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 (predefined) set of predicates. This range restriction allows for an algorithmic inference procedure where all (finite) concretizations of the gradual refinements are exhaustively checked. Due to exhaustive search we have at hand the concretizations (*i.e.* the concrete refinements) that rendered the program gradually type safe. We observe that these concrete refinements contain all the potential inconsistencies of the non-gradual refinement type checking, thus can be used for error explanation. We implemented gradual liquid types in `GuiLT`, a tool that interactively presents all safe concretizations to the user and we used it for Haskell to Liquid Haskell, user-aided migration of three commonly used list manipulation Haskell libraries.

1 Introduction

Refinement types [Rushby et al. 1998; Xi and Pfenning 1998] allow for lightweight program verification by decorating existing program types with logical predicates. For instance, the type `{ x:Int | x /= 0 }` describes non-zero values and can be used to validate at compile time the absence of division-by-zero runtime errors. Liquid types [Rondon et al. 2008] bring refinement typing in real programming [Vazou et al. 2014a] by allowing for efficient type inference. The type `{ x:Int | k }` now describes integer values refined with a variable predicate `k`, which will be solved based on the uses of `x` to a concrete refinement drawn from a finite domain of template refinements.

The attractiveness of liquid types is usability. Specification only requires refinements of top level program types and verification is automated by SMT solvers. Thus, in theory users should not be aware of the internal, refinement 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. The system should (potentially erroneously) decide whether to blame the function definition or the client on an wrong function application and error reporting inevitably exposes 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 `{ v:Int | ? }` describes integer values for each occurence of which there *exists* a proper concrete refinement. Thus, to decide type safety, the type system 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 Lehmann and Tanter [2017]'s gradual refinement types system where the gradual refinement ? ranges only over the finite domain of the liquid refinement templates. This (strong in theory but realistic in practice) restriction allows for an algorithmic implementation of gradual refinement types, via exhaustive search for the solutions of the user specified ?. More importantly, exhaustive searching also produces certificates of validity that we use to report errors on a liquid type system. When liquid type checking generates an error on a function, the user may insert ? on function preconditions and then the gradual cerificates are used to explain locally exactly what conditions should be met at each occurrence of each argument (§ 2). The idea of using gradual typing for error reporting is novel and, since it is agnostic of the refinement typing framework, we believe it just generalizes to standard typing.

Our first contribution is to formalize the semantics and algorithmic inference of gradual liquid types and prove that it correct and it satisfies the gradual criteria (§ 4).

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

Our third contribution is to use `GuiLT` for user-aided migration of three existing Haskell libraries (1260 LoC) from Haskell to Liquid Haskell. This experiment supports our claim that gradual liquid types allows for an efficient implementation (??s required for 228 functions) interactively used for error explanation (§ 6).

2 Overview

We start with an overview of gradual liquid typing. First we start by presenting refinement types (§ 2.1), liquid types (§ 2.2) that allow for refinement inference and gradual refinement types (§ 2.3) that allow for gradual refinements. We observe that liquid types suffer from bad error messages and gradual refinement types lack inference and an algorithmic implementation. We address these problems by combining both systems to gradual liquid types (§ 2.4).

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

2.1 Refinement Types

We start by describing how refinement types are used to check the absence of divizion-by-zero. For the shake of this overview, let's assume that the division operator is only defined for division with positive numbers.

```
(/) :: Int -> {v:Int | 0 < v} -> Int
```

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

```
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)
```

Refinement typing accepts the definition of the function `divIf` as safe, *i.e.* there are no violations of the preconditions of `(/)`. To take this decision, the system performs three verification steps. (1). First, it uses the code and the specifications to generate refinement subtyping constraints. (2). Then, it reduces the subtyping constrains to logical implications, *i.e.* verification conditions (VC), that if are valid, then the subtyping constrains hold. (3). Finally, it is using an SMT solver to automatically check the validity of the VCs.

Step 1: Constraint Generation In the first step, two subtyping constraints are generated, one for each call to `(/)`. Both constraints check that under the environment that contains 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.* `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 ∧ not 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, in the constraint generated inside the the branch the guard is known to be true (*i.e.* `b`), while in the **`else`** branch, it is known to be false (*i.e.* `not b`).

Step 2: Verification Conditions In the second step, each constraint is reduced to a logical verification condition (VC), that intuitively checks that assuming all the refinements in the environments, the refinement of the left hand side implies the one in 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 ∧ not b ⇒ v = 1-x ⇒ 0 < v
```

Step 3: Implication Checking In the final verification step, an SMT is used to check the validity of the generated VC, and thus decide decide the safety of the source program. Here, the SMT decides that both VCs are valid, thus the system concludes `divIf` is safe.

Extensive description of these three verification steps can be found in the literature [Knowles and Flanagan 2010; Vazou et al. 2014b]. which, for completeness, is summarized in the background section (§ 3.1) of this paper.

2.2 Liquid Types

Safety of `divIf` crucially relies on the guard predicate, as generated by the precise refinement of the result of `isPos`. But what if the guard function `isPos` was an imported, unrefined function? Does there exist a safe refinement type specification for `divIf`?

Let's now use liquid typing to infer a type for `divIf`. Before verification, liquid types are using refinement type variables, here k_x and k_o, for the unknown refinements of the input and the output types respectively.

```
import isPos :: x:Int -> Bool

divIf :: x:{ Int | k_x } -> {o:Int | k_o }
divIf x = if isPos x then 1/x else 1/(1-x)
```

Once the unknowns 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 initial program type checks. Inference proceeds in two steps. First, as in § 2.1 the proper subtyping constraints are generated. Second, the liquid variables `k` appearing in the constraints are solved, so that all the constraints are satisfied.

Step 1: Constraint Generation Exactly as in § 2.1, the first step of inference combines the refinement type specifications with the source code to generate subtying constraints, that now contain liquid refinement variables. Inference of `divIf` generates the below constraints.

```
x:{k_x}, b:{b}     ⊢ {v | v=x   } ≤ {v | 0<v }
x:{k_x}, b:{not b} ⊢ {v | v=1-x } ≤ {v | 0<v }
```

For space, we write `{v | p}` to denote `{v:t | p}` for some type `t` and we further omit the refinement binding from the environment, simplifying `x:{x | p}` to `x:{p}`.

Step 2: Constraint Solving In the second step, inference solves the liquid refinement variables `k` so that the constraints are satisfied. The solving procedure takes as input a finite set of refinement templates $\mathbb{Q}^\star$ that capture the refinements relevant for verification and are abstracted over program variables. For example, the below template set $\mathbb{Q}^\star$ describes ordering predicates.

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

Next, for each liquid variable, the liquid inference instantiates the set $\mathbb{Q}^\star$ with all the program variables in scope, to generate well sorted predicates. Instantiation of the above $\mathbb{Q}^\star$ for the liquid variables k_x and k_o leads to the following 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 conjustion of all the instatiated templates

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

Repeatedly is filters out predicates in 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 }
```

In § 3.2 we summarize the formal background of liquid type inference required for this paper, where we simplify the procedure to reason about instatiated set of templates (*i.e.* $\mathbb{Q} = \bigcup_{k^x \text{ liquid variable}} \mathbb{Q}^x$).

For now, we observe that the inferred (`false` precondition) type for `divIf`, though correct, is not intuitive, in that a used would rarely write such a type. So, to understand the inferred type, the user needs to now some theory behind the automated inference. Worse, the inferred type of `divIf` depends on its clients. For instance the existence of a positive client `divIf 1` would establish a positive precondition (*i.e.* `0 < x`) to `divIf` rendering the its definition unsafe and the client safe. Thus, liquid inference needs to (potentially erroneously) decide whether which code to blame in case of failure. These challenges are not unique in liquid typing, on the contrary they are appear in most type inference engines. Next, we use, for first time, gradual typing to the rescue.

2.3 Gradual Refinement Types

Here, we use a gradual, *à la* Lehmann and Tanter [2017], precondition to the type of `divIf`. The precondition is written as `x:{Int | ? }` and means that the program is type safe *iff* for each occurrence of the argument `x`, there exist a valid instantiation (*i.e.* concretization) of the gradual refinement.

```
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 can contain gradual refinements.

```
x:{?}, b:{b}     ⊢ {v | v=x  } ≤ {v | 0 < v}
x:{?}, b:{not b} ⊢ {v | v=1-x} ≤ {v | 0 < v}
```

Step 2: Gradual Verification Conditions Each subtyping reduces to a verification condition, where each type `x:{ ? }` translates intuitively to an existential refinement (`∃ p. p x`). For soundness and satisfaction of the gradual criteria, as summarized in § 3.3, the existentials cannot range over arbitrary refinements but only over refinements that 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.

```
(∃? p_then. p_then x) ∧ b     ⇒ v=x   ⇒ 0 < v
(∃? p_else. p_else x) ∧ not b ⇒ v=1-x ⇒ 0 < v
```

Step 3: Gradual Implication Checking Checking the validity of the generated GVCs is an unsolved problem. In the `divIf` example, we can easily find existential certificates that render the GVCs valid.

```
p_then x ↦ 0 < x     p_else x ↦ x ≤ 0
```

More importantly, since the certificates are logical predicates, we can present them to the user as the conditions under which `divIf` is safe.

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] describes 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 as the solution explanation.

2.4 Gradual Liquid Types

To algorithmically solve GVCs we use the finite predicate domain of liquid typing to simply perform exhaustive search. 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 constrains 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** $\mathsf{p_{then}}$ `x` $\mapsto$ `0<x`, $\mathsf{p_{else}}$ `x` $\mapsto$ `0<x`:

```
x:{0<x}, b:{b}     ⊢ {v | v=x   } ≤ {v | 0<v}
x:{0<x}, b:{not b} ⊢ {v | v=1-x} ≤ {v | 0<v}
```

2. **Concretization for** $\mathsf{p_{then}}$ `x` $\mapsto$ `0<x`, $\mathsf{p_{else}}$ `x` $\mapsto x \leq 0$:

```
x:{0<x}, b:{b}     ⊢ {v | v=x   } ≤ {v | 0<v}
x:{x≤0}, b:{not b} ⊢ {v | v=1-x} ≤ {v | 0<v}
```

Step 3: Constraint Solving After we concretized the constraints, 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 $\mathsf{p_{then}}$ `x` $\mapsto$ `0<x` and $\mathsf{p_{else}}$ `x` $\mapsto x \leq 0$ or $\mathsf{p_{else}}$ `x` $\mapsto x < 0$. Thus, type checking reports that `divIf` is safe, but also it provides to the user the certificates (concretizations of $\mathsf{p_{then}}$ and $\mathsf{p_{else}}$) as an explanation of the safety.

In § 4 we formalize these three inference steps and prove the correctness and 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 conjuctions 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 conjuctions 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 $2x(2^{|\mathbb{Q}|}) = 32$ constrains for complete inference ($2x|\mathbb{Q}| = 8$ checked by default in the implementation).

Summary To sum up, gradual liquid types address two main problems of gradual refinement types. They allow for algoritmic 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 for the valid conretizations which, as later discussed, can be used for error reporting (§ 5) and user-aided program migration (§ 6).

3 Technical Background

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).

Constants	c	::=	$\wedge \mid \neg \mid = \mid \ldots$
		$\mid$	`true` $\mid$ `false`
		$\mid$	$0, 1, -1, \ldots$
Values	v	::=	$c \mid \lambda x.e$
Expressions	e	::=	$v \mid x \mid e\ x$
		$\mid$	`if` x `then` e `else` e
		$\mid$	`let` $x = e$ `in` e
		$\mid$	`let` $x{:}t = e$ `in` e
Predicates	p	::=	e
Basic Types	b	::=	`int` $\mid$ `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.

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 a 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.

Denotations Each type of λ^e_R denotes a set of expressions [Knowles and Flanagan 2010]. 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:

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

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

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

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

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

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

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.)

where $\cdot \hookrightarrow^\star \cdot$ is the reflexive and 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* $\vdash e{:}t$. These rules include three kinds of relations 1. typing, 2. subtyping, and 3. well-formedness.

1. *Typing:* $\Gamma \vdash e{:}t$ *iff* $\forall \sigma \in [\![\Gamma]\!].\sigma \cdot e \in [\![\theta \cdot t]\!]$.
   ET:♣ could skip those "here's the formula in words"♣
   Under a typing environment Γ, e has type t *iff* for any substitution σ in the denotation of Γ, $\theta \cdot e$ET:♣ ...what? ♣. 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* $\forall e, \sigma \in [\![\Gamma]\!]$. if $e \in [\![\sigma \cdot t_1]\!]$ then $e \in [\![\sigma \cdot t_2]\!]$.
   Under a typing environment Γ, t_1 is a subtype of t_2 *iff* for every expression e and each substitution σ in the denotations of Γ, if e belongs to the denotation of $\sigma \cdot t_1$ then it also belongs to the denotation of $\sigma \cdot t_2$. 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:* A type t is well-formed in environment Γ, noted $\Gamma \vdash t$, *iff* all the refinements in t are well-typed boolean expressions.

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

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

Rule W-Base overloads the relation isValid(·) to refer to well-formedness on base refinements. For now, we define a base refinement $\{x{:}b \mid p\}$ to be 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.

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_R^e:
$\text{isValid}(\Gamma \vdash \{x{:}b \mid p_1\} \preceq \{x{:}b \mid p_2\})$ 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 assumptions of Γ, the predicate $\hat{p}_1$ implies the predicate $\hat{p}_2$.

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

iff

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

```
Infer :: Env -> Expr -> Quals -> Maybe Type
Infer Γ̂ ê ℚ = A <*> t̂
  where
    A = Solve C A₀
    A₀ = λκ. ℚ
    (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♣

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`.

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 $\mathbb{Q}$.

ET:♣ I'm HERE♣

3.3 Gradual Refinement Types

Gradual refinement types [Lehmann and Tanter 2017] extend the refinements of λ_R^e to include the dynamic refinement ? and describe sound type checking but no inference. With a goal to extend liquid inference for gradual refinements types,

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

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

here we summarize the basic concepts of gradual refinement typing as described in [Lehmann and Tanter 2017].

Syntax Figure 5 describes the gradual refinement typed language $\lambda^{\tilde{p}}_{G}$ that merely extends the predicates with the dynamic $?$. The predicates of $\lambda^{\tilde{p}}_{G}$ are either static predicates (p) (*e.g.* the predicates of λ^{e}_{R} or of $\lambda^{\hat{p}}_{L}$) or a conjuction ($p \wedge ?$) of a static predicate and the dynamic $?$. For the gradual predicate $p \wedge ?$ to be meaningful, the static part p should be local with respect to the value that is refined.

Locality In the context of a refinement $\{x{:}b \mid p\}$, the predicate p is local with respect to the variable x that is refined, when there exists a value v for which p is true. We use the explicit syntax $p(x)$ to explicitly declare the variable x refined by the predicate p. For example in $\{x{:}b \mid 0 \leq x\}$ the refinement predicate is $(0 \leq x)(x)$ while in $\{y{:}b \mid 0 \leq x\}$ the refinement predicate is $(0 \leq x)(y)$. Using the explicit syntax we define a refinement predicate to be local when there exist a value that satisfies the predicate, under and (well-typed) substitution σ:

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

Specificity To ensure that the gradual predicate $p \wedge ?$ respects the staticrefinem part p, the gradual predicate maps only to (local) predicates stronger than p. We say that p_1 is stronger (or more specific *à la* [Lehmann and Tanter 2017]) than p_2, written $p_1 \preceq p_2$ when p_1 is true when p_2 is true:

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

Concretization The concretization function $\gamma(\cdot)$ maps gradual refinement predicates to the set of the static predicates they represent. Concretely, we define the concretization of the explicit predicate $\gamma(p(x))$ to be the singleton set $\{p\}$ while the concretization of the gradual precidate the explicit predicate $\gamma((p \wedge ?)(x))$ is the set of local predicates more specific than p.

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

We naturally extend the concretization function $\gamma(\cdot)$ to map gradual refinement types and environments to sets of refinement types and environments, respectively.

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

$$
\begin{array}{rcll}
\check{p} & ::= & \hat{p} & \text{Static Liquid Predicate} \\
 & | & \hat{p} \wedge ? & \text{Gradual Predicates, where } \hat{p} \text{ is local and known}
\end{array}
$$

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

Denotations Each gradual type denotes the set of (gradual) expressions that have this type. The denotation of the gradual basic type $\{x{:}b \mid p \wedge ?\}$ includes all expressions that satisfy dome local predicate stronger than p. The denotations of the rest gradual types are the same as the respective static denotations.

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

Type Checking Figure 2 is used to type gradual expressions $\tilde{\Gamma} \vdash \tilde{e}{:}\tilde{t}$ with the validity predicate lifter to $\text{isV}\tilde{\text{alid}}(\cdot)$. Concretely, $\tilde{\Gamma} \vdash \tilde{t}_1 \preceq \tilde{t}_2$ is valid *iff* there are Γ, t_1, and t_2 in the respective concretizations of $\tilde{\Gamma}$, $\tilde{t}_1$, and $\tilde{t}_2$ so that $\text{isValid}(\Gamma \vdash t_1 \preceq t_2)$. Similarly, $\tilde{\Gamma} \vdash \tilde{t}$ is valid *iff* there are Γ and t in the respective concretizations of $\tilde{\Gamma}$ and $\tilde{t}$ so that $\Gamma \vdash t$.

$$
\begin{array}{rcl}
\text{isV}\tilde{\text{alid}}(\tilde{\Gamma} \vdash \tilde{t}_1 \preceq \tilde{t}_2) & \doteq & \exists \Gamma \in \gamma(\tilde{\Gamma}), t_1 \in \gamma(\tilde{t_1}), t_2 \in \gamma(\tilde{t_2}). \\
 & & \quad \text{isValid}(\Gamma \vdash t_1 \preceq t_2) \\
\text{isV}\tilde{\text{alid}}(\tilde{\Gamma} \vdash \tilde{t}) & \doteq & \exists \Gamma \in \gamma(\tilde{\Gamma}), t \in \gamma(\tilde{t}).\text{isValid}(\Gamma \vdash t)
\end{array}
$$

4 Gradual Liquid Types

Figure 6 defines gradual liquid core language $\lambda^{\check{p}}_{GL}$ whose predicates are gradual predicates, following Figure 5 where the static part is a liquid predicate from Figure 3. The elements of $\lambda^{\check{p}}_{GL}$ are both gradual and liquid. For example expressions $\check{e}$ could also be written as $\tilde{\hat{e}}$.

Our goal is to lift the liquid function `Infer` (defined in § 3.2) on gradual types, so that $\text{In}\check{\text{fer}}\ \check{\Gamma}\ \check{e}\ \mathbb{Q}$ returns a type $\check{t}$ so that $\check{\Gamma} \vdash \check{e}{:}\check{t}$. Next (§ 4.1), we apply the abstract gradual typing [Garcia et al. 2016] methodology to get the intuition of $\text{In}\check{\text{fer}}$'s methodology. Then, we provide an algorithmic description of $\text{In}\check{\text{fer}}$ (§ 4.2) and the properties that $\text{In}\check{\text{fer}}$ satisfies (§ 4.3).

4.1 Abstract Gradual Typing

We define the function $\text{In}\check{\text{fer}}$ using the abstract gradual typing methogology of [Garcia et al. 2016]. That is, to lift a function `f :: Type -> a` on types to a function $\tilde{\texttt{f}}$ `::` $\tilde{\text{Type}}$ `-> [a]` on gradual types, we apply `f` to all the elements of the concretization of the input type:

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

Direct application of this methodology raises two crucial questions:

1. If f is a composition of functions f = g · h, should we lift f, g, or h?
2. How do we algorithmically enumerate all the elements of the set $\gamma(t)$?

Next we answer these two questions in the specific application of gradual liquid types.

Our goal is to lift the function Infer from the static to the gradual language. As described in § 3.2, Infer calls the static functions Cons and Solve, which, in turn, calls the function isValid. Thus, to lift Infer, which exactly function should we lift? We answered this question via trial-and-error.

Assume we directly lift Infer. Then we have

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

This definition of $\check{\texttt{Infer}}$ is too *imprecise*, *i.e.* it rejects expressions that should be accepted. Consider for instance the following expression $\check{e}$ that defines a function f with a gradual user-provided specification:

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

Since there is no $e \in \gamma(\check{e})$ so that the definition of f above type checks, for any $\mathbb{Q}$ we will get Infer {} e $\mathbb{Q}$ = Nothing even though $\check{\texttt{Infer}}$ {} $\check{e}$ $\mathbb{Q}$ should simply return Int. Thus, Infer is not a good function to get lifted.

Our next attempt was to lift isValid, since gradual type checking (of § 3.3) already used the gradual validity checking $\tilde{\text{isValid}}$. To our surprise, lifting validity checking lead 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 conjuction:

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

This property does not hold for the lifted validity checking

$$\begin{gathered}\tilde{\text{isValid}}(\tilde{\Gamma} \vdash \{v{:}b \mid \tilde{p}\} \preceq \{v{:}b \mid \tilde{p}_1\}),\\ \tilde{\text{isValid}}(\tilde{\Gamma} \vdash \{v{:}b \mid \tilde{p}\} \preceq \{v{:}b \mid \tilde{p}_2\})\\ \not\Rightarrow\\ \tilde{\text{isValid}}(\tilde{\Gamma} \vdash \{v{:}b \mid \tilde{p}\} \preceq \{v{:}b \mid \tilde{p}_1 \wedge \tilde{p}_2\})\end{gathered}$$

Because, for any logical predicate q, that here represents the information captured in the typing environment and the left-hand-side predicate $(q \Rightarrow p_1 \wedge q \Rightarrow p_1) \Rightarrow (q \Rightarrow p_1 \wedge p_2)$, but $((\exists q.(q \Rightarrow p_1)) \wedge (\exists q.(q \Rightarrow p_1))) \not\Rightarrow (\exists q.(q \Rightarrow p_1 \wedge p_2))$.

Our final (, and successful) attempt was to lift Solve which lead to a provably (§ 4.3) sound and complete algorithm.

We lift $\check{\texttt{Solve}}$, as described in the Abstract Gradual Typing methodology, that is, we by solve for all static constraints in the concretization of the input gradual constraint.

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

We use $\check{\texttt{Solve}}$ to derive a gradual inference procedure

```
Infer :: Env -> Expr -> Quals -> Set Type
Infer Γ e ℚ = { t' | Just t' <- A <*> t
                   , A ∈ Solve A₀ C }
  where
    A₀ = λκ. ℚ
    (t, C) = Cons Γ e
```

First, the liquid method Cons derives the typing constraints $\check{C}$ and maybe 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, to get back the set of the gradual inferred types.

4.2 Algorithmic Concretization

We complete the definition of $\check{\texttt{Infer}}$ by defining an algorithmic concretization function for of a set of constraints $\gamma(\check{C})$.

Concretization of Predicates Recall that the concretization of gradual refinement predicates is defined as

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

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 of $\mathbb{Q}$. We define the algorithmic concretization function $\gamma_{\mathbb{Q}}(\check{p}(x))$ as an intersection of the powerset of the finite domain $\mathbb{Q}$ with the gradual concretization function.

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

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

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

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

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

Finally, we use $\gamma_{\mathbb{Q}}(\cdot)$ to define the algorithmic lifted solve function.

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

4.3 Meta-theory

We prove in [Material 2017] that the inference algorithm $\check{\texttt{Infer}}$ preserves the correctness properties of [Rondon et al. 2008] and the criteria for gradual typing of [Lehmann and Tanter 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 SMT-decidable logic, $\check{\Gamma}$ be a gradual liquid environment, and $\check{e}$ be 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 prove the theorems in § **??** of [Material 2017]. We note that, unlike the $\texttt{Infer}$ algorithm that provably returns the stronger 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 Criteria for Gradual Typing

Finally we prove that the gradual algorithm $\check{\texttt{Infer}}$ satisfies the criteria for gradual typing, as outlined in [Siek et al. 2015]. The proof can be found in § **??** of [Material 2017].

Static Gradual Guarantee First, we define use the algorithmic concretization function to define precision or gradual types.

Definition 4.2 (Precision of Gradual Types). $\check{t}_1$ is less precise than $\check{t}_2$, written as $\check{t}_1 \sqsubseteq \check{t}_2$ *iff* $\gamma_{\mathbb{Q}}(\check{t}_1) \subseteq \gamma_{\mathbb{Q}}(\check{t}_2)$.

We recursively extend the precision to expressions and environments and prove that less types are inferred for less precise expressions.

Theorem 4.3 (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 $\check{t}_{2i} \in \check{\texttt{Infer}}\ \check{\Gamma}_2\ \check{t}_2\ \mathbb{Q}$.*

Fully Annotated Terms Next, we prove that the gradual inference algorithm $\check{\texttt{Infer}}$ behaves exactly like the static algorithm $\texttt{Infer}$, when called with expressions and environments without gradual parts.

Theorem 4.4 (Fully Annotated Terms). *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, $\check{\texttt{Infer}}\ \hat{\Gamma}\ \hat{e}\ \mathbb{Q} = \emptyset$.*

Gradual Dynamic Guarantee Finally, we prove that inference always succeeds on *dynamic terms*, intuitively terms with only gradual (*i.e.* ?) specifications.

First, we define the function $\lfloor\cdot\rfloor$ that removes the refinements from types and terms:

Definition 4.5 (Unrefined Type & Terms). Unrefined types and terms represent base types and lambda calculus terms. The function $\lfloor\cdot\rfloor$ removes refinements from types:

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

We recursively apply $\lfloor\check{e}\rfloor$ on the types inside the expressions. We use the standard Hindley-Milner type inference on unrefined expressions. $\lfloor\Gamma\rfloor \vdash \lfloor e\rfloor{:}\lfloor t\rfloor$.

On the other side, the function $\lceil\cdot\rceil$ turns types and terms into dynamic, by replacing all the refinements with ?.

Definition 4.6 (Dynamic Type & Terms). Dynamic types are refined with only ?.

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

Dynamic terms cast all constants, casts and user provided specifications to dynamic types.

$$\lceil c\rceil = c :: \lceil ty(c)\rceil \qquad \lceil e :: t\rceil = \lceil e\rceil :: \lceil t\rceil$$
$$\lceil \texttt{let}\ x{:}t = e_x\ \texttt{in}\ e\rceil = \texttt{let}\ x{:}\lceil t\rceil = \lceil e_x\rceil\ \texttt{in}\ \lceil e\rceil$$

We prove that inference succeeds for any dynamic term $\lceil e\rceil$ as long as, e has a basic type and all the user specified refinements are local.

Theorem 4.7 (Gradual Dynamic Guarantee). *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 $\check{\texttt{Infer}}\ \lceil\Gamma\rceil\ \lceil e\rceil\ \mathbb{Q} \neq \emptyset$.*

The locality requirement is needed so each specified predicates p belongs into the concretization of $?\,p \in \gamma(?)$, which let us use subtyping to convert between refined and dynamic types $\Gamma \vdash t \preceq \lceil t\rceil$ and $\Gamma \vdash \lceil t\rceil \preceq t$.

5 Application I: Error Explanation

We implemented $\check{\texttt{Infer}}$ as $\texttt{GuiLT}$, 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, $\texttt{GuiLT}$ 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 $\texttt{GuiLT}$'s interface ET:♣ more than the interface, illustrate the use of the tool♣ via the list indexing example. Consider the standard list indexing function $\texttt{xs!!i}$ that returns the $\texttt{i}$th element of the list $\texttt{xs}$.

```
(!!) :: 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

Figure 7. GuiLT for error explanation from here.

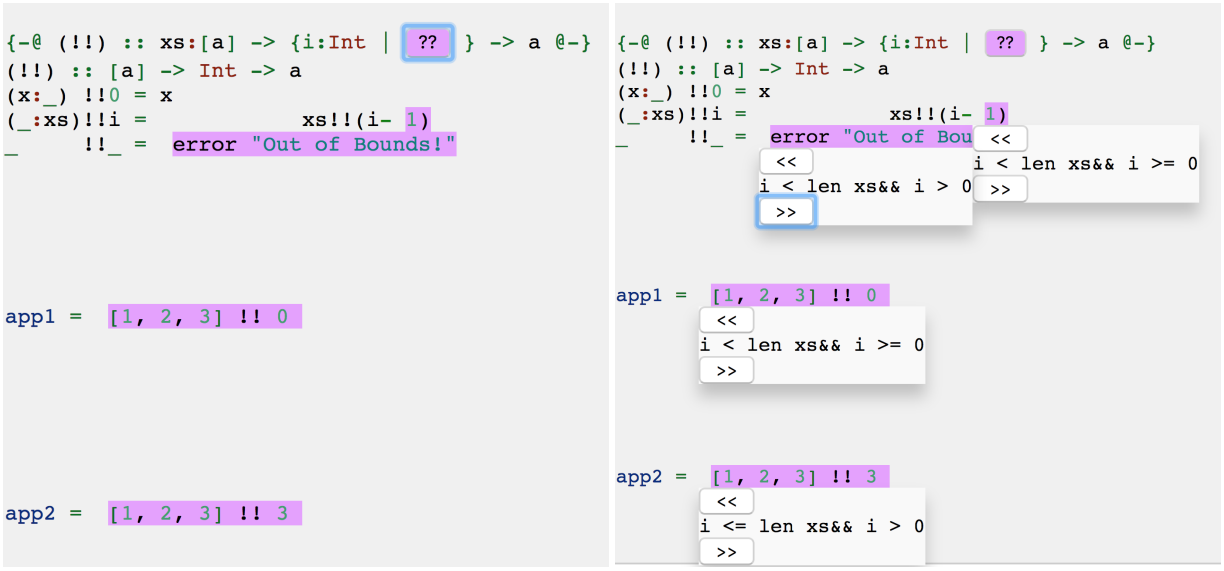
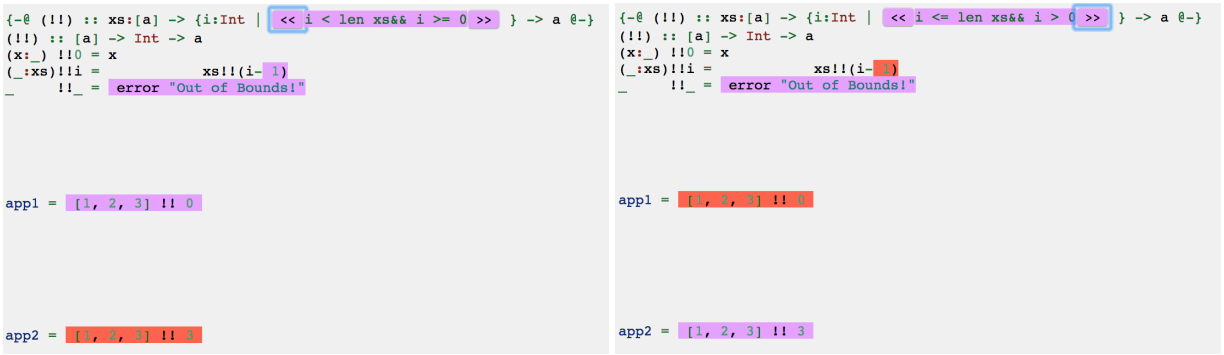


Figure 8. GuiLT for static typing from here.



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.

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

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.

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.

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.

`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

```
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

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

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

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

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

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *TSE*.

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

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. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. https://doi.org/10.1145/2692915.2628161

H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *PLDI*.