Gradual Liquid Types

Anonymous Author(s)

Abstract

We present gradual liquid types, an extension to refinement types with gradual refinements that range over a finite set of SMT-decidable predicates. This finiteness restriction allows for an algorithmic inference procedure where all possibly valid interpretations of a gradual refinement are exhaustively checked. Thanks to exhaustive searching we can detect the *safe concretizations*, *i.e.* the concrete refinements that justify that a program with gradual refinements is well typed. We make the novel observation that gradual liquid types can be used for static liquid type error explanation, since the safe concretizations exhibit all the potential inconsistencies that lead to type errors. Based on Liquid Haskell, we implement gradual liquid types in `GuiLT`, a tool that interactively presents all the safe concretizations of gradual liquid types, and demonstrate its utility for user-guided migration of three commonly-used Haskell list manipulation libraries.

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 | x /= 0 }` denotes non-zero integer values, and can be used to validate *at compile time* the absence of division-by-zero errors. Liquid types restrict refinements to decidable theories to achieve efficient type inference [Rondon et al. 2008] The type `{ x:Int | k }` now describes integer values refined with some predicate `k`, that will be solved based on unifying the constraints inferred at each use of `x`, resulting in a concrete refinement drawn from a *finite* domain of template refinements.

The attractiveness of liquid typing is usability. Verification only requires specification of top-level functions, while all intermediate types can automatically be inferred and checked. Thus, in theory, users could be ignorant of the internal, liquid typing verification procedure. In practice, ignorance of the verification internals makes it impossible for the user to understand and fix potential type errors. Liquid types, as most type inference engines, suffer from terrible error messages [Zhang et al. 2015]. At an ill-typed function application, the system should (potentially erroneously) decide whether to blame the function definition or the client; error reporting inevitably exposes some of the internals of the sophisticated verification procedure.

On a seemingly unrelated work, Lehmann and Tanter [2017] combined gradual and refinement types. The gradual refinement type `{ x:Int | ? }` describes integer values for each occurrence of which there *exists* a concrete refinement that causes the program to type check. Thus, type checking involves solving existentials over refinements (*i.e.* second order logic), rendering the implementation of a practical gradual refinement type system challenging.

In this paper we present Gradual Liquid Types, a restriction of the gradual refinement types of Lehmann and Tanter [2017], in which the gradual refinement `?` ranges only over the finite domain of liquid refinement templates. This restriction—seemingly strong in theory but realistic in practice—allows for an algorithmic implementation of gradual refinement types: we can exhaustively search for *safe concretizations* (called SCs for short), *i.e.* concrete refinements that can replace each occurrence of a gradual refinement to make the program type check. More importantly, we observe that the SCs can be used to explain liquid type errors. When liquid type checking of a function fails, the user may insert `?` on function preconditions and use the generated SCs to (locally) understand exactly what conditions should be met at each occurrence of each argument refined with `?` (§ 2). The idea of using gradual typing for error explanation is novel and, since it is agnostic of the refinement typing framework, we believe it just generalizes other typing disciplines.

Our first contribution is to formalize the semantics and algorithmic inference of gradual liquid types and prove that it is correct and it satisfies the expected static criteria for gradual languages [Siek et al. 2015] (§ 4).

Our second contribution is to implement gradual liquid types in `GuiLT`, an extension of Liquid Haskell that supports gradual liquid type inference. `GuiLT` takes as input a Haskell program annotated with gradual refinements and generates an `.html` user interface that presents all the SCs for each user-specified `?` in the program. The user can explore all suggested predicates and decide which one to replace a `?` with, so that their program refinement type checks (§ 5).

Error explanation naturally aids migrating programs to adopt liquid types. Our third contribution is to use `GuiLT` for user-guided migration of three existing Haskell libraries (1260 LoC) to Liquid Haskell. ET:♣ rephrase:♣ This experiment supports our claim that gradual liquid types allows for a practical implementation (??s required for 228 functions) interactively used for error explanation and program migration (§ 6).

2 Overview

We start with an overview of gradual liquid types: the intersection of gradual refinement types (§ 2.4) with liquid types (§ 2.2), *i.e.* a decidable fragment of refinement types (§ 2.1).

PL'17, January 01–03, 2017, New York, NY, USA
2017.

2.1 Refinement Types

To prevent runtime division-by-zero, we can refine the type of division to only accept positive integers.

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

Next we define a function `isPos x` that checks positivity of `x` and use it to define the function `divIf` that either divides `1` with its argument `x`, if it is positive or with `1-x`.

```
isPos :: x:Int → {b:Bool | b ⇔ 0 < x}
isPos x = 0 < x
divIf :: Int → Int
divIf x = if isPos x then 1/x else 1/(1-x)
```

`divIf` is well-typed because the precondition of `(/)` is satisfied. Type checking proceeds in three steps: (1) based on the code and the specifications, refinement subtyping constraints are generated; (2) subtyping constraints are reduced to logical implications, *i.e.* verification conditions (VCs); (3) an SMT solver, automatically checks the validity of the VCs.

Step 1: Constraint Generation For our example, two subtyping constraints are generated, one for each call to `(/)`. Both constraints 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`.

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

In both constraints the branching guard `b` is refined with the result refinement of `isPos` (*i.e.* `(b ⇔ 0 < x)`). Also, the guard is strengthened to be true (*i.e.* `b`) or false (*i.e.* `¬ b`) in the constraints generated for the **`then`** and **`else`** branch, *resp.*

Step 2: Verification Conditions Each subtyping constraint is reduced to a logical verification condition (VC), that intuitively checks that assuming all the refinements in the environments, 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.

```
true ∧ b ⇔ 0 < x ∧  b ⇒ v = x   ⇒ 0 < v
true ∧ 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 decides that both VCs are valid, thus `divIf` is well-typed.

ET:♣ can skip:♣ These three verification steps follow the literature [Knowles and Flanagan 2010; Vazou et al. 2014], which, for completeness, is summarized in § 3.1 of this paper.

2.2 Liquid Types

The safety of `divIf` crucially relies on the guard predicate, as propagated by the refinement type of `isPos`. But could `divIf` type check, if the guard function `isPos` was an imported, *unrefined* function?

Let us now use liquid typing [Rondon et al. 2008] to infer a type for `divIf`. Before verification, liquid types are using refinement type variables, here k_x and k_o, for the unspecified refinements of the input and the output types, *resp.*

```
import isPos :: Int → Bool
divIf :: x:{ Int | kx } → {o:Int | ko }
divIf x = if isPos x then 1/x else 1/(1-x)
```

Once the unspecified refinements are named, the inference procedure of liquid typing attempts to find a solution for the liquid variables (here k_x and k_o) so that the program type checks. Inference proceeds in two steps: (1) exactly as in § 2.1, the proper subtyping constraints are generated; (2) the liquid variables ks appearing in the constraints are solved, so 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`.

$$\mathtt{x}{:}\{\mathsf{k}_x\},\ \mathtt{b}{:}\{\mathtt{b}\} \vdash \{\mathtt{v} \mid \mathtt{v}{=}\mathtt{x}\} \leq \{\mathtt{v} \mid \mathtt{0}{<}\mathtt{v}\}$$
$$\mathtt{x}{:}\{\mathsf{k}_x\},\ \mathtt{b}{:}\{\neg\mathtt{b}\} \vdash \{\mathtt{v} \mid \mathtt{v}{=}\mathtt{1}{-}\mathtt{x}\} \leq \{\mathtt{v} \mid \mathtt{0}{<}\mathtt{v}\}$$

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

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.

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

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

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

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

$$A = \{\mathsf{k}_x \mapsto \bigwedge \mathbb{Q}^x,\ \mathsf{k}_o \mapsto \bigwedge \mathbb{Q}^o\}$$

Repeatedly it filters out predicates of the solution until all constraints are satisfied. In our example, the initial solution solves both liquid variables to false (since the contradictory predicates $0 < x$ and $x < 0$ are part of the initial solution). Thus, both constraints are valid, and inference returns with the valid solution $\mathsf{k}_x \mapsto$ `false`, $\mathsf{k}_o \mapsto$ `false`, or

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

ET:♣ skip? footnote?:♣ In § 3.2 we summarize the formal background of liquid type inference required for this paper, where ET:♣ I don't get that:♣ we simplify the procedure to reason about instantiated set of templates (*i.e.* $\mathbb{Q} = \bigcup_{k^x \text{ liquid variable}} \mathbb{Q}^x$).

The inferred type for `divIf`, though correct, is useless in practice, because a `false` precondition means that the function cannot be applied. For a user to understand this result, she needs to be aware of the internals of the inference procedure. In fact, the inferred type of `divIf` depends on the presence of clients in the considered code base.

For instance, in the presence of a "positive" client call `divIf 1`, the inferred precondition would be `0 < x`, thereby producing a type error on the *definition side* of `divIf`ET:♣ where exactly♣. This arbitrary blaming of client/definition is not unique to liquid typing; it frequently appears in type inference engines [Wand 1986], yielding hard-to-debug error messages.

2.3 Gradual Refinement Types

A key novel observation of this paper is that we can exploit gradual typing in order to assist inference and provide better support for error explanation and program migration. Let us use a gradual refinement Lehmann and Tanter [2017] for the precondition of the type of `divIf`, `x:{Int | ? }`. This 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. ET:♣ I'm HERE♣

```
import isPos :: x:Int → Bool
divIf :: x:{ Int | ? } → Int
divIf x = if isPos x then 1/x else 1/(1-x)
```

Type checking of `divIf` follows the three steps of standard refinement type checking, adjusted to accommodate the semantics of gradual refinements.

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 type `x:{?}` translates intuitively to an existential refinement (∃ `p. p x`). The solution of these existentials produces the SCs of the program. For soundness and satisfaction of the gradual criteria of Siek et al. [2015], as formalized in § 3.3, the existentials cannot range over arbitrary refinements but need to satisfy certain conditions (namely locality and precision). Here we informally use $\exists^?$ `p` to denote existential over predicates that satisfy the required conditions and call VCs that contain such existentials Gradual Verification Conditions (GVC). For example, the GVCs for `divIf` are the following.

```
(∃? pthen. pthen x) ∧ b      ⇒ v=x    ⇒ 0 < v
(∃? pelse. pelse x) ∧ ¬b ⇒ v=1-x ⇒ 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` p_{else} `x` $\mapsto$ `x ≤ 0`

More importantly, since these concretizations are logical predicates, we can present them to the user as the conditions under which `divIf` type checks.

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 loose the certificate generation, and thus the opportunity to use such certificates for error explanation.

2.4 Gradual Liquid Types

To algorithmically solve GVCs exhaustively search for SCs in the finite predicate domain of liquid types. The gradual liquid type system performs the two inference steps of §2.2 but in between concretizes the constraints, by instantiating the 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 Next, we exhaustively generate all the possible concretizations of the constraints. For example, the `x:{?}` in the constraints can be concretized with each predicate from the $\mathbb{Q}^x$ set of § 2.2, giving $|\mathbb{Q}^x|^2 =$ 16 concrete constrain 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, we merely invoke the Constraint Solving Step (*i.e.* Step 2) of liquid typing to find 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 p_{then} `x` $\mapsto$ `0<x` and p_{else} `x` $\mapsto x \leq 0$ or p_{else} `x` $\mapsto x < 0$. Thus, `divIf` type checks and also the inference provides to the user the SCs (concretizations of p_{then} and p_{else}) as an explanation of type checking.

In § 4 we formalize these three inference steps and prove the correctness and the gradual criteria of our algorithm.

Number of Generated Constraints For complete inference (*i.e.* if there exist a valid liquid concretization, then type inference finds it) it does not suffice to concretize each ? with the elements of $\mathbb{Q}$. Instead we need to try all possible conjunctions of these elements, $2^{|\mathbb{Q}|}$ in number, leading to $(2^{|\mathbb{Q}|})^2 = 256$ concretizations in our example. In our implementation (§ 5), we sacrifice completeness and, by default, we merely concretize with singletons from $\mathbb{Q}$, though the user could adjust the size of conjunctions that are considered.

As a crucial optimization, before concretization, we partition the constraint set based on its dependencies. In our example, the two constraints are independent (*i.e.* validity of each does not depend on the other), thus we independently concretize and solve them, leading to only $2\text{x}(2^{|\mathbb{Q}|}) = 32$ constraints for complete inference ($2\text{x}|\mathbb{Q}| = 8$ checked by default in the implementation).

Summary To sum up, gradual liquid types address two main problems of gradual refinement types. They allow for algorithmic solving of GVCs over arbitrary domains and also they co-exist with liquid type inference, thus the user does not have to specify the types for each intermediate expression. As an important side effect, type checking generates existential certificates (or SCs) for the valid concretizations which, as later discussed, can be used for error reporting (§ 5) and user-aided program migration (§ 6).

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^e_R. 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 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^e_R 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 $x{:}t_x \to t$, 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 $t_x \to t$ as a shortcut for $x{:}t_x \to t$ when x does not appear in t.

Constants	c	::=	$\wedge \mid \neg \mid = \mid \ldots$
		$\mid$	$\texttt{true} \mid \texttt{false}$
		$\mid$	$0, 1, -1, \ldots$
Values	v	::=	$c \mid \lambda x.e$
Expressions	e	::=	$v \mid x \mid e\ x$
		$\mid$	$\texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$
		$\mid$	$\texttt{let}\ x{:}t = e\ \texttt{in}\ e$
Predicates	p	::=	e
Basic Types	b	::=	$\texttt{int} \mid \texttt{bool}$
Types	t	::=	$\{x{:}b \mid p\} \mid x{:}t \to t$
Environment	Γ	::=	$\cdot \mid \Gamma, x{:}t$
Substitution	σ	::=	$\cdot \mid \sigma, (x, e)$

Figure 1. Syntax of λ^e_R.

Denotations Following Knowles and Flanagan [2010], each type of λ^e_R 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).

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

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.

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

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

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

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

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

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

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

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

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

Figure 2. Static Semantics of λ^e_R. (Types colored in blue need to be inferred.)

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 follows:

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

Of course, this semantic definition is undecidable, as it quantifies over all possible 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:

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

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.

$\hat{p}$	::=	$\texttt{true}$	True
	|	q	Predicate, with $q \in \mathbb{Q}$
	|	$\hat{p} \wedge \hat{p}$	Conjunction
	|	κ	Liquid Variable
A	::=	$\cdot \mid A, \kappa \mapsto \overline{q}$	*Solution*

Figure 3. Syntax of $\lambda^{\hat{p}}_L$. Completes Figure 1.

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 $q \in \mathbb{Q}$.

Syntax Figure 3 summarizes the syntax of *liquid predicates*, written $\hat{p}$. 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 embedding 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

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

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

Figure 4. Liquid Inference Algorithm ET:♣ cons and solve in appendix♣

assumptions of Γ, the predicate $\hat{p}_1$ implies the predicate $\hat{p}_2$.

$$\text{isValid}(\Gamma \vdash \{x{:}b \mid \hat{p}_1\} \preceq \{x{:}b \mid \hat{p}_2\})$$

$$\textit{iff}$$

$$\text{isSMTValid}(\bigwedge\{\hat{p} \mid x{:}\{x{:}b \mid \hat{p}\} \in \Gamma\} \Rightarrow \hat{p}_1 \Rightarrow \hat{p}_2)$$

Inference The liquid inference algorithm, defined in Figure 4, 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].

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 by 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: $A_0 = \lambda\kappa.\mathbb{Q}$.

The algorithm `Infer` $\hat{\Gamma}$ $\hat{e}$ $\mathbb{Q}$ is proved to be 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 the set $\mathbb{Q}$.

3.3 Gradual Refinement Types

Gradual refinement types [Lehmann and Tanter 2017] extend the refinements of λ^e_R 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 briefly summarize their basics.

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

Figure 5. Syntax of $\lambda^{\tilde{p}}_G$. Completes Figure 1.

Syntax Figure 5 describes the syntax of predicates in $\lambda^{\tilde{p}}_G$. 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 stronger (or *more specific*)ET:♣ any reason not to stick to specificity?♣ than p_2, written $p_1 \preceq p_2$ when p_1 is true when p_2 is true:

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

Locality Additionally, in order to avoid imprecise formulas to introduce 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 $p(x)$ ET:♣ maybe use 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. Formally:

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

Concretization Armed with specificity and locality, Lehmann and Tanter [2017] define the concretization function $\gamma(\cdot)$, which 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.

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

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

Figure 6. Syntax of $\lambda^{\check{p}}_{GL}$. Completes Figure 1.

Denotations While Lehmann and Tanter [2017] do not provide denotations for gradual refinement types, we can extend our denotations from Section 3.1. The denotation of a base *imprecise* gradual refinement $\{x{:}b \mid p \wedge ?\}$ includes all (gradually-typed) expressions that satisfy some local predicate stronger than p. ET:♣ this is somehow mixing concretization and type denotations... plus it's not needed, we can always pick $p' = p$...?♣ The denotations of the other types are unchanged.

$$\begin{aligned}
[\![\{x{:}b \mid p \wedge ?\}]\!] \doteq \{\tilde{e} \mid \vdash \tilde{e} : b,\ & \text{if } \tilde{e} \Downarrow \text{ and } \exists p' \preceq p.\mathrm{isLocal}(p'(x)),\\
& \text{then } \models p'[\tilde{e}/x] \text{ET:♣ check exists♣}\}\\
[\![\{x{:}b \mid p\}]\!] \doteq \{\tilde{e} \mid \vdash \tilde{e} : b, & \text{if } \tilde{e} \Downarrow \text{ then } \models p[\tilde{e}/x]\}\\
[\![x{:}\tilde{t_x} \to \tilde{t}]\!] \doteq \{\tilde{e} \mid \mid \forall \tilde{e_x} \in [\![\tilde{t_x}]\!].\tilde{e}\ \tilde{e_x} \in [\![\tilde{t}[\tilde{e_x}/x]]\!]\}&
\end{aligned}$$

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. $\tilde{\mathrm{isValid}}(\cdot)$ holds if there exists a justification, by concretization, that the static judgment holds. Precisely:

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

4 Gradual Liquid Types

We now address the combination of liquid type inference and gradual refinements. 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.

Figure 6 defines the syntax of predicates in $\lambda^{\check{p}}_{GL}$, a gradual liquid core language whose predicates are gradual predicates (as in Figure 5) where the static part of an imprecise predicate is a liquid predicate (from Figure 3), with the additional requirement that it be *local* (as defined in § 3.3). The elements of $\lambda^{\check{p}}_{GL}$ are both gradual and liquid; *e.g.* expressions $\check{e}$ could also be written as $\tilde{\hat{e}}$. Also, we write ? as a shortcut for the imprecise predicate $\mathit{true} \wedge ?$.

Our goal is to define $\check{\mathtt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ so that it returns a type $\check{t}$ such that $\check{\Gamma} \vdash \check{e}{:}\check{t}$. After deriving $\check{\mathtt{Infer}}$ using AGT, we provide an algorithmic characterization of $\check{\mathtt{Infer}}$ (§ 4.2), which serves as the basis for our implementation. We present the properties that $\check{\mathtt{Infer}}$ satisfies in § 4.3.

4.1 Lifting Liquid Inference

We define the function $\check{\mathtt{Infer}}$ using the abstracting gradual typing methodology [Garcia et al. 2016]. In general, AGT defines the consistent lifting of a type function `f` as:

$$\tilde{\mathtt{f}}\ \tilde{t} = \alpha(\{\mathtt{f}\ t \mid t \in \gamma(\tilde{t})\})$$

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

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

This definition of $\check{\mathtt{Infer}}$ 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 ℚ = Nothing`, and abstracting the empty set denotes a type error. This behavior breaks the flexibility programmers expect from gradual refinements (this example is based on the motivation example of Lehmann and Tanter [2017]). One expects that $\check{\mathtt{Infer}}\ \{\}\ \check{e}\ \mathbb{Q}$ should simply return `Int`. Note that this is the same reason that Garcia et al. [2016] do not lift the typing relation as a whole, but instead lift type functions and predicates used in the definition of the typing relation. 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?

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.*:

$$\begin{gathered}
\mathrm{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid p_1\}),\\
\mathrm{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq \{v{:}b \mid p_2\})\\
\Rightarrow\\
\mathrm{isValid}(\Gamma \vdash \{v{:}b \mid p\} \preceq p_1 \wedge p_2)
\end{gathered}$$

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

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 constraintET:♣ where is the syntax of (static) constraints?♣. This approach is successful, and leads to a provably sound and complete inference algorithm (§ 4.3).

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.

```
Infer :: Env → Expr → Quals → Set Type
Infer Γ̌ ě ℚ = { ť′ | Just ť′ <- A <*> ť,
                    A ∈ Solve A₀ Č }
  where (ť, Č) = Cons Γ̌ ě
```

First, function `Cons` derives the typing constraints $\check{C}$, and if successful, the template type $\check{t}$. Then, we use the lifted $\check{\texttt{Solve}}$ to derive solutions for each concretization of the derived constraints. 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 applications discussed in § 5 and § 6 make explicit use of the inferred set in order to assist users in understanding errors and migrating programs.[1]

4.2 Algorithmic Concretization

To turn the definition of $\check{\texttt{Infer}}$ into an inference algorithm, we need an algorithmic concretization function of a set of constraints, $\gamma(\check{C})$.

Recall that the concretization of gradual predicates is defined as

$$\gamma((p \wedge ?)(x)) \doteq \{p' \mid p' \preceq p, \text{isLocal}(p'(x))\}$$

In general, this function cannot be algorithmically computed, since it ranges over the infinite domain of predicates. In gradual liquid refinements, the domain of refinement 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.

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

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

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

[1] 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 § 5.

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

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

Finally, 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 $\check{\texttt{Infer}}$.

4.3 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].[2] The corresponding proofs can be found in supplementary material [Material 2017].

4.3.1 Correctness of Inference

The inference algorithm $\check{\texttt{Infer}}$ is sound and 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*

- ***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 $\nexists \check{t}.\ \check{\Gamma} \vdash \check{e}{:}\check{t}$.*
- ***Termination*** $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ *terminates.*

We note that, unlike the `Infer` algorithm that provably returns the strongest possible solution, we have not formalized whether or not the set of solutions returned by $\check{\texttt{Infer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ are stronger that the rest of the types that satisfy $\check{\Gamma} \vdash \check{e}{:}\check{t}$.

4.3.2 Gradual Typing Criteria

Siek et al. [2015] list three properties related to the static semantics of a gradually-typed language, which the gradual algorithm $\check{\texttt{Infer}}$ satisfies. These guarantees capture the fact that a gradual type system is *(i)* a conservative extension of a static type system, which is *(ii)* 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 `Infer` on terms that only rely on precise predicates. More specifically, if a term is inferred a static type with `Infer`, then $\check{\texttt{Infer}}$ returns only that type. Conversely, if a term is not typeable with `Infer`, it is also not typeable with $\check{\texttt{Infer}}$.

[2] 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].

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

(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 programs.)

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

The following embedding function turns simple types into gradual liquid types by recursively introducing the unknown predicate on every base type:

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

We appeal to its natural extension to type environments and terms.

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

(iii) Static Gradual Guarantee Finally, we prove the static gradual guarantee, which states that typeability is monotonous with respect to the precision of type information. In other words, making type annotations less precise cannot introduce new type errors.

We first need to define the notion of precision of gradual types, defined using 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* $\gamma_{\mathbb{Q}}(\check{t}_1) \subseteq \gamma_{\mathbb{Q}}(\check{t}_2)$.

The notion of type 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* $\check{t}_{1i} \in \check{\texttt{Infer}}\ \check{\Gamma}_1\ \check{e}_1\ \mathbb{Q}$ *there exists* $\check{t}_{1i} \sqsubseteq \check{t}_{2i}$ *so that* $\check{t}_{2i} \in \check{\texttt{Infer}}\ \check{\Gamma}_2\ \check{t}_2\ \mathbb{Q}$.

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 in a less precise environment.

5 Application I: Error Explanation

ET:♣ syntax: use of ?? vs. ?♣

We implemented $\check{\texttt{Infer}}$ as `GuiLT`, an extension to Liquid Haskell that takes as input a Haskell program with gradual refinement types specifications and returns an HTML interactive file that lets the user explore all gradually safe solutions.

ET:♣ general comment: clarify the use of the words "safe" and "unsafe". For me, and I suspect for many, safe means "no crash" – hence the name "unsafeCoerce" in Haskell.♣

Concretely, `GuiLT` uses the Liquid Haskell API to generate the constraints and the refinement templates. Then, it uses the templates to generate all the concretizations of the constraints. Finally, it uses the Liquid Haskell API to select the valid concretizations.ET:♣ here you used "valid" instead of "safe" – I prefer valid as well, but the main point is that we should be consistent in which words we use.♣

Next, we illustrate `GuiLT`'s interface ET:♣ more than the interface, illustrate the use of the tool♣ via the list indexing example. Consider the standard list indexing function `xs!!i` that returns the `i`th element of the list `xs`.

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

ET:♣ start by introducing the two clients (called app1 and app2 on the figures btw), before talking about the fact there are two possible interpretations for indices here♣ Solving `??` to `0 <= i < len xs` ensures that the `error` case is never called but renders the following client as unsafe ET:♣ here the meaning is "throws a runtime error" (which is not the same as unsafe by the definition above! unsafe = crash/seg fault) – or "does not type check"?♣

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

As an alternative, solving `??` to `0 < i <= len xs` renders the client function as safe, but the definition of `(!!)` is unsafe, since the `error` case is reachable. ET:♣ the error case was reachable in the original definition♣

ET:♣ suggestion: the Haskell definition of (!!) is essentially a *partial* function, where partiality manifests by the fact that the function can halt execution with a (uncatchable?) runtime error. Using refinements, we can make the function *total*, by making sure the error case is unreachable.♣

The above situation describes a characteristic challenge on error reporting, where the type system needs to decide whether to blame ET:♣ now safety is related to blame - again something with a technically overloaded meaning♣ the function definition or the client function for incompatible types.ET:♣ I'm not quite sure I buy that this example illustrates that. Clearly (!!) intends the index to be from to 0 to length-1. What's the meaning of the other interpretation?♣

ET:♣ explain why you only discuss these two possible instantiations♣

To address this challenge instead of blaming any of the two parties, we use the valid gradual solution concretizations to present all the valid refinement instantiations to the programmer.

Figure 7 illustrates the generated HTML interactive file for the indexing example above. The ?ET:♣ one ? or two ??♣ refinement is connected to all its usages by highlighting them using the same color. Once the user clicks on the ? button a (gradually valid) solutionET:♣ solution and instantiation are

used interchangeably♣ appears for each occurrence, combined with left/right navigations buttons that the user can use to explore potential solutions. This way, the error ET:♣ which error? there is no error unless I decide to view it as an error♣ is explained without having to blame a specific party.

As an alternative illustration, GuiLT allows the user to explore all the gradual solutions and accordingly adjusts the blame. ET:♣ prior illustration was: stick to unknown refinement, view the constraints generated at each occurrence; this one is: (based on the observed possible constraints) pick a (more?) precise refinement and visualize which occurrences would break♣ For example, in the GUI of Figure 8 the user navigates to the left or right to explore the previous or next solutions of the ? they placed. ET:♣ terminology: differentiate the "binding occurrence" and "usage occurrences" of a question mark♣ As the solutions changeET:♣ it's not really the solutions that change, it's the precise refinement that one picks to replace the unknown refinement♣, so does the blame (red in the figure) that is placed either on the client (left figure) or at the function definition (right figure). ET:♣ -1 is in red because (i-1) potentially violates the $i > 0$ condition, right? if so, what about in the other case? Since we're talking Int, and not Nat, $i - 1$ is not guaranteed to be ≥ 0 either – I'm confused♣

5.1 Practical Implementation

The implementation of GuiLT closely follows the theory of § 4 with some adjustments to make it practical.

Measures Following [Lehmann and Tanter 2017]ET:♣ use citet, not citep, when refering to the article explicitly♣, we adjust locality checking to accommodate uninterpreted functions, ET:♣ introduce+cite measures♣ like the `len` of a list. For instance, the predicate `0 < len i` is not local on `i`. This is because `exists i. 0 < len i` is not SMT valid, as there exists a model in which `len` is always negative. Thus to check locality of predicates that use uninterpreted functions we define a fresh variable ET:♣ eg. leni♣ for each function application ET:♣ eg. len i♣, and then check locality. For instance `0 < len i` is local on `i` because `exists leni. 0 < leni` is SMT valid.

Sensible Before checking for locality, the implementation is performing a heuristic "sensibility" testing to filter out automatically generated templates ET:♣ you explain generated templates below, so this paragraph should come after♣ that the user would not write. For instance we consider arithmetic operations on lists and booleans to be non-sensible and filter them out. ET:♣ be more specific: filter ill-typed templates, remove False and other contradictions – anything else?♣

Templates To generate the templates we use Liquid Haskell's API for template generation. The generated templates consist of a predefined set of templates for linear arithmetic (v [$< \mid \leq \mid > \mid \geq \mid = \mid \neq$] x), comparison with zero (v [$< \mid \leq \mid > \mid \geq \mid = \mid \neq$] 0) and `len` operations (len v[$\geq \mid >$]0, len v = len x, v = len x, v = len $x + 1$). Other than these default templates, templates are abstracted from each user-provided refinement type and the user can define their own templates.

Depth In the theoretical development, for completeness, we check validity of any solution—which is clearly not tractable in practice. In the implementation, we introduce an instantiation depth parameter. By default, depth is 1, meaning that we instantiate each ? refinement with single templates, and not their conjunctions. To get the conjunction of two templates, such as the conjunction of `0 <= i` and `i < len xs` in our `(!!)` example, the instantiation depth is set to 2.

Partitions An implementation detail that is critical for efficiency is that before solving, we partition the set of constraints based on their dependencies and solve each partition independently. ET:♣ explain what "dependencies" means here♣ That way, each partition has a limited number of gradual occurrences ET:♣ what is a "gradual occurrence" – need to fix/streamline the terminology♣, thus findings the combination of all instantiations is pragmatic.

ET:♣ I would tend to present all the techniques above in a different order. First, expressiveness (ie. measures). Second, efficiency, in an order that follows the general algorithm: partition, templates, depth, sensibility (is that the right/chronological order?)♣

ET:♣ I'm HERE♣

Table 1 provides a quantitative illustration of the indexing example. GuiLT used a template depth (Dep) of 2 to solve for the one (# ?) gradual refinement that occurred (Occs) four times in the generated constraints. Each occurence had 68 candidate solutions (Cands) out of which 38 were sensible (Sens), 34 were local (Local) and 34 were precise (Prec). Note that most non-local solutions were filtered out by the sensibility check. The constraints were split in 14 partitions (Parts), out of which 4 contained gradual refinements and had to be concretized. Each partition had 34 conretizations (# γ) out of which 22, 13, 10, and 13 were valid (Sols) for the four partitions. None of these solutions was common for all the occurences of the ?, thus the program has 0 static solutions (SSols). Finally, the running time of the whole process was 6.88 sec.

6 Application II: Migration Assistance

As a second application here we explain how the interface from § 5 was used to migrate commonly used libraries from Haskell to Liquid Haskell safe code.

Migration of a library from (refinement) untyped to typed code is an iterative procedure since errors propagate from the imported to the client libraries and back.

Figure 7. GuiLT for error explanation from here.

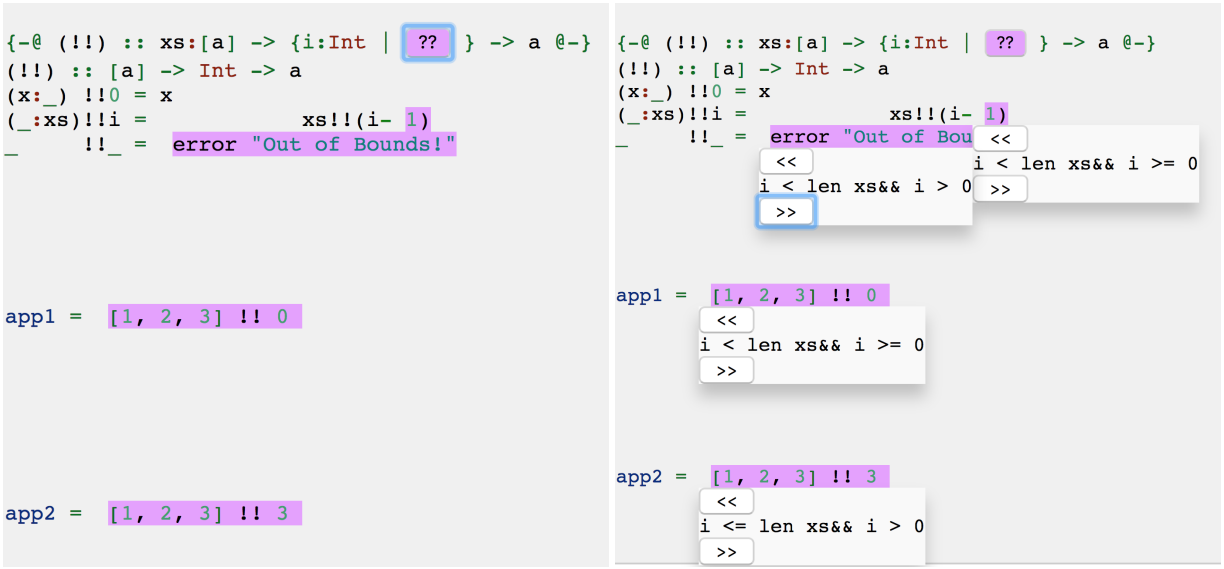
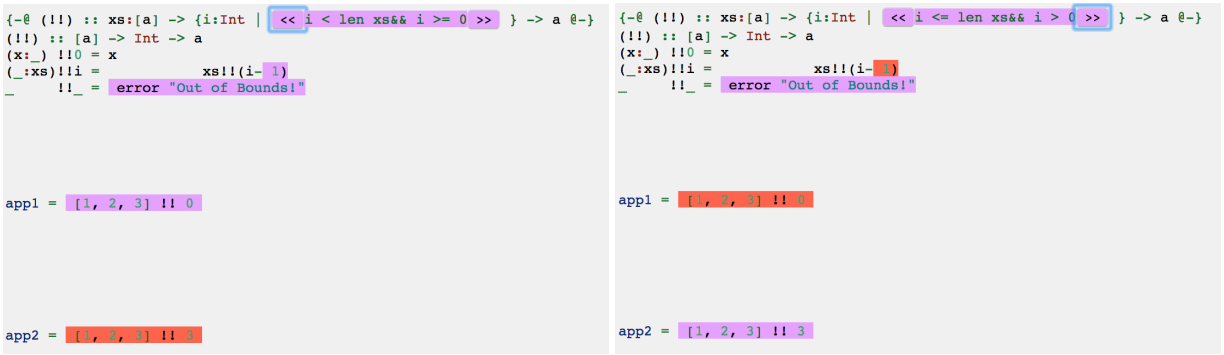


Figure 8. GuiLT for static typing from here.



Dep	# ?	Occs	Cands	Sens	Local	Prec	Parts	# γ	Sols	SSols	Time(s)
1	1	[4]	[12*]	[11*]	[11*]	[11*]	4/14	11*	[8,6,0,6]	1	3.39
2	1	[4]	[68*]	[38*]	[34*]	[34*]	4/14	34*	[22,13,10,13]	0	6.88

Table 1. Indexing example in Numbers

Benchmarks As a case study we used the interface from § 5 to aid the migration of three dependent, commonly used (thus safe) Haskell list manipulation libraries

- `GHC.List` (56 exported functions, 501 LOC) that provides the commonly used list functions available at `Prelude`,
- `Data.List` (115 exported functions, 490 LOC) that defines more sohpisticated list functions, like list transposing, and
- `Data.List.NonEmpty` (57 exported functions, 269 LOC) that lifts list functions to provably non empty lists.

Migration Process At the beginning of the migration process we run Liquid Haskell in the library to be migrated to get the initial type errors. There are two sources for these initial errors

1. failure to satisfy imported functions preconditions, *e.g.* the `error` function assumes by default the `false` precondition, and
2. incomplete patterns, that is a pattern-match that might fail at runtime, *e.g.* `scanr` matched the recursive call with a non empty list.

We note that by default, Liquid Haskell will also generate termination errors when the default termination check fails,

but to simplify our case study, we deactivated termination checking.

To fix an error in a function there are three potential approaches (other than fixing the buggy code that never happened at our commonly used libraries of our case study). We can

- infer function's precondition (*e.g.* `head` is only called on non empty lists), at which case we need to revisit the clients of the functions (inside and outside the library),
- strengthen imported function's post-conditions, at which case we can either enforce the imported type (thus gradual verification) or we could update and re-verify the library.
- strengthen function's post condition (*e.g.* `scanr` always returns non-empty lists) at which case no external conditions are violated.

Liquid Haskell is able to infer the strongest possible post conditions (case 3), which interestingly our technique is very bad at (`scanr1` time-outed because post-conditions means many different dependencies). Yet, the liquid type inference of Liquid Haskell is unable to infer pre-conditions (it only infers the strongest preconditions, *i.e.* false, for library functions that are never called) and further is it unable to infer types for imported functions. Thus the two techniques smoothly complement each other.

The i^{th} iteration of our interactive migration process is the following. We use Liquid Haskell on the to-be-migrated library to trace potential type errors, *i.e.* unsafe functions. We fix the errors by adding ? in either precondition of the erroring function or post-condition of the potentially imported, thus assumed, types the function is using. We use the framework of § 5 to interactively solve the ? to static refinements. We repeat until Liquid Haskell reports no errors.

Table 2 summarizes our migration case study in numbers. Rndis the number of times we needed to iterate inside until each library got verified. At each round the reported functions (Function) we reported unsafe, thus we annotated their specification with gradual refinements and run `GuiLT` to interactively inspect the proposed solutions.

`GuiLT` by default proposes only solutions as instantiations of the provided templates. Few functions (in our case study only `(!!)`) can be solved only using conjunctions of the templates. Depis the depth of the conjuctions used.

? is the number of ? we manually inserted. For each ? we report the number of constraints in which it appeared. For each occurrence we report the number of the generated template candidates (Cands) and how many of them are sensible (Sens), local (Local), and precise (Prec), where sensible is a heuristic we used to simplify away non-intuitive instantiations, *e.g.* comparison on booleans, tautologies and contradictions.

Partsis the number of partitions verification generated. For each partition we report the number of concretizations (# γ) generated, that is the combination of all instantiations. Further, for each partition we report the number of solutions (Sols). Finally, we combine together the solutions of different partitions to generate all possible static solutions. Timeis the whole time in seconds required for all the above process.

`GHC.List` Liquid Haskell reported three errors on `GHC.List`. The function `errorEmp` errored as is it merely a wrapper around the `error` function and `scanr` and `scanr1` had incomplete patterns assuming a non empty list post condition. `GuiLT` performed bad at all these three cases. It was unable to infer any solution for `errorEmp` , since the required false precondition is non local. It required more than one hour to infer the post condition of `scanr` and we timed-out it in the case of `scanr1`. The reason for this is that ? in post-conditions do not allow partition spitting leading to one partition with a huge number of potential combinations. NV:♣ I do not understand why this happens: is it expected or an implementation bug?♣ `GuiLT` was more efficient in the next two rounds. Specification of `errorEmp` introduced errors in eight functions that were trivially specified using `GuiLT`. One of this functions, `fold1` was further used by two more functions that were interactively migrated at the third and final specification round.

`Data.List` Migration of `Data.List` was simpler, only four functions errored due to incomplete patterns (`transpose` should return a list of non empty lists), while three other functions used functions with preconditions from `GHC.List`. Unexpectedly, `GuiLT` was unable to find any solution for `genericIndex`, a generalized version of `(!!)` that indexes a list using any integral (instead of an integer index) because it lacks linear arithmetic templates for integrals.

`Data.List.NonEmpty` Migration was more interesting for the `Data.List.NonEmpty` library that manipulates a data type that represents 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 can only be specified once `toList` returns non empty lists.

```
cycle = fromList . List.cycle . toList
```

Thus, to migrate `cycle`, we added ? to the result of `toList` and `GuiLT` quickly solved the specification.

Indexing non empty lists, invokes list indexing. Thus migration of non empty list indexing was more complex. since we needed to relate the length of empty and non lists (different data types) and further add templates that talk about non empty list length arithmetic operations.

Adding a non-empty list precondition to `fromList` fired errors to eight function that invoke them. All of them call list operations that return (unspecified) non empty lists. For example, `inits` on non empty lists is defined as follows

Rnd	Function	Dep	# ?	Occs	Cands	Sens	Local	Prec	Parts	# γ	Sols	SSols	Time(s)
`GHC.List as L` (56 functions defined and verified)													
1st	`errorEmp`	1	1	[1]	[[5]]	[[4]]	[[4]]	[[4]]	1/4	[4]	[0]	0	1.00
	`scanr`	1	1	[4]	[6*]	[5*]	[5*]	[5*]	1/5	[625]	[125]	1	4640.20
	`scanr1`	1	2	[4,6]	[12*,5*]	[5*,4*]	[5*,4*]	[5*,4*]	1/5	[2560000]	??	??	??
2nd	`head`	1	1	[1]	[[5]]	[[4]]	[[4]]	[[4]]	1/3	[4]	[1]	1	0.70
	`tail`	1	1	[1]	[[5]]	[[4]]	[[4]]	[[4]]	1/5	[4]	[1]	1	0.77
	`last`	1	1	[2]	[5*]	[4*]	[4*]	[4*]	2/4	4*	[4,1]	1	1.04
	`init`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/8	[16,4]	[16,1]	1	3.12
	`fold1`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/5	[4,16]	[1,16]	1	2.41
	`foldr1`	1	1	[1]	[[5]]	[[4]]	[[4]]	[[4]]	1/2	[4]	[1]	1	1.08
	`(!!)`	1	2	[4,4]	[5*,10*]	[4*,9*]	[4*,9*]	[4*,9*]	4/9	36*	[12,24,36,36]	6	7.81
	`(!!)`	2	2	[4,4]	[12*,47*]	[10*,17*]	[6*,16*]	[6*,16*]	4/9	96*	[52,64,96,96]	26	19,72
	`cycle`	1	1	[2]	[5*]	[4*]	[4*]	[4*]	2/6	4*	[4,1]	1	1.37
3rd	`maximum`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/4	[4,16]	[1,16]	1	3.38
	`minimum`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/4	[4,16]	[1,16]	1	2.80
`Data.List as DL` 115 functions defined and verified)													
1st	`maximumBy`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/5	[16,4]	[16,1]	1	2.24
	`minimumBy`	1	1	[3]	[5*]	[4*]	[4*]	[4*]	2/5	[16,4]	[16,1]	1	2.40
	`transpose`	1	1	[3]	[12*]	[5*]	[5*]	[5*]	2/11	[25,5]	[0,4]	1	51.02
	`genericIndex`	1	2	[6,6]	[2*,5*]	[1*,4*]	[1*,4*]	[1*,4*]	6/12	4*	[3,4,4,1,1,4]	0 ??	
`Data.List.NonEmpty` (57 functions defined and verified)													
1st	`fromList`	1	1	[2]	[5*]	[4*]	[4*]	[4*]	2/11	4*	[4,1]	1	1.41
	`cycle`	1	1	[1]	[[2]]	[[1]]	[[1]]	[[1]]	1/3	[1]	[0]	0	1.27
	`-toList`	1	2	[1,2]	[[2],5*]	[[1],4*]	[[1],4*]	[[1],4*]	2/3	4*	[1,3]	1	3.11
	`(!!)`	1	1	[4]	[10*]	[9*]	[9*]	[9*]	4/22	9*	[9,4,4,9]	1	5.96
	`(!!)`	2	1	[4]	[47*]	[17*]	[16*]	[16*]	4/12	16*	[16,16,11,8]	5	7.73
2nd	`cycle`	1	2	[1,1]	[[12],[5]]	[[5],[4]]	[[5],[4]]	[[5],[1]]	1/2	[5]	[2]	2	3.13
		1	2	[1,1]	[[5],[12]]	[[4],[5]]	[[4],[5]]	[[4],[5]]	1/2	[20]	[6]	6	10.54
	`lift`	1	1	[1]	[[6]]	[[5]]	[[5]]	[[5]]	1/2	[5]	[2]	2	2.90
	`inits`	1	1	[1]	[[6]]	[[5]]	[[5]]	[[5]]	1/5	[5]	[2]	2	2.95
	`tails`	1	2	[1,1]	[[5],[6]]	[[4],[5]]	[[4],[5]]	[[4],[5]]	1/5	[20]	[6]	6	10.22
	`scanl`	1	1	[1]	[[6]]	[[5]]	[[5]]	[[5]]	1/5	[5]	[2]	2	2.23
	`scanl1`	1	1	[1]	[[6]]	[[5]]	[[5]]	[[5]]	1/2	[5]	[1]	1	1.13
	`insert`	1	1	[1]	[[12]]	[[5]]	[[5]]	[[5]]	1/5	[5]	[2]	2	2.25
	`transpose`	1	2	[1,1]	[[7],[12]]	[[6],[5]]	[[6],[5]]	[[6],[5]]	1/7	[30]	[6]	6	37.48
3rd	`reverse`	1	2	[1,1]	[[5],[12]]	[[4],[5]]	[[4],[5]]	[[4],[5]]	2/3	[5,4]	[2,3]	5	0.96
	`sort`	1	2	[1,1]	[[12],[5]]	[[5],[4]]	[[5],[4]]	[[5],[4]]	2/3	[5,4]	[2,3]	5	1.03
	`sortBy`	1	2	[1,1]	[[12],[5]]	[[5],[4]]	[[5],[4]]	[[5],[4]]	2/3	[5,4]	[2,3]	5	0.97

Table 2. Evaluation of Migrations Assistance. Rndis the number of iteration inside the verified library. Functionis the name of the gradualized function. Depis the number of template combinations we used. # ? is the number of ? we manually inserted. Occsis the number each ? was used inside the function. For each occurrence we report the number of template candidates (Cands), and how many of them are sensible (Sens), local (Local), and precise (Prec). Partsis the number of partitions spitted. For each partition we report the number of concretizations (# γ) and the number of solutions (Sols). SSolsis the number of static solutions found and Timeis the whole time taken to compute the above.

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

To migrate such functions, we add a gradual assumed specification for the `List` library function. For example

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

and use GuiLT to solve the ?. Once the ? is solved (here to `0 < len o`) we could leave the gradual specification or revisit the library definition to strengthen the post-condition. This is not always trivial. For instance `transpose` only returns non empty lists when called with a non empty list of non empty lists, which was the case in the non empty list library.

As another interesting function, `lift` lifts a list to a non-empty list transformation.

```
-- /Beware/: If the provided function
    returns an empty list,
-- this will raise an error.
```

```
lift :: ([a] → [b]) → NonEmpty a →
    NonEmpty b
lift f = fromList . f . toList
```

To migrate `lift` we inserted a ? at a higher order position.

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

GuiLT's first solution was that `0 < len o`, which was non sensible (how can a function generate bs on empty list?) thus we selected the second solution `len i == len o` which was strong enough to verify the three users of `lift` in the third and final migration round.

7 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. This restriction makes program invariants that fall outside of the decidable fragment non-expressible in the logic, thus the user has to assume such invariants via unchecked yet trusted assertions at verification time.

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. 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 fully 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 application of the liquid inference procedure.

Gradual Type Inference. Many systems study type inference in presence of 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. Like us, 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 the liquid inference algorithm the constraints imposed by gradual refinements cannot be ignored, since unlike Hindley-Milner inference, the liquid algorithm starts from the strongest solution (*i.e.* false) for the liquid variables and uses constrains to iteratively weaken the solution.

Error Reporting. Inference algorithms are prone to misleading error messages [Wand 1986], thus many algorithms have been proposed to to improve error reporting employing various techniques like counter-example generation [Nguyen and Horn 2015] heuristics [Zhang et al. 2015], or learning [Seidel et al. 2017]. Unlike all these techniques, developed for the purpose of error reporting, we uncover a novel application of gradual typing for error explanation, after observing that concretizations of gradual types embed the inconsistencies that lead to refinement type errors.

8 Conclusion

We presented gradual liquid types a gradual refinement type system where the gradual refinements range over the finite templates of liquid typing. As a consequence of this range restriction the system allows for algorithmic checking and inference. More importantly, we describe how the concretizations of the gradual refinements can be used for interactive error reporting and thus aid program migration. We conjecture that our idea of using gradual typing for error reporting, which is agnostic of refinement types, generalizes to any gradual system.

References

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

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.*

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

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

Supplementary Material. 2017. Technical Report: Gradual Liquid Types.

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

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.*

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

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

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 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*.

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

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*.