `GuiLT`: in seven cases the generated SCs correspond to exactly one predicate that was used to replace the ??. One of these functions, `foldl`, was further used by two more functions that were interactively migrated at the third and final specification round.

In short, out of the 56 functions defined, 13 required refinement type specifications, since the rest were typed with the true-default types generated by Liquid Haskell. `GuiLT` was unusable in 2 cases, but synthesized exactly one, the correct, predicate in 10 of them. For the case of `(!!)`, `GuiLT` generated 6 correct predicates, out of which we manually picked the most general one. The two cases where `GuiLT` was unusable are due to many dependencies within the same partition; as future work, we should explore whether it is possible to devise a more advanced partitioning scheme.

`Data.List`*:* Migration of `Data.List` only required one round. Four functions errored due to incomplete patterns or violation of preconditions of functions imported from previously verified `GHC.List`. Unsurprisingly, `GuiLT` was unable to find any SCs for `genIndex`, a generic variant of `(!!)` that indexes lists using any integral (instead of integer) index, because it lacks arithmetic templates for integrals. To complete migration, the user needs to either manually provide the refinement type of `genIndex` or appropriately extend the set of templates.

`Data.List.NonEmpty`*:* Migration was more interesting for the `Data.List.NonEmpty` library that manipulates the data type `NonEmpty a` of non-empty lists. The first round exposed that `fromList` requires the non-empty precondition. The `GHC.List` function `cycle` has a non-empty precondition, thus lifted to non-empty lists does not type check.

```
 cycle :: NonEmpty a → NonEmpty a
 cycle = fromList . List.cycle . toList
```

To migrate `cycle`, we first gradually refined the result type of `toList :: NonEmpty a → {xs:[a] | ??}` for which `GuiLT` suggested the single static refinement of `0 < len xs`.

Verification of non-empty list indexing calls requires invariants that relate the lengths of the empty and non-empty lists. Similarly, `GuiLT` finds SCs only after predicate templates that express such invariants are added. In general, to aid migration on user-defined structures, `GuiLT` requires the definition of domain-specific templates.

On the second round, the non-empty precondition of `fromList` triggers errors to eight clients (step 4a), all of which call functions that return (non-provably) non-empty lists. For example, `inits` lifts `List.inits` to non-empty lists.

```
inits = fromList . List.inits . toList
```

To migrate such functions, we first assume an unknown specification, for example:

```
assume List.inits :: [a] → {o:[a] | ?? }
```

We then use `GuiLT` to solve the unknown specification to `0 < len o`. At this point, we can either update the imported function with the discovered static specification and recheck the library (step 4c), or stick to gradual checking and stay with the unknown specification for `inits`.

The function `cycle` reappears in the second round, due to the new precondition of `fromList`. Since the imported `List.cycle` already has a precondition `{i:[a] | 0 < len i}`, at this round we further strengthened the existing precondition with the imprecise refinement:

```
assume List.cycle :: i:{ [a] | 0 < len i && ?? } → {o:[a] | ?? }
```

This is the only case in our experiments where we used a gradual refinement with a static part and thus the only case in which some local templates are rejected as non-specific (here 3 out of 4 local templates are not specific).

Finally, we use `GuiLT` to derive higher-order specifications. The `lift` function lifts a list transformation to non-empty lists, `lift f = fromList . f . toList`, and comes with a comment that "If the provided function returns an empty list, this will raise an error.". Alerted by this comment, we use a ?? in higher-order position:

```
lift :: (i:[a] → {o:[b] | ??}) → NonEmpty a → NonEmpty b
```

`GuiLT` produces two static solutions `0 < len o` and `len i == len o`. We choose the second, which leads to type errors in three clients, which are resolved in later rounds.

To sum up, `GuiLT` indeed is aiding Haskell to Liquid Haskell migration of real libraries, since the user can place the initial ??s and choose from the suggested SCs, instead of writing specifications from scratch. Often times, there is only one possible SC coinciding to all concretizations, thus the choice is trivial. When no suggestions are generated, *e.g.* `errorEmp` or `genIndex`, the user can fall back to the standard verification process.

Automatic insertion of ??s would further reduce the required user input, but is left as future work. Theorem 4.3 suggests a complete way to automatically insert ??s so that the migrating libraries gradually type check. The types suggested by this theorem are very imprecise, yet they provide a sound starting point, upon which one can increase precision so long as the libraries type check.

8 RELATED WORK

Liquid Types. Dependent types allow arbitrary expressions at the type level to express theorems on programs, while theorem proving is simplified by various automations ranging from tactics (*e.g.* Coq [Bertot and Castéran 2004], Isabelle [Wenzel 2016]) to external SMT solvers (*e.g.* F* [Swamy et al. 2016]). Liquid types [Rondon et al. 2008] restrict the expressiveness of the type specifications to decidable fragments of logic to achieve decidable type checking and inference.

Gradual Refinement Types. Several refinement type systems mix static verification with runtime checking. Hybrid types [Knowles and Flanagan 2010] use an external prover to statically verify subtyping when possible, otherwise a cast is automatically inserted to defer checking at runtime. Soft contract verification [Nguyen et al. 2014, 2018; Tobin-Hochstadt and Horn 2012] works in the other direction, statically verifying contracts wherever possible, and otherwise leaving unverified contracts for checking at runtime. Ou et al. [2004] allow the programmer to explicitly annotate whether an assertion is verified at compile- or runtime. Manifest contracts [Greenberg et al. 2010] formalize the metatheory of refinement typing in the presence of dynamic contract checking. Lehmann and Tanter [2017] developed the first gradual refinement type system, which adheres to the refined criteria of Siek et al. [2015]. None of these systems support inference; on the contrary, because refinements can be arbitrary, inference is impossible in these systems. Here we restrict gradual refinements to a finite set of predicates, in order to achieve inference by adaptation of the liquid inference procedure.

Gradual Type Inference. Several approaches have been developed to combine type inference and gradual types. Siek and Vachharajani [2008] infer gradual types using unification, while Rastogi et al. [2012] exploit type inference to improve the performance of gradually-typed programs. Garcia and Cimini [2015] lift an inference algorithm from a core system to its gradual counterpart, by ignoring the unification constraints imposed by gradual types. In contrast, in gradual liquid inference the constraints imposed by gradual refinements cannot be ignored, since unlike Hindley-Milner inference, the liquid algorithm starts from the strongest solution for the liquid variables and uses constraints to iteratively weaken the solution. Our work is the first gradual inference work to be systematically derived from the static algorithm based on the Abstracting Gradual Typing approach (AGT) [Garcia et al. 2016], and proven to satisfy the static gradual guarantee [Siek et al. 2015].

Error Reporting. Properly localizing, diagnosing and reporting type errors is a long-standing challenge, especially for inference algorithms. Since the seminal work of Wand [1986], which keeps track of all unification steps in order to help debugging, many algorithms have been proposed to better assist programmers. A large variety of techniques has been explored, including slicing [Haack and Wells 2003; Tip and Dinesh 2001], heuristics [Zhang et al. 2015], SMT constraint solving [Pavlinovic et al. 2014], counter-example generation [Nguyen and Horn 2015; Seidel et al. 2016], machine learning [Seidel et al. 2017], and other search-based approaches, such as Seminal [Lerner et al. 2007] and counterfactual change inference [Chen and Erwig 2018].

Apart from Seidel et al. [2016], which focuses on identifying dynamic witnesses for static refinement type errors, none of the above approaches target refinement types. In this work, we uncover a novel application of gradual typing for error explanation, by observing that the notion of concretizations of gradual types that stem from AGT, embed useful justifications (or lack thereof) that can be exploited to identify refinement errors and guide migration.

Gradual Program Migration. Recently, Campora et al. [2018] exploit variational typing [Chen et al. 2014] to assist programmers in making their program more static, by analyzing the impact of replacing unknown types with static types. Variational typing greatly reduces the complexity of exploring all possible combinations. Though the considered type system is simple, it seems likely that the approach could be extended to refinements and combined with our technique. This might be an effective way to address the scalability issues we have encountered with our implementation approach in certain scenarios.

9 CONCLUSION

This paper makes the novel observation that gradual inference based on abstract interpretation can be fruitfully exploited to assist in explaining type errors and migrating programs to a stronger typing discipline. We develop this intuition in the context of refinement types, yielding a novel integration of liquid type inference and gradual refinements. Gradual liquid type inference computes possible concretizations of unknown refinements in order for a program to be well-typed. In addition to laying down the theoretical foundations of gradual liquid type inference, we provide an implementation integrated with Liquid Haskell. Thanks to a number of heuristics and optimizations, the current implementation of the interactive tool `GuiLT` proves useful for migrating existing Haskell libraries to the stronger discipline of Liquid Haskell. Our experience however also shows that enhancing the scalability of our prototype further is necessary; we believe that variational typing could be particularly helpful towards that goal. Similarly, integrating prior work on ranking suggested type error sources would help our inference algorithm suggest the most relevant concretizations first. Finally, we conjecture that the idea of using gradual typing for error reporting and migration generalizes to other typing disciplines.

REFERENCES

Y. Bertot and P. Castéran. 2004. *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *PACMPL (POPL)* 2 (2018), 15:1–15:29.

Sheng Chen and Martin Erwig. 2018. Systematic identification and communication of type errors. *JFP* 28 (2018), 2:1–2:48.

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *TOPLAS* 36, 1 (2014), 1:1–1:54.

B. Courcelle and J. Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach* (1st ed.). Cambridge University Press.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *PLDI*.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *POPL*.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing *(POPL)*.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. *POPL*.

Christian Haack and Joe Wells. 2003. Type Error Slicing in Implicitly-Typed Higher-Order Languages. In *ESOP*. 284–301.

K.W. Knowles and C. Flanagan. 2010. Hybrid type checking. *TOPLAS*.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types *(POPL)*.

Benjamin Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. In *PLDI*. 425–434.

Supplementary Material. 2018. Technical Report: Gradual Liquid Type Inference.

Phuc C. Nguyen and David Van Horn. 2015. Relatively complete counterexamples for higher-order programs. In *PLDI*.

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *ICFP*.

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft Contract Verification for Higher-order Stateful Programs. In *POPL*.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). *IFIP*.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding minimum type error sources. In *OOPSLA*. 525–542.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. In *POPL*.

P. Rondon. 2012. *Liquid Types*. Ph.D. Dissertation. UC San Diego.

P. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI*.

Eric Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *ICFP*. 228–242.

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to blame: localizing novice type errors with data-driven diagnosis. OOPSLA (2017).

Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Dynamic Languages Symposium*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL*.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*.

Frank Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type error. *TOSEM* 10, 1 (2001), 5–55.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: From scripts to programs. In *DLS*.

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order Symbolic Execution via Contracts. In *OOPSLA*.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with Refinement Types in the Real World. In *Haskell*.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014b. Refinement Types for Haskell. (2014).

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan Scott, Ryan Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. In *POPL*.

Mitchell Wand. 1986. Finding the Source of Type Errors. In *POPL*.

Makarius Wenzel. 2016. The Isabelle System Manual. (2016). https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2016-1/doc/system.pdf

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *PLDI*.

A AUXILIARY DEFINITIONS AND PROOFS

We now provide the full inference algorithm and prove the theorems of § 4.4.

A.1 The Inference Algorithm

We define Infeř as in § 4.4:

```
Infeř :: Ěnv → Exǩr → Quals → [Typě]
Infeř Γ̌ ě ℚ = { ť′ | Just ť′ <- A <*> ť
                     , A ∈ Solvě A₀ Č ℚ }
  where
    A₀ = λκ. ℚ
    (ť, Č) = Cons Γ̌ ě

Solvě :: Sol → [Cons̆] → Quals → [Maybe Sol]
Solvě A Č ℚ = {Solve A Ĉ  |  Ĉ ∈ γ_ℚ(Č)}
```

Note that unlike [Rondon et al. 2008], for simplicity, we assume that the set of refinement predicates $\mathbb{Q}$ is not a set of templates, but an “instantiated” set of predicates. The algorithmic concretization on a list of constraints is defined as follows

$$
\begin{aligned}
\gamma_{\mathbb{Q}}(\check{p}(x)) &\doteq 2^{\mathbb{Q}} \cap \gamma(\check{p}(x))\\
\gamma_{\mathbb{Q}}(\check{\Gamma}) &\doteq \{\hat{\Gamma} \mid x{:}\hat{t} \in \hat{\Gamma} \textit{ iff } x{:}\check{t} \in \check{\Gamma}, \hat{t} \in \gamma_{\mathbb{Q}}(\check{t})\}\\
\gamma_{\mathbb{Q}}(\check{\Gamma} \vdash \check{t}_1 \preceq \check{t}_2) &\doteq \{\hat{\Gamma} \vdash \hat{t}_1 \preceq \hat{t}_2 \mid \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t}_i \in \gamma_{\mathbb{Q}}(\check{t}_i)\}\\
\gamma_{\mathbb{Q}}(\check{\Gamma} \vdash \check{t}) &\doteq \{\hat{\Gamma} \vdash \hat{t} \mid \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t} \in \gamma_{\mathbb{Q}}(\check{t}), \}\\
\gamma_{\mathbb{Q}}(\check{C}) &\doteq \{\hat{C} \mid c \in \hat{C} \textit{ iff } \check{c} \in \check{C}, \hat{c} \in \gamma_{\mathbb{Q}}(\check{c})\}
\end{aligned}
$$

We complete the definitions by repeating the Solve and Cons algorithms from [Rondon et al. 2008].

The procedure Solve A $\hat{C}$ repeatedly weakens the solution A until the set of constraints $\hat{C}$ is satisfied, or returns Nothing.

```
Solve :: Sol → [Conŝ] → Maybe Sol
Solve A Ĉ =
  if exists ĉ ∈ Ĉ s.t. ¬isValid(A · ĉ)
  then case Weaken ĉ A of
         Just A′ → Solve A′ Ĉ
         Nothing → Nothing
  else Just A

Weaken :: Conŝ → Sol → Maybe Sol
Weaken (Γ̂ ⊢ {v:b | θ · κ}) A = Just $
  A[κ ↦ {q|q ∈ A · κ, isValid(A · Γ̂ ⊢ {v:b | θ · q})}]
Weaken Γ̂ ⊢ {v:b | p} ≤ {v:b | θ · κ} A = Just $
  A[κ ↦ {q|q ∈ A · κ, isValid(A · Γ̂ ⊢ {v:b | A · p} ≤ {v:b | θ · q})}]
Weaken _ _  = Nothing
```

The procedure Cons $\check{\Gamma}$ $\check{e}$ is following the rules of Figure 2 to generate a template type of the expression $\check{e}$ and the set of constraints that should be satisfied.

```
Cons :: Ěnv → Exǩr → (Maybe Typě, [Cons̆])
Cons Γ̌ ě =
  let (ť,Č) = Gen Γ̌ ě
  ($\check{t}$, Split $\check{C}$)
```

The procedure Gen $\check{\Gamma}$ $\check{e}$ is using the typing rules of Figure 2 to generate a template type and a set of constraints.

```
Gen :: $\check{\text{Env}}$ → $\check{\text{Expr}}$ → (Maybe $\check{\text{Type}}$, [$\check{\text{Cons}}$])
Gen $\check{\Gamma}$ $x$
  = if $\check{\Gamma}(x)$ = $\{v{:}b \mid \_\}$
     then (Just $\tilde{R}vbv = x$, $\emptyset$)
     else (Just $\check{\Gamma}(x)$, $\emptyset$)
Gen $\check{\Gamma}$ $c$
  = (Just $ty(c)$, $\emptyset$)
Gen $\check{\Gamma}$ ($\check{e}$ :: $\check{t}$) =
  let (Just $\check{t}_e$, $\check{C}$) = Gen $\check{\Gamma}$ $\check{e}$ in
  (Just $\check{t}$, ($\check{\Gamma} \vdash \check{t}_e \preceq \check{t}$, $\check{\Gamma} \vdash \check{t}$, $\check{C}$))
Gen $\check{\Gamma}$ ($\lambda x.\check{e}$) =
  let Just $x{:}\check{t}_x \to \check{t}$ = Fresh $\check{\Gamma}$ $\lambda x.\check{e}$ in
  let (Just $\check{t}_e$,$\check{C}$) = Gen ($\check{\Gamma}, x{:}\check{t}_x$) $\check{e}$ in
  (Just ($x{:}\check{t}_x \to \check{t}$), ($\check{\Gamma} \vdash \check{t}_e \preceq \check{t}$, $\check{\Gamma} \vdash x{:}\check{t}_x \to \check{t}$,$\check{C}$))
Gen $\check{\Gamma}$ ($\check{e}$ $y$) =
  let (Just ($x{:}\check{t}_x \to \check{t}$), $\check{C}_1$) = Gen $\check{\Gamma}$ $\check{e}$ in
  let (Just $\check{t}_y$,$\check{C}_2$) = Gen $\check{\Gamma}$ $y$ in
  (Just $\check{t}[y/x]$,($\check{\Gamma} \vdash \check{t}_x \preceq \check{t}_y$, $\check{C}_1$ ∪ $\check{C}_2$))
Gen $\check{\Gamma}$ if $x$ then $\check{e}_1$ else $\check{e}_2$) =
  let Just $\check{t}$ = Fresh $\check{\Gamma}$ if $x$ then $\check{e}_1$ else $\check{e}_2$ in
  let (Just $\check{t}_1$,$\check{C}_1$) = Gen $\check{\Gamma}$,_:$\{v{:}\text{bool} \mid x\}$ $\hat{e}_1$ in
  let (Just $\check{t}_2$,$\check{C}_2$) = Gen ($\check{\Gamma}$,_:$\{v{:}\text{bool} \mid \neg x\}$) $\check{e}_2$ in
  (Just $\check{t}$, ($\check{\Gamma} \vdash \check{t}$, $\check{\Gamma} \vdash \check{t}_1 \preceq \check{t}$, $\check{\Gamma} \vdash \check{t}_2 \preceq \check{t}$,$\check{C}_1$ ∪ $\check{C}_2$))
Gen $\check{\Gamma}$ (let $x = \check{e}_x$ in $\check{e}$) =
  let Just $\check{t}$ = Fresh $\check{\Gamma}$ (let $x = \check{e}_x$ in $\check{e}$) in
  let (Just $\check{t}_x$,$\check{C}_x$) = Gen $\check{\Gamma}$ $\check{e}_x$ in
  let (Just $\check{t}_e$,$\check{C}_e$) = Gen ($\check{\Gamma}, x{:}\check{t}$) $\check{e}$ in
  (Just $\check{t}$, ($\check{\Gamma} \vdash \check{t}$, $\check{\Gamma} \vdash \check{t}_e \preceq \check{t}$, $\check{C}_x$ ∪ $\check{C}_e$))
Gen $\check{\Gamma}$ (let $x{:}\check{t}_x = \check{e}_x$ in $\check{e}$) =
  let Just $\check{t}$ = Fresh $\check{\Gamma}$ (let $x = \check{e}_x$ in $\check{e}$) in
  let (Just $\check{t}_e$,$\check{C}_e$) = Gen ($\check{\Gamma}, x{:}\check{t}$) $\check{e}$ in
  (Just $\check{t}$, ($\check{\Gamma} \vdash \check{t}_x$, $\check{\Gamma} \vdash \check{t}$, $\check{\Gamma} \vdash \check{t}_e \preceq \check{t}$, $\check{C}_x$ ∪ $\check{C}_e$))
Gen _ _ =
  (Nothing, $\emptyset$)


Fresh :: $\check{\text{Env}}$ → $\check{\text{Expr}}$ → Maybe $\check{\text{Type}}$
Fresh $\check{\Gamma}$ $\check{e}$ = HM type inference
```

The procedure Gen is using Fresh, the Hyndler Miller, unrefined type inference algorithm to generate liquid type templates with fresh refinement variables for the unknown types.

Finally, Split C using the well-formedness and sub-typing rules of Figure 2 to split the constraints into basic constraints.

```
Split :: [$\check{\text{Cons}}$]→  [$\check{\text{Cons}}$]
Split ∅
  = ∅
Split (Γ̌ ⊢ {v:b | p},C)
  = (Γ̌ ⊢ {v:b | p}, Split C)
Split (Γ̌ ⊢ x:ť_x → ť,C)
  = Split (Γ̌ ⊢ ť_x, Γ̌,x:ť_x ⊢ ť,C)
Split (Γ̌ ⊢ {v:b | p_1} ⪯ {v:b | p_2}, C)
  = (Γ̌ ⊢ {v:b | p_1} ⪯ {v:b | p_2}, Split C)
Split (Γ̌ ⊢ x:ť_x1 → ť_1 ⪯ x:ť_x2 → ť_2, C)
  = Split (Γ̌ ⊢ ť_2 ⪯ ť_1, Γ̌,x:ť_x2 ⊢ ť_1 ⪯ ť_2, C)
```

A.2 Correctness of Inference

Next, we prove Theorem 4.1. Let $\mathbb{Q}$ be a finite set of predicates from SMT-decidable logic, $\check{\Gamma}$be a gradual liquid environment, and $\check{e}$be a gradual liquid expression.

The proofs rely on the properties of the functions `Solve` and `Cons`. Since these two functions operate on liquid types, and are ignorant of the gradual setting, we directly port the proofs from [Rondon 2012].

Lemma A.1 (Constraint Generation). *Let* $(\texttt{Just}\ \check{t}, \check{C}) = \texttt{Cons}\ \check{\Gamma}\check{e}.\ \check{\Gamma} \vdash \check{e}{:}\check{t}'$ *iff there exists* A *so that* $\check{t}' \equiv A \cdot \check{t}$ *and* $is\tilde{V}alid(A \cdot \check{C})$.

Proof. Following Theorem 4 of Appendix A of [Rondon 2012]. Since `Cons` is just the algorithmic version of the rules of Figure 2 the theorem holds for any refinement type system with refinement variables, when the isValid(·) relation is exactly the same as in the premises of the rules in Figure 2. □

Lemma A.2 (Constraint Solving). *For every set of constraints* $\hat{C}$ *and qualifiers* $\mathbb{Q}$,

(1) `Solve` $(\lambda\kappa.\mathbb{Q})\ \hat{C}$ *terminates.*
(2) If `Solve` $(\lambda\kappa.\mathbb{Q})\ \hat{C} =$ `Just` A *then* $isValid(A \cdot \hat{C})$.
(3) If `Solve` $(\lambda\kappa.\mathbb{Q})\ \hat{C} =$ `Nothing` *then* $\hat{C}$ *has no solution on* $\mathbb{Q}$.

Proof. Theorem 6 of Appendix A of [Rondon 2012]. □

Lemma A.3 (Gradual Validity).

(1) $is\tilde{V}alid(A \cdot \check{c})$ *iff* $\exists \hat{c} \in \gamma_{\mathbb{Q}}(\check{c}).isValid(A \cdot \hat{c})$
(2) $is\tilde{V}alid(A \cdot \check{C})$ *iff* $\exists \hat{C} \in \gamma_{\mathbb{Q}}(\check{C}).isValid(A \cdot \check{C})$

Proof.

(1) By case analysis on the shape of the constraint:
- $\check{c} \equiv \check{\Gamma} \vdash \check{t}_1 \preceq \check{t}_2$.

$$\begin{gathered} \text{is}\tilde{\text{V}}\text{alid}(A \cdot \check{\Gamma} \vdash A \cdot \check{t}_1 \preceq A \cdot \check{t}_2) \\ \Leftrightarrow \\ \exists \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t}_i \in \gamma_{\mathbb{Q}}(\check{t}_i).\text{isValid}(A \cdot \hat{\Gamma} \vdash A \cdot \hat{t}_1 \preceq A \cdot \hat{t}_2) \\ \Leftrightarrow \\ \exists \hat{c} \in \gamma_{\mathbb{Q}}(\check{c}).\text{isValid}(A \cdot \hat{c}) \end{gathered}$$

- $\check{c} \equiv \tilde{\Gamma} \vdash \tilde{t}$.

$$\begin{gathered}\text{is}\tilde{\text{Valid}}(A \cdot \check{\Gamma} \vdash A \cdot \check{t}) \\ \Leftrightarrow \\ \exists \hat{\Gamma} \in \gamma_{\mathbb{Q}}(\check{\Gamma}), \hat{t} \in \gamma_{\mathbb{Q}}(\check{t}).\text{isValid}(A \cdot \hat{\Gamma} \vdash A \cdot \hat{t}) \\ \Leftrightarrow \\ \exists \hat{c} \in \gamma_{\mathbb{Q}}(\check{c}).\text{isValid}(A \cdot \hat{c})\end{gathered}$$

(2) By the definition of $\text{is}\tilde{\text{Valid}}(\cdot)$ and concretization of list of constraints.

$$\begin{gathered}\text{is}\tilde{\text{Valid}}(A \cdot \check{C}) \\ \Leftrightarrow \\ \forall \check{c} \in \check{C}.\text{is}\tilde{\text{Valid}}(A \cdot \check{c}) \\ \Leftrightarrow \\ \forall \check{c} \in \check{C}.\exists \hat{c} \in \gamma_{\mathbb{Q}}(\check{c}).\text{isValid}(A \cdot \hat{c}) \\ \Leftrightarrow \\ \exists \hat{C} \in \gamma_{\mathbb{Q}}(\check{C}).\text{isValid}(A \cdot \check{C})\end{gathered}$$

□

Theorem A.4 (Soundness).
If $\check{t} \in \texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$, *then* $\check{\Gamma} \vdash \check{e}{:}\check{t}$.

Proof. Since $\check{t} \in \texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ then $\exists A$ so that

(1) $\check{t} = A \cdot \check{t}'$
(2) $\texttt{Just}\ A \in \texttt{So}\check{\texttt{l}}\texttt{ve}\ (\lambda\kappa.\mathbb{Q})\ \check{C}\ \mathbb{Q}$
(3) $(\texttt{Just}\ \check{t}', \check{C}) = \texttt{Cons}\ \check{\Gamma}\ \check{e}$

From (2), $\exists \hat{C} \in \gamma_{\mathbb{Q}}(\check{C})$ so that

(4) $\texttt{Just}\ A \in \texttt{Solve}\ (\lambda\kappa.\mathbb{Q})\ \hat{C}$

From (4) and Lemma A.2 we get

(5) $\text{isValid}(A \cdot \hat{C})$

By Lemma A.3 we get

(6) $\text{is}\tilde{\text{Valid}}(A \cdot \check{C})$

By (1), (3), and Theorem A.1,

$$\check{\Gamma} \vdash \check{e}{:}\check{t}$$

□

Theorem A.5 (Completeness).
If $\texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q} = \emptyset$, *then* $\nexists \check{t}.\ \check{\Gamma} \vdash \check{e}{:}\check{t}$.

Proof. Assume that there exists $\check{t}$ so that $\check{\Gamma} \vdash \check{e}{:}\check{t}$. Then, by Lemma A.1, there exists an A so that

(1) $(\texttt{Just}\ \check{t}', \check{C}) = \texttt{Cons}\ \check{\Gamma}\ \check{e}$
(2) $\check{t} = A \cdot \check{t}'$
(3) $\text{is}\tilde{\text{V}}\text{alid}(A \cdot \check{C})$

From (3) and Lemma A.3 $\exists \hat{C} \in \gamma_{\mathbb{Q}}(\check{C})$ so that

(4) $\text{isValid}(A \cdot \hat{C})$

From (4) and inverting 3 of Lemma A.2 we get

(5) $\texttt{Solve}\ (\lambda\kappa.\mathbb{Q})\ \hat{C} \neq \texttt{Nothing}$

By the definition of $\texttt{So}\check{\texttt{l}}\texttt{ve}$ we get

(6) $\exists A'.\texttt{Just}\ A' \in \texttt{So}\check{\texttt{l}}\texttt{ve}\ (\lambda\kappa.\mathbb{Q})\ \check{C}\ \mathbb{Q}$

By the definition of $\texttt{In}\check{\texttt{f}}\texttt{er}$ we get

$$\texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q} \neq \emptyset$$

Since we reached a contradiction, there cannot exist $\check{t}$ so that $\check{\Gamma} \vdash \check{e}{:}\check{t}$. □

Theorem A.6 (Termination). $\texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ *terminates.*

Proof. Since both the set of constraints $\check{C}$ and the set of refinement predicates $\mathbb{Q}$ are finite, the concretizations $\gamma_{\mathbb{Q}}(\check{C})$ are also finite. Thus, $\texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ calls $\texttt{Solve}\ \cdot\ \cdot$ finite times and by Theorem A.2 $\texttt{Solve}\ \cdot\ \cdot$ terminates, thus so does $\texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$. □

A.3 Criteria for Gradual Typing

Finally we prove the static criteria for gradual typing.

(i) Conservative Extension.

Theorem A.7 (Conservative Extension). *If* $\texttt{Just}\ \hat{t} = \texttt{Infer}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q}$, *then* $\texttt{In}\check{\texttt{f}}\texttt{er}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \{\hat{t}\}$. *Otherwise,* $\texttt{In}\check{\texttt{f}}\texttt{er}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \emptyset$.

Proof. If $\texttt{Just}\ \hat{t} = \texttt{Infer}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q}$, then there exists an A, so that
(1) $\hat{t} = A \cdot \hat{t}'$
(2) $\texttt{Just}\ A = \texttt{Solve}\ (\lambda\kappa.\mathbb{Q})\ \hat{C}$ Since the generated constraints $\hat{C}$ contain no ?, then $\{\hat{C}\} =$
(3) $(\texttt{Just}\ \hat{t}', \hat{C}) = \texttt{Cons}\ \hat{\Gamma}\ \hat{e}$
$\gamma_{\mathbb{Q}}(\hat{C})$.
Thus, $\texttt{So}\check{\texttt{l}}\texttt{ve}\ (\lambda\kappa.\mathbb{Q})\ \hat{C}\ \mathbb{Q} = \texttt{Just}\ A$. So, $\texttt{In}\check{\texttt{f}}\texttt{er}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \{\hat{t}\}$.
Otherwise, $\texttt{Infer}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \texttt{Nothing}$, because of a failure either at constraint generation or at solving. In either case $\texttt{In}\check{\texttt{f}}\texttt{er}$ will also return $\emptyset$. □

(ii) Embedding of Unrefined Terms.

Definition A.8 (Unrefined Type & Terms). Unrefined types and terms represent base types and lambda calculus terms that are typed using HindleyMilner inference: $\lfloor\Gamma\rfloor \vdash \lfloor e\rfloor{:}\lfloor t\rfloor$.

$$\lfloor\{v{:}b \mid p\}\rfloor = b$$
$$\lfloor x{:}t \to t\rfloor = \lfloor t_x\rfloor \to \lfloor t\rfloor$$

$$\begin{aligned}
\lfloor c\rfloor &= c\\
\lfloor \lambda x.e\rfloor &= \lambda x.\lfloor e\rfloor\\
\lfloor x\rfloor &= x\\
\lfloor e\ x\rfloor &= \lfloor e\rfloor\ x\\
\lfloor \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rfloor &= \texttt{if}\ x\ \texttt{then}\ \lfloor e_1\rfloor\ \texttt{else}\ \lfloor e_2\rfloor\\
\lfloor \texttt{let}\ x = e_x\ \texttt{in}\ e\rfloor &= \texttt{let}\ x = \lfloor e_x\rfloor\ \texttt{in}\ \lfloor e\rfloor\\
\lfloor \texttt{let}\ x{:}t = e_x\ \texttt{in}\ e\rfloor &= \texttt{let}\ x{:}\lfloor t\rfloor = \lfloor e_x\rfloor\ \texttt{in}\ \lfloor e\rfloor
\end{aligned}$$

Definition A.9 (Imprecise Types & Terms). Imprecise types are refined with only ?. Imprecise terms only use imprecise type annotations.

$$\lceil\{v{:}b \mid p\}\rceil = \{v{:}b \mid ?\}$$
$$\lceil x{:}t \to t\rceil = x{:}\lceil t_x\rceil \to \lceil t\rceil$$

$$\begin{aligned}
\lceil c\rceil &= c' \text{where } c' = c \wedge ty(c') = \lceil ty(c)\rceil\\
\lceil \lambda x.e\rceil &= \lambda x.\lceil e\rceil\\
\lceil x\rceil &= x\\
\lceil e\ x\rceil &= \lceil e\rceil\ x\\
\lceil \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rceil &= \texttt{if}\ x\ \texttt{then}\ \lceil e_1\rceil\ \texttt{else}\ \lceil e_2\rceil\\
\lceil \texttt{let}\ x = e_x\ \texttt{in}\ e\rceil &= \texttt{let}\ x = \lceil e_x\rceil\ \texttt{in}\ \lceil e\rceil\\
\lceil \texttt{let}\ x{:}t = e_x\ \texttt{in}\ e\rceil &= \texttt{let}\ x{:}\lceil t\rceil = \lceil e_x\rceil\ \texttt{in}\ \lceil e\rceil
\end{aligned}$$

Lemma A.10 (Well formedness of imprecise types). $\Gamma \vdash \lceil t\rceil$

Proof. Trivial, since true always belongs to the concretization of imprecise gradual refinements. □

Lemma A.11 (Imprecise Subtyping). *If all of the refinements in t are local, then $\Gamma \vdash t \preceq \lceil t\rceil$ and $\Gamma \vdash \lceil t\rceil \preceq t$.*

Proof. By induction on t.

- $\Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid ?\}$, since $\{v{:}b \mid \texttt{true}\} \in \gamma(\{v{:}b \mid ?\})$.
- $\Gamma \vdash \{v{:}b \mid ?\} \preceq \{v{:}b \mid p\}$, since $\{v{:}b \mid p\} \in \gamma(\{v{:}b \mid ?\})$.
- $\Gamma \vdash x{:}t_x \to t \preceq x{:}\lceil t_x\rceil \to \lceil t\rceil$, since $\Gamma \vdash \lceil t_x\rceil \preceq t_x$ and $\Gamma, x{:}\lceil t_x\rceil \vdash t \preceq \lceil t\rceil$ by inductive hypothesis.
- $\Gamma \vdash x{:}\lceil t_x\rceil \to \lceil t\rceil \preceq x{:}t_x \to t$, since $\Gamma \vdash t_x \preceq \lceil t_x\rceil$ and $\Gamma, x{:}t_x \vdash \lceil t\rceil \preceq t$ by inductive hypothesis.

□

Lemma A.12 (Imprecise Terms). *If all the refinements for constants and user types are local, then if $\lfloor\Gamma\rfloor \vdash \lfloor e\rfloor{:}\lfloor t\rfloor$, then $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil t\rceil$.*

Proof. The proof proceeds by induction on the derivation tree of $\lfloor\Gamma\rfloor \vdash \lfloor e\rfloor{:}\lfloor t\rfloor$.

- $e \equiv x$. By assumption, $x \in \lceil\Gamma\rceil$.
  - If $\lceil\Gamma\rceil(x) = \{v{:}b \mid \_\}$, then $\lceil\Gamma\rceil \vdash x{:}\{v{:}b \mid v = x\}$. By Rule T-Sub and Lemmas A.11 and A.10 $\lceil\Gamma\rceil \vdash x{:}\{v{:}b \mid ?\}$.
  - Otherwise, $\lceil\Gamma\rceil \vdash x{:}\lceil\Gamma(x)\rceil$.
- $e \equiv \lambda x.e'$. By inversion of hypothesis $\lfloor\Gamma, x{:}t_x\rfloor \vdash \lfloor e'\rfloor{:}\lfloor t\rfloor$. By inductive hypothesis $\lceil\Gamma, x{:}t_x\rceil \vdash \lceil e'\rceil{:}\lceil t\rceil$. By Lemma A.10 $\lceil\Gamma\rceil \vdash \lceil x{:}t_x \to t\rceil$. So, by rule T-Fun $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil x{:}t_x \to t\rceil$.
- $e \equiv e'\ x$. By inversion of hypothesis $\lfloor\Gamma\rfloor \vdash \lfloor e'\rfloor{:}\lfloor x{:}t_x \to t\rfloor$ and $\lfloor\Gamma\rfloor \vdash \lfloor x\rfloor{:}\lfloor t_x\rfloor$. By inductive hypothesis $\lceil\Gamma\rceil \vdash \lceil e'\rceil{:}\lceil x{:}t_x \to t\rceil$ and $\lceil\Gamma\rceil \vdash \lceil x\rceil{:}\lceil t_x\rceil$. By rule T-App $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil t\rceil$.
- $e \equiv c$. By definition of $\lceil c\rceil$ this case falls in the previous.
- $e \equiv \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. By inversion of the hypothesis $\lfloor\Gamma\rfloor \vdash x{:}\lfloor\{v{:}\texttt{bool} \mid \_\}\rfloor$, $\lfloor\Gamma\rfloor \vdash \lfloor e_1\rfloor{:}\lfloor t\rfloor$, and $\lfloor\Gamma\rfloor \vdash \lfloor e_2\rfloor{:}\lfloor t\rfloor$. By inductive hypothesis $\lceil\Gamma\rceil \vdash x{:}\lceil\{v{:}\texttt{bool} \mid \_\}\rceil$, $\lceil\Gamma\rceil \vdash \lceil e_1\rceil{:}\lceil t\rceil$, and $\lceil\Gamma\rceil \vdash \lceil e_2\rceil{:}\lceil t\rceil$. By weakening, Lemma A.10 and the rule T-If $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil t\rceil$.
- $e \equiv \texttt{let}\ x = e_x\ \texttt{in}\ e'$ By inversion of the hypothesis $\lfloor\Gamma\rfloor \vdash \lfloor e_x\rfloor{:}\lfloor t_x\rfloor$ and $\lfloor\Gamma, x{:}t_x\rfloor \vdash \lfloor e'\rfloor{:}\lfloor t\rfloor$. By inductive hypothesis, rule T-Let, and Lemma A.10, $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil t\rceil$.
- $e \equiv \texttt{let}\ x{:}t = e_x\ \texttt{in}\ e$ By inversion of the hypothesis $\lfloor\Gamma\rfloor \vdash \lfloor e_x\rfloor{:}\lfloor t'_x\rfloor$ and $\lfloor\Gamma, x{:}t_x\rfloor \vdash \lfloor e'\rfloor{:}\lfloor t\rfloor$. By hypothesis, Lemma A.11 and rule T-Sub $\lfloor\Gamma\rfloor \vdash \lfloor e_x\rfloor{:}\lfloor t_x\rfloor$. By inductive hypothesis, rule T-Spec, and Lemma A.10, $\lceil\Gamma\rceil \vdash \lceil e\rceil{:}\lceil t\rceil$.

□

Theorem A.13 (Embedding of Unrefined Terms). *If all the refinements in constants and user provided specifications are local, then if* $\lfloor\Gamma\rfloor \vdash \lfloor e\rfloor{:}\lfloor t\rfloor$, *then* $\texttt{In}\check{\texttt{f}}\texttt{er}\ \lceil\Gamma\rceil\ \lceil e\rceil\ \mathbb{Q} \neq \emptyset$.

Proof. Since by Lemma A.12 the theorem is proved by completeness of our inference algorithm, *i.e.* Theorem A.5. □

(iii) Static Gradual Guarantee.

Definition A.14 (Precision of Gradual Types). $\check{t}_1 \sqsubseteq \check{t}_2$ *iff* $\gamma_{\mathbb{Q}}(\check{t}_1) \subseteq \gamma_{\mathbb{Q}}(\check{t}_2)$.

Theorem A.15 (Static Gradual Guarantee). *If* $\check{\Gamma}_1 \sqsubseteq \check{\Gamma}_2$ *and* $\check{e}_1 \sqsubseteq \check{e}_2$, *then for every* $\check{t}_{1i} \in \texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}_1\ \check{e}_1\ \mathbb{Q}$ *there exists* $\check{t}_{1i} \sqsubseteq \check{t}_{2i}$ *so that* $\check{t}_{2i} \in \texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}_2\ \check{t}_2\ \mathbb{Q}$.

Proof. Since $\check{t}_{1i} \in \texttt{In}\check{\texttt{f}}\texttt{er}\ \check{\Gamma}_1\ \check{e}_1\ \mathbb{Q}$ then $\exists A$ so that

(1) $\check{t}_{1i} = A \cdot \check{t}_1$
(2) $\texttt{Just}\ A \in \check{\texttt{Solve}}\ (\lambda\kappa.\mathbb{Q})\ \check{C}_1\ \mathbb{Q}$
(3) $(\texttt{Just}\ \check{t}_1, \check{C}_1) = \texttt{Cons}\ \check{\Gamma}_1\ \check{e}_1$

From (2),

(4) $\exists \hat{C}_1 \in \gamma_{\mathbb{Q}}(\check{C}_1).\texttt{Just}\ A \in \texttt{Solve}\ (\lambda\kappa.\mathbb{Q})\ \hat{C}_1$

Since Cons is preserves ?

(5) $(\texttt{Just}\ \check{t}_2, \check{C}_2) = \texttt{Cons}\ \check{\Gamma}_2\ \check{e}_2$
(6) $\check{t}_1 \sqsubseteq \check{t}_2$
(7) $\check{C}_1 \sqsubseteq \check{C}_2$

By (4) and (7) we get

(8) $\hat{C}_1 \in \gamma_{\mathbb{Q}}(\check{C}_2)$

So,

(9) $\texttt{Just}\ A \in \check{\texttt{Solve}}\ (\lambda\kappa.\mathbb{Q})\ \check{C}_2\ \mathbb{Q}$

By (5) and (9) we get

(10) $A \cdot \check{t}_2 \in \check{\texttt{Infer}}\ \check{\Gamma}_2\ \check{e}_2\ \mathbb{Q}$

By (6), (10) and since A preserves ?

$$A \cdot \check{t}_1 \sqsubseteq A \cdot \check{t}_2$$

□